GRIFON - A Graphical Interface to an Object Oriented Database.

A Thesis by: Padraig Moran B.Sc.

Supervisors: Mr. Renaat Verbruggen,
Mr. Michael Ryan.

Submitted to the
School of Computer Applications
Dublin City University
for the degree of
Master of Science
August 1991

DECLARATION : No portion of this work has been submitted in support of an application for another degree or qualification in Dublin City University or any other University or Institute of Learning.

Acknowledgements

I would like to thank IBM Ireland Information Services Ltd. (IISL), for their generous scholarship, help and support which made it possible for me to pursue this research. I would like to express my gratitude to the staff at IISL, especially Mr. John Dermody for his guidance and interest, and Mr. John Murphy for his invaluable advice and support. I would like to sincerely thank my supervisors, Renaat Verbruggen and Michael Ryan for their interest, support, patience, advice and encouragement. Thanks are also due to Dr. Alan Smeaton for materialising relevant research papers, seemingly out of the blue. I would like to express my thanks to Rev. T. Moran for the use of his computer, which substantially helped the writing of this thesis. Special appreciation is due to all in number 9, and especially Joanne, without whose support and sustenance this thesis would never have materialised.

Finally I would like to acknowledge the support of my family, colleagues, and friends.

To Mam, Dad and Nan who between them gave me the encouragement to pursue my studies and the invaluable chance to do so.

Abstract

The aim of the research outlined in this thesis is to establish what type of interface would be most suitable for object oriented databases. In particular it examines how graphical interface technologies might be used to present the database in a clearer form.

In support of the research, a prototype interface system has also been developed to a commercial database to illustrate the practicality of the development of such an interface, and the increased effectiveness of the resultant system.

The thesis outlines the features provided by the interface, the benefits accrued from such a system, and the problems associated with its development.

Finally, it examines how such a system fits into the current work being carried out in the area of user interaction with databases.

Table of Contents

APPENDICES

Figures

Chapter 1 - Introduction

1.1 Background

The design of traditional database applications has largely been determined in response to the needs of typical business applications. Before the advent of databases, applications supported persistence of the application's data, but this data tended to be stored in an ad-hoc filing system constructed by the application itself. These filing systems tended to have their own idiosyncratic format and structure and usually were incompatible with the data from other applications. Consistency was definitely not a feature of such systems with data often being stored redundantly.

Creation of new applications required difficult extraction procedures for information from many disparate sources. Programs were very much dependent on the structure of the data, making these structures difficult to change or improve without substantial modification to the application.

Although applications are still very diverse, requiring different data, the introduction of database systems has improved the application-development process in large data intensive environments through their provision of a single, uniform view of data expressed in structure independent terms. In addition to this SQL (Structured Query Language) or other query languages offer a simple, yet powerful interface to the data. Because of their highly structured and defined format and the features provided for data sharing, databases facilitate the creation of integrated applications more easily than ever before. With the advent of database management systems (DBMS), the problem of data being replicated in many different application data files is minimised, as the DBMS acts as a single repository of all the data, making it available to all applications requiring it. Furthermore there is one set of highly tuned routines for data formatting and access rather than separate sets of routines, of varying quality, provided by each application program.

Most available and extensively used databases adhere to the relational data model. Although they are adequate for most applications, in particular applications with large quantities of similarly structured data, they do not provide facilities for all application areas. Business applications tend to require the storage of large quantities of similarly structured data with efficient access methods to it. This data tends not to be very complex in structure and the relational model copes adequately with it.

In relational databases there are normally 3 data access/query languages provided -

- **SQL - Structured Query Language.**
  This is probably the most recognisable from a user's point of view as it provides the standard interface for accessing/querying the data, for most applications.

- **DML - Data Manipulation Language.**
  In a relational DBMS (RDBMS) this would be a level below SQL. This would be a programmatic language provided for manipulation of data/indices etc.

- **DDL - Data Definition Language.**
  Initially some form of language is required to create the database. This includes both the creation of the database schema and the insertion of data into this structure.

The data in an relational database (RDB) is stored in a table like structure with records being stored in tuples, or rows, consisting of attributes.

General concerns about the maintenance of the data are also provided for in the form of integrity constraints, security management, concurrency and transaction management and recovery issues, being dealt with. In the scenario where there are thousands of records of the same structure without complex inter-relationships in the data (eg. Bank accounts), the facilities provided by a RDBMS are more than adequate.

However with the continuation of the computer revolution, computers have been moving into application areas more diverse in data requirements than business and finance.

Recently the availability of high-performance graphics workstations has increased the breadth and complexity of data-intensive applications that are being attempted. Examples of these are

Computer Aided Design (CAD)

Computer-Aided Software Engineering (CASE)
Office Information Systems (OIS)
Computer Aided Engineering (CAE)

In an electrical CAD system, the typical environment includes tools such as *Schema-Capture Editors, Design-Rule Checkers* and *Circuit-Layout Programs.* With all of these sub-systems, massive amounts of data storage will be required. In addition to this, the storage, the level of complexity of these programs, and of the data used, has grown far beyond what traditional database systems are prepared to handle.

For example, the circuit-layout programs would include graphical representations of various electrical components, such as resistors, ICs, transistors etc., the design-rule checker would include complex pre-conditions about the positioning and inter-connection of such electrical components. In addition to this, performance details on the components needs to be stored, to mimic a designer's drawing board. Impressive high-quality graphics need to be included in the schema-capture editor. All of this data is very diverse and as such has differing storage requirements from a database. For a successful system to be developed, these individual tools need to be interlinked in terms of the different data, making the programs extremely complex.

Currently, the application programs in design environments store data in application-specific file structures. The state of the art here is roughly at the same stage that existed, in the 1960's, before the emergence of database technology in the business data-processing world. There is wide agreement [ZDO90] that what is needed is an extension of current database technology that can provide the same boost for developing these complex applications now, that occurred in the commercial data-processing world back then. The development of *Object-Oriented Databases*, to a large extent, has been driven by this need.

In the area of CASE, some form of data storage is required to aid programmers in the design and development of large systems. Many of these systems require large amounts of data storage, facilitating the representation of the complex inter-connections between the system modules (eg. E-R diagrams, data-flow diagrams etc.). Computer-Aided Software Engineering tools are examples of such complex systems, as are CAD systems.

With respect to the engineering field, the introduction of high-powered workstations has promoted the graphical representation of *real-world* engineering systems on the computer. In addition simulation of external effects on such engineering structures can be created and investigated. This is very prevalent in such areas as car & aircraft manufacture and building design. Apart from the benefits that accrue to the engineer, problems of data structuring and storage present problems to the S/W designer/programmer.

As an example of the complexity of such new systems, consider the following [SME91]. Boeing Avionics in the USA are currently developing the new 777 aircraft. This is a 390 seat revolutionary aircraft which is due to be launched in 1995. It will have 130,000 unique parts, all digitally designed using an advanced, distributed CAD system, running on 8 of IBM's largest mainframes. This system will be accessed by 2,200 workstations. Each component will be modeled in 3-D in the system, and information on the dimensions, weights, strengths, etc. of each will be stored. The system will be developed to such a level, that components can be assembled on the screen. This level of digital assembly can continue to the extent that a digital model of the aircraft can be built on the screen. When it is complete, the CAD information can spill over into a CAM system (Computer-Aided Manufacture) which will deal with the manufacture of the plane.

The designer's work involves designing components digitally, calling up other parts and testing them together. Constraints on the positioning of parts will need to be taken into account. In addition to all this, some form of storage will be required to keep track of old versions of parts.

The design system entails imaging, simulation and inter-connecting different components. An advanced storage system would be required to support this type of application. It would need to take into account the make-up of components, the complexity of individual parts, the logical and meaningful grouping of parts together. In addition, copies of parts would be required, to allow subsequent access to old versions.

Although this is an extremely large system, applications of its scale and type, and indeed smaller ones have highlighted the need for databases supporting increased semantics and complex-data support.

In the area of factory facility management, much work is being carried out into completely automating all stages of production from design to manufacture to stock control. In conjunction with this, the area of plant management is being examined, to aid the maintenance departments. The aim is to be able to represent the complete layout of a factory floor on a computer, including the power, gas and water lines, the plant and machinery positioning and constraints on such positioning. Graphical workstations facilitate the graphical representation of such a system but the problems involved with the representation of the data is more difficult. In addition to this the types of queries which would be used are also more difficult. They would tend to be more of a *Spatial* nature, referring to positions of things and distances rather than just requiring pieces of data from the database.

e.g.

List all the machines within 6 feet of Power Line 123.

or

Where is Machine X located ?

The answers to such queries are not readily accessible. The manner in which the data is stored (ie. with data inter-connections) would make the retrieval of such answers significantly easier. The facility for a database to be able to carry out an operation or calculation on the data in the database itself in order to return the required result, would be useful, removing such processing from the application.

Cartographic databases are becoming very popular, even in cars. They represent a map with all of its details in a computer database. Again here spatial-type queries are used, asking such questions as :

List all the Hospitals within 10 miles of point (10,10).

To facilitate the processing of such queries through a relational database, the data might be stored containing :

Details of each item on the Map

Position of each item on the Map

The application would then need to deal with the establishment of the distances of the item from another position on the map, applying the scale. What is needed is some database facility which can inherently connect different pieces of data in the database and return processed information from it rather than just acting as a retrieval system. Object-oriented systems, support the idea of methods associated with the data. So in this case, a method could be associated with a point which will calculate its distance from another point, applying a scale automatically. The idea of adding some intelligent processing power to the database has the dual effect of adding more functionality to the database, making more information available to different applications, and simplifying applications which access the database, as some of the functionality which they would have previously provided are now provided by the database.

In a relational database all entities which can have a location would be required to have a *point* attribute holding the co-ordinate. There is nothing in this structuring to reflect the semantics of the situation. In the factory management example, outlined above, information would be maintained about the locations of all the machinery. However, all the machinery and facilities can be classified. Yet each piece of machinery has the common feature that it has a location or a point. In relational databases, this meaning cannot be captured. In object-oriented based systems it can.

In an object oriented database, the data to represent the objects, machinery and facility lines in a factory might be structured as shown in figure 1.1. As queries might be addressed to all items referring to position and distance between it and other items, the data concerning location can be extracted into a general class which all other objects in the system can inherit.

All of these application areas have one thing in common - they place many demands on database technology, including the ability to model very complex data and the ability for the data to evolve without disruptive effects on the current application base, which conventional databases cannot cater for. These demands in turn place a requirement on the system to provide an appropriate level of extensibility to easily capture application-specific data semantics and mechanisms for incremental development of the database structures. In addition to this, as mentioned in the examples, these application areas contain many complex interconnections with many complex constraints on the way these interconnections can be made.

Having agreed that current relational database technologies are not adequate for complex data-intensive applications, where increased semantics can help simplify the data representation, over the past 10 years, much work has, and still is, being carried out in the search for an alternative to relational databases for such application areas, which can not only cater for such

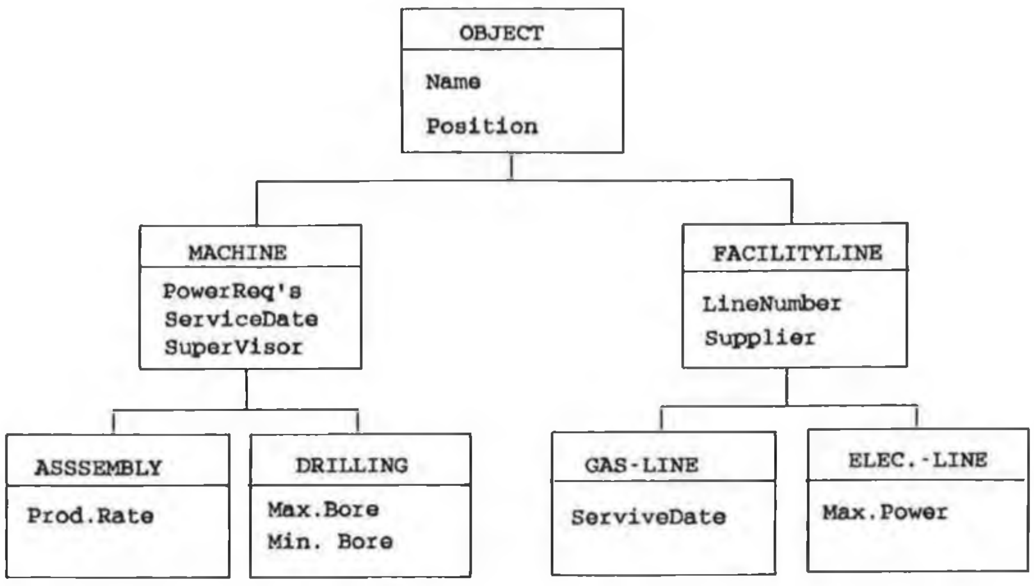


Figure 1.1 - Classes in facility management example.

increases in data structure complexity, but also enable the capture of more meaning of the application-environment in the data. To date the database technology which best caters for these needs is Object Oriented.

1.2 Overview of Object-Oriented Databases (OODB)

A problem exists with the current research into OODB's because of the lack of standard and agreement on exactly what an object-oriented database is. In fact there is a large degree of confusion about exactly what the term *object-oriented* is. This lack of agreement is can largely be traced back to the foundations of the whole object-oriented area. Its origin is largely attributable to three areas of research. Firstly in *Programming Languages*, then in *Artificial Intelligence* and finally in the *Database* area.

1.2.1 Programming Languages

OO programming languages can be traced back to what is regarded as the first OO language Simula-67. This was developed as a simulation language and included specific constructs for object-oriented programming [DAH68]. From there, research into OO programming languages has taken two different paths. The first, was the development of new object oriented languages being built from the ground up on object-oriented principles with no direct relationship to existing languages. The most notable result of this research is Smalltalk [GOL83]. Smalltalk is generally regarded as the best example of the implementation of the OO concepts in a language and tends to be used as a yard-stick against which all other OO languages are measured. Smalltalk is implemented around the idea of an object and implements the main OO principles of :

Encapsulation
Inheritance
Data Abstraction
Information Hiding

The intent of an object is to **encapsulate** the representation of a problem domain entity which changes state over time. **Abstraction** deals with how an object presents this representation to other objects, suppressing non-essential details. In most OO languages, this is done through the provision of methods or operations to operate on the core details. The stronger the **abstraction** of an object, the more details which are suppressed by the abstract concept. **Information hiding** means that such details should be kept secret from other objects, so as to better preserve the abstraction modeled by the object. **Inheritance** makes provision for the inheritance of the state and behaviour (representation) of one class of object by others. All of these principles are outlined in more detail in chapter 2.

Many other languages were developed along these lines, eg. EIFFEL [MEY83]. However, by far the most popular area of language development was the addition of OO concepts and features to conventional programming languages. Three existing programming languages contributed to the development of new OO programming languages. LOOPS and Object-Lisp developing from LISP, C++ and Objective-C from C and CLASCAL and Object Pascal from PASCAL. Many advantages accrue from this approach, mainly the success and expertise experienced in these languages. However OO concepts and the development procedure is totally different from development in procedural languages and development in one of the OO-procedural languages (e.g. C++, Object Pascal) can result in unstructured and inefficient code. Unlike the *'Pure OO Languages'*, the implementation of the OO principles is less rigid, and they are based around an essentially non-OO language. This is often regarded as only a *half-hearted* approach. However, their success cannot be denied and often they serve as a stepping stone to a purer OO development environment.

1.2.2 Artificial Intelligence

In the area of artificial intelligence, the work carried out by Minsky [JAC83] into frames as a method of knowledge representation have resulted in knowledge representation languages such as KEE and ART, both being based on OO principles. Many experimental knowledge representation languages have been developed in the last 10 years, and many of them are frame based. The fundamental organising principle in such schemes is the packaging of both data and procedures into structures related by some form of inheritance mechanism. This is very similar to the aforementioned **encapsulation** ideas. Frames in themselves, exhibit many OO features with the *A-KIND-OF* relationship to support inheritance and the *IS-A* relationship to support the OO concepts of attributes and the objects. For Example, A BMW IS-A-KIND-OF Car, and as such inherits all the properties of a Car.

1.2.3 Database

In the database area, in the early 1980's, the limitations of relational database technology were recognised and work went on into finding a new model. It is commonly agreed that the method of data representation in a RDB is not very realistic, representing data from the application environment in an abstract manner. Very little of the true meaning of the data is captured, with a significant mapping of the data from the application area to the database. Features in the data such as commonality of composition of entities have been ignored. The fact that in the real world most objects tend to be specialised versions of a more general object, is omitted.

e.g. - Current A/C and Savings A/C are specialisations of a Bank A/C.
- Employee and Student are specialisations of Person.
- Car and Truck are specialisation of Vehicle.

These facts spurred on the development of a data model encompassing more of the meaning or semantics in the data - the semantic data model was a response to these needs. SDM [HAM81] was one database system developed based on the semantic model. Although its implementation never became widely popular, it did act as a *spring-board* for future database systems implementing many of the SDM ideas. The core ideas behind SDM are outlined in chapter 2.

Since their initial development in the early 1980's, semantic databases have not really proved successful as an alternative to the existing relational databases. Existing database systems tended to be concerned primarily with large quantities of similarly structured atomic data. This data consisted of text or numbers on which queries could be issued. The relational data model proved sufficient for such storage.

However, in the last five years, the power of personal computers and workstations has increased to such a level that new applications are placing ever-increasing demands on their underlying data storage facilities. Developments in the areas of artificial intelligence, computer-aided design, computer-aided engineering, office automation, computer-aided manufacture, and many others has highlighted the inadequacies of current database technology. The semantic data model served to highlight the lack of real semantic power of such systems. The new applications required efficient storage for complex structures, consisting of graphical images, CAD drawings, design plans, schedules, digitised speech and many other new dynamic and non-atomic data types.

These new object based databases were initially used in a transparent manner. The user had no conception of how the data was stored. Initial systems using them tended to plug them in as a *back-end* storage facility. As the uses for OODBs have expanded in the last few years, database researchers have become interested in developing interaction techniques to the new databases, in a similar manner to the way relational databases provide SQL, DDL, DML and 4GL-type interfaces. However, since the initial development of SQL [CHA76],*Human-Computer Interaction* techniques have developed substantially.

1.3 Overview of Human-Computer Interaction

In the early days of computers, only qualified personnel could access and program them through switches, paper-tape and punch-cards. Over the years, keyboards and visual display units (VDU) have become more commonplace supporting textual interfaces, initially, and in recent times, graphical interaction techniques. The saying ***A picture says a thousand words,*** has inspired the development of new and revolutionary computer interfaces. Over the years, with the new and more powerful computing hardware, software developers have been developing new techniques by which users can express their needs and computers can express the answers, in simple, clear and user-friendly manners. Databases in particular have proved difficult to interact with for the uninitiated. The are repositories of information. However the retrieval of the required information is not a simple matter.

1.4 Conclusions

This thesis outlines an approach taken to representing the data structures and information contained in a commercially available OO database, utilising the current hardware and software tools to the full.

Chapter 2 introduces the principles of object-oriented database, indicating their development from the core ideas of semantic databases. As no standard data model exists for OODBs, the features of an OODB which are regarded by many experts as the core elements are described. Commercially a number of OODB systems have been developed. Four such systems are described. They represent four different approaches taken to implementing OO principles in database systems. One of these systems, ONTOS, formed the basis for the development of the prototype user-interface - GRIFON (GRaphical InterFace to ONtos).

In chapter 3, a look is taken at human-computer interaction techniques. The different interaction techniques are examined, outlining their merits and demerits. Some of these approaches to interface design have been applied in some database interface systems, and three such systems are described.

GRIFON was developed in an attempt to see the feasibility of developing graphical user interface to an OODB, and to determine the form which such an interface would take. Chapter 4 describes GRIFON, outlining the features which it provides, highlighting the benefits to be accrued by such an interface.

The development of GRIFON posed a number of problems, and brought up a number of issues concerning the development of windowed graphical interfaces. The hardware and software used in the development are described in chapter 5, as is the development process and the representation of the database in GRIFON. The tools used in the development brought the complete development procedure under an object oriented framework and added a new simple structure to the normally complicated process of developing windowed applications. The manner in which GRIFON interacted with ONTOS is described, presenting an outline of the processes involved to carry out the different GRIFON tasks. For the graphical representation of the database structures in a simple manner, various node positioning and graph drawing algorithms needed to be developed. These are presented, detailing their functionality and the benefits accrued from them.

Although GRIFON presented the features of the database in a clear and pleasant manner, a number of issues arose concerning OODBs, user-interfaces and their use together. In addition, a number of possible future developments which might be carried out in this area became apparent. These are discussed in chapter 6.

Chapter 2 - Object Oriented Databases.

2.1 Introduction

Object-Oriented Databases (OODB) have, as outlined in the previous chapter developed as a result of 3 main areas of study : Artificial Intelligence, Programming Languages and Databases. Database research and development is probably primarily responsible for current form of OO databases.

In this chapter the principles behind the Semantic Database Model will be examined, which, although not very successful in the early 1980s when it appeared, has since become the corner stone for most of the current OO databases. Although no standard formal OO data model exists, at present, many of the reputed experts in the OODB field agree on a number of core elements which must be include to warrant a database being termed *Object-Oriented*. Many of these elements are inherited from the developments in the semantic database area. These elements will be outlined and a number of supplementary features which might be included to enhance any basic OODB system to enhance it, will be described.

So far very few software development companies have got involved in the development of OODB systems. Much of the work has been carried out in academic research environments. However, of those that are available, four of them will be described in this chapter. The four systems are representative of the different approaches that have been taken to the development of new OODB systems. They are :

POSTGRES - extending the relational data model into the OO realm [ROW87c][STO86b].
EXODUS - providing DB tools for developing a customised DBMS [CAR86b][CAR88].
GemStone - adding persistence to Smalltalk V, creating a database [MAI86a][MAI90].
ONTOS - supplying database functionality to C++ [AND90][ONT90].

These systems are commercially available, and although very different in their implementation, they each implement most of the core OO features outlined.

As mentioned above, semantic database technology has contributed significantly to the development of OODBs. This chapter will now proceed to outline the features of the semantic data model. Many of the features associated with semantic Dbs have been adopted into what has been called the core elements of the OO data model. These will be outlined, as will some supplementary features which are advantageous in OODBs. Finally the four sample commercial OODB systems will be described, outlining their individual features, stating how they build on the core and supplementary features recommended for OODBs.

2.2 Semantic Database Model

Semantic data models have emerged from a requirement from engineering, scientific and computing environments for more expressive conceptual data models. Current generation data models lack direct support for relationships between the data, data abstraction, inheritance, constraints, unstructured objects, and the dynamic properties of an application - the data requirements of the application is not always known at the time of its development. Although the need for data models in both industry and with richer semantics is widely recognised, no single approach has won general acceptance. Since the mid-1970s a number of semantic data models have been proposed. These have ranged from the Entity-Relationship model [CHE76] which is basically a semantic model that unifies features of the traditional models to facilitate the incorporation of semantic information, to TAXIS, SDM [HAM81] and DAPLEX [SHI81] which implement new principles. SDM, as with the others, implements the following common features [HAM81][KIN84][SHI81]:

(i.) A database is to be viewed as a collection of entities that correspond to the actual objects in the application environment. There is a close association between the real world and it's database representation.

(ii.) The entities in a database are organised into classes that are meaningful collections of entities. This allows for the grouping of similarly structured data together,
e.g. Class Person would be a class to group all entities of the same structure, representing Persons.

(iii.) The classes of a database are not in general, independent, but rather are logically related by means of *inter-class connections*. The make up of one particular entity may include another.

For example, in the figure 2.1, a Company class may include the fact that it has a president who is an entity of the Employee class. Therefore there is a connection between the Company class and the Employee class.

(iv.) Database entities and classes have attributes that describe their characteristics and relate them to other database entities. An attribute may be derived from other attributes in the database. This last point encompasses the representation of the data

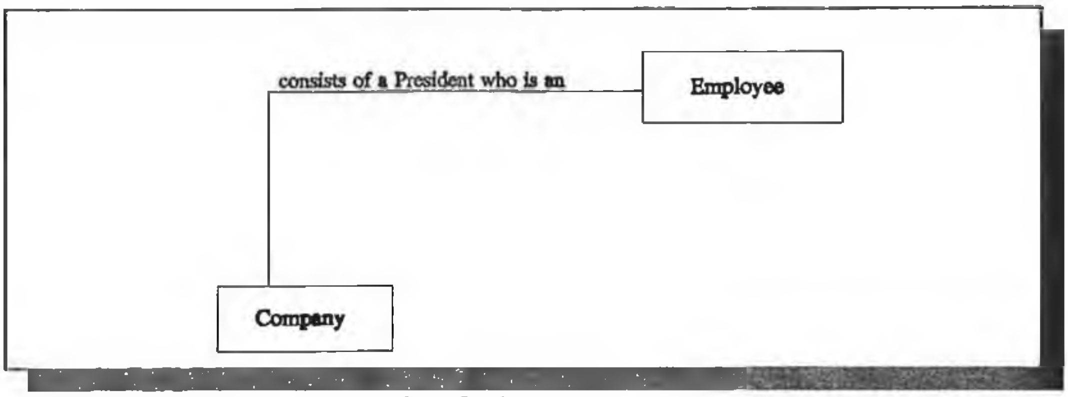


Figure 2.1 - One class may consist of others.

in a general structure with this generalisation acting as a core for data specialisation.

e.g. *Bank A/C* would be a general structure with attributes associated with it. However, *Current A/C* would be a specialisation of it, inheriting the attributes and features of a Bank A/C while adding the extra features which make it different (i.e. Check Book, Zero Interest Rate, Overdraft etc.).

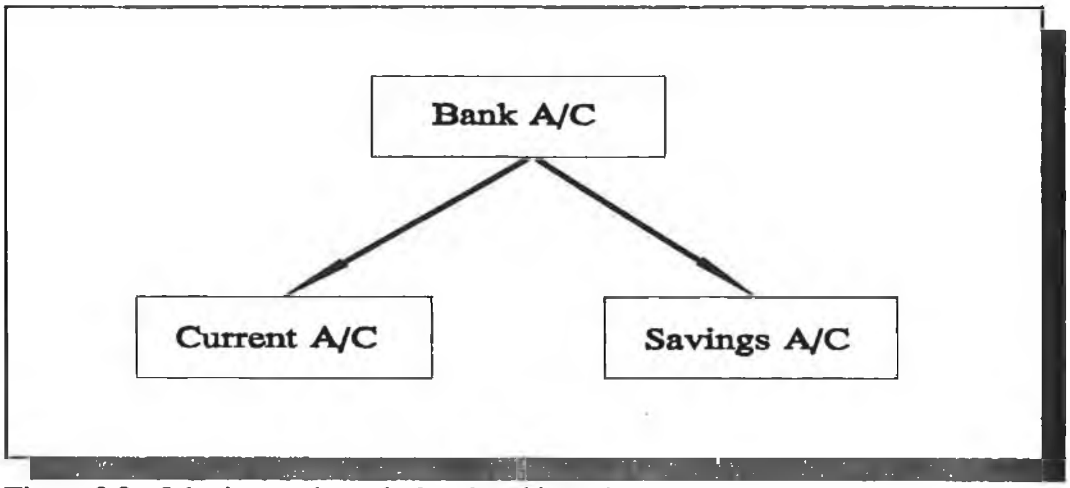


Figure 2.2 - Inheritance through the class hierarchy.

In the figure 2.3, class Vehicle is made up of an Engine and Manufacturer which themselves are instances of other classes. This represents the facility for one class to be composed of objects of other classes, enhancing the modelling ability of the system.

The different inter-class connections like inheritances from class to class and the facility for the make-up of one class with others, provides a good capturing of the semantics of the data. In addition to this, it provides a number of ways of viewing the data. Such facilities permit the simple construction of more complex structures.

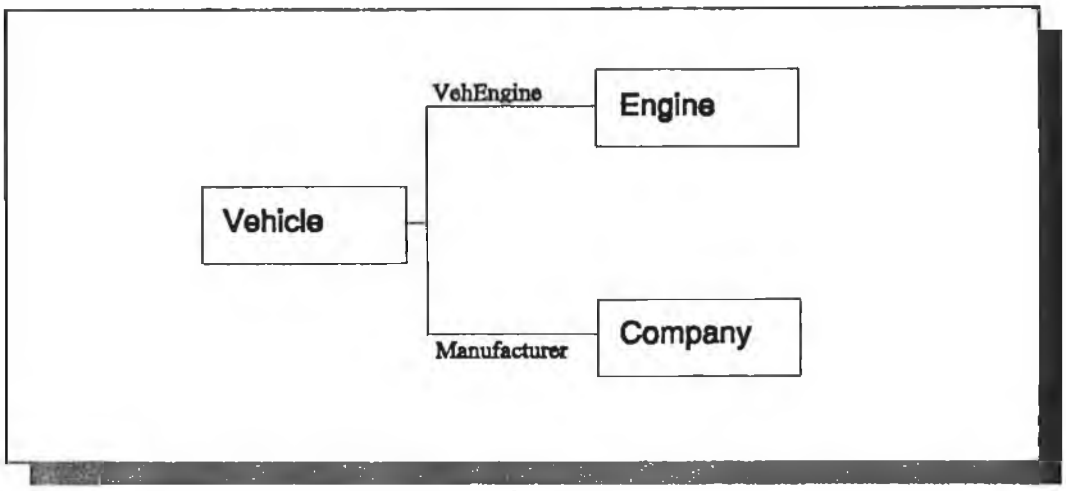


Figure 2.3 - A class may be composed of others.

(v.) There are several ways of defining inter-class connections and derived attributes, corresponding to the most common types of information redundancy appearing in the database applications. These facilities integrate multiple ways of viewing the same basic information, and provide building blocks for describing complex attributes and interclass relationships.

DAPLEX [SHI90], was developed as an alternative to SDM [HAM81]. It adheres to the same principles as SDM and like SDM provides a data programming language as well. Many others have carried out work in this area, CACTIS a Semantic/OO database [HUD90], Sembase [KIN84].

2.3 Object-Oriented Databases

Although many object-oriented database systems have been developed, with all of them incorporating many of the concepts associated with semantic databases, no standard data model has been arrived at. No OO data model has been established yet, unlike relational databases based on the relational model.

However, as research into OODB's has progressed, many of the proponents and experts in the field have established what core elements must be included to constitute a database being termed object-oriented. According to Won Kim [KIM90], much of the core object-oriented data model is viewed as a subset of the semantic data model. Yet, further features are included in the core OODB model to adhere to the generally accepted OO concepts.

2.3.1 Core Object-Oriented Database features.

Although the experts are in disagreement about exactly what an object-oriented database should be, they are all in agreement that the existence of the following features in a database, is required to allow it to be termed object-oriented [KIM90],[ATK89].

(i) **Object & Object Identifier.**
(ii) **Attributes & Methods**
(iii) **Class**
(iv) **Class Hierarchy & Inheritance**
(v) **Late Binding**
(vi) **Extensibility**
(vii) **Computational Completeness**
(viii) **DBMS Components**
(ix) **Query Facility**

2.3.1.1 Object and Object Identifier

The uniform treatment of any real-world entity as an object, simplifies the user's view of the real world. In addition, each object is associated with a unique *Object Identifier*. The idea of representation of real-world entities as corresponding objects in the database is as outlined in point (i) above in Semantic Databases.

In a RDB, a tuple has a key attribute associated with it as a means of access to the data. The key value would normally be unique, acting as a means of identifying a particular tuple in the database. However, this means that the identity of a tuple is based purely on it's value. In contrast to this, in OODBs, the inclusion of an object identifier means that an object has an existence which is independent of it's value. In addition to this, in an *Identity-Based* model, two or more objects can share a component. This is an important and useful feature especially where integrity is concerned.

Consider the following example, as shown in figure 2.4. A Person object has a name, an age and a set of children. Assume Peter and Ann both have a 15-year-old child named John. In real-life two situations may arise, Peter and Ann are parents of the same child or they are parents of two different children.

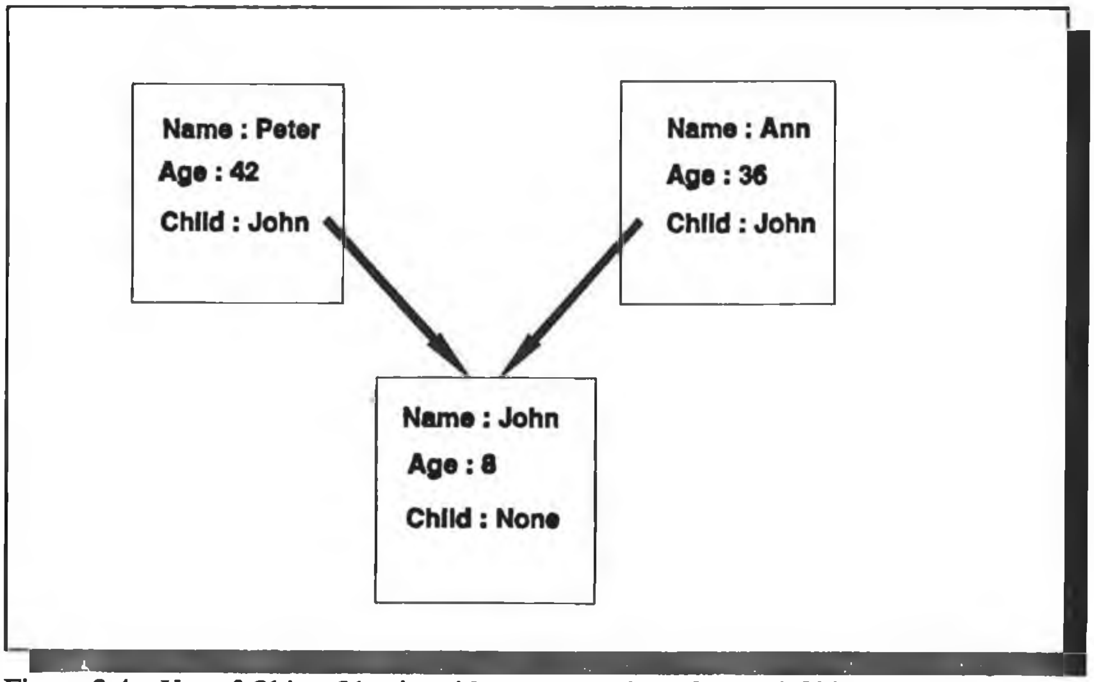


Figure 2.4 - Use of Object Identity aids representation of shared Objects.

In a model without object identity it would be impossible to represent the fact that Peter and

Ann are the parents of the same child. However in an identity based system two structures can share a common part if necessary, thus capturing either situation. From a database point of view, object sharing can result in an increase in the level of integrity in the database, with a change to one object with a shared part, automatically resulting in a change to the other objects involved. This integrity problem is something more difficult to cope with in models not supporting object-identity, and will require some integrity-rules to cope with it [ATK89].

Many commercial object oriented databases support an *Inverse-Relationship* feature which allows one attribute in a particular class to be linked to an attribute in a different class.
e.g. To represent the fact that Tom is Mike's manager in an Employee System, Mike can be stated as Tom's boss in Tom's object. Tom can be declared as Mike's sub-ordinate in his object. If an inverse relationship link is declared then any changes to the Manager attribute in Tom's object will result in a corresponding change in the value of the sub-ordinate attribute of Mike's object. ONTOS [AND90][ONT90] and GemStone [MAI90][MAI86a][MAI86b] are two commercial systems which support this idea.

2.3.1.2 Attributes and Methods

Every object has a state and a behaviour [KIM90]. The *State* of the object consists of the values taken on by the attributes of an object. The *Behaviour* of it is the set of methods which can operate on the state of it. For consistency, the attributes of an object are objects themselves (i.e. an attribute *Name* might be an object of class *String*). An attribute corresponds to a column of a relation in a relational database. The domain of an attribute may be any class, user-defined or primitive (see below). This contrasts with the relational model where the domain of an attribute is restricted to a primitive class (ie. Real, Character, Integer, Boolean etc.). The support for the storage of complex data is one of the more important features of OODBs.

For Example:
An Object of Class *Vehicle* might be 'Ford Escort'. It might take on the following attributes: The 2000 represents the value of the attribute *Weight*, FORD is the identifier of an object of the Class COMPANY, which is the value of the attribute *Manufacturer*, X1243 is the identifier of an object of Class VehicleDriveTrain which is the value of the attribute *DriveTrain* and Escort is the value of the attribute *Name* which is a character string.

Attribute Name		Domain	
Weight	(2000)	Real Number	(Primitive Class)
Manufacturer	(FORD)	class COMPANY	(User-defined Class)
DriveTrain	(X1234)	class VehicleDrvTrn	(User-defined Class)
Name	(Escort)	Character String	(Primitive Class)

Figure 2.5 - The construction of Class Vehicle.

The idea of attributes representing the state of an entry is consistent with the approach put forward in Semantic Databases, however it's implementation is more defined and practical. The one major difference between an OODB and a Semantic Database is the inclusion of methods.

Unlike database models proposed up to now, OODB's include the code to operate on the data, in the database also. This idea is totally foreign to conventional databases.

With respect to the area of programming techniques, in conventional 3GL programming languages, the application is designed around the functionality of the system rather that entities which are involved. OO programming concentrates on the data entities primarily and the functionality is looked at from the point of what operations are to be carried out on the data. So any application is designed around the data involved, with the application's functionality being represented by operators (methods) on this data.

In OODB's, the same is true, and instead of just storing the data associated with an application, the methods which operate on it are stored also. The advantages of this are evident when one considers the OO programming scenario. Now the whole of the applications including the data and the code (methods) are stored in the database. Code modularity and re-usability are obvious with an increase in productivity inherent. Methods consist of the code to access/modify the attributes in the object. The concept of data abstraction is adhered to, as access to the attributes is only possible by invoking the appropriate method. From a practical database point of view, the implementation of constraints on the data can be simply dealt with by some code in one of the methods, which, as mentioned above, is stored in the database also. Such constraints would normally be included in the application code accessing the data. So if a number of different applications need to implement similar constraints on the data to be written to the database, like a restriction on an age value (18 <= Work_Age < 65), then each of them needs to include it's own code to carry out the procedure. This implies redundancy of data. Such replication of code can be eliminated using an OODB.

In the context of the example as shown in figure 2.5, the methods associated with this Class (*Vehicle*) might be *DisplayName*, *DisplayDTrain* or *SetWeight*. These would be the only way that the application has of altering the values of the attributes in this object.

The concept of attributes, and the methods which operate on them is in accord with the OO principle of *Data Abstraction*. The behaviour of an abstract data object is fully defined by a set of abstract operations defined on the object - its interface. The user of the object does not need to understand how these operations are implemented or how the object is represented [SNY86]. The operations are, in effect, the methods provided.

Locality of data is ensured, as the data is known only inside it's own class and accessible only through the methods attached to the class. As the variables of classes are only known inside the classes in which they are declared, and by the methods which operate on them, the data is maintained locally.

2.3.1.3 Class

In Relational DB terms, a relational schema can be likened to a class in OODBs, providing a general structure for the storage of data. The records (tuples) in this relation can be compared to objects of a class in OO terms. However, where the relation is a means of structuring and storing data, the class, in an OODB is much more. It is an effective means of grouping together all objects of the same structure. This includes the same attributes and methods. Every object must belong to only one class and is an instance of that class. The value of an attribute of an object, since it is necessarily an object, must belong to a class. This class is called the *Domain* of the attribute. The notion of a class consisting of a set of attributes and methods which are permitted to operate on this data is in agreement with the OO principle of *Data Encapsulation.*

For Example, with the Class Vehicle in figure 2.5, the interface to an object of this class would be through function calls to either of the methods :

DisplayName, *DisplayDTrain* or *SetWeight*

Apart from these 3 methods, there is no other way in which a client module can access the attributes of an object of class Vehicle.

2.3.1.4 Class Hierarchy and Inheritance

Probably one of the most crucial features of OO systems is the facility to create new classes which are derived from existing classes. The new class (sub class) inherits all the attributes and methods of the existing class, called the **super class** of the new one. The user may specify extra attributes and methods for the new class, thus building on the super class - '*specialisation from the general*'. A class may have any number of sub classes. However, depending on implementation of the Database, a sub class may or may not be allowed to have more than one super class. Some systems restrict a class to having at most one super class. This is called *Single Inheritance. Multiple Inheritance* allows a class to inherit attributes and methods from more than one super-class. An obvious problem arises here if there is a conflict of names of attributes or methods which are inherited. Again as no formal model has been arrived at, the treatment of such a conflict is dealt with in a manner decided upon by the manufacturer of the database system. Some ordering of the super classes in the declaration of the new class will usually be used to decide how to solve the conflict [BAN87],[SNY86],[KIM90]. Of course this problem of conflict in inheritance can also occur in single inheritance, if there is a clash with the name of a new attribute or method being the same as the one which is inherited. However, this problem is easier to solve. The usual procedure would be to select the attribute or method nearest to the bottom of the tree (in the sub-class). In the multiple inheritance scenario the selection of the attribute/method is made more difficult because the two conflicting classes may be at the same level in the hierarchy and both equally entitled to have their attribute/method pass down the hierarchy. The process of inheritance results in a hierarchy being created, with a class inheriting data and operations from superClasses. In single inheritance this hierarchy is in the form of a Tree Structure called the inheritance tree.

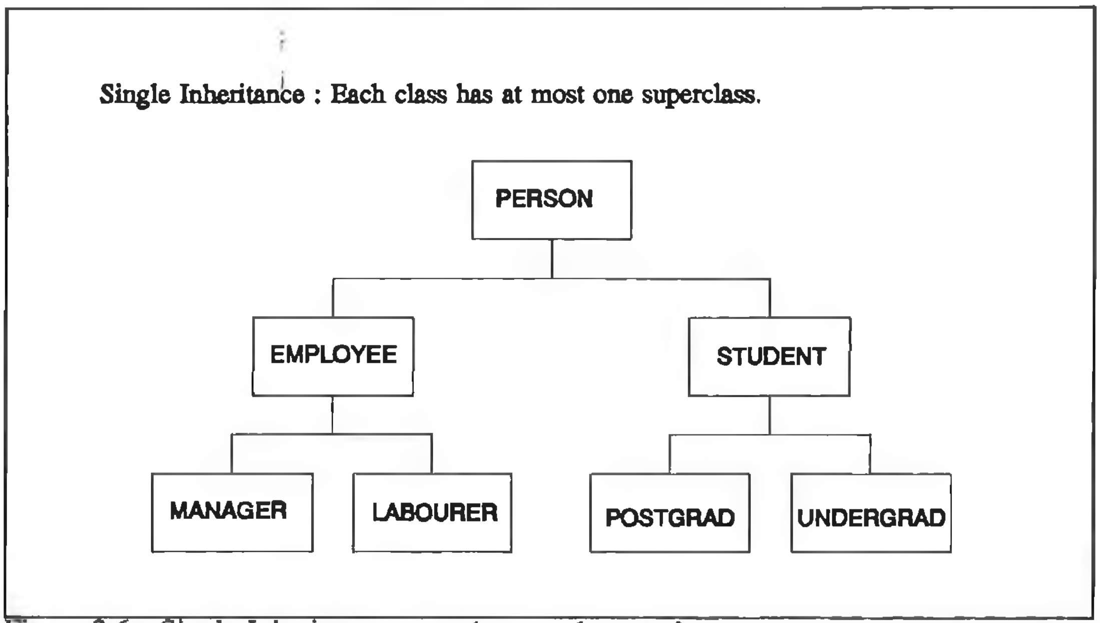
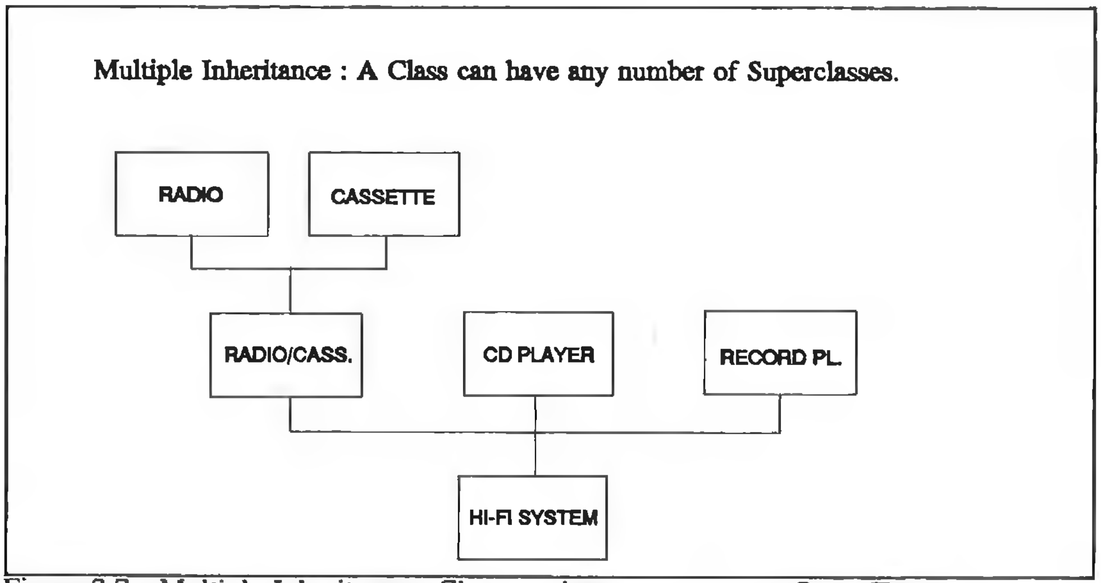


Figure 2.6 - Single Inheritance tree - At most 1 superclass.

Where there is *Multiple Inheritance* the Hierarchy is in the form of an *Inheritance Lattice*.[CAR90]


Multiple Inheritance : A Class can have any number of Superclasses.
RADIO
CASSETTE
RADIO/CASS.
CD PLAYER
RECORD PL.
HI-FI SYSTEM
Figure 2.7 - Multiple Inheritance - Class can have zero or more SuperClasses.

The advantages of the concept of inheritance and a class hierarchy cannot be overstated. The facility to have specialisation of data down through the hierarchy means a more natural structure for the data is defined.

For example, consider a database system to store information on Employees and Students. Employees consist of Names, Ages and Salaries and Age must be greater than 18. Students consist of a Name, Age and a Grades Point Average (GPA), but the Age must be greater than 13 and less than 19 years. Both Students and Employees can die, get married. Students can have their GPA computed and Employees their salary increased.

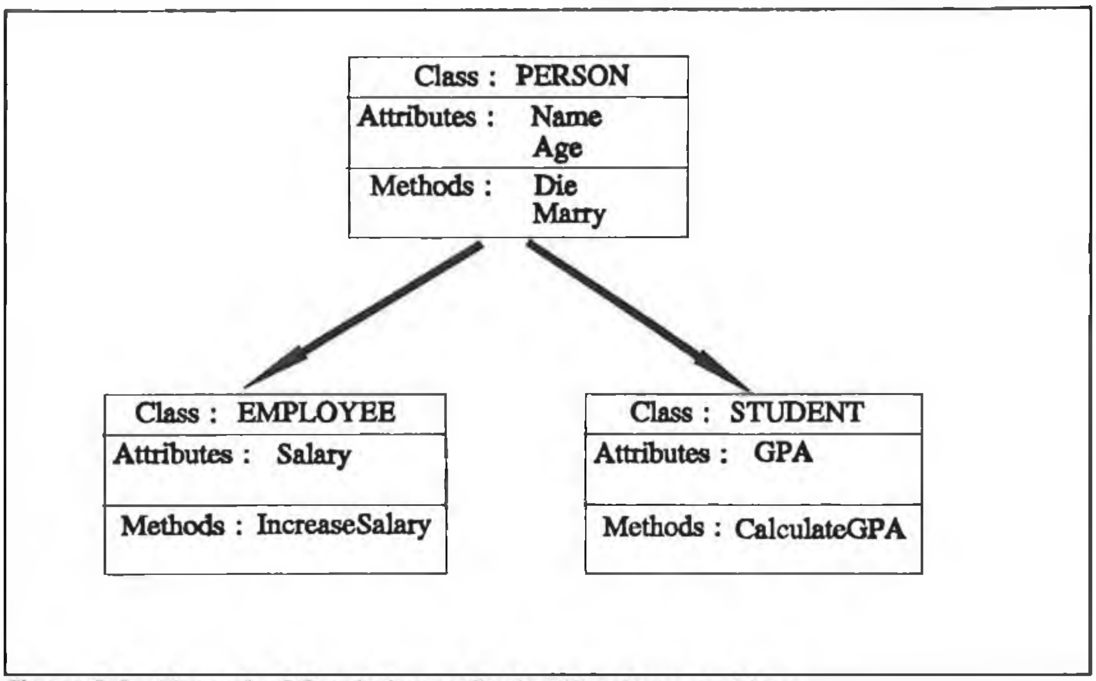


Figure 2.8 - Example OO solution to Student/Employee problem.

To store this information in a relational database there would be a relation for Employee with three properties, and a separate relation for Student, similarly with three properties. In the application there would be functions written to deal with the death of an Employee, his/her marriage and the salary increase.

Similarly the application would include code to deal with a Student's death, his/her marriage and the calculation of the grade points average. code would also be needed to deal with the validation of the Age value when an Employee/Student is being created.

Although the application and the data storage here would be quite adequate, the structure of the data in the RDB does not take into account the common properties in the two relations. The properties of Name and Age are common to both Employee and Student, however the structure of the data in the relational system treats it as if there were no similarities.

In an OO database system where there is inheritance, the commonality in the data could be extracted and maintained on its own, with the classes Employee and Student containing the data which is unique to them, while inheriting the common features from the parent class (e.g. Person). Such a structure in an OODB might be as displayed in figure 2.8. Here the methods Die() and Marry() deal with the situation where either an Employee or Student dies or gets married. The code to deal with validating the age of an Employee/Student will be included in the *Constructor*[1] for the appropriate class, which will also be included in the database. The OO technique of structuring the data and code results in a more modular system. The inheritance facilitates the grouping of common attributes and methods together, with more specific classes inheriting these attributes and methods from more general ones [ATK89]. The class hierarchy and inheritance concepts are central to OO models, however in addition to the class hierarchy, the *Class-Composition* hierarchy is included. The class-composition hierarchy is orthogonal to the class hierarchy. It usually has nothing to do with inheritance of attributes and methods. This hierarchy shows the relationships between attributes in a class and their domains. This concept is outlined in figure 2.9.

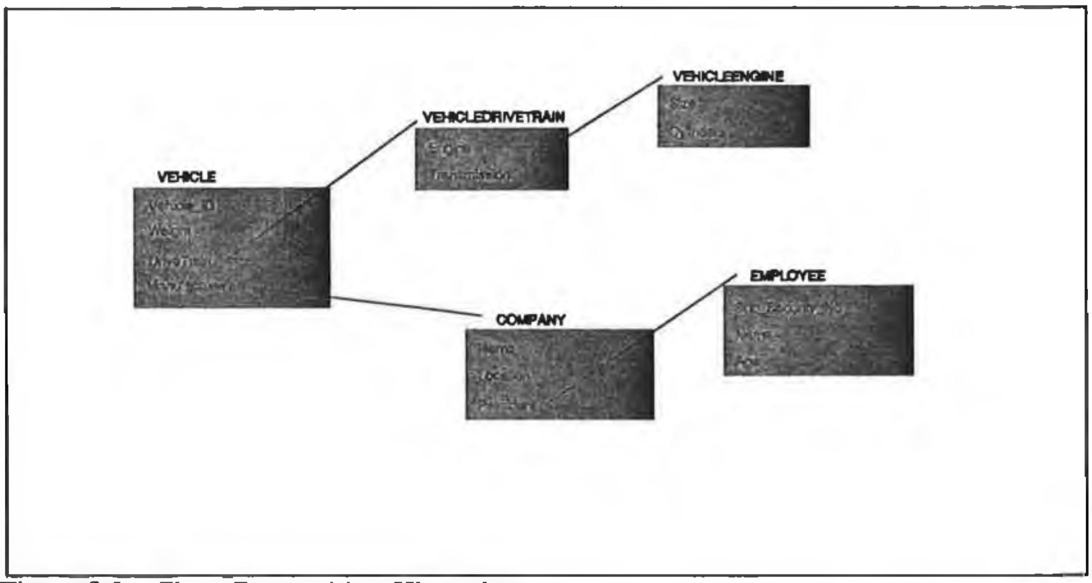


Figure 2.9 - Class-Composition Hierarchy.

Class Vehicle is composed of four attributes. **DriveTrain** is an object of another class -

[1] A Constructor function or method is invoked whenever a new object of a class is being created. In C++, the constructor function must be given the same name as the class.

VehicleDriveTrain, while **Manufacturer** is an object of class **Company**. Similarly the other classes are made up of other objects. It is these relationships which are displayed in the class-composition hierarchy.

The aforementioned points outline the generally accepted criteria for a DB to be classed as OO. However according to Atkinson et al [ATK89] a number of other features must be present.

2.3.1.5 Late Binding

The name of an operation should not be bound to a particular program until the last possible moment. For example, an object-oriented database is used for storing objects to be displayed in a CAD system. Such a system could exploit the generalisation to specialisation structure of the OO hierarchy. Graphical objects could be built up from a simple point, to a line, to a shape etc. In a conventional application the program needs to know what type of object it is currently dealing with. So to display a square requires different processing from displaying a circle. If the system consists of hundreds of different types of shapes, different named functions need to be provided for the display of each one and applications need to cope with knowing which type is to be displayed.

In an OO CAD system, each class of object might have specific code to deal with displaying it. A method would be provided for the display of a square, and similarly, one would be provided for the display of a circle. However, these two methods could be called *Draw()*. In the application program using the database, a call might be made to draw an object. The program does not need to know what type of object it is drawing, all that is required is to call the *Draw()* function, and the system will deal with executing the appropriate code associated with the object invoking it.

According to Atkinson [ATK89], the decision as to which program to execute to draw a shape should not be made until run-time to allow the maximum amount of flexibility in the program. In the diagram below, the class *Shape* is a generic one. It provides a method for displaying a *Shape* object. *Circle* and *Square* are specialised sub-classes of a *Shape*. They inherit a display method from *Shape*, but this is superseded by new methods of the same name, *Draw()*, a specific one for each class. The application programmer can develop the system to deal with

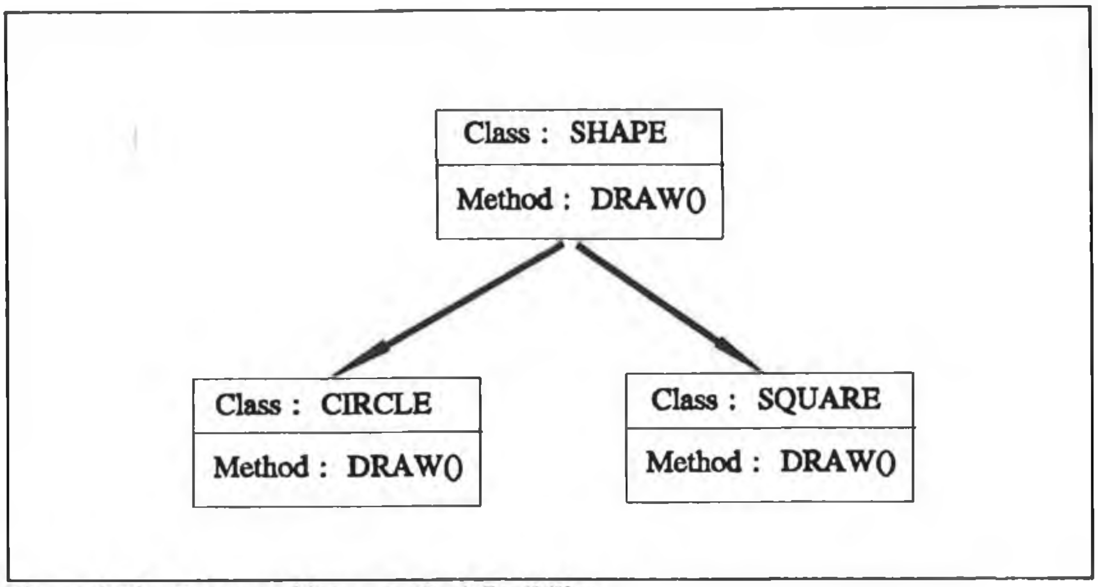


Figure 2.10 - Polymorphism - method flexibility.

objects of class *Shape*, when the program is executed, if these shapes happen to be squares or circles, the database will dynamically bind the appropriate function to the object. The binding will occur at run-time, thus increasing the flexibility of the application and making the application simpler, in that this feature directly supports the OO concept of *Polymorphism* where different classes can have the same message passed to them but result in a different function being executed [STR86]. *Different messages or function calls mean different things to different classes.*

In the *Circle* and *Square* example,

Circle->Draw()

and

Square->Draw()

are having the same Draw message passed to them but the result will be different with a circle being drawn in the first case with a square being drawn in the second.

2.3.1.6 Extensibility

Through the facilities for complex types, class hierarchies and inheritance outlined above the feature of extensibility is implied. New types can be created based on existing types - the new types being more complex. It is important that there is no distinction in usage between system defined and user-defined types. In the implementation of the database itself, there may be significant difference between the way system and user-defined types are supported but this should be invisible to the application and the application programmer. Atkinson et al [ATK89] feel that the extension through the provision of user-defined types/classes is enough.

However, in the search for extensibility, much work has gone into the development of Extensible Databases which not only allow for the extension of the data in the database in both structure and content, but also facilitate the incremental and extensible development of the database system itself. GENESIS [BAT86] and later EXODUS [CAR88] make provisions for such extensibility. They both allow for the creation and extension of the query language and storage/access mechanisms. Not only are they extensible in provision of user-defined types but the programmer/database creator can build a database management system tailored to the application areas needs.

2.3.1.7 Computational Completeness

This feature tends to be supported sufficiently by most of the available OODB's as they are connected closely to computationally powerful programming languages. Most available OO programming languages support statistical and trigonometric operations in addition to basic mathematical functions. The computational power of programming languages is through their ability to combine sequences of operations to create more complex ones. It would be preferable that the system provide more than just simple computations on atomic values.

In relational databases, SQL provides basic addition, subtraction, multiplication and division facilities which can be included in a query [DAT83][DAT86]. As a programming language, it would not be regarded as computationally complete. However, the majority of applications to RBDs involve embedded SQL calls. This means that the computational power of the application language can be exploited.

Operations on arrays, statistical operations, business functions and scientific functions would enhance a database query language, but with respect to OO databases they provide facilities to enhance the computational power through the close links they have with programming languages. For example, GemStone [MAI89] gets much of it's computational power from its close links with Smalltalk. ONTOS [AND90] is computationally powerful because of C++'s facilities [STR86]. POSTGRES [ROW90], gets it's power through its facility to implement user-defined procedures, extending the available mathematical facilities of the underlying INGRES model.

2.3.1.8 DBMS Components

Any OODB must implement the features associated with database management systems to deal with the management and maintenance of the data.

The DBMS must ensure that the life of the data extends beyond the life of the application if required. Persistence must be catered for. In OODBs, the data is virtually identical in structure to the data in the application. In contrast to relational systems, the process of deactivating the data from the application to the database may be less explicit, as there is no need for conversion of the data structures, to be written out. In other words, there is no **impedance mis-match** to deal with.

The database management system should cope with the management of the secondary storage, maintaining indices, hashing structures and tree structures where required. The location of the data in secondary storage should be transparent to the application. This should be an issue dealt with by the DBMS in an optimal manner.

Issues such as **Concurrency, Distribution** and **Recovery** in the database should be managed, to facilitate maximum safe usage of the database. Considering the application areas where OODBs might be used, these requirements would be essential.

2.3.1.9 Query Facility

Although there are many commercial OO databases available, the provision of a query facility, if any, is absent in many of them. As with an OO data model, no decision has been made on the form a query language to an OODB should take.

It is agreed that some form of query facility should be provided [KIM90][ATK90]. It is generally felt that the query facility should satisfy the following criteria:

(i) It should be high-level, ie. the user should be able to express simply, non-trivial queries concisely.

(ii) It should be efficient. It should lend itself to some form of query optimisation. Currently most of the commercially available OO databases do deal with query optimisation. EXODUS [CAR89], provides a query optimiser generator, allowing the user to develop a database and query language which executes optimised queries.

(iii) It should be application independent. This is akin to the development of SQL for the Relational Data Model. This feature would mean that the query should work with any type of database. It would not be necessary that this feature consist of a query language. A graphical browser would be sufficient to fulfil this functionality.

Different available databases provide different query languages. The majority of languages support a similar SQL-type query language. Queries are addressed on classes [ONT90], [BAT86], [CAR88]. This is akin to queries being addressed on relations in a RDB. However some of the OO databases allow queries to be addressed to groups of objects as well as classes [BAN88]. One interesting challenge is to define a query model for object-oriented databases which will admit operations equivalent to relational joins and set operators and which will honour all fundamental principles of the object-oriented systems [KIM89].

Obviously the lack of a query language with a set of operators akin to (SELECT/PROJECT/JOIN) of RDBs makes the development of a query optimiser extremely difficult. In addition, many systems which have implemented a query language so far, seem to have done so in an attempt to offer some form of query language rather than just offering a good, well thought-out language. Much work needs to be carried out in this area to arrive at a query language designed and optimised to suit OODBs.

2.3.2 Supplementary Features

In addition to these core features, a number of extra features have been described as recommended in any OODB system. The most important one is ***Versioning***. OODBs currently available, provide facilities for the maintenance different versions of the same data. Although not an essential part of an OO data model, the feature is decidedly useful. In a CAD application scenario, the ability to maintain back versions of designs facilitates retrospective modelling, being able to use old versions of a current design without the need to load up an old database - a practical necessity in many design situations.

Although the experts agree that the model should be established for Object Oriented Databases and that the above features should be included, these experts have tended to go off in a direction of their own and develop their own OODBs. Although different, these systems have tended towards a number of different directions.

Although, as has been said a number of times, no OO data model has been decided upon, some software companies have developed their own products in an attempt to introduce OODB facilities to commercial markets. Four of these are outlined in the following section. They all adopt different approaches to offering object orientation to application programmers, yet they all either implement the core OO features discussed above , or provide facilities by which they can be implemented.

2.4 Available Object Oriented Databases

The developers include people such as Maier and Stein who developed *Gemstone* [MAI86a], Ontologic Inc. who developed *ONTOS* [ONT90] and Banerjee, Kim et al. who developed *Orion* [BAN87a].

These databases have a number of things in common. Primarily, they represent commercial attempts to produce viable database management systems or database tools which implement the fundamental features of the OO data model. However, their primary goal, in their development, was to satisfy current application requirements for data storage.

The approaches taken to implementing new OO models have gone in three main directions. Systems like POSTGRES [ROW87c][STO86b] have aimed at adding object-oriented principles to the relational model. EXODUS is developed as an extensible database providing tools for the development of any type of database system.

Finally GemStone and ONTOS have been developed as totally new database systems being based around some of the new object-oriented programming languages. GemStone is developed as a new system around Smalltalk and ONTOS is developed as a persistence model around C++.

2.4.1 POSTGRES

2.4.1.1 Background

POSTGRES is a next generation extensible database management system developed at the University of California [ROW87c][STO86a][STO86b]. The data model is based on the idea of extending the relational model developed by Codd [COD70], with general mechanisms that can be used to simulate a variety of semantic data modelling constructs. The mechanisms include :

(1.) Abstract Data Types (ADT)

(2.) Data of type procedure.

(3.) Rules.

These extensions have been built onto an existing successful RDB, INGRES [STO76]. INGRES has a query language QUEL. As an obvious development of this POSTGRES includes POSTQUEL.

2.4.1.2 System Structure

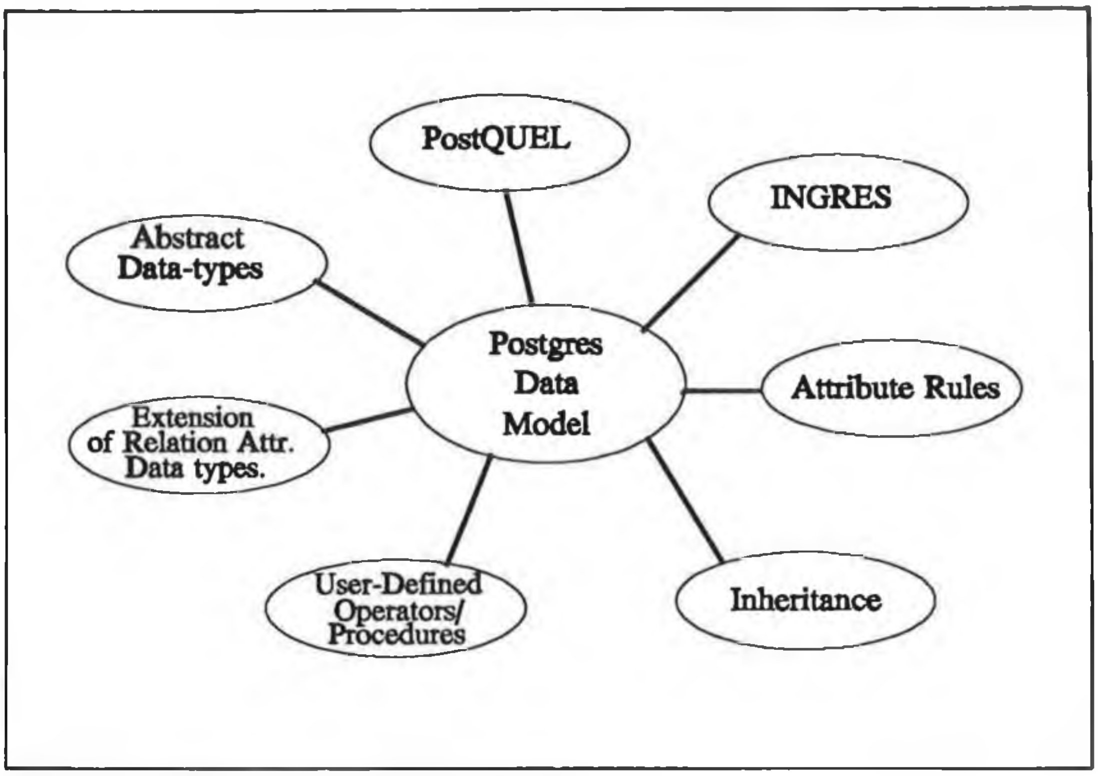


Figure 2.11 - The architecture of POSTGRES.

POSTGRES is an example of the application of object-oriented features to an existing data model. The data model is the relational model, but has been extended with abstract data types including user-defined operators and procedures, relation attributes of type procedure, and attribute and procedure inheritance.

The new data model is built around INGRES, the popular RDBMS and adds a number of extra features to it. In addition to the basic relational model, abstract data-types have been added. Data types can be specified by stating their storage requirements, operators which operate on them and their external interface to an application. Procedures can also be defined which are associated with attributes in a relation. This is akin to the association of methods with classes of data in object-oriented theory. To improve the semantic power of the relational model, inheritance has been added. Relations can inherit attributes and procedures defined on a relation from parent relations. QUEL, the original query language with INGRES has been enhanced to cope with the added features to produce POSTQUEL. To control the data stored in the relations and to add flexibility, rules can be applied to attributes and attribute contents can be determined by user-defined procedures.

2.4.1.3 System Features

A database is composed of *relations* that contain tuples which represent real-world entities (e.g., documents and people) or relationships (e.g., authorship). A relation has attributes of fixed types that represent properties of the entities and relationships (e.g.,the title of a document, or a person's name), and a primary key. The type of an attribute in an RDB tends to be atomic (e.g., Integer, Real, String etc.). POSTGRES adds inheritance to this model. A relation inherits all the attributes from it's parent(s) relation unless an attribute is overridden in the definition. The level of inheritance implemented is multiple inheritance, as described above, and also applies to procedures or operators associated with relations, as outlined below.

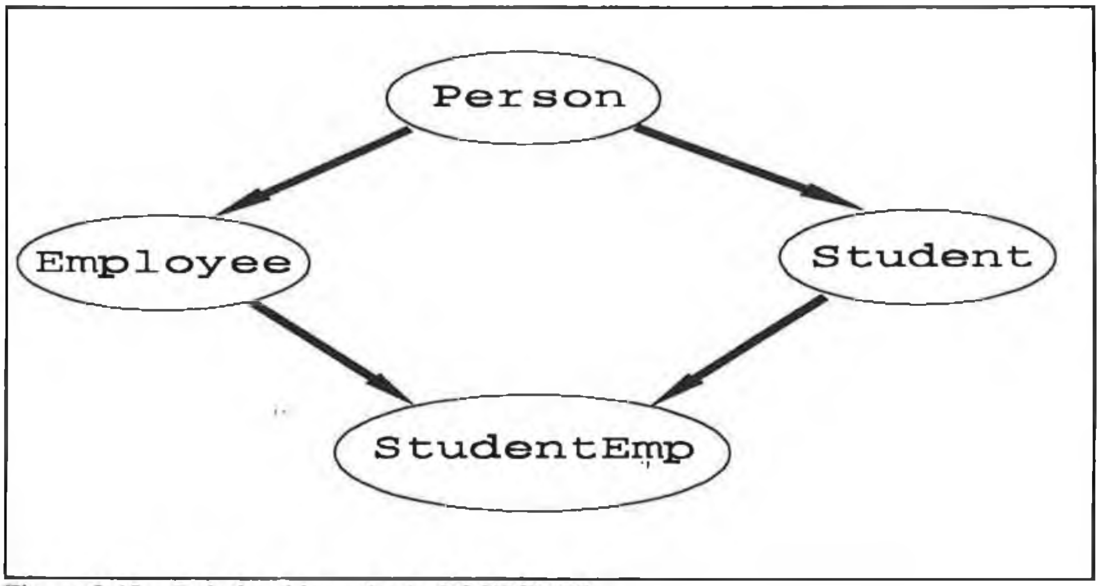


Figure 2.12 - Relation hierarchy in POSTGRES.

The POSTGRES query language is a generalised version of QUEL, the query language associated with INGRES, the original relational database, called POSTQUEL. QUEL was extended in several directions, to cope with the extensions to the data model [STO84].

2.4.1.3.1 Versioning

POSTGRES supports the idea of versioning in two different manners. The first manner is not explicit versioning, but results in a version of a relation at a particular time. POSTGRES saves a copy of data deleted from or modified in a relation so that queries can be executed on historical data. For example, in a STUDENT relation, a list of all of the students could be retrieved from the version of the relation on March 1st 1991 using the following statement:

```
retrieve (S.Name)
from S in STUDENT ["March 1,1991"]
where S.City = "Dublin"
```

The version of the STUDENT relation as it was on March 1st is used in this query. Although this is a type of versioning, POSTGRES actually allows the user to create an explicit version of a relation or part of the relational hierarchy at any moment. Versions can be made from views of a relation also. In pure OO manner, a version can be taken of a particular base relation, which will include the state of all of it's sub-relations, inherited from the specified one.

2.4.1.3.2 Data Types

One of the important aims in the development of object oriented databases was to make up for the problems associated with the representation of complex structured data in relational databases. In response to this inadequacy of RDBs, POSTGRES provides an abstract data type (ADT) definition facility. An ADT is defined by specifying the type-name, the length of the internal representation in bytes, procedures for converting from an external to internal representation for a value and from an internal to external representation, and a default value. In addition to this aggregations can be attributes, e.g., arrays, unions, sets.

2.4.1.3.3 User-defined Procedures

Procedures can be added to the database. These are written in a conventional programming language. They are used to implement ADT operators or to move a computation from a front-end application process to the back-end DBMS process. The procedure is defined in the database, stating the name of the C function program file, it's return value and the name of it in POSTGRES. These details are stored in the system catalogue and the system dynamically loads the object code when it is called in a query. The following query uses a procedure called *AgeInYears* which originally coded in C is passed a date and returns the number of years since that date. The query is used to list all the people in the database and their ages in years :

```
retrieve (S.Name,
          Age = AgeInYears(S.Birth))
from S in STUDENT
```

Procedures can also be written to take complete tuples as parameters. So for example a procedure (e.g., CalcBonus) may be passed a tuple of the EMPLOYEE relation and based on a number of it's attributes (e.g., Salary, Job Title, Status) calculate some value (e.g.,Bonus). This idea of the tuple as a parameter, is akin to the definition of a function or method as applying to a particular class in pure OO systems. As attributes of a particular relation can be inherited down through the relation hierarchy, so can procedures. So if the procedure CalcBonus was defined to be passed an EMPLOYEE tuple, then it could also be passed a STUDENTEMP tuple as this relation inherits all attributes and procedures from EMPLOYEE, as indicated in the above diagram.

2.4.1.3.4 Rules and Attributes

RDBs are restricted to a large extent by their ability to store only atomic values as attributes. POSTGRES, in contrast to this, as mentioned above, provides for the storage of new datatypes in a relation. In addition to this, rules can be applied to attributes, determining their contents, or restricting or validating their values. In addition to this, user-defined procedures can be associated with individual attributes. So for example, an attribute may not actually contain an explicit value but a method which can be used in the determination of the value. In a database with employee details, the following details might be included :

Name
Marital Status
Years-Service
Age
Salary
Bonus

Bonus is a field which is determined by factors like salary, marital status, age and years service. The *Bonus* attribute has a method associated with it which calculates the bonus. This is a slightly more restrictive version of methods in object-oriented languages like C++ or Smalltalk.

2.4.1.4 Conclusion

Although it could be argued that POSTGRES is not an object-oriented database in the true sense of the word, it does contain many of the key OO features as outlined above. It does attempt to extend the relational model to add more functionality and meaning to it. However, there are limitations inherent in the relational model and adding OO features to it does not necessarily solve these problems. The relational model is good for data requirements where the ratio is large between the data and the attributes in the database, where there is a large quantity of similarly structured data. Object-oriented databases could never expect to encroach too far into that area. Their domain is set in the area of high-powered applications sporting complex data with a small data to attributes ratio. This area may only account for 5% of the overall software market. In this light, the development of POSTGRES may really only be an attempt to improve the way relational databases cope in their current applications areas rather than offering a solution to the complex CAD, CAM, etc. applications with substantially different data storage requirements.

2.4.2 EXODUS Extensible DBMS

2.4.2.1 Background

Until recently, research and development efforts in the database system area have focussed primarily on supporting traditional business applications. New applications such as AI, CAD, CAM, Image & voice systems and statistical and scientific applications require more complex storage facilities. Such systems as POSTGRES [STO86a], outlined above, provide enhanced data storage and manipulation facilities. These extensible databases let the database develop to match the extension of the application.

EXODUS [CAR86a][CAR86b][CAR88] is also an extensible database, but unlike POSTGRES or PROBE [DAY86], it is a modular and modifiable system rather than being complete, end-user DBMSs for handling all new applications. EXODUS provides a collection of kernel DBMS facilities together with software tools to enable the semi-automatic construction of an application-specific DBMS for a given new application area. This provision of low level database construction tools facilitates the creation of relational, OO, or any other model based databases which will suit the required application area.

2.4.2.2 System Structure

EXODUS, unlike most other extensible database systems, is designed to be a *toolkit* type system, that can be easily adapted to satisfy the needs of new application areas. For true extensibility, EXODUS provides as many generic components as possible. At the lowest level is the Storage Manager. This is primarily concerned with the storage management of objects of any size. The type of data actually being stored is of no importance to the Storage Manager, as it deals only in bytes.

The system consists of 5 main components, provided to the database engineer (DBE) to create a system :

(1.) The Storage Manager.

(2.) The E programming language and its compiler.

(3.) A library of type-independent Access and Operator Methods.

(4.) A rule-based Query Optimiser Generator.

(5.) Tools for constructing query language front ends.

At the bottom level of the system is Storage Manager. The basic abstraction at this level is the storage object, which is an untyped, uninterpreted, variable-length byte sequence of arbitrary size. Included in this module are buffer management, concurrency control, and recovery mechanisms for operations on shared storage objects.

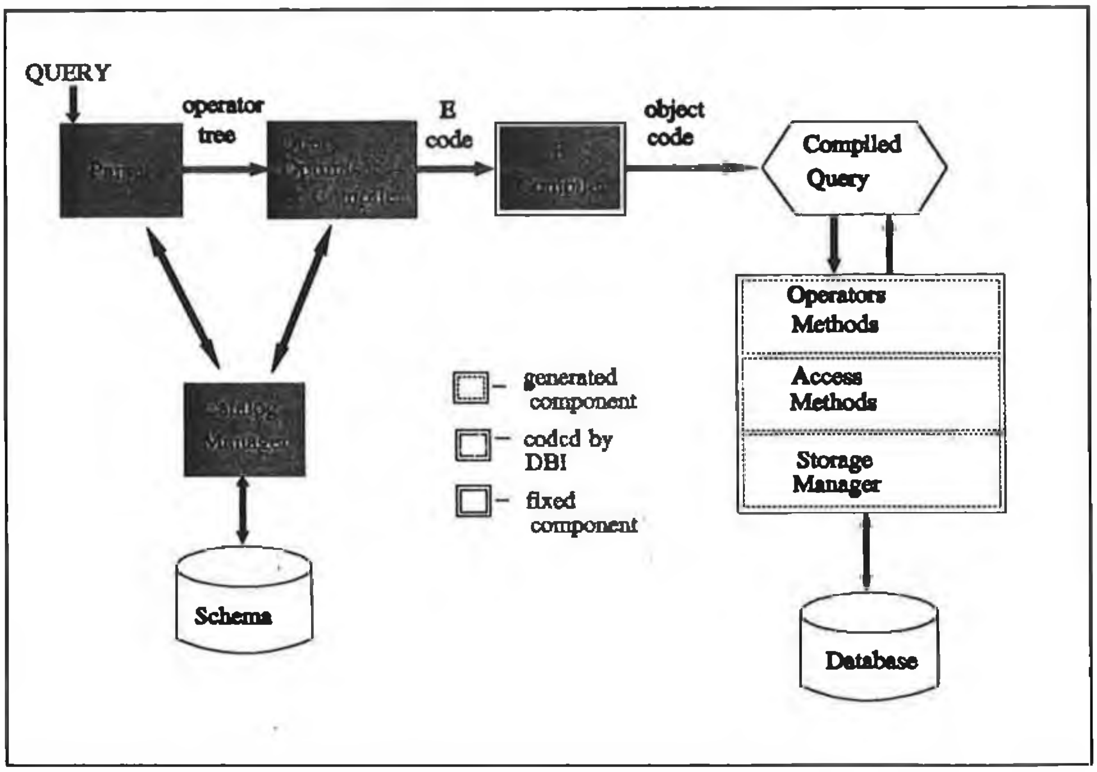


Figure 2.13 - General EXODUS database system structure.

E is the implementation language for all components of the system for which the DBE must provide. E extends C++ by adding generic classes, iterators, and support for persistent object types to the C++ type facilities and control constructs. E makes reference to persistent objects transparent. The compiler inserts the code to deal with referencing stored objects as opposed to memory resident ones. The objective of E is to simplify the development of *internal* systems software for a DBMS. Layered above the Storage Manager is a collection of access methods that provide associative access to files of storage objects and further support for versioning. Such structures as B-trees, linear hashing facilities are provided, in E code. These can be replaced by DBE written E access methods. These intermediate access methods shield the DBE from having to map main memory data structures onto storage objects and from having to deal directly with other low-level details of secondary storage. The third level are the operator methods. This layer contains a collection of methods that can be combined with one another in order to operate on *typed* storage objects. In general the DBE will have to implement one or more methods for each operator in the query language associated with the target application.

2.4.2.4 Conclusion

EXODUS is not strictly an OODB system, however, the facilities which it provides do allow it to support the development of application specific database management systems. The development of OODBs using EXODUS would seem sensible through the provision of the E programming language which is an extended version of C++, because of the strong OO flavour in the language.

2.4.3.2 System Structure

GemStone consists of two individual parts. *Gem* and *Stone* correspond roughly to the object memory and virtual machine of the standard Smalltalk implementation. Stone provides secondary storage management, concurrency control, authorisation, transactions, recovery, and support for associative access. Stone is built on the underlying VMS file system. It provides only the operators for structural access and update of the database.

Gem sits on top of Stone and elaborates Stone's storage model into the full GemStone model. Gem also adds capabilities of compiling OPAL methods (Query/Procedure language), executing code, user authentication and session control. The Procedural Interface Module (PIM) is a set of routines to facilitate communication from different programs written in different languages to GemStone.

The structure of the system has Gem and Stone running on a VAX under VMS. While a GemStone system has a single Stone process, it maintains a separate Gem process for each active user, and the PIM handles communication on a per-application basis.

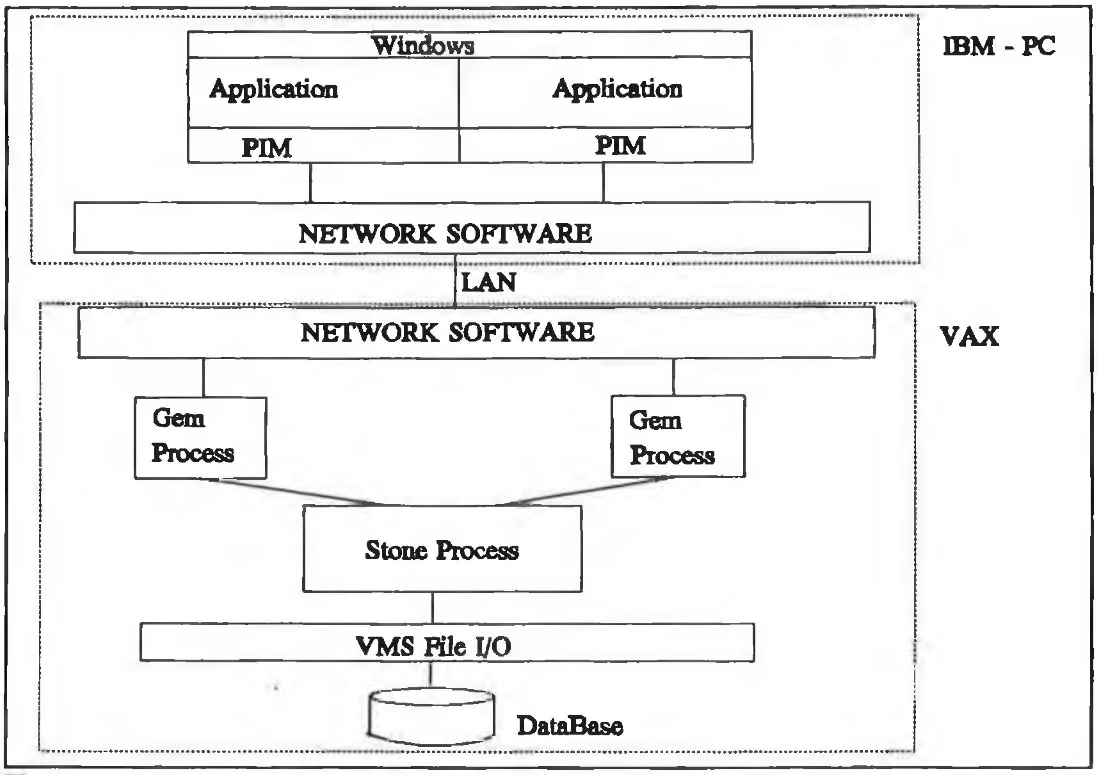


Figure 2.14 - The architecture of GemStone.

2.4.3.3 System Features

In simple terms, Gemstone implements a persistent version of Smalltalk, and includes the following features :

- It implements the ideas of Classes, Objects and Methods.

- In contrast to Smalltalk, Gemstone is a Multi-User, Disk-Based Environment. As with any good database, intelligent staging of data between disk and memory is employed. This procedure aims to anticipate which objects in main memory are likely to be used again soon, and organize its query processing to minimise disk traffic. Obviously to support the Multi-User feature, concurrency needs to be implemented. For a schema of classes accessible to a number of users, if a single user modifies some of the classes, then a copy of the altered classes is made into the user's storage segment. This is essentially a shadow copy of the original classes and as in the versioning mechanisms adopted by other databases, is linked to the original schema. This idea is outlined in the figure 2.15.

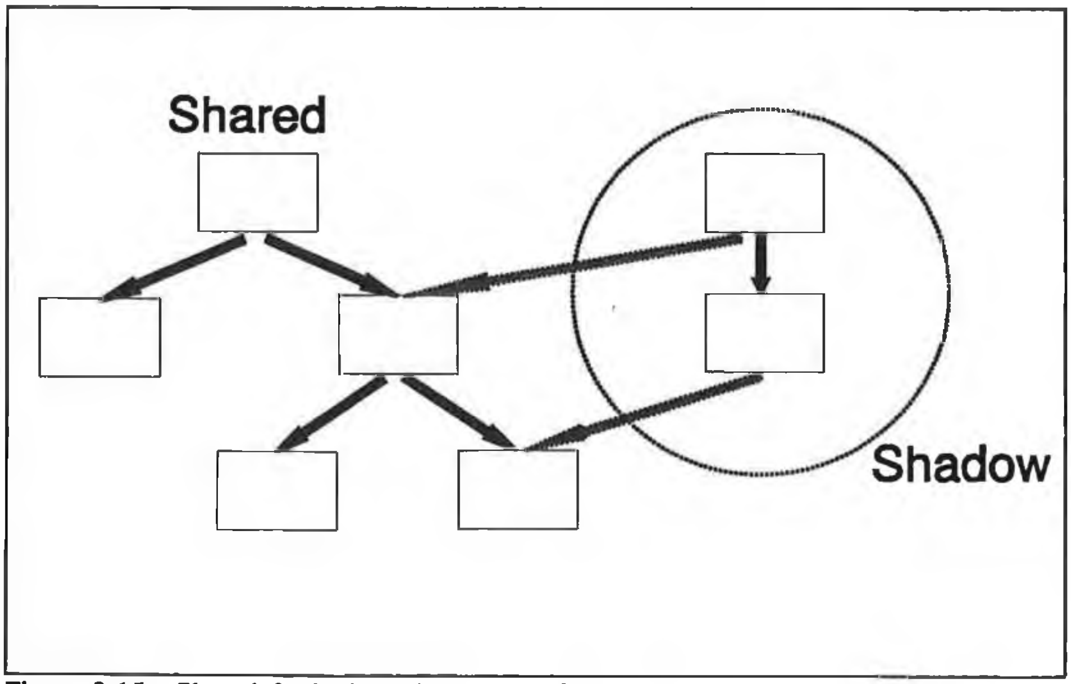


Figure 2.15 - Shared & shadow classes or objects.

- Stone deals with recovery and transaction management.

- Includes a programming language called OPAL, which is an extension of Smalltalk-80, to facilitate the creation of Class Hierarchies.

The standard ideas of data authorisation and granting and revoking privilege on data are included. Every user has his/her own segment and as such has authorisation on it. He/She can grant access to this data to any other user.

One feature which seems to be prevalent in OODBs is large object space. Gemstone supports up to 2^{31} objects and each object can have up to 2^{31} instance variables.

2.4.3.3.1 Query Language

Gemstone provides a limited calculus sub-language. However the language has been constructed in such a manner that associative queries can be viewed as procedural OPAL code. In an object-oriented model, there is no need for many of the joins used in relational systems, as these joins often serve to recompose entities which were decomposed for data normalisation. Entities are not decomposed in the first place in an object-oriented model; most joins are replaced by path-tracing, which Gemstone supports.

2.4.3.4 Conclusion

GemStone, like many of the new OO databases is based purely on a successful and tried OO programming language. With the current OO programming languages like Smalltalk, C++, Object-C and LISP (OO to a certain extent) gaining in popularity, the requirement for database systems to back them up, is essential. The features provided by GemStone would seem to fit nicely on top of Smalltalk, extending it's capabilities, and in the process resulting in a powerful database management system which is capable of dealing with very complex application domains.

2.4.4 ONTOS - Object Database

2.4.4.1 Background

ONTOS is an object database which adds persistence to the C++ programming language. It is the successor to VBASE, one of the first commercial object databases to appear [AND87]. It is primarily a *Client-Server* database system designed to be distributed over a network, initially a homogeneous network but in later versions, a heterogeneous network of architecturally diverse nodes and servers. In addition to C++ as a programming language, an SQL type query language is provided to attain some of the benefits associated with RDB query facilities. ONTOS initially developed for UNIX workstations now is available for platforms ranging from VAX minicomputers to Workstations to PCs.

2.4.4.2 System Structure

Ontos is a distributed *Object Database* and uses the client-server style of data interaction [CAR86]. The server side manages the data store: the client side provides the interface to user processes and manages mapping of data to the application process's virtual memory space. The entire database, whether it is all contained on a single node or distributed over several nodes, has a single *User ID* (UID) space and thus operates as a single, very large, random access memory. In the database itself, each object may have a name associated with it. These names are mapped to the UIDs by the hierarchy of name directories. The name object's full name is a pathname, tracing a path from the root directory to a leaf directory containing the object name. Yet these pathnames are logical and never refer to physical devices.

The task of the server process is to manage the underlying storage of it's portion of the database and responds to client requests over the network. A primary server also maps each object to it's respective server. The server also responds to database open and close requests, and controls the multi-server commission of transactions.

The client is implemented as a function and class library that is linked into the application process. It manages communications between the application and one or more database servers on the network. Obviously, the client needs to translate Object UIDs to virtual memory space addresses for processing and back again when the processing is complete.

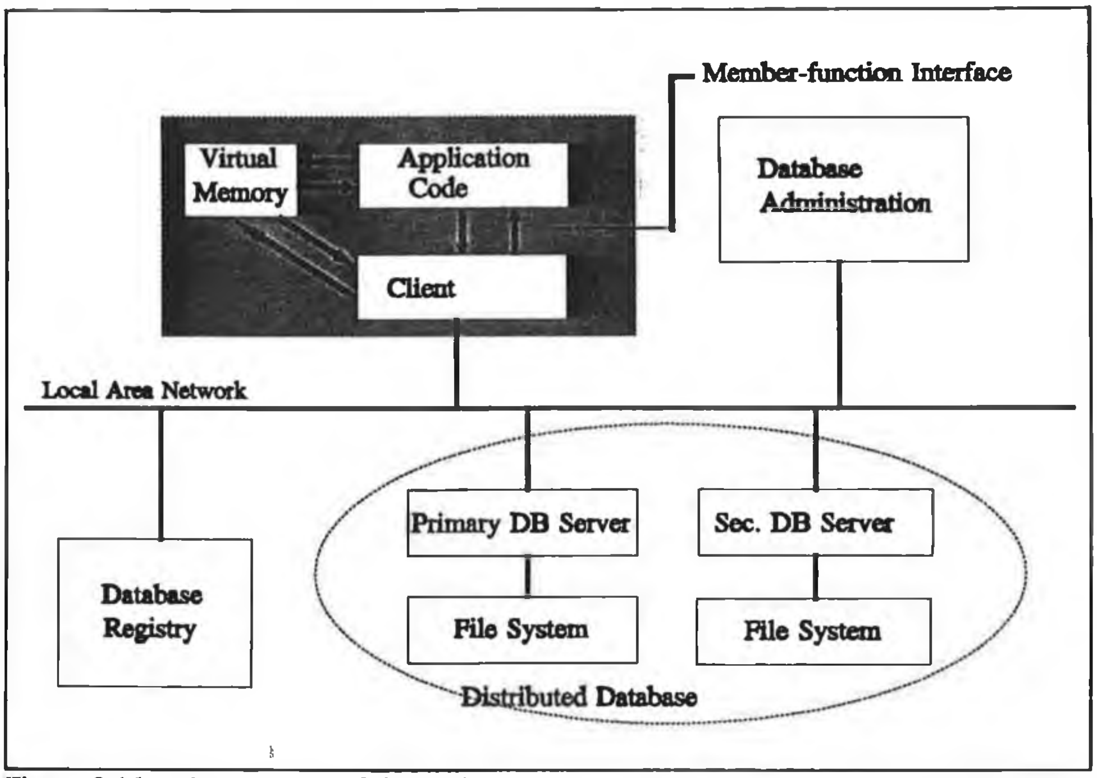


Figure 2.16 - The structure of ONTOS.

In addition to this client/server model, a Registry is included on the servers to give the information on the logical databases and their mappings to physical files. A set of database administrator utilities is provided to facilitate the maintenance of the databases, with reference to location of files, names of databases, privilege etc.

2.4.4.3 System Features

ONTOS is built around the C++ language, and as such includes all the features of that language. However, the database has a number of extensions.

2.4.4.3.1 The Database

ONTOS can be thought of as consisting of collections of properties and functions. As mentioned above, each object is identified by it's own identifier or UID, guaranteed to be unique and consistent within an ONTOS database. The properties or attributes associated with an object are embedded in the object or are represented by UID references to other objects. When an application requires an object to be taken into memory for processing, it is *Activated.* When it is no longer required, it is *DeActivated.* Activation involves transferring object state from the database to memory. All references contained by the activated object to other read objects are translated from their UID form to high performance virtual memory-based references. Deactivation is the reverse process - the translation of memory references back to UIDs and the writing of objects back to the database. Activation can affect the performance of the application substantially. This process is put at the programmer's control. ONTOS facilitates the activation of single objects, a *logical cluster* of associated objects or even a *closure*, consisting of all objects reachable directly or transitively from a given object (see diagram below). Deactivation can adopt the same approaches.

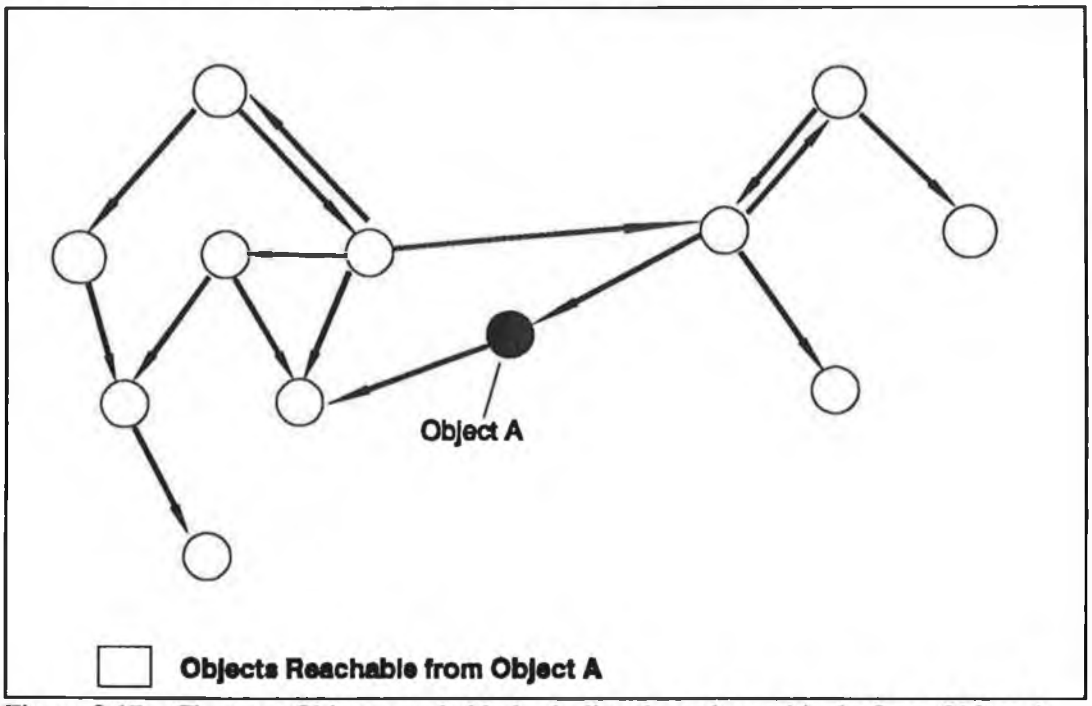


Figure 2.17 - Closure : Objects reachable both directly and transitively from Object A.

ONTOS also supports transparent references (TRefs). If objects are used in this manner, they are automatically activated when required if they are not already in memory.

2.4.4.3.2 Transaction Model

The activation and deactivation of objects discussed so far implicitly assumed a single user. In a database system with many concurrent users, conflicts between users must be detected and resolved. ONTOS handles such conflicts through a transaction mechanism. As with most database systems, the transaction mechanism insures atomicity of change. Either all changes comprising a transaction are made or none are made. This prevents the database from becoming inconsistent. When objects are activated, it is possible to lock out other processes from access. This idea of read/write privilege association with objects is akin to table and tuple privilege in relational databases. It is also possible for the process to permit a conflict situation to occur. If it does occur, the application can specify the action to take, as part of the *transactionStart* statement.

Transactions are implemented to the extent that cooperating processes can share a single transaction. This facility allows dealing with tasks which are atomic from the standpoint of changes to the database but are implemented more easily by a number of cooperating processes. It is particularly convenient for multiple window applications in which a separate process operates each window.

2.4.4.3.3 Data Manipulation - Creation,Deletion,Modification & Accession

ONTOS, as with any database, provides facilities for the creation of data in the database. In object oriented database programming, this tends to be a two-fold process. Initially the database schema needs to be defined. In a relational database this involves the definition of the database relation. In ONTOS this entails the creation of the classes of data - the definition of the class attributes, the procedures (methods) associated with the class. It is also necessary to specify the superclass of the new one, if any. In addition to this, classes may be composed of other classes. This class-composition relationship is something that gives object based databases their power, but is omitted from relational databases.

ONTOS is based around the OO programming language C++ [STR86], developed as an object oriented extension to C. A simple C++ program to create an employee class is given in Appendix A. When this program is executed, the Employee object is created, but when the program terminates all the data associated with the program is lost. ONTOS provides the facility to make this program data persistent beyond the execution of the program. With this in mind, the program is modified to facilitate the addition of the class to a database and the insertion of the new Employee object into the it. Appendix B gives the Employee program again, but it has been modified to facilitate the making persistent of the data. When this program is executed, the new class is created in the same manner as in the non-database program. When the program is compiled, an extra pre-processor is applied and the class definition and it's associated method C++ code is added to the database. New objects are created in the same manner as before, but to add it to the database requires only the method *putObject()* to be applied to the new instance. As can be seen, little conversion is required to go from a substantial non-OODB C++ system to a converted OODB one. Ontologic, the ONTOS manufacturer, report a customer who converted a 100,000 line system to be fully operable with ONTOS in only 3 days of modification [AND90]. This is believable when one compares the two programs, the increase in functionality is definitely not reflected in the increase in program complexity. This form of ONTOS class creation is static. In other words, the class is compiled into the database at development time. For many applications where the database acts as a backend support system for a well defined application where the DB schema will not change substantially, this form of schema definition is quite sufficient. However, if the application is to be dynamic and requires the schema to be created or manipulated at run-time, some other approach needs to be taken.

ONTOS provides a number of methods whereby a class can be created at run-time by the user. Methods can be associated with this new class, but these are required to have been pre-compiled into the system. If this is the case, they will tend to be general functions and not really directly associated with the class which is created. Currently no facility is provided for editing and creation of the class methods at run-time, but this is due to be included in a later version of ONTOS. Even so, the creation of classes dynamically at run-time and the facilities provided for the access to the class details proves sufficient for many current applications. Appendix C contains a sample program which creates the Employee class. Unlike the program in Appendix B, this is created dynamically and as such makes the program very flexible. However, with this increased flexibility in functionality, there is a corresponding decrease in programming flexibility as the coding is no longer just an application of database functions to existing C++ classes.

The dynamic creation of the objects in the database is carried out in a similar manner. When the class was created in the database, a constructor function to be used in the creation of objects was created in the database automatically. This function is then accessed when the new object is being created.

ONTOS also introduces the concept of *Iterators* for moving through the data in the database. For accessing the Objects associated with a class, the properties or methods making up a class definition, the values of instances or any item with multiple parts, iterators are usable.

2.4.4.3.4 Exception Handling

Object programming allows code to be more robust, more interchangeable and more portable than it can be using non-object techniques [AND90]. Based on this it is therefore reasonable to expect that object programs will need a better defined, more regular and robust exception handling mechanism as well. The common system employed by many systems of passing back error codes as special values within the domain of a function's return value has well known drawbacks. Perhaps the largest drawback of the traditional system arises because the code detecting the error cannot, in the general case, recover from it. Frequently, the point in the program at which there is enough context for error recovery is in a high-level function while actual error detection occurs in a low-level function. The two may be separated by arbitrarily many levels of function calls. An elegant solution to this problem [MIL88] that can be implemented without any language extensions uses a two part protocol. The essence of the protocol is this:

> The high-level function specifies an exception scope. It specifies an exception handler and exception paths for normal and abort processing. If any function that runs within this scope detects an error, it dispatches directly to the error handler. The error handler either corrects the problem and returns to the point at which the exception was raised or determines that the error cannot be corrected and aborts processing. An abort involves clearing up data objects created in the exception handler scope and taking the abort branch. ONTOS implements this approach to exception handling. Error conditions are specified by classes. The root class, called Failure, defines the necessary error raising functions. More specific errors are represented by classes derived from Failure. Thus, in keeping with the OO flavour, the exception handling facility allows the definition of a systematic hierarchy of exception conditions that the system will handle and the linking of these exception conditions with appropriate exception handling functions.

2.4.4.3.5 Versioning Mechanism

The ONTOS versioning and alternatives mechanism allows a single object to exist in any number of versions. All objects of the same version are collected into a *Configuration* object. Configurations are related to each other through *derivations links*. Each configuration (except the first) has a parent and may have any number of children. Thus both alternatives and serial versions are supported. Objects in leaf configurations may be changed freely.

2.4.4.3.6 Query Facility

In the current version of ONTOS, a provision is made for querying the database through OSQL. OSQL adds a predicate-based iteration style of interaction to the database to the

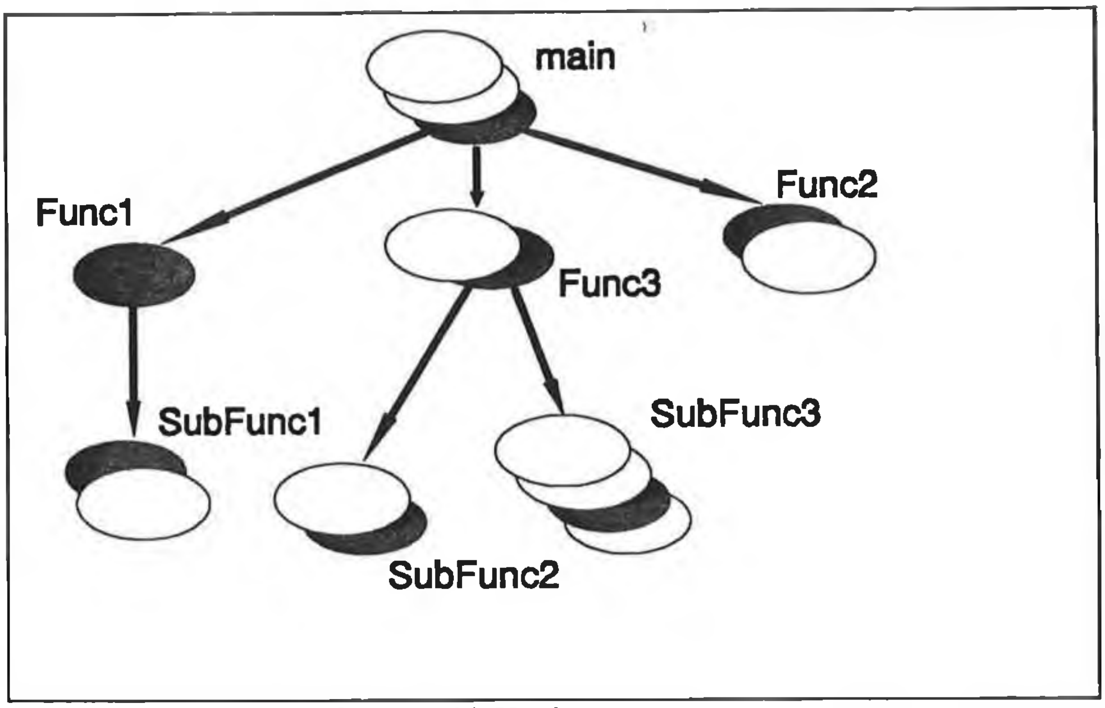


Figure 2.18 - A Software System with Versions.

navigational style characteristic of object systems generally and C++ specifically. OSQL is intended both for access from applications programs when the SQL style of query is more convenient and as a backend for an interactive end-user query facility.

In keeping with the OO flavour, the query facility is designed around a persistent *QueryIterator* class, allowing queries to be stored as objects in the database. Each instance of the iterator represents a particular query. (The argument to the *QueryIterator* is an OSQL query). Similarly having created an instance of the *QueryIterator*, the result of it can be obtained by applying the *yieldRow* member function. Each time it is called, it returns another row of the query result, until all rows of the result have been returned. An example of an ONTOS OSQL query is given in the Appendix D.

ONTOS currently consists of a purely programmatic interface from C++, with OSQL calls being made through C++ programs. In version 2, currently being released, support is being included for Pascal programs. In addition to this, a number of utility programs are included to allow the interactive creation of the database in a graphical manner. A class browser is also provided. This is akin to the Smalltalk development environment. ONTOS would seem to be developing into a complete database management system with a full development environment for application creation.

2.4.4.3.7 Inverse Relationships

One feature of ONTOS, which is also contained in GemStone, is the inverse relationship facility. For example, one class may contain the following attributes :

Name, Salary, Manager, Sub-Ordinates.

Attributes like Manager and Sub-Ordinates will refer to other objects. The manager of Tom will himself be an employee. Similarly, the sub-ordinates of Tom will also be employees. Tom's manager's sub-ordinates will include Tom. This fact in most database systems would need to be explicitly dealt with each time one attribute changed.

In ONTOS, when the employee class is created the programmer would explicitly state that there is an *inverse-relationship* between the manager attribute and the sub-ordinate one. So subsequently if Tom's manager is set to be Mike, Mike's sub-ordinates will change to include Tom.

Obviously such an inverse relationship facility helps to improve the semantic capturing ability of the database.

2.4.4.4 Conclusion

ONTOS, in its current release, provides most of the features outlined as core-features above. However, as it is designed to be used as an extension to C++ programs, closely linked to it, it makes the development of persistent applications which use the database's facilities easier to develop for the application programmer. However, because of its programmatic interface only, it does not readily provide an interface to the user. Later releases of ONTOS are expected to provide some form of class browser, but this will need to include a very high level of functionality to put it on par with those user-interfaces provided with many of the RDBMSs, which provide a number of different interaction techniques to the user.

2.5 Conclusions

Based on the way in which the new OODBs are developing, it would appear that in the five to ten years time, they will have evolved to the stage that RDBMSs currently are - providing a complete data management and application development solution.

The principles behind the current object-oriented databases stream from the necessity for new data models to capture more meaning from the environments, and the necessity to facilitate the storage and modelling of complex data.

OO databases have evolved from the semantic databases which appeared in the early 1980's. With the addition of method support and polymorphism, the increased functionality has found a niche for them. Currently relational database technology accounts for 90% of the databases in operation. The application areas to which OO databases would be geared would probably account for less than 10% of the market. However, this 10% is made up of very technically advanced applications, placing great demands on the data storage facility.

This chapter has endeavoured to present an outline of the facilities associated with OO databases, and in addition examine some commercially available OODB systems. The systems which have been outlined represent four different approaches to the implementation of OO principles. They each deal with object orientation in different manners, yet they all aim to implement the core OO features in some form.

The features which these database systems exhibit are not based on any formal standard OO data model, as none has yet been established, yet it would appear that any resulting model will have its constituent parts seriously influenced by these initial commercial developments.

OO databases, and OO principles in general, are very firmly based around the structuring of the data into hierarchies, with data inter-relationships being made explicit. This structuring presents a number of opportunities to application and interface developers regarding the manner in which the underlying data can be represented. With OO databases, a pictorial representation of the database is not only possible, but in many circumstances may be preferable.

Chapter 3 will look at current user interface techniques. It will examine how they have evolved over the years, and the advantages offered by each. Certain interaction techniques are more suited to graphical representation of structures than others, and these will be examined. Some commercial and research systems have been developed which have attempted to represent databases in a graphical context, allowing the user to interact with the database schema and data through graphics. These will be outlined.

Chapter 3 -
Human Computer Interaction & User Interfaces

3.1 Introduction

In the early days of computers, they were being used purely by computer programmers. They programmed these monsters by means of electrical switches, paper tape or punched cards. Programming using punched cards involved a pile of punched cards on the input side and a similar pile of punched cards on the output side. These were the only interface between the computer and the programmer. With the introduction of the Cathode Ray Tube (CRT), computer screens offered a simple and comprehensible method by which the computer could report its operations to the programmer or operator. The introduction of the keyboard gave a similarly simple means by which the programmer/operator could give his/her instructions to the computer. Because of the relative scarcity of computers with their restriction to the research departments of large academic institutions or data processing departments of large corporations, the necessity for them to be easy to use was of minimal importance.

However, in the past 20 years, with the extent of computer usage moving from science laboratories to most desks, factory floors, and even classrooms, much work has been put into finding a simple, yet effective manner by which both computer professional and computer inexperienced personnel can utilise the computer's power to the full, in as simple and stress-free an environment as possible.

As a result of the computer revolution, a whole new area of study and research appeared.

Human Computer Interaction (H.C.I.) deals with the way in which Users communicate with computers [HEL88]. It is concerned with the manner in which commands and instructions to the computer are represented by the user and how the computer relays its information to the user. However, in addition to this, H.C.I. encompasses **Cognitive Engineering.** *Cognitive Engineering is about human behaviour in complex worlds* [WOO88]. This study of human behaviour provides a lot of tips to computer user-interface designers regarding the mental 'train of thought' of users in certain situations, enabling them to fine tune interfaces to match the user's thought processes.

Excluding the knowledge gained about H.C.I. and cognitive engineering, most of the developments in user-interfaces have been driven by two factors:

Software Developments

Hardware Developments

3.1.1 Software Developments

As obvious as it may seem, most of the advances in user-interface development have been driven by the changes in software. Initially when computers were new and scarce, applications tended to be mathematically based, in the case of science research laboratories or administrative based applications, like personnel or payroll systems in the case of large corporations. The emphasis was on what they did - their output, rather than on how they looked, how easy they were to use or how the user liked using them.

With the introduction of high powered workstations and personal computers, sporting high resolution graphics adapters and screens, and the abundance of networking hardware and software, computers have been made available to professionals previously working without them, firstly due to the lack of suitable equipment and secondly due to the exorbitant cost of what was available. Professionals such as engineers and architects have been aided with computer-aided design (CAD) and computer-aided engineering (CAE) systems. Accountants and financial controllers are helped by advanced financial management and planning systems. With the help of networking and communications facilities currently available, they have been offered links to the international business world, in a way never before experienced.

Even in application areas not obviously requiring computers, they have been introduced to cope with mundane record keeping and data storage. In many environments, they have been adopted as advisory tools for professionals there, or as partial replacements for professionals unavailable, in the form of expert systems. In many areas where experts are unavailable or where the working environment is inaccessible or harsh, such as nuclear power plant maintenance, expert systems have been linked to peripheral devices such as sensors and robots to carry out maintenance tasks in an intelligent manner.

Although all of this new computer installation in new application environments has benefitted the personnel concerned, it has posed many problems for the software developers creating them. No longer can applications be *paper tape or punched card based,* but they need to adopt a much more advanced approach to human computer interaction. For a CAD package, the software must display a realistic image of what is being designed, yet such an image must be easily manipulated, requiring little new learning from the professional concerned, to be able to use the system. Similarly for *Expert Systems,* the user should be able to relate to the system and use it as a tool.

Computers should be thought of as tools and devices which make our work easier and less tedious. Nevertheless, because of the way software has tended to be designed and developed, users unfamiliar with computing, have tended to regard computer use as a chore, with them becoming confused, annoyed and regularly worried that they'll do something wrong. Some of these problems are due to unfamiliarity on the user's part but, by-and-large, the manner in which the software represents itself is the main contributor. Much work needs to be put into developing software with the user in mind as opposed to developing it with the task to be carried out in mind. In an attempt to enact this aim, software is moving in a number of different directions with user-acceptance of the software being regarded as one of the main criteria for a successful package. These approaches and differing philosophies on user interfaces will be dealt with below, outlining the application areas suited to the different types.

3.1.2 Hardware Developments

Although the way the application presents itself to the user is determined by the way the application was written, the hardware available to be exploited by the software ultimately determines the form and content of the user interface.

For example, if a system is being developed to run on a computer with nothing more than a keyboard and text screen, the user interface will be purely text based with little or no graphical content. However if the computer has a large high resolution graphics screen, graphics pad, pointing devices (mouse), plotters etc., then the basic tools are there for an altogether different and advanced interface with much non-keyboard entry being provided for and impressive graphical output supplied.

Although the peripheral devices ultimately determine the form or the interface, the aptitude of a particular application to a certain interface technique and the way in which the designer exploits the available hardware will finally determine the interface.

Over the last number of years, with the improvement of cost/benefit of computers, they have been implemented in new environments. These new environments have warranted the development of new peripheral hardware. CAD and general design applications have resulted in the development of graphics pads and graphical input equipment like scanners to support them. Similarly with the introduction of vision equipment, high-resolution screens, mice etc., the way in which applications represent themselves has changed to cater for these.

The suitability of different hardware to different applications has resulted in different types of user interfaces being suitable to them.

In the remainder of this chapter, I will endeavour to examine some of the theories behind different user interface approaches, outlining what they are and their suitability to different application areas.

Finally, I will look briefly at some available different user-interface techniques to databases. These have applied some of the techniques discussed to produce radically different ways of interacting with databases.

3.2 Current Types of User Interfaces

With the abundance of computers and the diversity of their fields of application, there are nearly as many types of interface types as there are application environments. The following headings cover most of the current interfaces in vogue :

Structured Command Languages

Natural Languages

Menu Based Interfaces

Windowed Interfaces

Direct Manipulation Interfaces

3.2.1 Structured Command Languages

The earliest interactive human-computer dialogues relied overwhelmingly on commands or abbreviations entered by the user [BAR88]. So it is not surprising that much early work in the field of H.C.I. focused on the creation and use of command names.

With respect to current computers, command languages still play a very important part in the way we use computers.

Command languages date back to the first computers. Programming languages are essentially command languages. In early computers analog switches and punched cards represented commands to be executed by the computer. When the keyboard and screen appeared, it was only natural that there would be a direct transfer of the commands to the new computers, with the addition of some new ones. Since then command languages have thrived. Operating systems such as MS-DOS, VMS or CP/M have appeared offering the user an interface to the base computer operations for file, directory and peripheral manipulation through pseudo-English commands -

PRINT, TYPE, COPY, DIR, RENAME etc.

In recent years UNIX appeared, offering the same facilities to the user, again in a command language format but with less evident, more cryptic commands :

lpr, cat, cp, ls, mv etc.

In recent years, in an attempt to offer the facilities of the operating systems to a wider user-base, windowing systems such as Windows from Microsoft, Presentation Manager from IBM/Microsoft and X-Windows in all its guises from M.I.T. have appeared. These will be looked at below, but they essentially act as a buffer zone between the user and the command language.

Command Languages have been predominantly prevalent in operating systems, but in applications too they are widespread. Probably the best known example of command languages being applied to applications, is query languages (QL) in database management systems.

3.2.1.1 Query Languages (QL)

A query language is a special-purpose language for constructing queries to retrieve information from a database of information stored in the computer [REI88]. It is usually intended to be used by people who are not professional programmers. Query languages are usually high-level. Much of the work carried out has been on SQL (Structured Query Language) [CHA77] and QBE (Query-by-Example) [ZLO75]. In the case of both of these, the data being queried is assumed to be stored in the form of tables or relations [COD70]. The tables have names, as do column headings. In the examples given below I use a database consisting of a table, *EMPLOYEE*, with columns labelled *NAME*, *DEPTNO* and *SALARY*. Each row in the *EMPLOYEE* table relates an employee's name, department number, and salary.
For this database the following English instruction :

Find the names of the Employees in department 50

would be written using the keyword SQL commands SELECT, FROM and WHERE as shown in figure 3.1 below. This query will return a list of employee names who meet the criterion stated in the WHERE condition.

To give the same question in QBE, the user will fill in a copy of the EMPLOYEE table displayed on a CRT screen, as shown in figure 3.1. The underlined word "*Brown*" is an "example element". Any such example element can be chosen by the user instead of a variable. Thus the entry "p.Brown" in the NAME column is the equivalent of SELECT NAME in SQL. The symbol "p" stands for print. More complex questions can be expressed by using other functions of the two languages.

SQL-type query languages are nearer to programming-type languages and may tend to be more difficult for a non-computer literate user to use. QBE on the other hand removes the need for the user to have a knowledge of the syntax of the query language. The query procedure is just a form fill-in method.

QUERY LANGUAGE	Example query for "Find the Names of Employees in Department 50"
SQL	Select Name from Employee where Deptno = 50;
QBE	

Employee	Name	DeptNo	Salary
	p.Brown	50	

Figure 3.1 - Example of SQL and QBE queries.

Query languages like SQL are to all intents and purposes programming languages. To execute any substantial query requires that a number of separate queries be joined or unioned together, resulting in a set of instructions not unlike a short computer program. SQL-type languages demand that the user learn the syntax of the language. In addition the user needs to be able to formulate the query requirements in a logical manner.

Originally many of the relational databases such as ORACLE, INFORMIX, DB2 and later versions of INGRES only provided a programming type environment for SQL with the user being required to formulate queries in the programming languages.

However in later releases of all of these products form fill-in interfaces and 4GL interfaces have been provided to remove the need for the user to know SQL. INFORMIX, for example offers form design, menu design and table design through menu selection of possibilities. A recently released version of ORACLE facilitates the creation of the database through a graphical designer [ORA90]. The system is intended to store data associated with the Computer Aided Software Engineering (CASE) process. The database is created through four main diagramming tools :

Entity-Relationship Diagrammer
Function-Hierarchy Diagrammer
Dataflow Diagrammer
Matrix Diagrammer (Inter-type Relationships)

Although the interface is designed primarily for use by a software engineer, the fact that it facilitates the design and creation of a database through the drawing of diagrams presents a new and simple manner by which inexperienced users could interact with relational databases.

It is interesting to note that even with the provision of this graphical interface, ORACLE still provides a direct SQL interface to the data.

3.2.1.2 Natural Language Interfaces

The goal of most natural language systems is to provide a program interface that minimises the training required. To most this means supplying a system that allows the use of the words and syntax of a language used in common non-computer discourse, such as English. There is some disagreement as to the amount of "understanding" or flexibility that is required in a natural language system. For example, systems have been proposed that provide natural language by permitting the user to construct English sentences by selecting words from menus [TEN83]. Although the individual words are natural and their linkage may result in a natural language sentence, many experts including Woods [WOO77] argue that a system using English in an artificial format could not be considered a natural language system. Woods assumes that a NL system should have an awareness of discourse rules that allows the omission of details that can be easily inferred. In a natural language interface system, four different domains need to be considered :

Conceptual Domain
Functional Domain
Syntactic Domain
Lexical Domain

3.2.1.2.1 Conceptual Domain

This refers to the application domain of the language. It defines the objects and actions covered by the interface. Users may reference only the objects and actions the system is capable of processing. For example, if a system contains knowledge only about assembly-line employees in a company, a NL query like :

What is the salary of Joe Blogg's manager ?

is invalid and will not be handled by the system, as it doesn't know about managers. A language could expand the conceptual domain of the underlying system by recognising concepts (eg. manager) that exceed the system's coverage and respond appropriately [COD74].

3.2.1.2.2 Functional Domain

The functional domain is defined by the constraints on what can be expressed within the language and without elaboration. For example, in the above sample query the database may contain all of the information on both employee and manager salaries. However the system may not have the functionality to evaluate the above query. It may be necessary to express the query in the following two queries:

Who is the manager of Joe Bloggs ?

The system returns: **Tom Smith**

Then apply the following query :

What is the salary of Tom Smith ?

So although the initial complete query is conceptually valid, as all the data is available, the system does not have the information to know about the salary of Joe Blogg's manager.

3.2.1.2.3 Syntactic Domain

The syntactic domain of a language is determined by the number of different paraphrases of a given command that are acceptable. For instance a system may not be able to understand:

What is the salary of Joe Blogg's manager ?

because of the possessiveness of the statement. However if the above statement was paraphrased :

What is the salary of the manager of Joe Bloggs ?

the system might respond correctly to it.

3.2.1.2.4 Lexical Domain

Finally, a sentence may not be allowed because the words are not in the system's lexicon. For example, in the salary query, this may be rejected by the system if it doesn't know the word ***Salary***. But if the word ***Earnings*** was used instead, the system might respond positively.

Since no systems will be able to cover all possible utterances of a natural language, they are in some sense a type of formal computer language. Therefore these systems must be compared against other formal language systems as regards function, ease of learning and recall, etc. One study carried out by Jarke et al [JAR85] dealt with a comparison of SQL with NL queries on a particular database. The findings indicated that the participants of the study using the NL type queries experienced difficulty using it. The main reasons cited for the problems were lack of functionality of the System. The subjects were attempting to execute queries which were outside the functional domain of the system. Problems also arose with attempts being made to use words in the queries which the system knew nothing about.

In an experiment carried out by Bell and Rowe at UC Berkeley [BEL90], they compared the performance of three systems, a Natural Language System, SQL and a Graphical Query System. The people involved in the experiment included:

Computer Novices

End Users
Programmers
Database Experts
Interface Experts

Each participant was required to carry out a set of queries, each in the different interfaces.

With respect to Structured Query languages versus Natural Languages, the following conclusions were drawn :

- The best performance was achieved by experienced SQL interface users.
- N.L. results were mixed.

In general, N.L. shows promise as a better interface than SQL. However, the performance with NL is too unpredictable. This is due to the inability of current N.L. systems to cover <u>all</u> functional, syntactic and lexical domain possibilities.

Although it is accepted that N.L. systems are not currently developed to such an extent that they can easily be implemented as interfaces to all sorts of applications, much work is still going on to perfect it as far as possible. In the area of **Text Retrieval** much work is going on in an attempt to provide NL query facilities from which a list of appropriate documents will be returned.

3.2.2 Menu Based Interfaces

The distinction between menu-driven and command-based interfaces can be a fuzzy one. The fuzziness comes about because menus have many characteristic features, but seem to lack defining features that are either necessary or sufficient. For my purposes, a menu can be defined as a set of options, displayed on the screen, where the selection and execution of one (or more) of the options results in a change in the state of the interface [PAA88]. Menu screen panels usually consist of a list of options. The options may consist of words or icons. The word or icon is not arbitrarily chosen, but conveys some information about the consequences of selecting that option. When one of the options is selected and executed a system action occurs that usually results in a visual change on the screen. The range of options is usually distributed across a number of different menus.

3.2.2.1 Menus vs Commands

Although there are many similarities between menus and command languages, there are some distinct differences :

- ***Menus*** simply require that the user be able to understand or recognise the options, whereas ***commands*** require the user to learn and recall the command names and argument structure.

- ***Menus*** guide the user, step by step, suggesting viable options and hiding inappropriate actions, whereas ***commands*** must be learned and cannot prevent the user from trying options in incorrect contexts.

- ***Commands*** are highly flexible permitting the user to reorder the actions into procedures or descriptions never anticipated by the designer, whereas ***menus*** need to be organised into structures that can limit their flexibility.

- ***Commands*** require very little screen space, whereas ***menus*** can be very demanding of space and may require the user to navigate through several panels.

- Because ***commands*** are faster and more powerful, but require more a priori knowledge and provide less guidance, ***commands*** should be better for experienced users, whereas *menus* should be easier for the beginner.

As can be seen from the above list of points, both types of interface have their relative merits. For a new user, using a system with many possible alternatives or for users using the full extent of a package, menus may prove easier to use and possibly more productive. However, for an experienced user the process of navigation through a number of menus to execute a single task may prove tedious and slow when typing in a single command would be more effective.

Many systems recognise these relative merits and make provisions for both. For example, *Dbase III+*, a PC-DOS database management system, provides a menu driven method of creating and manipulating the databases. In addition to this, when the user has got a good enough grasp of the system's commands, he/she may leave the menu and work from a system prompt where the same operations can be carried out but by means of command entry.

Another PC based database package, *DataEase*, applies only a menu based approach to database operations. Screen and report layouts are defined through menus, as are the database structure and queries.

Menu interfaces are clear, and simple in use but for a system applicable to all classes of users, the necessity exists for the provision of both menus and commands.

3.2.3 Windowing Interfaces

In today's software advertising, the term *"Windows"* appears almost as frequently as "*user-friendly*". Windowing systems, according to their proponents, are inherently easy to use and tend to be conducive to productivity. But, while it is clear that some windowing systems offer benefits to some users under some conditions, we still have little understanding of the implementation, user, and task parameters that lead to increased productivity and satisfaction. A window may be defined as an area on a computer display, usually rectangular and usually delimited by a border, that contains a particular view of some data in the computer [BIL88]. With current computing power and depending on the implementation, windows may represent different host computers, different operating system environments (ie. MS-DOS, OS/2 and UNIX in 3 different windows on the same screen), different files in the same application domain or different views of the same file. Often windows tend to accommodate features of other interface techniques. By Billingsley's definition [BIL88], *pull-down menus*, *pop-up menus*, *dialog boxes* and *message boxes*, used to separate specific segments of the user-system dialog from the main application, can also be considered windows. The term ***Windowing*** has been used extensively to describe any system which can display a window. Although many windowing systems offer concurrent execution of the applications in the separate windows, concurrency is not a necessity of such a system.

The concept of windowing is appealing because if supports the way people really work. Office-based information workers, for example, routinely monitor and manipulate data from a wide variety of sources. They typically spend a great deal of time synthesising, summarising, and reorganising information [CAR85]. It was also observed that people tended to position papers on their desktop to reinforce the way they had categorized tasks. As work proceeded, task materials were frequently rearranged to reflect changing priorities. When these workers use computer screens, they also deal in tasks with changing priorities. People seldom tend to carry out one task, from start to finish, but tend to move between tasks. This applies to both computer and non-computer based office tasks. Most conventional non-windowing systems have a number of drawbacks when considered in the light of the way people work :

- Users can view only one screen worth of information from one source at a time. At any one time, they have only access to one part of their overall task domain. Their on-screen work materials do not reinforce task groupings, remind them of unfinished tasks, or reflect task priorities.

- Users cannot switch between tasks or sub-tasks without changing or, more typically, replacing the current display. This involves terminating the current process and initiating another.

- Re-starting a process normally involves recreating a previous working context. Most such systems do not provide any facility for saving the state of an environment and process at the time the process was halted.

- The integration of information from a number of different systems involves the memorisation of information from these different sources or some other sub-optimal procedure for integration. [CAR85]

Based on these problems, it would appear that an environment which supports windowing would eliminate most of these problems. In answer to these problems, windows offer the following solutions:

- Windowing systems allow users to apply what they know about spatial management of printed materials on a desktop to the arrangement of electronic data on a computer display. If windows overlap (see figure 3.2), there is a great correspondence between the desktop and screen based work.

- Windows make it possible to change the focus from one task to another with little effort, since components of both tasks can be viewed on the display simultaneously.

- The need to re-establish context between windows is removed as the individual windows preserve their current state.

- Windowing systems provide a visible memory cache. This feature makes them particularly useful for tasks in which users must: (1) integrate information from a secondary file or application into their primary task domain, (2) monitor changes in a secondary process while they perform a primary task, (3) transfer information from a specific location in one file to a specific location in another. [BUR85]

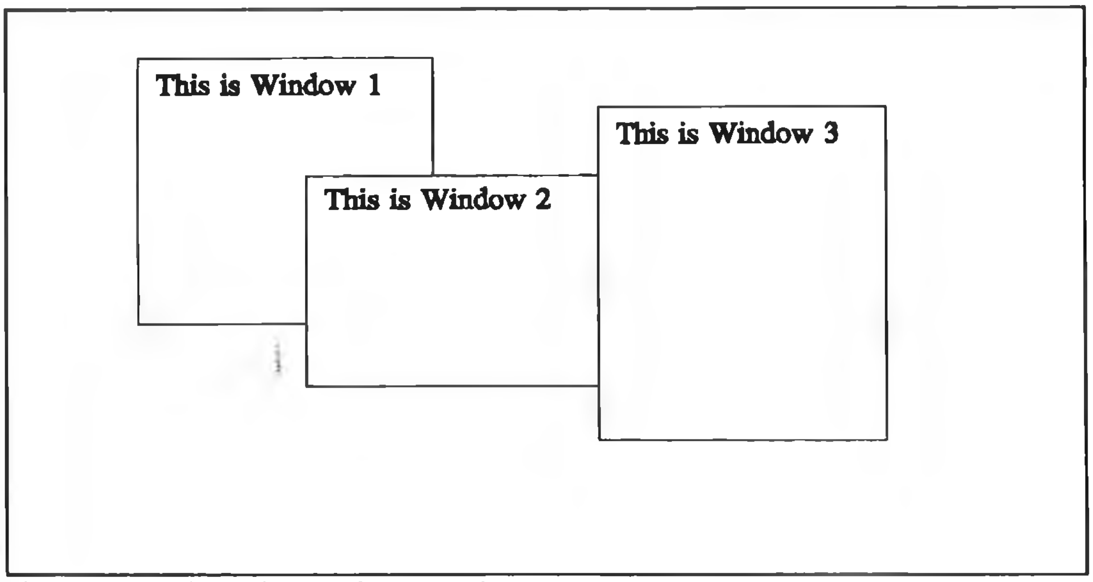


Figure 3.2 - Windowing configuration in an overlapping system.

The first commercially-available windowing system included good functionality and ease-of-use. Created by Dan Ingalls, it was part of the Smalltalk [GOL77] programming environment developed at the Xerox Palo Alto Research Centre (PARC) in 1975. It included overlapping windows and a direct manipulation interaction style. This was novel in itself, and due to the fact that this was all applied to the first O.O. programming language environment, made the system revolutionary. Since that original development, Xerox have maintained much of these principles in Systems like Xerox Star (8010) office workstation. Other manufacturers like Apple adopted this environment for their all in one office workstation called LISA in the early 1980s. This windowed environment was further developed and is part of the friendly environment which has made the Macintosh so successful. The release of the Apple Macintosh spurred on many software manufacturers to get on the Windowing 'bandwagon'. Manufacturers like Sun Microsystems, Microsoft and Quarterdeck have developed their own Windowing environments which sit on top of MS-DOS and UNIX. Many application developers have added windowing interfaces to their individual products. Problems started to arise as manufacturers tried to outdo the each other. The public stepped in and with so many diverse interfaces, the necessity for a windowing environment standard arose. User interface standards are unlikely to emerge from the computer industry itself, although there is a strong impetus to standardise the underlying architecture of windowing systems. This stems, in part, from the increased use of windowing systems in networked environments. In such environments, problems can arise when windowing systems must accept input from a number of different applications, running on different host computers. Standardised window management protocols, such as , are gaining wide acceptance as potential solutions to these problems.

The X Windows system, for example, or simply X, is a hardware-independent windowing system for workstations. It was developed in 1984, jointly by MIT and Digital Equipment Corporation and has been adopted by computer industry as a standard platform for graphics application [NYE90].

It attempts, like IBM's SAA project, to provide the same interface protocol for many different hardware configurations.

However, although in the last number of years has grown in popularity significantly, industry-wide standards for user interfaces to windowing systems are likely to emerge more slowly, at least partially because of the following facts :

- Many developers are proud of the originality of their interfaces. They fear that the implementation of standards will threaten both their creativity and its potential monetary rewards.

- Some companies are perceived as trying to promote standards based on their own interface design conventions, out of self interest. This has caused some other companies to be wary of any standardisation efforts.

- Certain companies have brought, or threatened to bring, legal action against any company which adopts their interface design conventions. In recent times the best example of such an action is when Apple sued Microsoft claiming that the Microsoft Windows product was too similar to Apple's environment. Fear of legal action may encourage even those who favour standardisation to opt for their own different interfaces.

- The last, and possibly the most important, reason why standardisation will be slow, is the fact that little research has been done into design alternatives and their impact on usability. There is no guarantee that the current standardised interfaces would be the proper approach. Without such a body of research, it is difficult to shift the focus of debate from industry politics to user considerations.

In addition to all this, current windowing systems tend to place many demands on the hardware available. Many experts argue that current technology is not well enough advanced to support substantial windowing environments. Their arguments rest on three hardware drawbacks :

- **Screen Size** - To display a number of different windows on one screen at the same time demands a reasonably large screen. Up until recently such screens have been unavailable and even today larger screens are beyond the reach, in terms of price, of most users.

- **Processing Speed** - Monitoring the activities of a number of different windowed sessions at the same time places many restrictions on the CPU. Microprocessors like the Intel 8088/86, Motorola 6800 & 68000 or the Zilog Z80 have proved ideal for many conventional single session activities. Applying these to the concurrent processing of windowed applications as well as the management of the windowing system results in unbearable delays and possible reduced functionality in the overall system. In the last three years, this problem has been somewhat remedied through the introduction of true 32-bit, multi-tasking processors, eg. Intel 80386,80486 and the Motorola 68030,68040. Many manufacturers have opted for new Microprocessor designs, with Reduced Instruction Set Chips (RISC) becoming popular, eg. IBM RS-6000 chip-set, Sun SPARC, or Acorn ARC. These new processors are built for multi-tasking, so in many respects a windowed environment is a natural front end for them. However, here again price is a stumbling block, computers sporting these CPUs are far from cheap, restricting their uses.

- **Low Screen Resolution** - Many of the computers in use comprise relatively low resolution display units. Until recently the highest PC graphics standard was CGA sporting a maximum of 500 x 300 approx. in 2 colours. This would be totally inadequate for a windowing environment, with the contents of the individual windows becoming totally illegible. New standards like VGA & super VGA introduced 1000 x 700 approx. with up to 256 colours. The facilitates the display of legible text in much smaller fonts.

These drawbacks have all had solutions mentioned, but much of the computer equipment installed in industry today would not be able to cope with the hardware requirements imposed by windows.

Windowing environments are novel in themselves, but with the introduction of windowing environments, a new interaction technique was developed - ***Direct Manipulation.***

3.2.4 Direct Manipulation Interfaces

With the development of the windowed environment at PARC, they also developed a new input device which could easily manipulate the windows and the contents in them. The ***mouse*** incorporated a hand-sized unit, containing a ball and 2 or 3 buttons. Moving the mouse on the desktop resulted in a corresponding movement of a pointer icon on the screen. Pressing one of the buttons on the mouse might initiate some computer operation due to the positioning of the mouse pointer on the screen. Icons of familiar office tasks were presented on the screen, with the user activating the required task by pointing at the icon using the mouse and selecting it using a mouse button.

Shneiderman [SHN82][SHN83] decided on the phrase *Direct Manipulation* for this approach to HCI and established that for a user interface to be classed as a direct manipulation interface, it should exhibit the following characteristics :

- Continuous representation of the object of interest on the screen, whatever its representation.

- The use of simple physical activities or of labelled buttons to carry out required processes/actions instead of the conventional use of languages and commands with complex syntax and command names.

- Facility provided to allow reversal of operations incrementally on an object, with all operations on the object being immediately apparent on the objects representation.

Shneiderman took the approach to direct manipulation interfaces as being interfaces which accurately and closely modelled the real world equivalent [SHN82][SHN83].

For example, in wordprocessing terms, many wordprocessors employ the "***What you see is***

***what you get*"** philosophy. Here any slight change to the appearance of the document on the screen would result in a corresponding change to the printed document, ie. a direct manipulation of the document through its screen representation.

In Spreadsheets, a change to a particular cell on the worksheet might result in an appropriate change to many other cells on the worksheet. The state of the Worksheet is always up to date with respect to the representation.

Shneiderman felt that direct manipulation interfaces could be identified by a set number of characteristics. However in 1986, Hutchins, Hollan and Norman took a more heuristic approach to the categorisation of direct manipulation interfaces [HUT86]. They referred to it as an *Orienting Notion.* They considered the existence of a gulf between the Goals and intentions of a user interacting with a system and the concepts and operations represented in the system. Two major problems arise -

- ***The Gulf of Execution*,** This refers to the required transformations required to turn the users goals into the input actions for the system.

- ***The Gulf of Evaluation*,** This deals with the problems of representing the system's reactions in a manner which can be perceived understood and correctly evaluated by the user with respect to his/her goals [HUT86].

Where Direct Manipulation fits in is to bridge the gap between the two gulfs and facilitate the use of the system's outputs directly as components of the user's input language. The user can then use a representation of the output as a representation on which further manipulation can be carried out.

Out of this direct manipulation technique, and its integration with windowing environments, these sorts of windowing environments came to be called **WIMP** interfaces, standing for Windowed Icon Mouse Pull-down menu interface. The Apple Lisa and later the Macintosh became the first widespread incarnations of such systems.

3.3 Databases and Interfaces

Databases have, since their initial development, tended to be purely textually based for the user. The data stored has always been purely textual or numeric. Large volumes of similar data have predominated. Indeed, relational databases have been based on this principle. Interfaces to these applications have also tended to be purely textual. Systems such as DB2, INGRES or INFORMIX have provided text-only interfaces to the user, with at best query editors, form designers for data entry or menu designers being provided to simplify the creation of, manipulation of, and access to the database.

With the increase in the power of computers, developers, and indeed users, have seen new openings for computers and databases. The new graphical capabilities of them can offer a different, simple and to a large extent self explanatory user-interface to the data. With these advances, the adage *A picture is worth a thousand words* can now be applied to database access.

With the introduction of semantic and object-oriented databases which are no longer just tables of similarly structured data, but hierarchies with inter-relations between different classes and objects, the applications of graphics to database interaction seemed a natural progression.

Although still relatively new, a lot of work has been carried out in the development of interaction techniques and systems to object-oriented database and object storage systems.

In the remainder of this chapter, I will examine some of the approaches that have been taken to both enhancing current interface techniques to data storage and database systems, and applying new interface techniques to new classes of systems. I will outline two interface systems developed for semantic databases - SNAP and ISIS, and look at an approach taken by Oracle to develop a design tool based around their relational databases - Case*Designer.

3.3.1 SNAP: A Graphics-Based Schema Manager

3.3.1.1 Introduction

SNAP [BRY86] is a system designed to provide simple interface techniques to the IFO data model, a variation of the semantic data model. It is a general-purpose schema manager for the IFO model which provides a coherent paradigm to support the three activities of schema design, schema browsing and query specification. SNAP has, through its development, characterised a number of features which should be present in any graphical database schema access package.

(i) permit the simultaneous, coherent display of all types of relationships arising in the underlying data model.
(ii) permit a modular perspective of the schema.
(iii) display the schema at several levels of abstraction.
(iv) permit flexible visual rearrangement of the schema.
(v) facilitate returning to visually familiar, static representations of schema components, easily.

In addition to this, SNAP makes a fundamental contribution in the area of schema representation and its offering to graphics-based query specification.

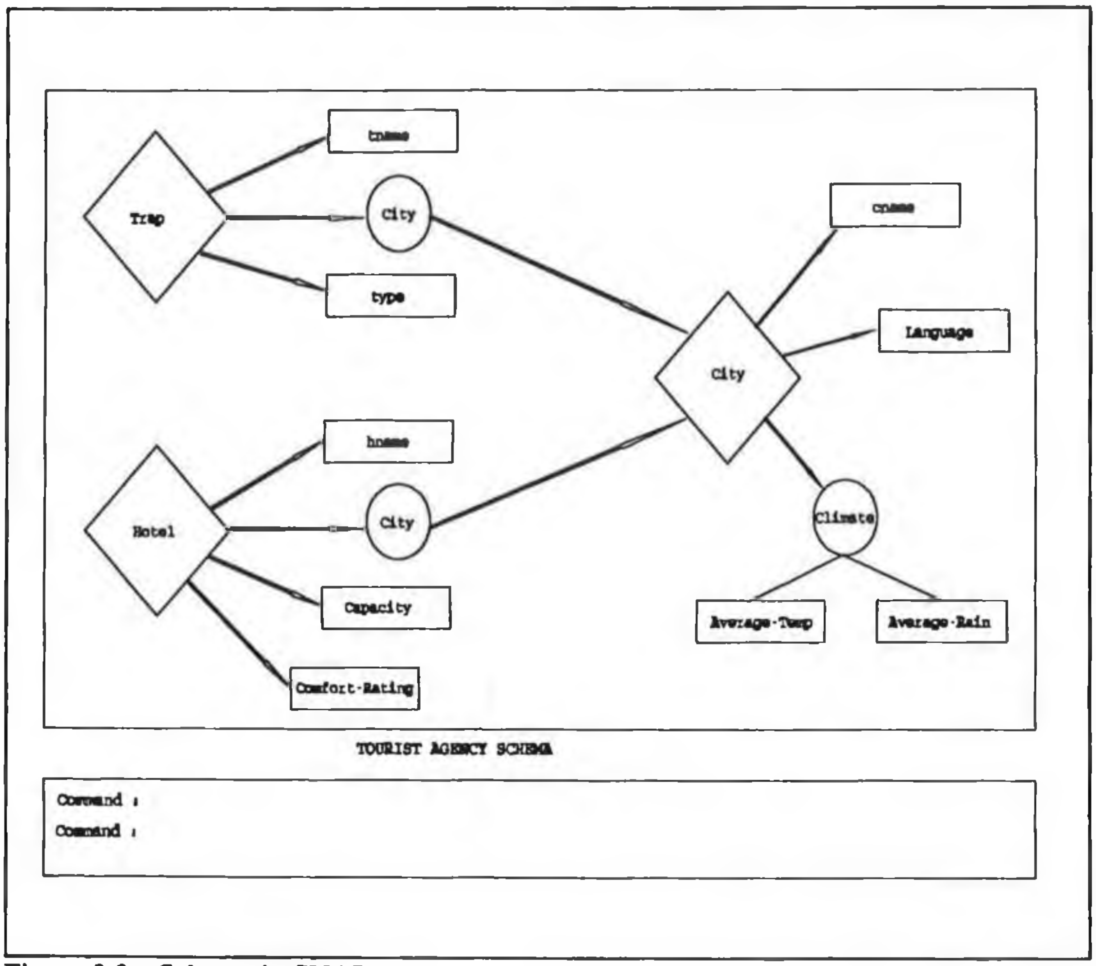
3.3.1.2 System Description

SNAP represents the underlying semantic database in a graphical manner, providing facilities for designing the schema and adding new classes to it. In addition, through this graphical representation the user can browse through the schema and the data included in it.

3.3.1.2.1 Schema Design in SNAP

The user communicates with SNAP through four windows: a graphics windows for the schema and the query specification, an enhanced text-based window for displaying query results , and a text-based window for special user interactions.

To create new nodes in the schema, the user positions the mouse pointer at the required position on the schema, pop-up menus allow the entry of the attribute details for this new node (class). This may prove too simple for experienced users, so, there are also plans to develop a *hybrid* approach to schema definition, in which the user could initially specify a fragment representation of the new schema section using a compact text-based syntax, and subsequently modify the graph corresponding to that specification using mouse-based commands.



```
Command :
Command :
```

Figure 3.3 - Schema in SNAP.

Rules are built into SNAP to prevent it from violating the construction rules for IFO schema. Therefore when constructing the schema, users are prevented from carrying out any option which will invalidate the IFO rules.

3.3.1.2.2 Schema Browsing in SNAP

As mentioned in the introduction, SNAP aims to make flexibility in viewing the schema a key factor in its implementation. The visual representation of schemas in SNAP include fundamental features which support both modularity and different levels of abstraction. The

SNAP system combines these features with the power of interactive graphics to provide a rich set of basic commands for schema browsing.

Direct manipulation is of the utmost importance in viewing and manipulating the schema. In general, facilities are provided for re-positioning objects, hiding/displaying objects, panning and zooming around the schema and automatically reformatting the ISA hierarchies and complex object representations. There is often more than one method for accomplishing a single particular task. For example, there are four distinct methods for re-positioning objects by dragging them with the mouse : move individual nodes, move fragment representations, move entire type-sub-type lattices, and move all nodes in a specified region of interest.

Finding and displaying nodes of particular interest is also facilitated. Nodes which are off-screen can be displayed, as can sub-types or super-types of a particular node and even specific links between nodes.

3.3.1.2.3 Query Specification in SNAP

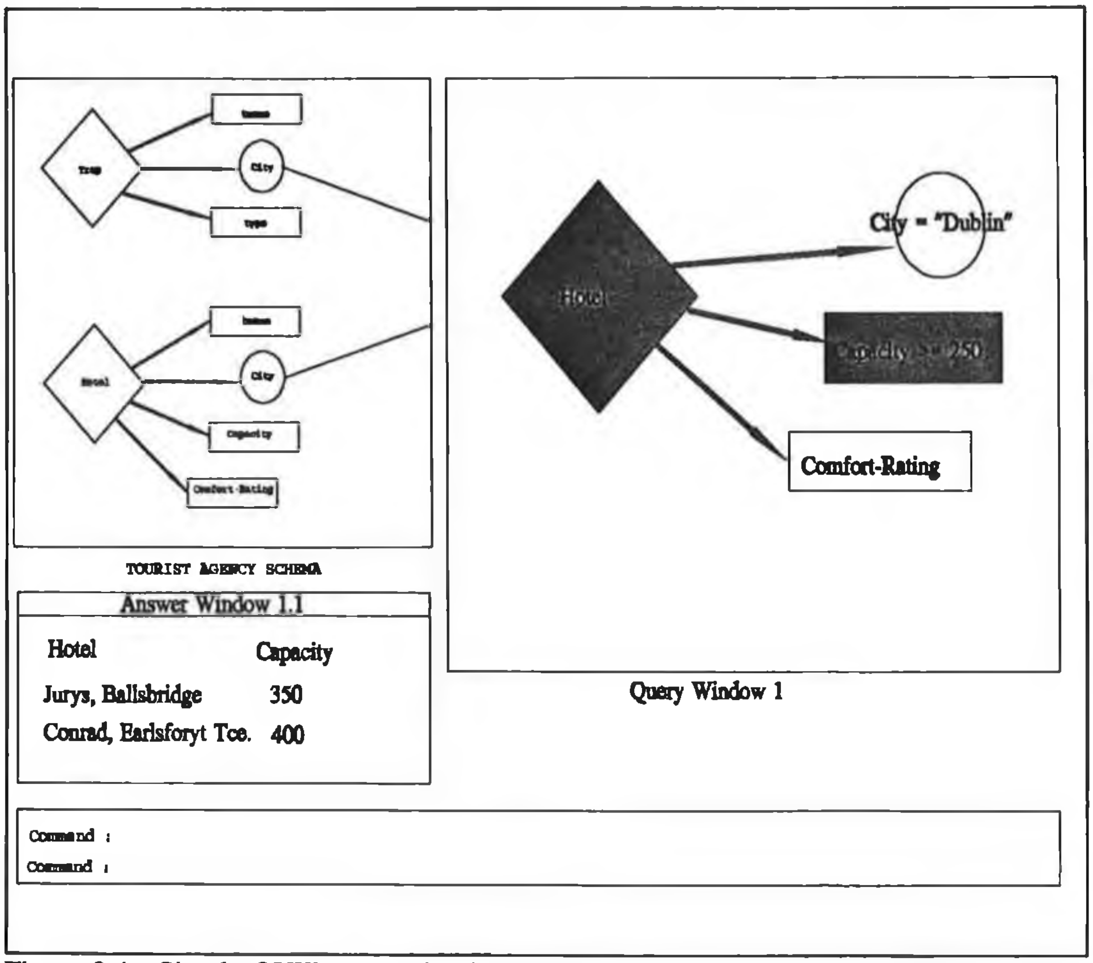
Queries in SNAP tend to be issued in a ***Query-by-Example*** [ZLO77] manner.

In SNAP, there are at usually at least four windows displayed when a query is being created and executed. The first window is a display of the relevant sub-section of the schema. The second is a larger interactive window which displays the actual type or types on which the query is being executed. The third is the answer window in which all the results from the query are displayed and the fourth is the command window which is displayed during most operations.

In SNAP there are four main types of queries which can be executed. These are all currently created in a graphical manner.

(i) **Simple Query Creation**

This involves the creation of a query in a QBE manner. The required section of the schema is selected. The attributes to be returned in the query results are selected, these are the highlighted ones in the diagram (Hotel and Capacity). The attributes on which the query conditions are to be based, have their conditional values filled in.



Answer Window 1.1

Hotel	Capacity
Jurys, Ballsbridge	350
Conrad, Earlsforyt Tce.	400

```
Command :
Command :
```

Figure 3.4 - Simple QUERY creation in SNAP.

(City and Capacity)

When the query is executed the results are displayed in the answer window with the selected attributes - Hotel and Capacity heading the columns of the answer window.

(ii) Queries using Comparator Arcs

Comparator Arcs facilitate the linking of two or more types and basing the query on the linkage. So, for example, the *comfort-rating* of an Hotel might be linked with a comparator-arc to the *comfort-quotient* of the Tourist type. This might have the effect of listing all of the hotels in a particular city which have a comfort-rating which is greater than or equal to the comfort-quotient given by the tourists. The practical effect of such a link is to list the only the hotels meeting a certain level of comfort, in the required city.

(iii) **User-defined composed and inverse functions**

The user can define composed and inverse functions in a visual manner. For example, the user is interested in creating a function, mapping each trip to the set of languages that will be used during that trip, so that he/she can compare the languages needed for trips with the languages that travel guides. This mapping is created in the interactive graphical manner, involving only the selection of nodes and the drawing of links.

(iv) **Complex Queries**

It is essential for any query system that complex data requests can be made to the system. SNAP provides for this in the same manner as the rest of the queries. The user can extract details from many different types based on some common attribute. For example, Get the hotels, languages and scenic-spots for given cities. Here the city is the common attribute and the query joins three different types together to arrive at the result. The user can control the format of the output here, determining how it is laid out, the order of the display etc.

Combining any or all of these types of querying mechanisms provides for complete coverage of data querying requirements.

Although SNAP is currently applied as a graphical interaction mechanism for IFO, a semantic database, the developers feel that with minimal modification it could provide simple interfacing with relational databases. There is a relatively straight-forward translation from a natural sub-set of IFO schemas into third-normal form relational schemas, in much the same way as E-R diagrams are mapped to relations (See ORACLE CASE*Diagrammer below).

3.3.2 ISIS: Interface for a Semantic Information System

3.3.2.1 Introduction

ISIS is a system that exploits the visual dimension for database programming. It allows users to construct, maintain, and query a database using a graphical interface and a consistent operational paradigm. As with SNAP, ISIS is based on a high-level semantic data model. ISIS integrates several forms of database programming into a single interface that is rich in capability yet intuitive enough for non-experts to use.

The construction of database retrieval systems constitute a very important part of programming in commercial data processing. A system like ISIS allows a broad class of users to become "*database programmers*" and can substantially reduce the amount of time required to construct programs of this type.

Many of the database query languages that have appeared suffer from the fact that they are textually oriented and very formal. Although simple queries are reasonably straightforward, slightly more complex queries exceed the capabilities of a novice user. The use of the visual dimension seems to hold promise as a way of providing a more intuitive interface in the context of a two-dimensional syntax. ISIS uses the visual dimension to integrate three aspects of database programming. With ISIS, a user is able to build a database or modify an existing one, to browse through the contents of a database in order to answer questions about the data or the schema, and to construct queries that can be saved for later use. All of these activities are accomplished using the same style interface and the same iconic representations, so that a user is able to move easily from one activity to another at any time.

3.3.2.2 System Description

ISIS provides multiple views of the database schema, as well as different views of the data itself. The screen structure is made up of different *views*. *Views* can contain (i) **menus**, (ii) **text-windows**, (iii) **windows**.

(i) **Menus**

Menus provide a consistent list of commands. The commands available are the same from view to view, with the same semantics, but the actual execution of the commands is determined by the current view.

(ii) **Text-Windows**
Text-windows are used for textual input and output. They are used for displaying error messages as well as prompting the user for input from the keyboard, mouse or function keys.

(iii) **Windows**
Windows contain the graphical representation of the schema or subsets of it. Commands are provided for changing the display, eg. panning, zooming, etc.. A graphical editor is provided, when appropriate, for changing the representation of the schema.

ISIS operates at two levels, the *schema level* and the *data level.*

Classes are represented in ISIS as rectangular nodes on the screen with them shaded in a class-unique manner. Attributes are represented by their name being displayed, with the background pattern to the name indicating the class which is its domain. Inheritance in the hierarchy is represented using lines, linking classes to sub-classes.

3.3.2.2.1 Schema Representation

The main schema representation structure is the **Inheritance Forest View.** This structure represents the schema indicating the class hierarchy. The attributes associated with each class are indicated in that class's node. A specific pointer is used to highlight the selected node.

In the figure 3.5, a sample musical instrument database schema is displayed. In the actual ISIS system each of these nodes would be displayed using a different background pattern. In addition to this, menu options would be provided at the right-hand side, and bottom edge of the window. These would include icons and menu headings, which, when selected would display an enhanced list of options.

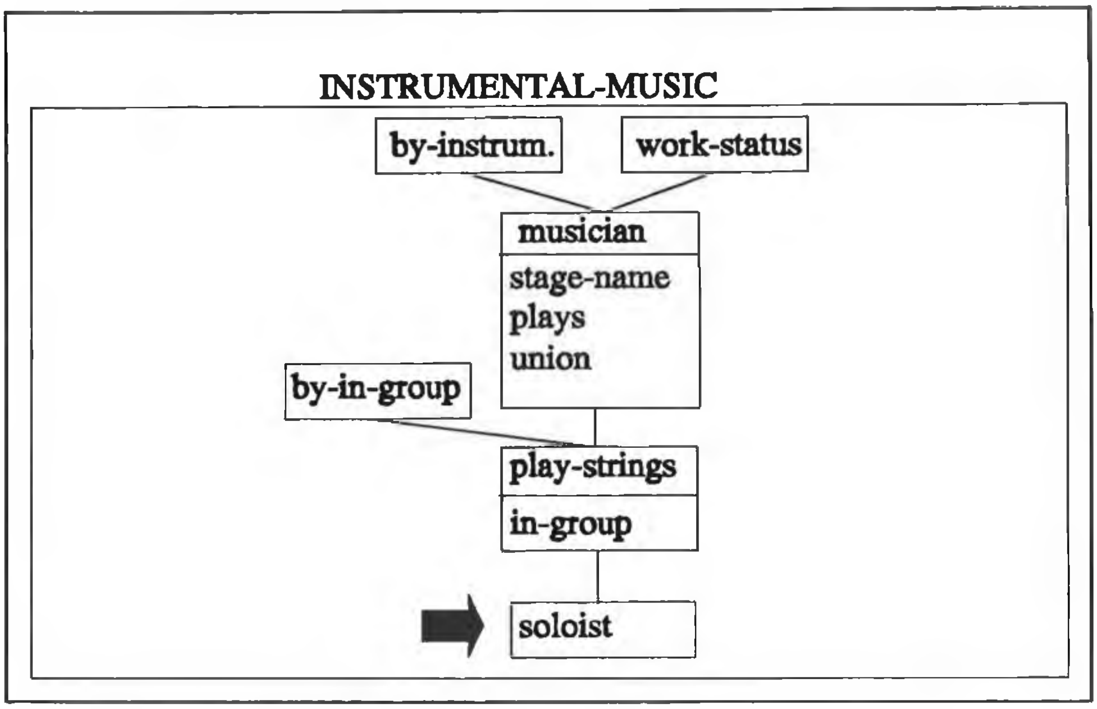


Figure 3.5 - ISIS Inheritance Forest (without shading).

3.3.2.2.2 Data Representation

The data in the database is represented with overlapping windows. The selected class is displayed in one window. In this, is a list of the class's attributes, both its own and its inherited ones. In a larger window, underneath and to one side of this, is a list of the member data objects of the selected class. This list may be panned. Selected members are highlighted with bold text. Further information can be displayed on the selected member, by making a further selection from one of the menus.

3.3.2.2.3 Database Manipulation and Extension

The schema hierarchy, as mentioned above, displays the classes in the inheritance-forest, displaying links between them, while displaying the attributes associated with each.
Sub-classes can be added to the hierarchy by selecting the super-class. Then attributes can be created and associated with this class.

ISIS does not provide a query facility in the conventional notion of one, but does facilitate the creation of predicates which define memberships of a new class. The predicate constructor is akin to a graphical query screen and allows a number of different classes to be involved. The new class's members are determined by a set of rules applied to the other classes involved and these contribute the members to the new predicate class. A predicate, if created intelligently, can work as efficiently as a query, grouping all the required data matching a set number of conditions being grouped together.

In summary, ISIS integrates several aspects of database programming. In particular, it allows users to construct schemas, to browse through the database at both schema and the data level, and to formulate *queries* through predicate creation, that can be stored as part of the schema and reused at some stage in the future.

3.3.3 CASE*Designer

3.3.3.1 Introduction

CASE or Computer-Aided Software Engineering, represents a comprehensive philosophy for modelling systems by combining software tools and structured system development methodologies. A methodology defines the process of engineering a system, and the approach and techniques to be used. The CASE tools provide a database for the system engineers and set of facilities that automate many of the techniques during the entire system life-cycle.

3.3.3.2 System Description

CASE*Designer is the name given to a number of illustrative tools that have been developed to support the Computer-Aided Systems Engineering concepts. It provides a multi-windowed, multi-user, graphics interface to the development database - CASE*Dictionary sitting on Oracle.

Following a structured method, CASE*Designer uses diagrams to model the business, its activities and how it uses information to support these activities. The following diagramming tools are provided :

- **Entity-Relationship Diagrammer.**
  This facilitates the creation of diagrams to represent entities, the vital business relationships between them and the attributes used to describe them.

- **Dataflow Diagrammer.**
  This is used to create diagrams to show the flow of information within an organisation, things that affect the organisation and where the data is stored.

- **Function Hierarchy Diagrammer.**
  This may be used to transform notes taken during interviews into structured functions. These functions describe what the organisation does or needs to do, irrespective of how it does it. This diagrammer creates and arranges the functions in a strict hierarchical order.

- **Matrix Diagrammer.**
  It allows the development of a matrix showing the associations between two types of information; for example, functions and entities, or functions and business units. This is useful in recording associations between these types of information, or to conduct thorough cross-reference and completeness checks.

These tools facilitate the creation of a database which contains the relevant data associated with the development of a software system.

CASE*Designer is in itself an application tool for designing software. However, in its internal construction it is essentially a high-powered graphical user-interface to the underlying Oracle database. Figure 3.6 shows the internal construction of CASE*Designer.

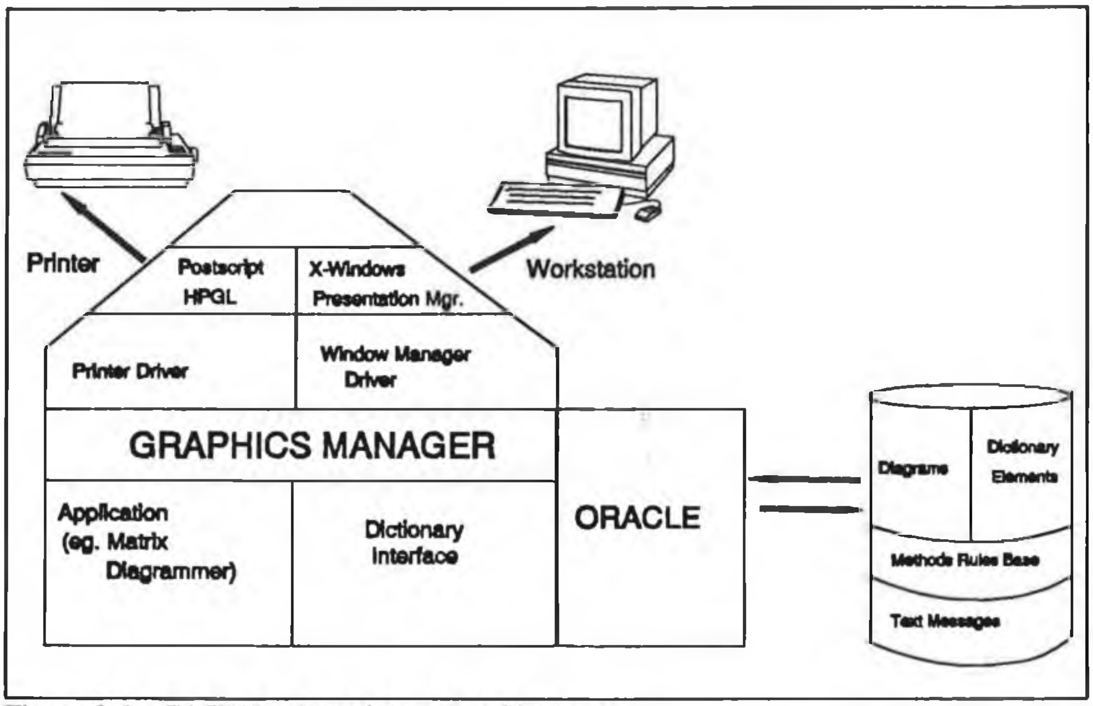


Figure 3.6 - CASE*Designer internal architecture.

CASE*Designer consists of an advanced **Graphics Manager** at its core. Above this is a set of interface tools which support the use of workstations running either IBM Presentation Manager or X-Windows. On the printer side, support is provided for Postscript and Hewlett-Packard HPGL printer formats. Applications which run, such as Matrix Diagrammer, E-R Diagrammer etc., are removed from the graphical environment in which they operate and make calls to the Graphics Manager which controls the windowing environment. This provides the graphical interaction between the user and the application. For the data storage requirement associated with the application, it interacts with the Dictionary Interface which deals with constructing the database according to the application's instructions. The dictionary interface is responsible for the actual construction, and this is done on Oracle.

The types of information stored in the database includes the diagrams associated with the individual application tools, data-dictionary information concerning the entities in the system and a number of rules concerning the operation of the system.

Although CASE*Designer is essentially a set of CASE tools which interact with the user in a graphical manner and use the information entered to construct a complex database, it does illustrate the manner in which graphics can be applied to the construction of a database. In this case the database is relational. Up to now interfaces to RDBs have tended to be textual with at best some form of intelligent interface being applied.

CASE*Designer does remove the user from the underlying database model to a certain extent. Oracle is transparent to the user and is only accessible, in a transparent manner, through CASE*Designer's tools. Its set of graphical design tools does, however, show us how future users of RBDs may be able to interact with their databases, through graphical representations of the problem space which is automatically converted into the underlying database structure.

In a similar vein, INGRES has produced a 4th generation language interface to their relational database product. **INGRES/Vision** consists of two primary components :

- **Frame Flow Diagrammer.**
  This enables the user to create the applications structure visually, in a graphical manner, as if designing an organisational chart, by specifying the frames to be used within the application.

- **Visual Query Editor.**
  This links into the frame diagrammer, with the user defining the data to be accessed and the operations that will be made available within a given frame.

This approach is similar to CASE*Designer, but applies to general applications development. It does, however, show how the creation and manipulation of a relational database can be substantially simplified by allowing interaction in a graphical manner.

3.4 Conclusion

HCI will, without doubt, prove to be one of the most important areas of computer research over the next number of years. As computers become more powerful and more widespread, they will be applied to new and different application areas. To ensure their future acceptance, it is essential that they are made easy to use and do not require the user to assimilate lots of new information to use them adequately.

The areas of research discussed in this chapter outline the approaches being taken to giving software applications a nicer face.

At the present time, windowing environments seem to have become the new 'craze' in application software. In the last year, the release of Microsoft Windows 3 for PCs has resulted in a large increase in the sales of applications for this environment. In addition, with the increase in the number of UNIX systems being installed, X-Windows is also increasing in popularity. It would seem that over the next number of years, windowing environments on applications will become a standard. Indeed, IBM, through their SAA policy intend to apply the same windowing interface to many of its different computer systems. They are placing a lot of financial resources into the idea of a consistent windowed front-end.

Applications like CASE*Designer and interfaces like SNAP and ISIS illustrate the extent to which new interface technologies are being put - applying graphical interfaces to databases. In the next chapter, I will outline the prototype graphical interface which I developed to facilitate easy interaction with ONTOS. It will apply many of the ideas outlined in this chapter, having much in common with systems like SNAP and ISIS.

Chapter 4
GRIFON - A GRaphical InterFace to ONtos

4.1 Introduction

Object-oriented databases which have been developed to-date have tended to be programmatic extensions to existing programming languages, whereas DB2, Informix, Oracle [ORA90] or Ingres provide a user interface to their relational databases products, systems like ONTOS [ONT90], GemStone [MAI90][MAI86a] and Orion [BAN87b] only facilitate the access to the database by means of programs written in their underlying programming languages. Systems like ISIS [GOL85] and SNAP [BRY90] have attempted to provide some form of graphical representation to their underlying OO databases. SNAP, for example, attempted to provide a mechanism by which the user could create a database schema simply. The extension and maintenance of this schema would be done purely through a graphical format. Similarly, ISIS acted as an interface to a semantic database. It represented classes and relationships between classes through a graphical picture of the database. A shade and colour system was used with ISIS to act as a unique identifier for different classes. In this manner a link could be displayed between attributes of one class and their domains which might be other class, by means of a different pattern as a background for the attribute.

In chapter 3, we have seen a relational database product - Oracle's CASE*Designer [ORA90] allowing the construction of a database schema through graphical CASE tools. Their approach concentrated on the application area for the database and concentrated on providing a number of CASE tools for graphically representing the relationships between the entities in the problem space. These tools subsequently created the appropriate Oracle database schema for the relationships presented by the user. CASE*Designer is an interface to Oracle, but in its practical use is more probably a CASE system modelling application which sits on top of the relational database.

GRIFON (GRaphical InterFace to ONTOS) was developed based on my research into object-oriented databases and the approaches taken to providing user interfaces to them. GRIFON, unlike many of the current user interaction techniques, aims to give the user a picture of the database as it is. It endeavours to provide a simple, user-friendly yet powerful interface to ONTOS, an OODB closely linked to the C++ programming language (See Chapter 2). This chapter outlines GRIFON, the features provided, the philosophy behind these features and the practicality of its implementation. It also looks at the features which were taken into account but due to limitations in the current release of ONTOS could not be fully implemented.

4.2 GRIFON - GRaphical InterFace to ONTOS

4.2.1 Introduction

ONTOS, as it currently exists is an OO database for use with C++. It provides a set of classes and methods by which the user can access and manipulate the database. In its current release, it provides no user interface, only facilities accessible from C++.

GRIFON is a prototype system developed with the intention of taking the good features of current interface techniques to database systems, combining them with the facilities provided by object oriented databases, and in particular ONTOS, to produce a simple graphical interface to an OODB.

GRIFON is very much a research project. It has been developed with the features of OODBs very much in mind. Certain aspects of object oriented databases make themselves suited to graphical representation. GRIFON aims to take advantage of these.

4.2.2 Aims of the System

In the development of a prototype interface to an Object-Oriented database, I was attempting to establish a number of different things.

- **What sort of interface would suit an OO database ?**

  This is determined by the use to which it is put. However, for a general purpose interface, the natural structure of the schema, the concept of inheritance through a hierarchy, with data being transferred from class to sub-class and the manner in which classes may be composed of other classes would indicate that a graphical interface would be appropriate. Systems like ISIS [GOL85] and SNAP [BRY90] have further illustrated this point through their simplification of the interaction process with semantic databases. SIG [MAI87], with its use of windowing illustrates that even through the use of simple graphical representations to underlying databases, the user can gain a better picture of the database structure.

  In a study carried out by John Bell [BEL90], into the evaluation of different computer interfaces to databases by different classes of users, he found that graphical interfaces performed best by most different categories of users. The categories tested included End users, familiar with applications but not programming, programmers with no database experience, database experts and interface experts. The findings indicated that users and experts alike performed better and more easily by means of a graphical interface.

- **What level of information could be represented by such an interface ?**
  OO databases by their structure are capable of storing complex information with this data being inter-connected. Data is inherited from class to class and classes are made up of other classes. In addition individual classes have attributes and methods associated with them. In an SQL type interface the information returned is purely dependant on the information requested. In a graphical interface the potential for the representation of more varied information without it being explicitly requested is substantial. The user is presented with more options to choose from.

- **To what extent could direct manipulation and mouse interaction could be used?**
  In the development of GRIFON, I always aimed to make the interaction between the user and the database as trouble-free as possible. One important feature of this was to minimise the amount of keyboard input necessary to carry out any database operations. The interface was to be windowed, presenting the data simply. The mouse was always going to be the main input device, with the interface allowing all activities to be carried out using it.

4.2.3 Reasons for Development of GRIFON

GRIFON was developed as a test of the feasibility of a graphical user-interface to an object-oriented database. Interfaces like SNAP and ISIS were built on semantic databases while CASE*Designer was developed on a relational database. Object-oriented databases up to now have been used purely as backend storage facilities for large applications. They do contain a lot of semantic power, being able to represent the real world problem space accurately. The data in them is usually highly complex and can be interpreted in a number of different ways. GRIFON aims to demonstrate the practicality of representing these various interpretations in a graphical manner. Simplicity is all important. The saying "*simple ideas are the best*" is true to a large extent where human-computer interaction is concerned. Graphical representation of the database is all important in helping the user visualise the data. It is important that consistency can be maintained by allowing the user to interact with these representations.

GRIFON aims to provide simple methods of extracting information from the graphical database representations and allow data to be added in a straight forward manner. It serves as a test-bed to see if a database could be extended, both in schema and data terms through a graphical interface.

The underlying database structure should be exhibited to its full extent. The user should be able to visualise the structure of the data. Because of the increase in the semantic power of the underlying OO database, through the graphical interface, the user is able to get a better grasp of the database representation of the problem and of the problem itself. To this extent GRIFON can serve as a tutorial system about OO databases and how they can be used to model real world situations. Through an accurate realistic picture of the database structure, users and students can gain a better understanding of concepts like inheritance and class-composition and how objects relates to these.

One major worrying factor for users of any system is having to learn a command language or query language. One aim of GRIFON was to attempt to simplify the use of the provided ONTOS query language. In keeping with the overall graphical feeling of the interface, a simple query construction facility was provided. However, to allow flexibility for the experienced ONTOS user, queries can be constructed in textual manner from outside GRIFON and be subsequently executed through GRIFON.

4.2.4 Features provided

The features provided in GRIFON can broadly be broken down into four categories:

Schema Representation

Schema Extension

Data Creation

Query Creation

Consistency is the key word where any user interface is concerned and this is the case with this system. GRIFON offers a number of different and diverse features but they are all presented to the user in a consistent manner. Consistency makes the user feel comfortable with the system. Consistency in applications like Appel's MacWrite, MacPaint and MacDraw on the Apple Macintosh computer has meant that users familiar with one application have little difficulty learning another. Users capable with one of the applications would tend to adapt better to one of the other applications sporting the same interface standards and techniques, than to a totally different application doing the same job [PET89].

4.2.5 Limitations in the development

As with any interface type system which relies on the facilities of the underlying software, GRIFON's facilities were very much determined by the features offered by ONTOS. ONTOS is an object-database. In the current release, it supports classes, objects, and methods. However, strict limitations are cast over the provision of methods. C++ functions can only be associated with classes before the class is compiled and added into the database. This cannot be done at run-time. It must be carried out at compile time. In this case, this would be when the interface is being compiled. Some minor support is provided for dynamic association [1] of methods with classes. Here again, the functions must already be present in the database. They will not be class specific functions but will be general C++ functions which just happen to be associated with the chosen class. With this form of binding of functions to classes, the database acts as no more than a repository of C++ functions.

Because of the nature of an interface, it is necessary to provide for dynamic operations on the database. Such operations as class and object creation require that the database be modifiable at run-time. As no satisfactory method is provided for the association and binding of methods

[1] Dynamic Association refers to the binding of a C++ function to a class at run-time. Normally functions and methods would be bound to new classes before the application would be compiled, however, this would not be appropriate for a system like GRIFON.

to classes at run-time, I have decided to concentrate solely on the attribute side of the database and in that light GRIFON is an interface to a semantic-type object-oriented database with no use being made of class methods.

If methods could dynamically be created and bound to new classes, I would envisage GRIFON having a text editor for creation of the methods. In such a system, these methods would need to be compiled during the execution of GRIFON, before being written to the database. The facility for such database manipulation may be provided in a later release of ONTOS. More on the features of ONTOS will be outlined in the next chapter.

4.2.6 The Schema Representation

GRIFON is built around the notion of a schema hierarchy [2]. The hierarchy is as described in chapter 2, indicating the inheritance tree of classes in the database. Every operation on the database starts from a representation of the hierarchy. According to Bryce [BRY86], a good graphical database schema access package should, among other things, facilitate returning to visually flexible, static representations of the schema components. SNAP implements this idea. A system which implements this strategy makes it increasingly difficult for the user to feel lost while using the system.

In GRIFON, as I said, all operations start from the familiar database hierarchy representation. The system provides three forms of the hierarchy which can be viewed and manipulated by the user :

The Class Inheritance Hierarchy.
The Class-Instance Hierarchy.
The Class Composition Hierarchy.

4.2.6.1 Conformity of representation and operation

No matter which hierarchy is being displayed on the screen, a number of items remain constant.

- **The hierarchy is not bound by the size of the window in which it is displayed.**

[2] Schema Hierarchy refers to the structure of the database. OODBs tend to be hierarchical in structure with classes acting as sub-classes of others.

On a large database schema, only a portion of the complete hierarchy may be displayed in the window at a single time, due to hardware restrictions. However, scroll bars are included at the side and bottom of the window to allow movement around the hierarchy. Clicking [3] once on the arrows at the end of the scroll bars or dragging [4] the slider along the scroll bar will move the hierarchy in the appropriate direction. This, in the case of a large hierarchy, may result in a new portion of the screen being displayed. Figure 5.2 gives a diagram of the structure of a window in the system.

- **More information about a class can be displayed by selecting a class node from a hierarchy.**
  The information displayed is determined by the hierarchy on which the selection was made. For example, choosing a particular node on the Class-Inheritance Hierarchy will give more information about the construction of the chosen class.

- **All operations to be carried out on the database start from a similar position.**
  Creating a new class starts from the class inheritance hierarchy where a class is selected to act as the super-class for the new one. The same applies to creating instances etc.. They all start from a hierarchy display.

- **An information window is included at the top of the main application window, just below the menu list.**
  This window is used to inform the user of relevant information to the operation currently being carried out. For example, when hierarchies are displayed, their names are displayed in this window. When the user is to enter some details, this window gives information regarding what input the computer is requesting.

[3] Clicking, with reference to selecting items off the screen, involves the use of the mouse to position the screen pointer over a particular item on the screen. The left button on the mouse is pressed once and this has the effect of selecting the operation associated with the screen icon or button.

[4] Dragging refers to an operation of moving the mouse with the left button held down. This is often used in windowing environments for moving items around the screen or by means of a scrollbar moving the displayed information in a particular direction.

4.2.6.2 The Class Inheritance Hierarchy

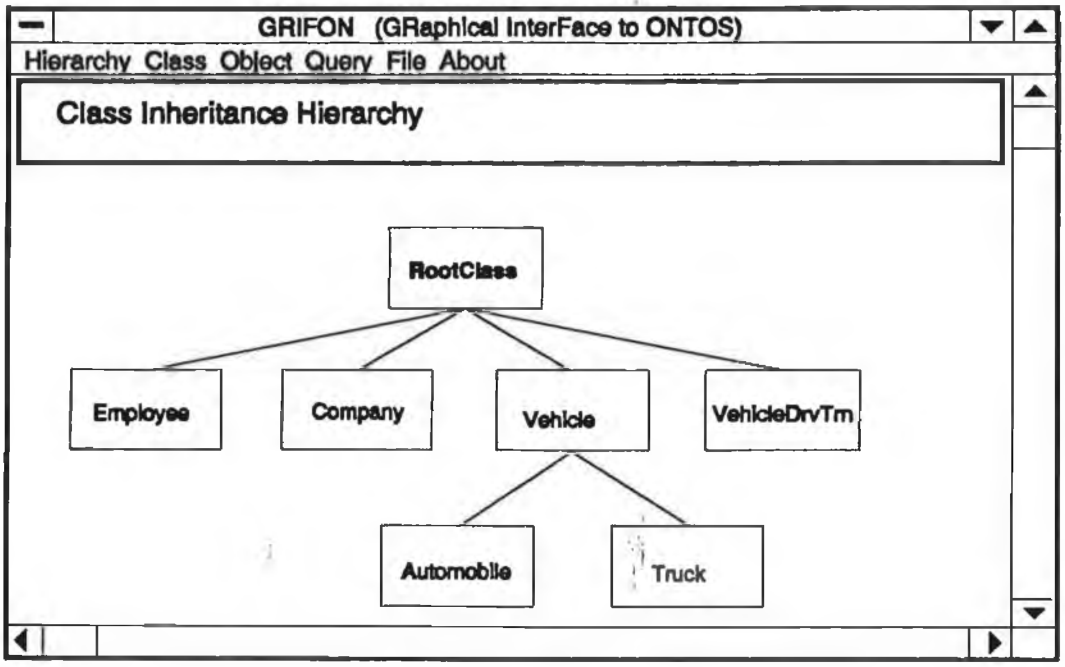

GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Class Inheritance Hierarchy
RootClass
Employee
Company
Vehicle
VehicleDrvTrn
Automobile
Truck

Figure 4.1 - Class-Inheritance Hierarchy Representation.

The class inheritance hierarchy is a representation of the class-subclass relationship in the database. As discussed in chapter 2, a database schema can support single or multiple inheritance. ONTOS only supports single inheritance in the current release, so the class inheritance hierarchy is a tree structure. As can be seen in figure 4.1, the hierarchy is simple in its representation. The graphical positioning of the nodes and the inter-connection of them serves to indicate the class-subclass relationships. Only the name of each class is displayed in each node. For consistency in the implementation of the system, and to keep the schema as a tree structure, GRIFON creates an artificial root node called **RootClass**. New classes created, ultimately are traced back to this class. This is a barren class in that it has no attributes or methods associated with it. It serves solely as an anchor point for the hierarchy.

4.2.6.2.1 Practicality of Representation

The class inheritance hierarchy is the simplest picture of the database. This is equivalent to a list of the relation names which might be displayed in a relational database system. However, where a list of relations just gives the names of the tables in the database, the class hierarchy gives much more information.

- **It is a graphical picture of the database.**
  The user has no difficulty conceptualising the database when it is in this format. It is simple to see the database structure.

- **It is a graphical picture of the problem space represented by the database.**
  Not only is the hierarchy a graphical picture of the database, but it is also a graphical picture of the real-world situation which it represents. A **Car** and a **Truck** are both types of **Vehicle**, in the real world environment. This factor is clearly represented in the hierarchy with **Car** and **Truck** both inheriting all the features associated with a **Vehicle.**

- **It represents a logical breakdown of the problem space.**
  The hierarchy is a representation of the breakdown of the entities in the problem with their general features being extracted and maintained as particular classes and a specialisation of data in classes as one moves down through the hierarchy.

  For example, in the class inheritance hierarchy displayed in Figure 4.1, the overall problem represented would appear to be the Vehicle manufacturing business. However it is clear to see that Vehicles can be sub-divided into Automobiles and Trucks, which in turn can be sub-divided further. This sub-division of the problem space is carried out in the creation of any database, but it is very difficult to represent in a coherent manner.

4.2.6.3 The Class-Instance Hierarchy

The class-inheritance hierarchy is essentially a picture of the database at a schema level. It deals purely with the structure of the database. In contrast to this, the Inheritance Hierarchy is concerned with the data in the database. However, the representation of the **Class-Inheritance Hierarchy** in GRIFON is a combination of schema and data representation. The hierarchy, as represented in figure 4.2, is essentially the same in appearance as the class-inheritance hierarchy. But in addition to this, classes which have instances created of them in the database are highlighted, and the number of objects of this class is displayed in the class node, under the class name. The class nodes are normally displayed in a blue-check colour. However, any classes with objects, are highlighted in red-check, and, as mentioned above, the

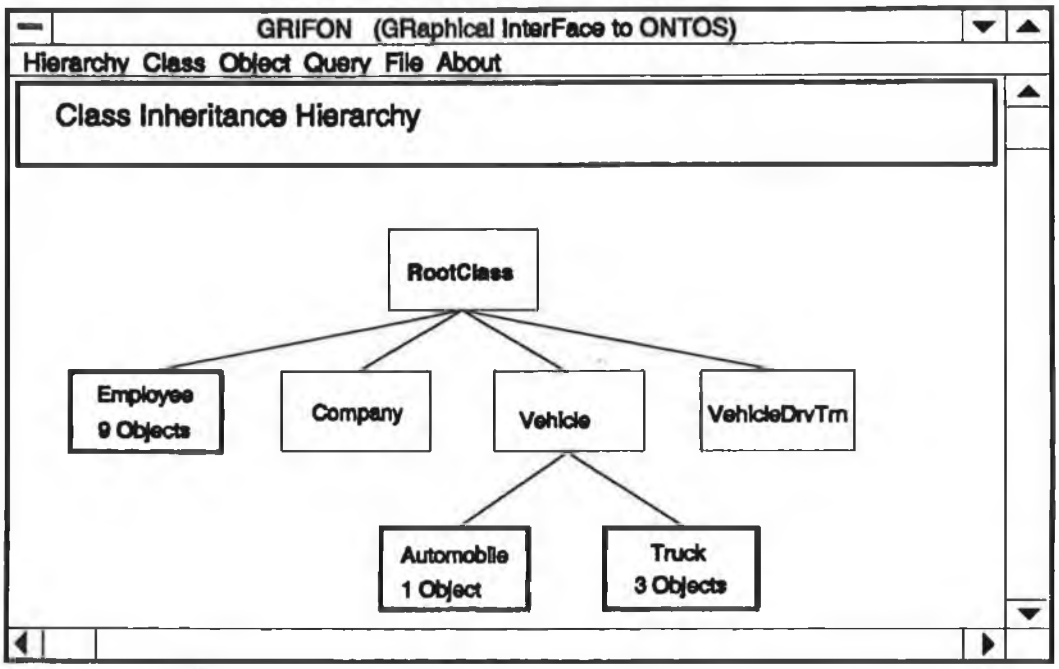

GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Class Inheritance Hierarchy
RootClass
Employee
9 Objects
Company
Vehicle
VehicleDrvTrn
Automobile
1 Object
Truck
3 Objects

Figure 4.2 - Class-Instance Hierarchy Representation.

actual number of objects which they have is displayed. In figure 4.2 class Employee has 9 objects, Automobile 1 object and Truck 3 objects.

4.2.6.3.1 Practicality of Representation

Although the screen display is just an extended version of the class inheritance hierarchy, there are a number of important extra benefits accrued from the inclusion of the number of objects in classes.

- **It gives an indication of the database size.**
  The information displayed in the class nodes allows a user or the database designer to get a feeling for the amount of data in the database. The number of individual entities in the database is the sum of all the class object counts.

- **It gives valuable statistics about the data.**
  In a relational database, to find the number of employees in the employee relation would require issuing a query applying the *count* function to count the number of tuples in the table. In GRIFON, the class-instance hierarchy displays statistics on individual classes. For example, in our sample Vehicle business database, a user can glance at the hierarchy and see that there are 9 employees in the industry (somewhat

unrealistic !), 1 model of Automobile and 3 models of Trucks. So relatively detailed knowledge about the database can be gleaned without the user ever needing to construct a query.

- **It presents important schema management information.**
  In any large database system, the database administrator (DBA) needs to try to optimise the usage of the storage facilities provided, yet servicing the users' requirements. Information about the current usage of the schema can provide valuable information to the DBA to help him/her optimise the secondary storage usage. The class-instance hierarchy provides such information. It gives an accurate picture of where the data is in the database. It allows the DBA to see if certain sub-trees of the database schema are redundant. If they are, few or no objects will have been created of the classes in these sub-trees. This will help him/her prune the database. Such a procedure will simplify the database structure, and improve storage and access times to the data. In our example, classes Company and VehicleDrvTrn have no objects. The DBA's attention will be drawn to these, through this hierarchy. He can then decide if they are unnecessary and remove them from the database, thus compacting the schema and simplifying the hierarchy, if required. This operation would only be done in a mature database when the overall picture of the database usage becomes clear.

4.2.6.4 The Class-Composition Hierarchy

One of the features of object-oriented databases which gives them their power is their ability to model complex information. Relational databases tend to deal in simple, atomic data where as OO databases can handle structured information. As outlined in chapter 2, a class consists of attributes. These attributes can be atomic, i.e. numeric, character, etc., or they can be objects. Their domains can be other classes in the database. This idea of attributes being objects of other classes facilitates complex modelling. Applying these ideas, GRIFON facilitates yet another view of the database schema, this view being orthogonal to the inheritance hierarchy. The **Class-Composition Hierarchy** represents the links between a selected class in the database and other classes which are domains of attributes in the selected one. As with all operations, the processing and display of the class-composition hierarchy commences from a familiar view. This familiarity being the class-inheritance hierarchy. When this is displayed, the user is instructed to select [5] any class node in the hierarchy whose class composition he wishes to display. The external class-composition of this class will be displayed.

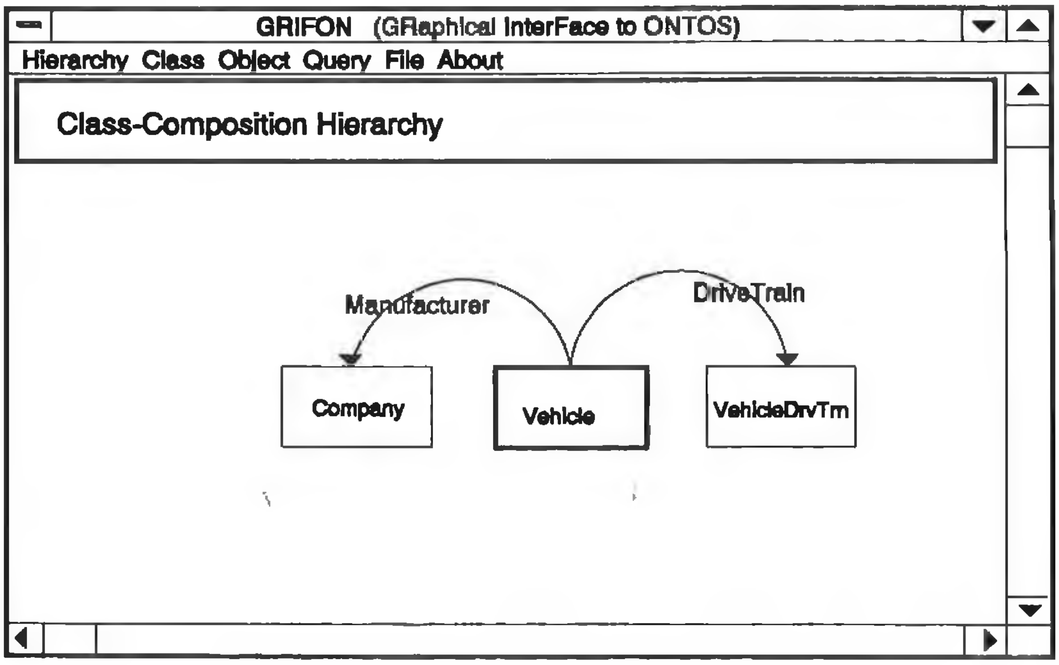

GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Class-Composition Hierarchy
Manufacturer
DriveTrain
Company
Vehicle
VehicleDrvTrn

Figure 4.3 - Class-Composition Hierarchy for class Vehicle.

Figure 4.3 illustrates this point. Class Vehicle has been selected. It is composed of an object of class Company which is the Manufacturer of the vehicle and an object of class DriveTrain which is the VehicleDrvTrn attribute of Vehicle.

Strictly the class-composition hierarchy is a representation of the links between all classes in the database, highlighting what classes are made up of other classes. This view, however, would tend to be awkward for large schemas and would inhibit the extraction of meaningful information from the database, so I have decided to restrict the class-composition display to that of a selected class.

[5] Selecting classes from the hierarchy is carried out using the mouse. The mouse-pointer on the screen is positioned over the required class-node, ie. inside the node's rectangle. Then the left button on the mouse is pressed once.

4.2.6.4.1 Practicality of Representation

The Class-instance hierarchy displays a data based view of the database, illustrating the usage of the database, and its make-up in terms of how the classes are being used. The class-composition hierarchy, like in class-inheritance hierarchy, gives the user or database designer more information about the structure of the database.

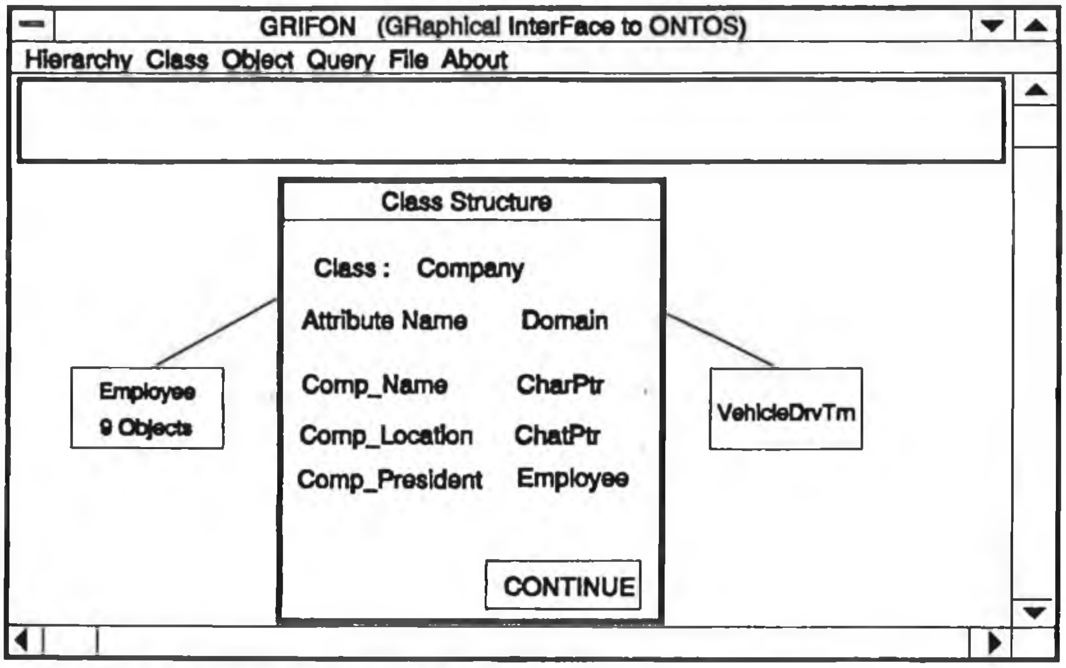
- **It demonstrates the effect one class has on another.**
  For a database administrator pruning the database, deleting classes or moving classes etc., the effects on the database as a whole would need to be considered when changing one class. In the figure 4.3, any changes to the Company class might affect the Vehicle class. So based on the display, it can be seen that removal of the Company class from the schema would require that the construction of the Vehicle class be modified to no longer refer to it. So the inherent inter-linking of the database classes is made explicit through this representation.

- **It gives a clear and concise view of a class structure.**
  The user is given a clear and concise picture of the structure of a particular class. This picture only refers to the classes which have domains which are external to the class, i.e. they are not atomic. Class Vehicle consists of a company as manufacturer. This fact is represented graphically. Clearly, to now get an expanded picture of the structure of Vehicle, one might look at the composition of the Company class. So instead of looking at a textual list of attributes of Vehicle and then getting a list of the attributes of Company, the system explicitly prompts the user as to which other classes to examine.

4.2.6.5 More Information on the hierarchies

Selecting a class node from the class-inheritance hierarchy, when choosing the class-composition hierarchy option, results in the class-composition for the selected class being displayed. However, similar extra information is available from the other two hierarchies, by selecting a particular class.

In the class-inheritance hierarchy, selecting a class node from the hierarchy using the mouse, by clicking once on it, will result in a window being displayed which contains a list of the attributes making up this class, including their name and their domain. Figure 4.4 shows the display if class Company was chosen off the hierarchy. This display is useful for displaying the structure of a class. It is simple and easy to use and as with the other facilities does not require the issue of any queries.


GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Employee
9 Objects
VehicleDrvTrn
CONTINUE

Class Structure

Class : Company

Attribute Name	Domain
Comp_Name	CharPtr
Comp_Location	ChatPtr
Comp_President	Employee

Figure 4.4 - Information on selected Class.

As outlined above, the class-instance hierarchy displays the inheritance hierarchy but each class with objects is highlighted and the number of objects displayed. If the user selects one of the class nodes with the mouse, the information window at the top of the main application workspace displays a message informing the user in a longer form of the number of objects that the selected class has.

The provision of more information about the classes or objects in the database, through the selection of classes in the hierarchy, using the mouse, is intuitively pleasing and is very much in keeping with the OO idea of moving from the general to the specific, as is done in the class-inheritance hierarchy.

4.2.7 Database Operations

The core element of GRIFON is the representation of the hierarchy. As with any database system, the user or administrator will want to add data, and access data in the database. As mentioned above, the aim of GRIFON was to provide a simple, user-friendly manner for interacting with the database. Consistency is very important to user-friendliness. Therefore the manner in which data is added to the database and accessed is consistent. As with all operations, creation or addition of data starts from a familiar hierarchy display. The hierarchy displayed is relevant to the selected operation. So for example, if something is to be done to the schema of the database, the operation would start from the class-inheritance hierarchy. If however, data is to be added to, or accessed, then the class-instance hierarchy is displayed.

4.2.7.1 Creating a New Class

As with the modification of the structure of any database system, OO, relational or otherwise, some initial information needs to be provided regarding the parameters affected and the data required to be entered. In the creation of a new class, the user needs to provide the following information :

(i) The name of the class which is to be the super-class or parent for the new one.

(ii) The attributes of the new class.
- their names and their domains.

(iii) Information on the attributes to be used in the creation of new objects.

Point (iii) above may seem unnecessary, as there would be no point in creating attributes for a class unless they were to be used in the creation of a new object. However, in object-oriented systems, data is inherited from super-classes. It may be possible that a class may be created with the intention of it being used as a super-class for other classes. It may never have objects created of it, itself. All of these attributes may not necessarily be required to be used in the creation of new objects in the database. So to facilitate the presentation of the required attributes in the instantiation of a class, the user will need to specify the attributes to be used, selected from a list of those of the new class and those inherited, in the creation of future new objects of the new class.

4.2.7.1.1 Procedure Involved

To create a class in GRIFON, select the **Create** option from the **Class** entry on the menubar displayed across the top of the application window[6]. This results in the class-inheritance hierarchy being displayed. The information window at the top of the screen will instruct the user to select a class from the hierarchy to act as the super-class for the new one. When the user selects the super-class, a small window appears asking the user to enter the name for the new class. This screen display is shown in figure 4.5.

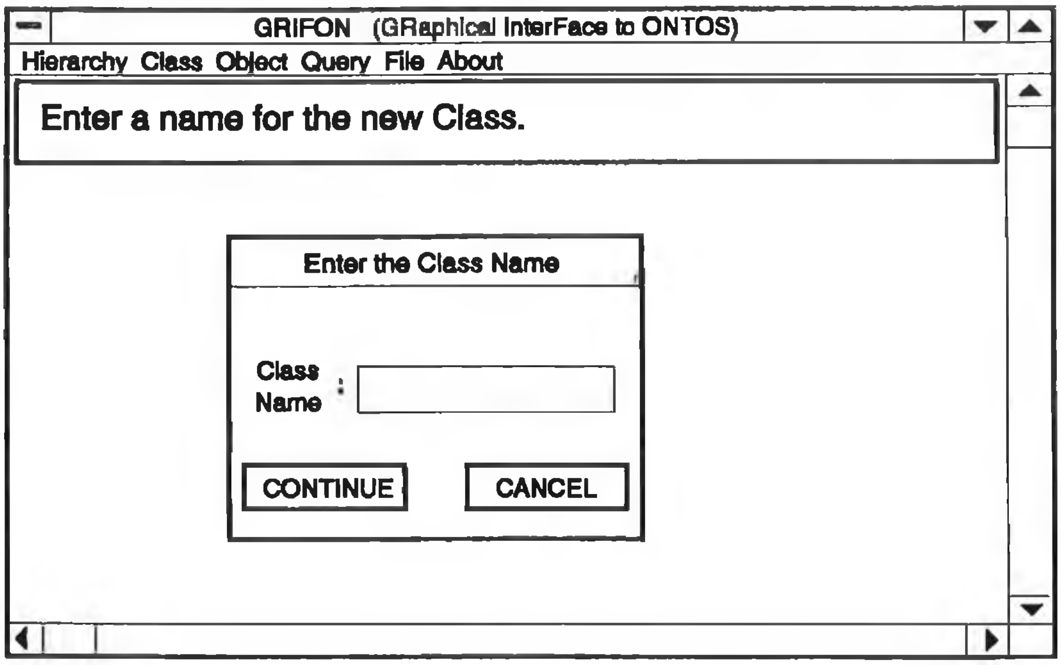

GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Enter a name for the new Class.
Enter the Class Name
Class Name :
CONTINUE
CANCEL

Figure 4.5 - New Class Name Entry Window.

Entering the new class name and pressing <CR> will display the *attribute-entry window* as shown in figure 4.6. This window allows the user to enter the names and domains of the new attributes to be associated with the new class. In addition to this the user can specify a number of other features associated with this attribute. These include choosing whether a value must be entered for this attribute, when a new object is created. The user can also specify if the attribute is to be unique. This means that no two objects of this class can have the same value for this attribute. If the attribute is to have a text domain, the option is provided to have the entry field as a multiple-line text entry field when the value is being

[6] The menubar displays the list of available options or sub-menus available to the user. Clicking once on a menu-bar entry will either execute some operation or display a pull-down menu with more entries. To select one of these, click once on the required one. It will become inverted for a second and then the operation associated with it will be executed.

entered for this attribute. These three extra features are selected by means of check-boxes [7].

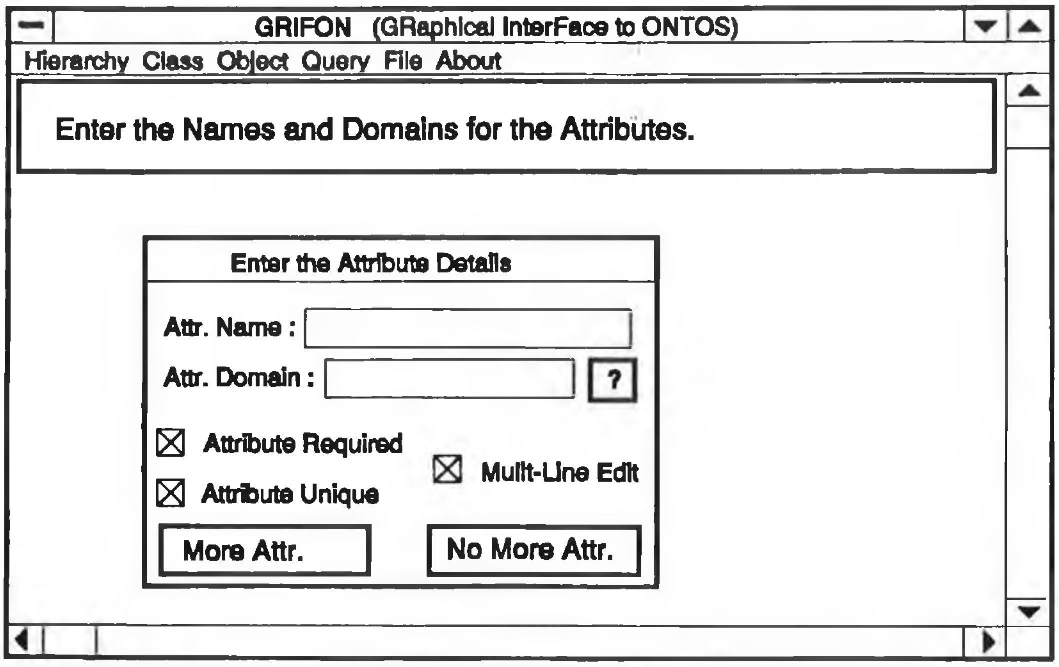

GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Enter the Names and Domains for the Attributes.
Enter the Attribute Details
Attr. Name :
Attr. Domain :
?
Attribute Required
Attribute Unique
Multi-Line Edit
More Attr.
No More Attr.

Figure 4.6 - Attribute-Entry Window.

To simplify the entry of the attribute's details, there is a button beside the domain entry line. If this button is selected using the mouse, a list of possible domains available is displayed in a separate window. Clicking twice on the chosen one in the list will return to the attribute-entry screen with the selected domain filled in. This facility ensures that the domain chosen by the user will be a valid. It also removes the need for the user to memorise all of the available domains. In addition, the user is not required to use the keyboard - the source of many errors.

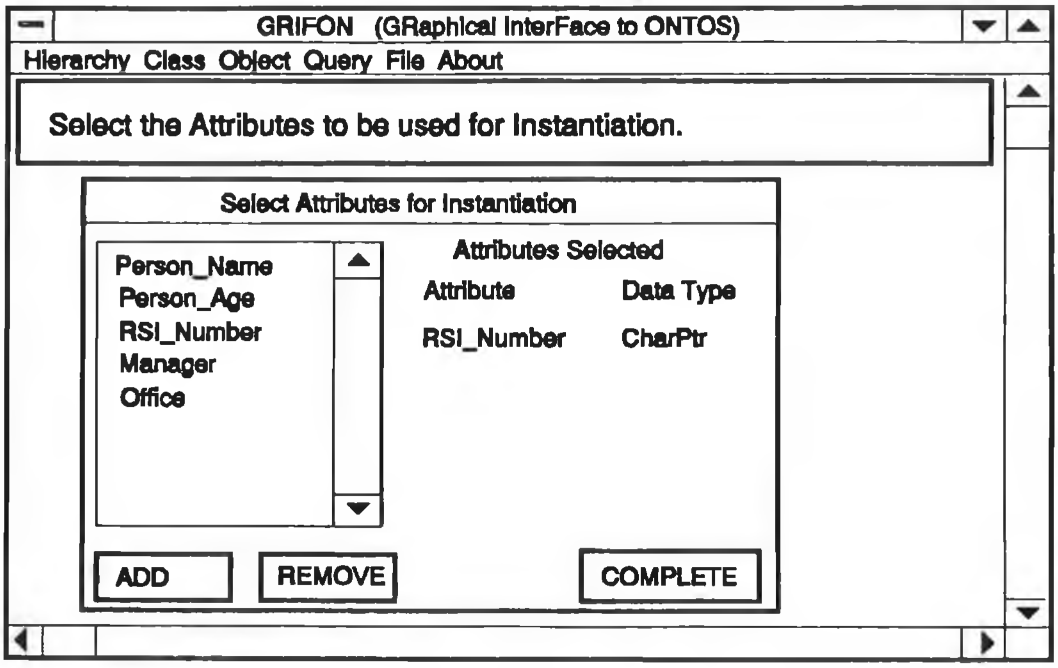
Once the details have been filled in for this attribute, pressing the **More Attr.** button [8] will re-display the attribute-entry window, but this time it will be blank. This can now be filled in, in the same manner as before, with the details of the next attribute.

[7] A Check-box is similar to a button. Clicking inside it with the mouse, has the effect of selecting it, or switching it on, if it is de-selected or off. There will be a 'x' in the box if the features is selected. It will be blank otherwise.

[8] Pressing a button in GRIFON refers to using the mouse to select a button on the screen. The mouse is used to move the screen pointer over the button icon on the screen. When the left mouse button is pressed, the screen button is pressed. The icon representation on the screen, temporarily changes to indicate it's being pressed. The operation associated with it is then executed.

When all the attributes have been entered, pressing the **No More Attr.** button will result in the *Instantiation-List* window being displayed.

This window allows the selection of the attributes to be used in the creation of new objects of this class, in the future. The list of attributes to choose from includes both the new ones, just defined, and those which are inherited from superclasses. Figure 4.7 shows the window on which this attribute selection is carried out.


GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Select Attributes for Instantiation
ADD
REMOVE
COMPLETE

Select the Attributes to be used for Instantiation.

Person_Name
Person_Age
RSI_Number
Manager
Office

Attributes Selected

Attribute	Data Type
RSI_Number	CharPtr

Figure 4.7 - Select Attributes for Object Creation.

The user selects those attributes from the list which he feels will be required when an object of this class is being created. This procedure simply involves selecting an attribute from the list on the left, by either *clicking twice*[9] on the chosen one, or clicking once on it to select it, and subsequently pressing the **ADD** button at the bottom of the list to add it to the selected

[9] Clicking twice on a particular item, icon, list-entry or button, involves positioning the mouse pointer over the required item and pressing the left button on the mouse twice, very quickly in succession. This is known as **double-clicking.** It is often used as an alternative, faster method of selecting an item, rather than having to select the item and then click once on a selection button.

attribute list [10]. When an attribute is selected, in either of the ways, it is added to the list on the right of the window. Its domain is also displayed. This selection of attributes can be reversed by selecting an attribute from the list on the left which has already been added to the list on the right, by clicking once on it, and pressing the **REMOVE** button. This has the effect of removing it from the list on the right and therefore from the list of attributes to be used when creating a new object.

Once all the required attributes have been selected, pressing the **COMPLETE** button will close this window and create the new class in the database.

Message windows will be displayed on the screen during the creation to inform the user what the computer is currently doing. With the new class added to the database, the hierarchy is re-calculated to include this, and finally to complete the addition procedure, the new, updated, hierarchy is displayed.

To the user, this approach to class creation minimises the amount of typing which he needs to do. Anywhere a set number of possibilities is available to an entry, i.e.,

- Entering the Attribute Domain and
- Selecting attributes for instantiation,

a list of the available options is displayed. This speeds up the creation process for the user and similarly requires less memorisation by him.

All these factors add significantly to the usability and user-friendliness of the interface and therefore the database.

4.2.7.2 Creating a New Object

The creation of a new object in the database is a simple operation from the user's viewpoint. However, it takes information created by other facilities in the system and subsequently is potentially the most difficult from a programmatic viewpoint.

[10] In GRIFON, an item can be selected from a list in two possible ways. The mouse pointer can be positioned over the item to be selected and the mouse button can be double-clicked. Alternatively, with the mouse pointer positioned over the item to select, press the button once. This provisionally selects the entry. Then press the Select or ADD button under the list, with the mouse.

In keeping with the interface in general, this procedure is simple. It makes any choices which the user must make as simple and trouble-free as possible.

4.2.7.2.1 Procedure Involved

In non-database terms, the creation of a new object involves specifying the class of object being created (ie. its class) and then entering the required data. In GRIFON, this is exactly how the creation is carried out.

When the option to **Create an Object** is selected from the **Object** menu on the menu-bar, line on the screen, the class-instance hierarchy is displayed. From this the user selects a class from the hierarchy which is to act as the domain for the new object. As with all selection procedures, this involves clicking on the appropriate class-node with the mouse. This has specified which class is going to have an object created.

A window is displayed with the list of attributes to have data entered. This window is shown in figure 4.8. The first field to fill in is the **Identifier**.

Although object-identifiers in OODBs are value independent unique identifiers for all objects, ONTOS allows the user to enter his own unique identifier. Although not a key-field, as in relational databases, it does act as a unique identifier on which we can access objects in the database.

The other fields displayed on this screen, for entry, were determined by the attribute-selection phase of the class creation procedure as explained above. The attributes which were selected for use in creating new objects of the class being created are those which are now being displayed, prompting the user for entry.

The selection of the attributes in the class-creation phase has facilitated the creation of a data-entry form automatically for this class. So any subsequent objects of this class being created will have their data entered through this entry-form.

Validation is carried out on the entry fields associated with the attributes. So for example, if the domain of the Person_Age field is an integer, the user is prevented from entering any character other than those in the range '0' to '9'. The same applies to real numbers etc..

Once the data has been entered in all of the entry-fields, pressing the **INSTANTIATE** button using the mouse will create a new object in the database. If the values of any attributes which were previously defined to be unique are found to exist on the database in other objects of this class, then this new object will not be created and the user will be informed of this fact.

Again with this creation option, once the new object is created, the hierarchy is re-calculated to update the object count on the class-instance hierarchy for the addition of the new object. As with the class-creation option, the last operation to be carried out is the re-displaying of the updated class-instance hierarchy.

Here again, the amount of textual entry by the user is minimised. The selection of the class to instantiate is done through the direct manipulation process. Once the object has been created successfully, the hierarchy is updated to represent this fact with the object count on that class being incremented.

4.2.7.3 Displaying Object Details

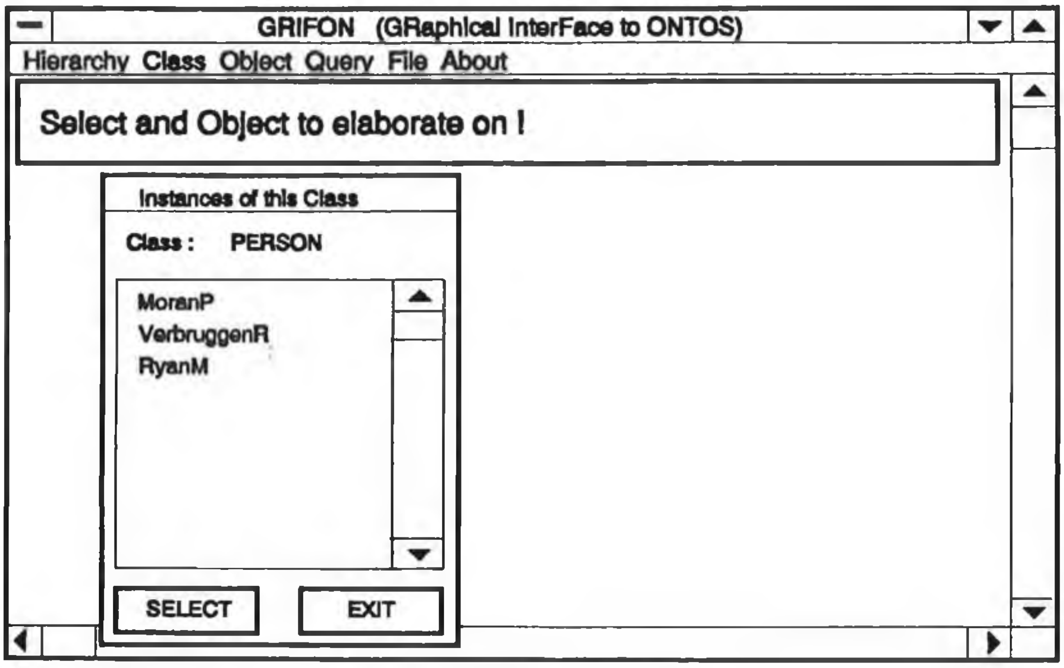
In any database system it is inconvenient to have to construct and execute a query to get access to the details of a single entity in the database. PC-based database packages like Dbase III [ASH85], have provided browse and edit facilities for listing all the records for a particular relation. Many of the larger mini-computers and workstation based databases do not provide such a facility. To access this information requires the creation of a query, with the user being expected to know the name of the relation to be queried and the fields or attributes being displayed.

GRIFON, provides a query facility, but in addition to this the user can access the object details of a particular class, solely through using the mouse. The user need never touch the keyboard.

4.2.7.3.1 Procedure Involved

The menu option for displaying information on objects is the last option on the **Object** sub-menu on the applications window menu-bar. When the user selects this option, using the mouse, the class-instance hierarchy is displayed. This is for conformity with all the other options.

The user must specify the class whose objects he wishes to have displayed. As with the other options, this is done by clicking on the appropriate class node on the hierarchy. Figure 4.8 shows the screen display. This list-box window lists all of the object identifiers of the objects of this class.


GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Instances of this Class
SELECT
EXIT

Select and Object to elaborate on !

Class : PERSON

MoranP
VerbruggenR
RyanM

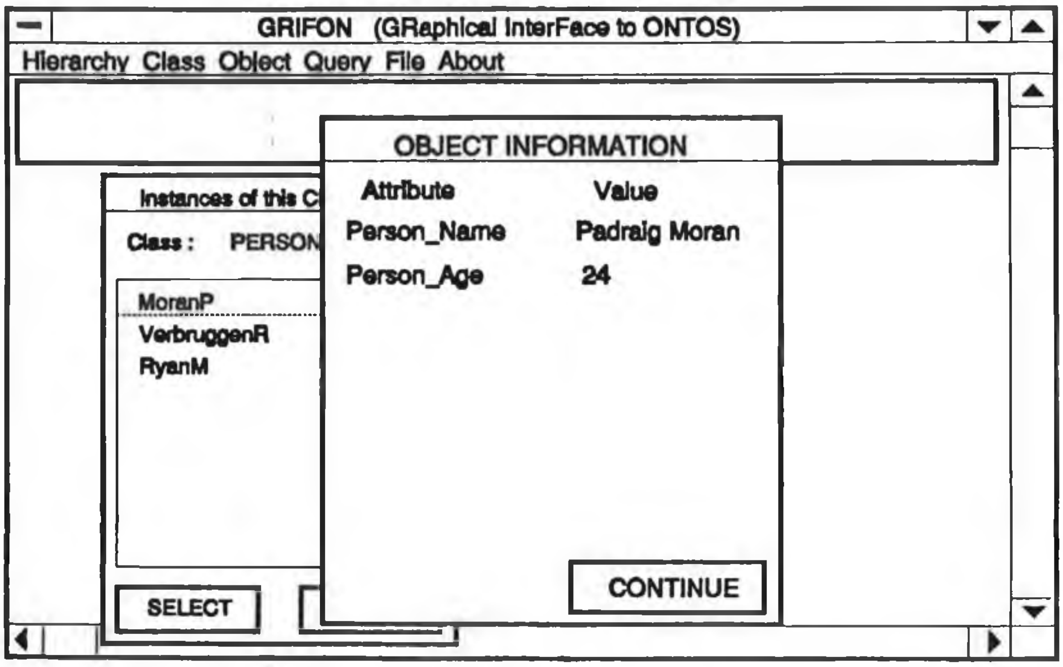
Figure 4.8 - List of objects for selected class.

To get more information on one particular object, select it from the list.

Selecting an entry from this list will retrieve the information in the database associated with this object. This will be displayed in another window, as shown in figure 4.9.

The object information window lists the attributes associated with this class with the values filled in for the attributes.

From here, the user can return to the list of object identifiers and select another object to get the details on, or return from this option completely and re-display the class instance hierarchy.



Attribute	Value
Person_Name	Padraig Moran
Person_Age	24

Figure 4.9 - Information on selected object.

4.2.7.3.2 Practicality

There are a number of features associated with this form of information presentation which give it advantages over existing systems.

- **It gives clear information about the data.**
  The user can simply select the class required. He is presented with a list of objects of this class, from which he selects one to elaborate on. This process can be repeated as required. This approach is simple. It presents the information in the database in a clear and as far as possible, concise manner. The user is required to know the minimum of operations and commands to get at the information in the database.

- **There is no keyboard interaction required.**
  The data associated with any object in the database can be retrieved by two or three clicks on the left mouse button. No use is made of the keyboard whatsoever. The keyboard is often regarded with trepidation by users unfamiliar with computers. With over eighty keys on even the most basic keyboard, unfamiliar users regard them as eighty possible sources of mistakes.

- **There is no need to learn a query language.**
  For complex and conditional access of data in the database, the user would need to become familiar with the basic concepts of query construction, whether in the interactive query constructor or through a textual editor. However, for many casual users of data systems, they just want to be able to access the data as simply as possible. The idea of getting a list of employees, and then selecting one on which to get more information is familiar to most users, even those unfamiliar with computer databases. This is the manner in which it might be carried out manually, leafing through a list.

4.2.8 Queries in GRIFON.

A database is useless unless the data stored in it can be accessed in some intelligent manner. The facility for displaying details of the objects of a particular class is very useful for quick accesses to the database, where the user is willing to sift through information to get at the required data.

GRIFON, provides a number of different ways in which the user can execute a query. Additionally, a query can be constructed in a number of different ways.

Users of different levels of experience and ability prefer different approaches to query creation and execution. GRIFON tries to provide for these.

4.2.8.1 Interactive Query Construction

ONTOS provides OSQL, an object-oriented structured query language, for query construction. This is a textual query language and queries are ultimately issued in this format against the database. Like SQL, it does have a structured syntax and requires a user to be relatively experienced in its use to be able to create complex queries.

No interface can replace OSQL, as this is part of the database system. However, GRIFON attempts to put a nicer face on OSQL for the inexperienced user. In keeping with the other parts of the interface, it tries to minimise the amount of keyboard entry required. The construction of a query is done through the selection of attributes from lists, pressing buttons to state conditions and the entry of text where required. The actual OSQL query is also displayed while it is being constructed, to help the user get a feel for the syntax and form of queries.

4.2.8.1.1 Process involved in interactive query construction.

The query creation process starts from the class-instance hierarchy. This displays the hierarchy illustrating what classes have objects, giving the user a slight prompt as to which classes are available on which to base the query. Currently GRIFON allows a query to be executed against a single class in the database. This is for simplicity to show how a query can be created interactively.

To create the query the user must select a class. Having done this the *query creation* screen is displayed. This is a dialog window which facilitates the creation of the query by means of selecting items from a list and pressing buttons. Figure 4.10 shows the query creation window.

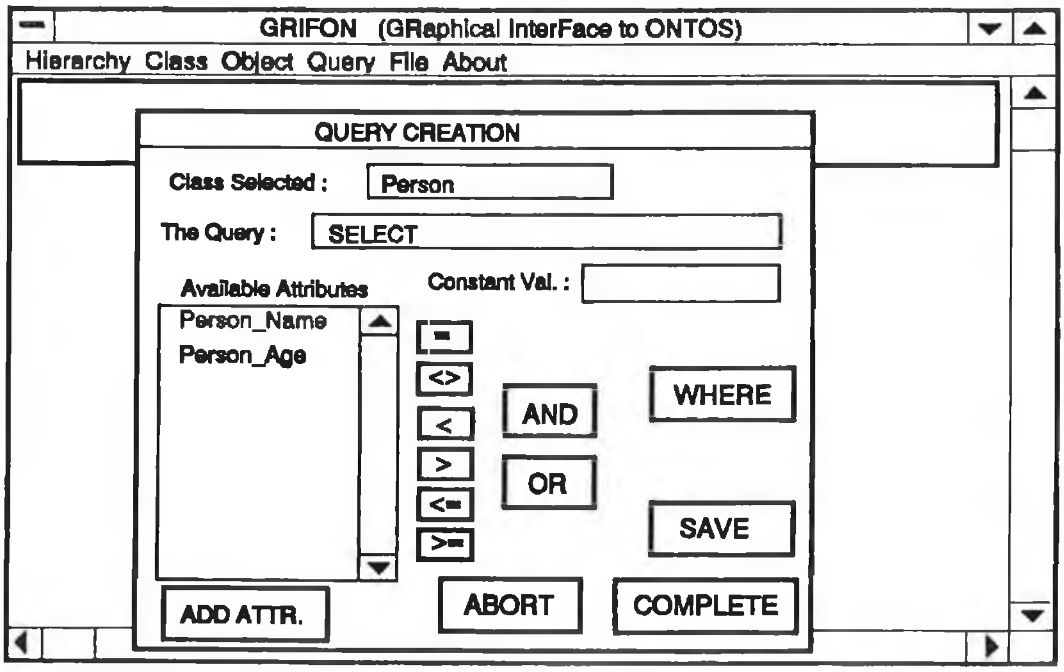


Figure 4.10 - Query Creation Window.

The structure of the query screen is as follows :

- **Edit Lines**
  Edit lines at the top of the screen give the name of the class being queried and the text of the query itself. This is changed as the query is created. A constant field is also included where the user is prompted to enter numbers or text where appropriate in the query construction.

- **Attribute List**
  The attribute-list gives all the available attributes whose values can be returned in the query. The user can select these from the list to add to the query, when required.

- **Query Operator Buttons**
  These include relational operator buttons and logical operator buttons. If these are pressed during the query construction, they result in the appropriate operator being added to the query. This can be seen in the text in the query edit line on the screen. In addition a **Where** button is used to indicate when the selection of attributes to query is complete and the creation of the condition is to start. In addition an **ADD ATTR.** button is provided to allow the user to select attributes from the list.

- **Query Control Buttons**
  These are general buttons which allow the user control the query construction. Buttons are added to allow it to be saved, to be executed (**COMPLETE** button) and to abort the complete query creation process and return to the hierarchy.

The simplest way to understand how the query creation process works is to follow through the process with a sample query.

Example Query Session
As mentioned above GRIFON allows a query to be executed against a single class in the database.

Consider the following sample query request :
Get the names and ages of all the people in the database who are greater than 24.

Based on this requirement we can see that the following are the case :

- Class being queried is **Person.**
- Name and Age attributes are required.
- Condition is Age > 24.

Now based on this information we can proceed with the creation of the query.

Initially select the required class from the hierarchy. For this query, select the **Person** class from the hierarchy by clicking once on it. This will display the query creation window. Its initial form will be as shown in figure 4.10. The name of the class which is being queried is displayed at the top and the attributes available for this class are listed in the list box on the left-hand side of the window.

Having selected the class, now select the attributes to be queried, from the list. As mentioned earlier this can be done by clicking twice on the attribute or once on it and once on the **ADD ATTR.** button. As the attributes which we wish to query are the Name and Age, we will select the **Person_Name** and **Person_Age** attributes. To do this, chose the **Person_Name** attribute from the list, in the manner described above. This will be added to the query line. Then repeat the procedure for the **Person_Age** attribute. The query as displayed in the query line will now look as follows :

SELECT E.Person_Name, E.Person_Age

The system automatically adds the ***E.*** to the query as this is the format in which ONTOS expects it. Now that the class and the attributes have been chosen, the condition can be stated. Firstly, however, press the **WHERE** button using the mouse. This adds extra information to the query and prepares GRIFON for the entry of the conditions. The query will now look like:

SELECT E.Person_Name, E.Person_Age FROM Person E WHERE

a condition is of the following form :

<Attribute><Relational Op.><Constant Value>

Again for simplicity the third part of the condition must be a constant value and not another Attribute. However if the query could be addressed against a number of different classes the facility to compare attributes from different classes would become important in the creation of conditions.

Conditions can be connected together using the logical operators AND and OR.

In GRIFON, once the **WHERE** button has been pressed, the first condition can be created. This is done by first selecting the attribute from the list. This is added to the query text. Then press one of the relational operator buttons (=, <>, <, >, <=, >=). This too is added to the list. At this stage the user is prompted by the **Constant Value** field to enter a value for comparison against the attribute. This is the only keyboard entry part of the whole procedure. If there is a second condition, then having entered the constant, pressing the **AND** or **OR** buttons using the mouse will allow the repetition of the condition creation.

In our sample **Person_Age** would be chosen from the list. Then the '**>**' button would be pressed using the mouse. The cursor would then start flashing in the **Constant Value** field prompting us to enter the value **24**. As we do not want to add any more conditions, we can press the **COMPLETE** or **SAVE** button to signify the end of the query. At this stage the query is as follows :

SELECT E.Person_Name, E.Person_Age FROM Person E WHERE E.Person_Age > 24;

Pressing the **SAVE** button prompts the user to enter the name of the file to which the query is to be saved. A Window is displayed with an edit field, into which the user enters the name. This facilitates the execution of the file later.

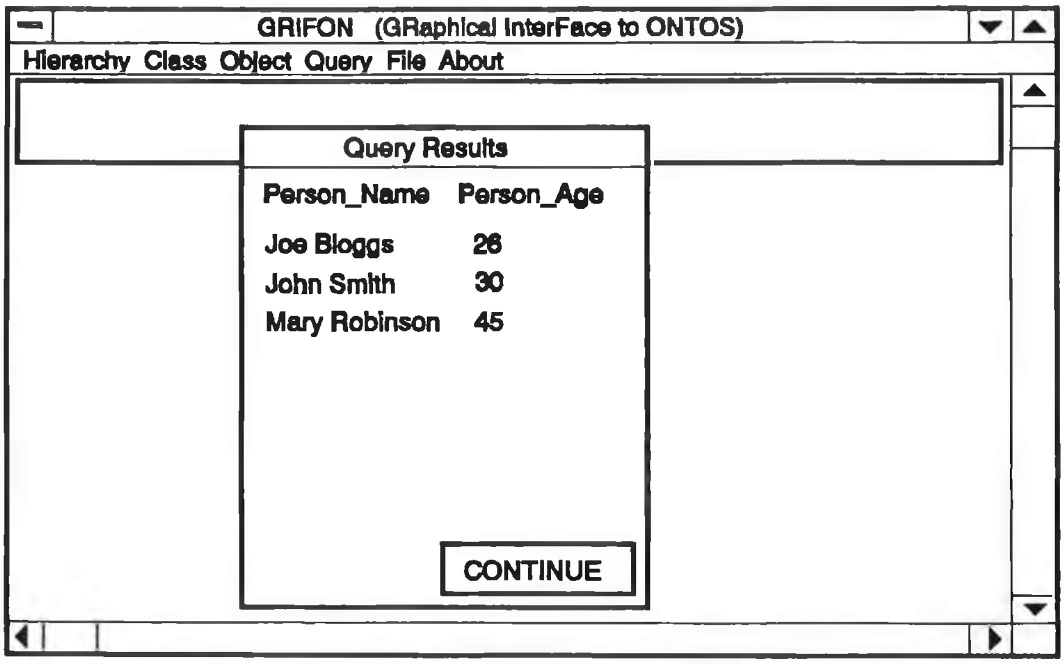
Once the file has been saved, the query is then executed. The results of it are displayed in another window. The format of this is as shown in figure 4.11.

Pressing the **CONTINUE** button in this window will effectively end the query session and return the user to the class-instance hierarchy.

If the **COMPLETE** button is pressed after the query has been created then the query is executed without it being saved. This may be useful if the query is a *once-off* request being made to the database.

4.2.8.1.2 Features of this facility.

This form of query creation allows the user to create complex queries in a simple manner, requiring them to do the minimum amount of keyboard work. To help the user along, a


GRIFON (GRaphical InterFace to ONTOS)
Hierarchy Class Object Query File About
Query Results
CONTINUE

Person_Name	Person_Age
Joe Bloggs	26
John Smith	30
Mary Robinson	45

Figure 4.11 - Query Result Window.

number features have been built in.

- **Strict control on buttons.** To ensure that the user cannot make mistakes in the creation of the query, only buttons appropriate to the current stage of query construction will have any affect on the system. For example, pressing the **AND** or **OR** buttons will not affect the query unless they are pressed after a condition has been entered, ie. after a constant has been entered. An attribute cannot be selected when the system is expecting a relational operator like '=' or '<>' etc..
  This feature minimises the amount of errors which the user can make. The errors cannot be syntactic as this is tightly controlled.

- **Query text display.** The query text is displayed permanently in the second line of the window. This is continually being updated as the user selects attributes and pressed buttons. It scrolls across, so the piece of the query currently being created is displayed. Although the system holds the user inside tight controls while creating queries interactively, the query can be manipulated or changed through the query edit line. The text of the query can be moved through using the left and right arrows and changes can be made in it. Any changes will affect the query itself but will not affect the stage of query development which the user is currently at. For example, if the second condition was being entered interactively, and the first condition was entered incorrectly, pressing the left arrow on the keyboard would move the cursor back to allow the alteration of the first condition. However as far as the system is concerned, the stage of development is still the second condition and the allowable buttons are associated with that.

- **Class Selection Flexibility.** The class is originally selected from the class-instance hierarchy. If during the creation of the query, the class needs to be changed the **Class Selected** field can have its value changed. This will affect the final query. However, this is not advisable as the attributes will also need to be modified to reflect the change in class.

The system allows the user to create queries in a simple manner, with minimal keyboard use. It also facilitates the alteration of the query if required, without over-complicating the interactive process. Once the simple sequence of steps has been mastered, the system can be used to create all sorts of queries, from simple condition-less ones like :

Select E.Person_Name from Person E;

to complex ones with multiple conditions.

4.2.8.2 Writing queries externally.

The interactive query creation facility allows the user to create, save and execute queries. For experienced users, the process of creating queries by means of pressing buttons and selecting attributes from a list may be tedious and slow. If they know the OSQL language, they can use an external text editor and create the query itself. GRIFON allows the execution of queries from external files. These files may can have been created interactively or typed in externally.

4.2.8.3 Executing Queries.

Any queries in text files in the system can be loaded through GRIFON and executed. The third option on the QUERY sub-menu, off the main system menubar provides this facility. Choosing this option, displays a window with a data-entry field on it allowing the user to enter the name of the query file to execute. Entering the name and pressing the **CONTINUE** key using the mouse will execute the query in that file, and the results will be displayed in the format similar to that in figure 4.11. This facilitates the creation of a number of files containing commonly executed queries. When any of these need to be executed, the user just enters the name of the appropriate query file and the query results are displayed.

4.2.9 Other Facilities

GRIFON provides some extra features which can be applied, but due to the database, they cannot be used.

4.2.9.1 Opening a Database

Although not mentioned above, opening a database is the first operation carried out on running GRIFON. When the system is executed, a window appears prompting the user to enter the name of the database. The user enters the name of the database, eg. **EMPLOYEEDB** and hits the <CR> key or presses the **CONTINUE** button using the mouse. The database is opened if it exists. The process of calculating the hierarchy then commences. This is a multi-step process. Initially all the classes in the database need to be read in from the database. From these the class hierarchy is constructed. Many different pieces of information are extracted from the database at this stage to calculate the hierarchy. The different processes involved will be outlined in the next chapter, when I will outline the representation of the hierarchy and the classes in GRIFON.

Once the hierarchy is calculated, the user can select an operation to carry-out, from the menu.

The facility is provided to allow the user to change from one database to another during a session in GRIFON. The user can close a database in the system by selecting the appropriate option off the menu. Similarly a database can be opened by selecting the correct menu option, in the same manner as outlined above. However, ONTOS will not allow a database to be opened, closed and another one opened during a single program. Based on this when GRIFON is executed, the user can only use one database during the execution of the system. To examine a different one would require him to exit GRIFON and start it again, or spawn GRIFON again as a new process.

The deletion or modification of classes from an object-oriented database hierarchy is a complicated process. In a relational database, removing a relation has very few repercussions on the rest of the database. Relations are very much independent entities.

Classes, in contrast, are very much inter-dependent. This can be seen from the different hierarchies displayed by GRIFON. Classes can inherit attributes and methods from super-classes. They can be composed from other classes through their attributes. Therefore it is easy to see how removing a class from the database can have an affect on the integrity of other classes.

GRIFON allows sub-classes to inherit attributes from super-classes higher in the inheritance hierarchy. If a super-class is removed, then the structure of some or all of the sub-classes needs to be changed. In addition to this, if a class is composed of other classes, eg. Class Vehicle has an attribute *Manufacturer* with class Company as its domain, if class Company is deleted from the database, the condition of all objects of class Vehicle will be inconsistent, referring to objects that don't exist. Here again much thought needs to be put into the removal.

For these reasons, GRIFON, provides on the menu for the deletion of classes, but does not provide the facility. The same applies to the modification of classes.
For modification purposes, I have no reason to expect that the manner of modifying the class would appear to the user to be significantly different from the process involved in the creation of a class.

4.3 Conclusion

GRIFON is a graphical interface to an object oriented database. That database is ONTOS. GRIFON does attempt, to and succeed in presenting the user with a clear and concise manner in which the database can be created, accessed and manipulated. The interface provides ease of access to the underlying database. I feel that this is attributable to the following features:

- **Pictorial representation of the database.**
  'A picture is worth a thousand words' sums up the reason for the pictorial representation. The user can get a concrete picture of the data, the database structure, and due to it being pictorial, can retain the information longer.

- **Removal of keyboard entry as far as possible.**
  The keyboard is no longer the primary input device, only being used when unavoidable. Unfamiliar users need not hold any fear of keyboard, as most of the interaction is carried out using the mouse.

- **Direct Manipulation interaction using the mouse.**
  The mouse is used to manipulate nodes on the screen affecting the underlying database. Selecting a class, when creating a new class, updates the database to indicate that the selected one is to act as the super-class for the new one.

- **Friendly windowed user-interface.**
  The options available are presented clearly and are accessible using the mouse. An information window is displayed to prompt the user as to what to do next. Where choices have to be made, a list of available options is displayed where possible.

- **Consistency maintained throughout.**
  All operations on the database start from a class hierarchy diagram. The user is always returned to a hierarchical view of the database on completion of operations. Through the windowed environment, user-interaction is done through buttons, lists, edit-fields and dialog windows which facilitate input/output of the relevant information.

GRIFON represents the physical realisation of how I think a user-interface to an object-oriented database should perform. It has been inspired by three main things :

- Current Interface technology
  - Available interfaces to database,
  - Current user interface environments.
- Object oriented database
  - The facilities they provide,
  - Their suitability to different forms of representation
- My personal feelings on the area
  - How I think a GUI to an object oriented database should be.

Although GRIFON does not necessarily represent the perfect interface to an object oriented database, I do think that it does present a way in which other databases can be represented. It does present a user with a friendly means of interacting with the database, through the features described above.

Although not a final system in itself, it may serve to inspire other developers about the possibilities for interfaces. User interfaces to complex systems, do not necessarily need to be complex !

The ideas implemented in the creation of the interface were very much affected by the facilities provided by the software and hardware available. The tools available to a large extent determined the extent of the development of GRIFON.

In the next chapter I will examine the environment in which the development took place. My choice of development tools will be outlined, explaining my reasons for their choice. In the development of the GRIFON, I found it necessary to development a number of algorithms to simplify the representation of the data. These will be outlined. As far as was possible, GRIFON was developed in an object oriented manner. I will look at how I represented some of the entities in the system in an object-oriented manner.

Chapter 5
The development of GRIFON.

5.1 Introduction

GRIFON, the system, was developed to give some indication of what form a user interface to an object-oriented database might take. The resultant system was very much determined by the available software and hardware.

In this chapter, I will outline the software and hardware configurations which I used in the development of GRIFON. Their choice was determined by a number of factors, and these will be described. GRIFON is a merging of a number of complex tools under an object-oriented framework to produce a complicated system. As the system interacts with a database, in order to represent it in a clear graphical manner, the way in which the database objects are represented by the system is of the utmost importance. This representation is outlined in this chapter, as are the methods by which GRIFON interacts with the database.

The steps involved in calculating the class-hierarchies are outlined. A number of graph drawing algorithms needed to be developed to facilitate the presentation of the hierarchies clearly. These algorithms are described.

5.2 Development Environment

GRIFON, is a system which facilitates a novel method of interacting with OO databases. In addition, its implementation is equally as novel, combining a number of new object oriented tools and packages, on a high powered modern personal computer workstation.

5.2.1 The Software Environment

GRIFON is the result of the merging of a number of different and totally diverse software development tools and environments. It is built around three main components :

(i) IBM OS/2 Operating System and Presentation Manager Windowing Environment.

(ii) Glockenspiel C++ Version E1.2 and Commonview 1.1 Class Constructor.

(iii) Ontologic Inc.'s ONTOS Version 1.42 Object-Database.

(i) IBM OS/2 is the latest operating system for the new range of IBM personal computer's - the PS/2. These machines contain 32-bit micro-processors and internally support multi-tasking. OS/2 is designed to exploit this new technology.

OS/2 is a single-user operating system which supports multi-tasking, dynamic linking, advanced memory management, inter-process communication and contains an impressive filing system. It is supplied with a windowing environment to allow access to all of these facilities in a relatively simple manner - Presentation Manager. Presentation Manager (PM) is a windowed environment which applies the WIMP philosophy. It is primarily mouse driven and allows the user to access the facilities provided by the underlying operating system in a clear, friendly manner.

Applications can be developed to use the PM windowing system as their interface. PM provides a large library of, in excess of, 500 C functions which allow access to both the windowing and graphics facilities provided by it, but also the base operations of the operating system. This library is provided in the PM development kit and makes up the Application Program Interface (API) to PM's facilities.

The features provided by this API, OS/2 and PM are outlined in Appendix F.

(ii) Glockenspiel's C++, is a complete version of the AT&T standard C++ language. It is a pre-processor which takes C++ programs and converts them into a form which can subsequently be compiled by IBM C/2 or Microsoft C Ver 5.1. Appendix G contains an outline of the C++ language, and the features which it provides.

From a programmer's, viewpoint, C++ provides all the facilities of C, with the added bonus of the structuring features of object orientation, such as data encapsulation and abstraction, polymorphism and inheritance. Classes facilitate the inheritance of data through the class-hierarchy.

In addition, Glockenspiel C++ is optionally supplied with a library of classes for accessing the facilities provided by PM in an object-oriented manner. **CommonView**, as this class library is called, presents the main features of PM to a C++ programmer in an object oriented manner. PM provides functions and operations to allow the creation and manipulation of buttons, icons, lists, windows, dialog-boxes, scroll-bars, and many more controls. However, to access these features requires the writing of complex code.

Appendix H contains a simple C program which makes calls directly to the PM API. This program simply, displays a window on the screen and displays the message **Hello World**. This program is, by no means, simple to follow with the inclusion of cryptic function calls, constants and structuring. To develop application under the PM API directly, requires the structuring of the application around the structure required by PM.

In contrast to this, Appendix G, contains a program written in C++, using the CommonView class library structures, which performs the same thing. The C++ program is not only shorter, but is significantly easier to understand and debug. It is much clearer through its enhanced structure. For an application developed in C++, the subsequent provision of the windowed interface through the inclusion of CommonView functionality, is a straightforward procedure.

CommonView removes many of the aspects of PM programming which make it complex, like the cryptic messages and constants, the complex sequence of steps in doing relatively simple tasks, such as displaying a window. Although CommonView is a well developed product, there are many features of PM which it does not support. It is concerned primarily with providing the programmer with a framework within which the windowed environment can be accessed. However, some of the specific underlying features, usually graphical, supported by the windowing environment are not supported directly through CommonView. The main reason for this is to aid portability. CommonView is designed to sit on top of many different windowing environments, so the provision of graphical functions specific to particular windowing systems, or specific hardware, may remove the large degree of portability which CommonView now possesses. However, because C++/Commonview programs are ultimately converted into C code, calls can be made to the underlying windowing functions, in our case PM API, indirectly from the C++ program. The benefit of this is that one has the structuring and simplicity of C++ and Commonview and where necessary the extra functionality of PM. CommonView is described in more detail in Appendix G.

(iii) ONTOS, as outlined in Chapter 2, is an object-database developed for use with C++. In its current form it comprises no-more than a library of classes, iterators and functions which provide the C++ programmer with database storage facilities for their programs.

From a developmental viewpoint, ONTOS is important as it is the first commercially available object-oriented database for C++. ONTOS requires a number of lines of C++ code to carry out even the simplest task. It is tightly linked with the C++ programming language and is really an extension of it. This is in contrast to relational databases which provide for remote SQL function calls from within the application code. These calls make requests from the database at a very high-level. ONTOS, in contrast, is low-level, integrating closely with the programming language. It acts as a persistent storage facility for C++ programs.

For example, consider the C++ program included in Appendix A which creates a class Employee and then instantiates one object of this class. When the program terminates, the class definition and the object which is created from it is lost.

The function **main()** creates the new object of the Employee class. However, as I mentioned above, when the program terminates the class definition and the object are '*no more*'.

Because ONTOS is so close to the C++ programming language, very little modification is required to the C++ code itself to make the class and its object persistent, and write them to the database. An extra step would be needed in the compilation stage of the development to use the database, but this is only a single line which needs to be included in the project *MAKE*[1] file. The modified version of the **main()** function is presented in Appendix B, with the new ONTOS related lines highlighted.

Indeed to a programmer unfamiliar with ONTOS, it would appear on reading this program that external C functions are being accessed. The database is accessed at such a level that it has become part of the language.

[1] A MAKE file consists of a list of the source files, the windowing environment resource files, the icon, pointer or bitmap files and the compiler and linker switches to be applied when compiling and linking the application.

It would subsequently be relatively simple to write a program which would read back the class into memory and make it accessible to another program.

A description of ONTOS is included in chapter 2, and a further technical outline is included in Appendix J, outlining the facilities which it provides to the programmer, many of which are used in GRIFON.

These are the tools which were used to develop GRIFON. The operating system and the database are complex, requiring large amounts of secondary storage and processing speed. In reply to these requirements, the computer on which GRIFON was developed boasted a very high specification.

5.2.2 The Computer Architecture

The platform on which these tools and environment operated was an IBM PS/2 Model 70-121. This is a high-powered 80386 based personal computer. As the storage requirements for the database are quite high, a large hard-disk drive was installed containing 120 Mbytes of diskspace. For speed, an 80387 mathematics co-processor was installed and an additional 8 Mbytes of RAM primary storage. The system operated at a clock speed of 20 Mhz, making it extremely fast.

To facilitate the clear display of the windowed environment, a VGA graphical adapter was fitted to the computer, and an IBM 8513 VGA monitor. This facilitated a screen resolution of up to 640 x 480 pixels or dots on the screen, with up to 256 colours, more than adequate for GRIFON's output requirements.

5.2.3 Reasons for using this development environment.

When developing any application, the features of the target system need to be considered when determining the appropriate environment and tools on which to develop it. GRIFON is no different in this respect. The development tools reflect, to a large extent, the features provided by the application.

Judging from current studies and surveys, as outlined in chapter 3, and looking at the currently available technology, windowing environments would seem to present themselves as ideal environments upon which to base new applications. Users like them and find them easy to use. Most of them provide advanced and impressive graphical facilities which can be easily exploited by the programmer, allowing many different tasks to carry on at the same time.

GRIFON is to present ONTOS in a graphical manner, providing various means of manipulating these representations, simply and concisely. Currently available windowing environments and tools ideally suit themselves to this sort of application. However, programming for windowing applications is not easy and this discourages many programmers from getting embroiled in the internals of it. In an attempt to take the pain out of such development some software houses have attempted to make the process easier through the development of windowing tools. CommonView is just one such example.

As outlined above, CommonView is a windowing interface class library for the development of windowed applications. Where Smalltalk [GOL83] has its own distinctive windowing appearance and structure, CommonView acts as a development tool sitting on top of an existing windowing environment. Like C++, CommonView is available for a number of different environments. It is a non-specific class library, working with a wide range of windowing environments. Those compatible with CommonView include, Microsoft Windows, IBM Presentation Manager, X Windows environments like, HP NewWave, OSF Motif and AT&T Open Look, Apple Macintosh and Adobe Display PostScript. The way the windowing objects are programmed using CommonView is independent of the GUI's look and feel.

This makes for easy porting of applications developed on one system using CommonView to a totally different system also having the CommonView library available.

The fact that C++, CommonView and ONTOS are all available for a wide variety of different systems, makes the possibility of porting GRIFON to other environments a real possibility.

The choice of PM is primarily due to its availability on a personal computer and the wide extent of facilities which it provides for graphical display. Appendix K contains a list of the graphical primitive functions provided by the PM API, the functionality which these provide make PM an ideal environment on which to develop any graphical applications. The functions in this list range from point plotting to complex spline construction, many of which are used in GRIFON. As mentioned above, CommonView neglects to provide functionality directly for certain PM features. However, the provision for the inclusion of PM API calls in C++/CommonView programs aleviates this inadequacy. The following extract of code illustrates the simplicity with which a call can be made to the PM API to draw a line, from a CommonView program.

This function is a function associated with the ExampleWindow class, defined from the CommonView class library. It makes a direct call to the PM API function to draw a line, passing the handle or address of the current window.

```
void ExampleWindow :: DrawLine(Point from, Point to)
{
  POINTL  the_point[2];                          // The start & end points in PM

  the_point[0].x = from.X();                     // Copy Commonview to/from points
  the_point[0].y = from.Y();                     // to PM point format.

  the_point[1].x = to.X();
  the_point[1].y = to.Y();

  /* Include the code in here to draw the line. */

  HPS = WinGetPS(Handle(API_CLIENT_HWND));       // Get the presentation space
                                                 // for this window.
  GpiMove(hps, &the_point[0]);                   // Move to the position.
  GpiLine(hps, &the_point[1]);                   // Draw the Line.

  WinReleasePS(hps);                             // Release the presentation space.

}
```

This function could now be called from anywhere inside the C++ program by passing a message to an object of class ExampleWindow in the following manner :

```
        ExampleWindow  *expwin;
                .
                .
                .
                .
        expwin->DrawLine(Point(10,10),Point(100,100));
```

This illustrates how functions contained in the PM API can be incorporated into class methods under the CommonView structure.

As regards the choice of programming language to use for the development, this was very much reliant on the database which was to be used. Current commercially available object-oriented databases are few and far between. The primary choices on which an interface could be based were ONTOS or GemStone, with both implementing OO principles on OO languages.

GemStone, as described in chapter 2, is based on a DEC VAX with a PC linked to it. Applications would be written in Smalltalk and access the database through a link between the PC and the mini-computer.

There are a number of issues which arose when considering GemStone as the database on which to implement a graphical interface.

- GemStone is based on storage of the database on a mini-computer. Object oriented databases are required to solve many of the problems associated with data storage in currently available complex applications. Many CAD systems, for example, are available on PCs, and relying on the backing storage of a mini-computer for these applications is unrealistic and impractical.

- The current emphasis in computing is being placed on personal computer workstations which would communicate with others over a local area network. Many proponents of OODBs feel that they should support distribution over a network. This is more apparent in ONTOS rather than in GemStone.

- Finally, the development of a windowed interface to GemStone would rely on the windowing capabilities of Smalltalk. Although Smalltalk did initially appear sporting the first real WIMP environment, its windowing environment has not become standardised. Any application developed under this environment would be very difficult to port to another system and Smalltalk is not available for all environments.

ONTOS, in contrast to GemStone is available on a large number of different platforms and operates under a wide variety of operating systems, i.e. OS/2, UNIX, PC-DOS (Microsoft Windows 3), VMS, and many more. It is primarily designed to be distributed over a network, although will operate perfectly well on a single machine. Although currently, when distributed over a network, the network must be homogeneous[2], future versions of ONTOS will support heterogeneous[3] networks.

ONTOS is based around the C++ programming language. C++ is growing in popularity among programmers and software houses. In fact, with the increasing popularity of object oriented programming and design, C++ is fast becoming the standard object oriented programming language. This popularity is reflected by the fact that there are versions of C++ available for virtually all development platforms and operating systems.

This combination of facilities provided by OS/2-PM, C++/Commonview and ONTOS favour their use in the development of GRIFON. Although very good tools individually, they all work very well together. The development of a graphical user interface to an object oriented database with the development tools all being object oriented seems natural. In addition, the availability of a very powerful personal computer sporting all the required hardware made it virtually impossible for any other hardware/software configuration to be chosen.

[2] Homogeneous networks are those which have nodes all of the same form. This will usually mean that the nodes attached to the network will all be operating under the same operating system.

[3] Heterogeneous networks can have nodes attached which are operating under diverse and incompatible operating systems, for example, PC-DOS, UNIX, VMS etc.

5.3 The Implementation of GRIFON

GRIFON is a prototype interface to an OODB - ONTOS. The facilities it provides are described in the previous chapter. The manner in which these facilities were provided owes a lot to the manner in which the system was designed and represented.

GRIFON was developed using object-oriented development tools and language, and the development technique employed in its creation was also object oriented. This applies to the manner in which the various components of the system were represented internally.

In the remainder of this chapter I will outline a number of important features concerning the development and implementation of GRIFON. These will be dealt with under the following:

- The representation of the system.
- GRIFON's interaction with ONTOS.
- How the hierarchy is displayed.
- How queries are processed in GRIFON.

5.3.1 The representation of the system.

In a system developed in an object oriented manner, the way in which the individual components of the system are constructed and designed is of the utmost importance to the performance of the system.

In GRIFON, there are a number of different separate components involved :

- The representation of the windows involved in the interaction.
- The representation of database structures.

The manner in which the system responds to the problems of representing each of these, can be dealt with separately, looking at each in turn.

5.3.1.1 The windows in GRIFON.

CommonView provides a hierarchy of window classes from which programmers can define new sub-classes applicable to their individual applications. Figure 5.1 shows the hierarchy of window classes provided in the CommonView class library. The three window classes generally accessed and used by applications are the three leaf node classes - **DialogWindow,**

TopAppWindow, ChildAppWindow. These three are all windows but all exhibit different characteristics and are used for different purposes. DialogWindows are used where there is some input or output to be carried out. The processing of the application ceases until the input/output has been completed.

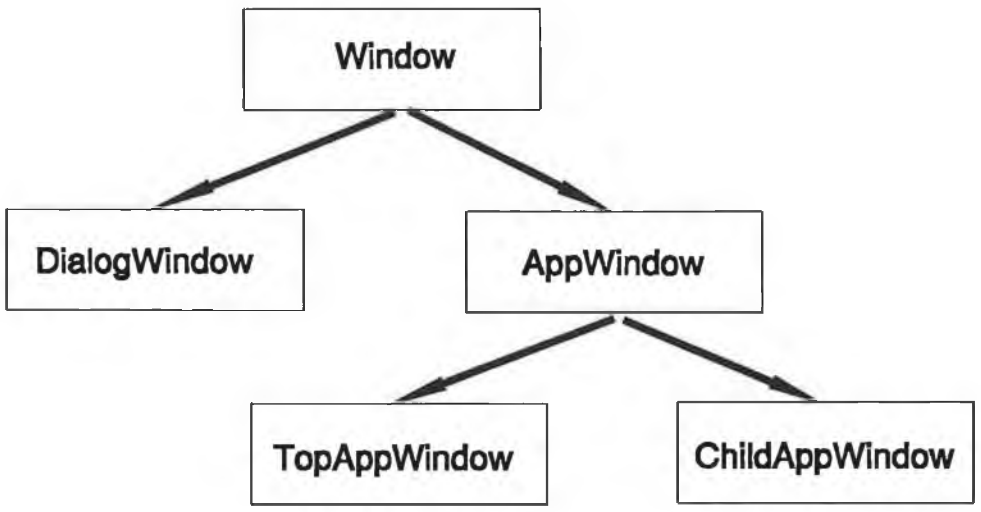


Figure 5.1 - CommonView's window class hierarchy.

This is generally signified by the pressing of a button in the object of the DialogWindow class. These types of window are referred to as *modal.* The user cannot access any other window on the screen until the processing on this window has been completed.

TopAppWindows and ChildAppWindows are *modeless.* They can be interacted with, without affecting the current thread of execution of the application. The user can interact with other windows while still having the Top or Child AppWindows available for access.

TopAppWindows and ChildAppWindows need to have sub-classes defined from them in the application to be used correctly, as they do not facilitate the instantiation of object of their classes directly, only of sub-classes. TopAppWindows are used as super-classes for the main application window while ChildAppWindows act as super-classes for any other modeless windows. In the case of GRIFON, such modeless windows tend to be information windows or message windows displayed while the system is carrying out some other processing, e.g., constructing the class hierarchy or creating a new class.

Figure 5.2 presents the window hierarchy created in GRIFON from the three accessible window classes offered by CommonView.

```
TopAppWindow
        SystemWindow
ChildAppWindow
        ChildWindow
        StatWind
DialogWindow
        OneButtonDialog
                Intro_Window
                Instant_Info_Window
                Class_Data_Window
        DoubleButtonDialog
                Entry_Window
                DoubleButton_EditLine_Window
                        DB_Name_Window
                        Class_Name_Window
                DoubleButton_List_Window
                        Domain_List_Window
                        Attr_List_Window
                        Inst_Info_List_Window
        Query_Window
        Instantiation_Window
```

Figure 5.2 - Window hierarchy in GRIFON.

From the ChildAppWindow class, two new classes are created. StatWindow is the information window at the top of the application window. It is used to display relevant help information for the user of GRIFON. ChildWindow is the class from which processing message windows are created.

GRIFON makes use of many different types of DialogWindow configurations. All DialogWindows need at least a single control (a button, list-box etc.) on them to complete their execution and return to the normal thread of execution in the application. Therefore, as can be seen from figure 5.2, two sub-classes of DialogWindow - *DoubleButton_Dialog* and *OneButton_Dialog* have been created. These act as super-classes for most of the dialog windows used in GRIFON. The *DoubleButton_* classes as their names indicate are dialog windows with two buttons (at least) on them, which facilitate the termination of the execution thread associated with them. One of these buttons will usually refer to the successful completion of the processing associated with the dialog, while the other will probably be associated with the cancellation of the current dialog. *OneButton_* classes, similarly, have one button by which to terminate their thread of execution. The *DoubleButton_Dialog* class has in turn two other classes of sub-classes - *DoubleButton_EditLine_Window* and *DoubleButton_List_Window*. Many of the dialog windows in GRIFON require the user to enter some details on the window before pressing the appropriate button. All of these windows have two buttons (continue or cancel). Similarly a number of these two button windows have listboxes as well, presenting the user with a list of options from which to choose. Actual objects of these classes may have more controls associated with them, like extra edit lines or more than two buttons, but they inherit the features associated with the common *DoubleButton_EditLine_Window* or the *DoubleButton_List_Window*.

QueryWindow and InstantiationWindow are declared as being direct sub-classes from DialogWindow, because they are more special cases of dialogs. *QueryWindow*, for example, contains three editlines, a listbox and thirteen buttons, so it is clearer, from a developmental viewpoint to consider it as a directly inherited dialog window from the *DialogWindow* class.

Most of the sub-classes of DialogWindow have many features in common.

The following is the declaration of the DB_Name_Window. This is one of the simpler dialogWindow classes created, yet it contains many of the class methods which are common to the other window classes created :

```
//************************************************************************************
//*
//*       Class:    DB_Name_Window
//*       Descr.:   This window presents the user with the dialog box
//*                 in which to enter the name of the database to
//*                 be accessed.
//*
//************************************************************************************

class DB_Name_Window :: public DoubleButton_EditLine_Window

{
        char*     db_name;            // Stores the name entered.

        void far  ButtonClick(ControlEvt);     // Respond to a button press.
        void far  KeyUp(KeyEvt);               // Respond to a key press.
  public:
        DB_Name_Window(pWindow, ResID, char*);      // Constructor function.
        void      DatabaseSelected();               // Functions specific to the
        char*     GetName() {return(db_name);}          // functioning of this window.
};
```

This class is inherited from DoubleButton_EditLine_Window, so in addition to the attribute db_name, it also has two PushButtons and an EditLine, so the functions defined on this class can access these inherited attributes.

The functions declared as protected -

```
        void far ButtonClick(ControlEvt)
                and
        void far KeyUp(KeyEvt)
```

are event activated functions. ButtonClick is called whenever a button on this window is pressed. This function should deal with processing the button press and carrying out the appropriate activity. KeyUp is the keyboard equivalent, it is automatically activated whenever a key is pressed while this dialog window is active. The constructor function for this class is passed a pointer to the parent window and the ResID (resource ID) for the new window.

The structure of the window is designed in the PM Dialog Window Editor, so a dialog window can be designed using the editor and subsequently associated with an application through its ResID. The character pointer passed to the constructor function is the default name for the database, which is usually blank.

The other dialog windows in the system are defined in a similar manner, with each window having its own ButtonClick function to deal with processing the buttons pressed in the individual windows.

5.3.1.2 The representation of the database structures.

To facilitate the interaction with the database, GRIFON needs to represent the data in the database in a manner which can be accessed and manipulated simply. In chapter 4, the features provided by GRIFON are outlined. The main feature of the system is the class hierarchy. All operations commence from the class hierarchy, in one of its forms.

Internally in GRIFON, the manner in which the class hierarchy is represented is of similar importance. The declaration of the Hierarchy class is as follows :

```
//*******************************************************************************
//*
//*        Class :   Hierarchy
//*        Descr.:   This is the declaration of the Hierarchy class which
//*                  contains information both the database and the
//*                  graphical representation of the classes in the hierarchy.
//*
//*******************************************************************************

class Hierarchy

{
   pWindow        parent;              // Pointer to the window on which it is displayed.
   Node*          list[50];            // An array containing pointers to all the class nodes.
   short          highlighted;         // Index of the currently selected one in the array.
   short          last_Highlighted;    // Index of the last selected one in the array.
   ChildWindow*   messageWindow;       // Message Window used to display processing messages.
   short          no_Nodes;            // Number of class nodes in the hierarchy.
   short          no_Levels;           // Number of different levels in the hierarchy.
   short          max_Level_Value;     // Number of nodes at the bottom level.
   short          max_Level_Name;      // What is the bottom level ?
   char           hierarchy_Type;      // What type of hierarchy is currently displayed.
   short          x_origin;            // The current X Origin of the hierarchy.
   short          y_origin;            // The current Y Origin of the hierarchy.

 public:
   Hierarchy(pWindow);

   //
   // Functions to open & close the database.
   //
   void    Initialise_Database(pWindow, char*);
   void    Terminate_Database();

   //
   // Functions to calculate the hierarchy, and extract information from
   // it.
   void    Calculate_Hierarchy(pWindow,char*);
   void    ReCalculate_Hierarchy(pWindow);
   void    Search(Type*, char);
   short   No_Nodes() {return(no_nodes);}
   short   Hierarchy_Depth();

   //
   // Functions to calculate the graphical representation of the hierarchy -
   //  the positioning of the class nodes, the lines between them
   //  and the number of instances for each class node.
   //
   void    Calculate_X_CoOrd();
   void    Calculate_Y_CoOrd();
   void    Calculate_Lines();
   void    Calculate_Instances();

   //
   // Return a pointer to a particular node in the hierarchy.
   //
   Node* Get_Node(short i) {return(list[i];}

   //
   // Functions to set/return the current origin position of the diagram.  These
   // values will change if the hierarchy is moved around (scrolled).
   //
   short   X_Origin() {return(x_origin);}
   void    X_Origin(short x) {x_origin+=x;}
```

```
  short   Y_Origin() {return(y_origin);}
  void    Y_Origin(short y) {y_origin+=y;}


  //
  // If a class node is selected using the mouse, then it is highlighted.  These
  // functions deal with setting/returning the index of the last node
  // highlighted.
  //
  short   LastHighlighted() {return(last_Highlighted);}
  void    LastHighlighted(short l) {last_Highlighted = l;}

  //
  // There are 3 types of hierarchies, these functions set/return the
  // flag value indicating what hierarchy is currently being displayed.
  //
  short   Hierarchy_Type() {return(hierarchy_type);}
  void    Hierarchy_Type(short h) {hierarchy_type=h;}
}
```

The main components of the hierarchy are the class nodes. These are contained in the variable **list**, an array of pointers to objects of class **Node**. The other information in the hierarchy class is concerned with statistics on the hierarchy, like the number of nodes, the number of levels, etc.. This information is used by the various options of GRIFON to display the hierarchy and the information on it.

For display purposes, a value indicating the hierarchy type is included. A value of 1 would indicate the current display is the class-inheriatance hierarchy, 2 the instance hierarchy etc., In addition, the origin screen position for the display is maintained in this class. This is updated whenever the hierarchy is scrolled. When the hierarchy display is scrolled or moved around the display, a different section of it is displayed in the window. The origin is used to keep track of where the different class-nodes are. All points in the hierarchy are maintained relative to the origin, so any movement is done by means of adding or subtracting values from the origin's value. This is essential so that the system can easily detect what class is being selected when the mouse is clicked over a class node in the hierarchy.

The hierarchy consists of an array of 40 pointers to nodes. These nodes are the rectangular class representation units in GRIFON. The C++ definition of class Node in GRIFON is as follows :

```
//****************************************************************************
//*
//*        Class :   Node
//*        Descr.:   This class will store the details on each class in the
//*                  hierarchy.  In addition information is stored on displaying
//*                  the object - its size, points etc.
//*
//****************************************************************************

class Node

{
   char*                  name;            // Name of this Class.
   char                   level;           // Level in the hierarchy.
   short                  x_value;         // Rel. X position of node in hierarchy.
   short                  y_value;         // Rel. Y position of node in hierarchy.
   RectangleObject*       the_node;        // Rectangle representing the node.
   LineObject*            the_line;        // Line to parent of this node.
   Point                  point_here;      // End-point of line on this node.
   Point                  point_from;      // End-point of line on parent node.
   short                  no_instances;    // Number of objects of this class.
   char                   is_referenced;   // Is class a domain of another class in the
                                           // class-composition hierarchy.

  public:

   Node (char*, char*);
   char*          Name() {return(name);}         // Get the name of this class.
   char           Level() {return(level);}       // Get the level at which it resides.

   short          X_Value() {return(x_value);}   // Get the X co-ordinate.
   void           X_Value(short x){x_value=x;}   // Set the X co-ordinate.

   short          Y_Value() {return(y_value);}   // Get the Y co-ordinate.
   void           Y_Value(short y){y_value=y;}   // Set the Y co-ordinate.

   Point          Here_Point() {return(point_here);}      // Get the line
   Point          From_Point() {return(point_from);}      //  end points.

   Bool           CheckClick(MouseEvt);       // Was the mouse clicked inside this node ?

   //
   //      Highlight and Unhighlight the node on the screen, when required.
   //      These operations set/reset a flag in the Hierarchy class.
   //
   void           Highlight_Node(pWindow, short, short, short, HANDLE);
   void           UnHighlight_Node(pWindow, short, short, short, HANDLE);

   //
   //      Display the node in its various different forms.
   //
   void           Display_Class_Rectangle(pWindow,short,short,HANDLE);
   void           Display_Instance_Rectangle(pWindow,short,short,HANDLE);
   void           Display_Attr_Rectangle(pWindow,short,short,HANDLE);

   //
   //      Display the name of this class.
   //
   void           Display_Name(char*, Point, pWindow);


   //
   //      Calculate the line position.
   //    - this calculates the two end-points for the line.
   //
   void           Calculate_Line(Node*);
//
//      Set or Query the number of instances of this class.
//
short               Instances() {return(no_instances);}
void                Instances(short i) {no_instances=i;}

//
//      Set or Reset the is_referenced flag for this class, indicating whether
//      this class is a domain of another's attributes.
//
char                Referenced() {return(is_referenced);}
void                Referenced(char r) {is_referenced=r;}

}
```

Every object of class Node in GRIFON represents a class in the database. Because this system is a prototype system, the hierarchy is restricted to up to 40 nodes.

A Node object contains information required to display it, like its representation - a rectangle, the line to connect it to its superclass, and its position. It also contains information about the class which it represents, like the name of the class, its level in the class hierarchy, the number of objects of this class in the database, and if this class is referenced by another class in the class-composition hierarchy. Methods are provided to calculate these features, as outlined in the class definition.

The *Hierarchy* and *Node* classes are used in the representation of the database to the user. They are updated by the database only and GRIFON uses this information to represent the structure of the database to the user. Any additions or modification to the database itself, are done by issuing commands to the database directly, this change will then be passed on from the database to the *Hierarchy* and *Node* representation of the database.

For example, if the user creates a new object of Class X and adds it to the database, this is done directly by entering the data and creating the object. Once the object has been created, the state of the hierarchy updated. It is updated from the information in the database, so in effect it acts as a confirmation of the creation of the object. The hierarchy cannot be updated directly by the user. This is so that the integrity of the representation with respect to the database can be maintained.

5.3.2 GRIFON's interaction with ONTOS.

As described in the previous chapter, GRIFON interacts with the database to carry out a number of operations. Initially when the hierarchy is being calculated a lot of information needs to be gleaned from the database in order to display it. When creating new classes or objects or when information is being displayed on particular classes or objects, data needs to be transferred to or from the database. All of these procedures have very distinct operations which need to be carried out by GRIFON on ONTOS.

5.3.2.1 Creating the Hierarchy

In GRIFON, every hierarchy has an artificial root class called **RootClass**. This is to facilitate the consistent anchoring of the hierarchy. In the calculation of the class hierarchy, the fact that there is a consistent root class is very useful and exploited to the full.

ONTOS, is based around the idea of *iterators* which can be used to access the data available. This, and other, features of ONTOS is outlined in Appendix J. One particular iterator which is useful when calculating the hierarchy is the *SubTypesIterator*. Using this, passing a class pointer in the database to this iterator will facilitate the return of pointers to all classes which are sub-classes of this. Using this approach, and applying the ***depth-first search algorithm***, a list of the names of the classes in the database can be constructed.

A recursive version of the ***depth-first search*** is used to loop through the classes in the database and subsequently construct Node objects to in memory to represent them. The search implemented is as follows :

```
void Hierarchy :: search(Type* class_pointer, char level)

{
  SubTypesIterator* subClassPointer;  // iterates through the sub-classes.
  char              next_level;       // keeps track of the levels.
  Type*             the_class;        // pointer to each class returned by iterator.

  next_level = level + 1;       // Have entered function again, so must be another level.

  //
  // Get the sub-class iterator for the given super-class.
  //
  subClassPointer = new SubTypeIterator(class_pointer);

  if (!(subClassPointer)) return();       // Have reached a leaf node in the hierarchy with no
                                          // sub-classes.

  while(subClassPointer->moreData())  // while more sub-classes in iterator ...
    {
      the_class = (Type*)subClassPointer->operator()();        // Get a pointer to the sub-class.

      search(the_class, next_level);        // Now search the sub-tree from this class in the
                                            // depth-first manner.

      // If we are at a leaf node, through recursion, add the details of the
      // current class to the list.

      char* tempstr = new char[30];
      strcpy(tempstr, the_class->typeName());      // Add the name of this class to the list.

      list[no_nodes++] = new Node(tempstr, next_level);
      delete tempstr;
     }

   delete  subClassPointer; subClassPointer = NULL;
}
```

The close integration of C++ and ONTOS makes the construction of this recursive function relatively simple, considering the complicated operation which is carries out.

After the completion of this function, the **list** variable in the Hierarchy contains pointers to the node representations of the classes in the database, containing the names and levels of each of the classes in the class-hierarchy and the variable **no_nodes** contains the number of classes in the class-hierarchy.

These two pieces of information from the database are essentially enough to construct the display of the hierarchy.

The calculation of the hierarchy, up to a stage where is can be displayed, is carried out in five steps :

- Searching the database for the class information (as outlined above).
- Calculating the X co-ordinates in the representation of the hierarchy for the different nodes.
- Calculating the Y co-ordinates for the positioning of the nodes.
- Calculating the end-points of the lines between the appropriate class-nodes, in the class inheritance hierarchy.
- Getting the numbers of instances or objects of each class in the hierarchy.

The Y co-ordinate for each node is established by the level on which that node falls, ie. a node on level 3 of the hierarchy would be positioned below nodes of levels 1 and 2 on the screen and on the same level as other nodes of position 3. However, the establishment of the X co-ordinate is slightly more difficult. The problem arises because of the fact that there are different numbers of nodes at different levels and certain nodes should be positioned centrally above a number of others to indicate that it is a super-class. It would prove relatively simple to draw a hierarchy representing the database, but to draw a perfectly spaced one requires more effort. The restrictions which are to be applied in drawing the diagram are as follows:

- The super-classes of nodes are to be positioned exactly centre way above their sub-classes.
- The nodes on a particular level are to be spaced out equally, as far as possible.
- The lines from super-classes to sub-classes should not cross, thus producing a clear and straight forward graph.

These restrictions demanded that a graph drawing algorithm be developed to take these into account these restrictions when positioning the nodes on their different levels.

The algorithm to calculate the X co-ordinates for the nodes is shown below.
The explanation of some terminology used in the algorithm may help to simplify it :

Run : This is used to describe the number of nodes which all have the same parent.

Current Level : This is the level of the Node currently being processed.

Previous Level : This is the level (in the hierarchy) of the last node which was processed. This is NOT necessarily the level above the current node.

Bottom Level : This is the bottom level in the hierarchy, not the Root Level. It would generally be the level of the leaf nodes.

Node-Width : This is the width of one node plus the gap between one node and the next one. It is the pixel distance between the left point of one node and the left point of the next one on this level, the two nodes being the minimum allowable distance apart.

The Class-Node positioning Algorithm

```
do
 {
  Get a Node from the list.
  if (Not the Bottom Level) AND (Not a Parent Node)
    {
    /*
            This ensures that non-parent nodes do not appear above nodes, preventing the bad
            positioning of nodes.
    */

    Move the next position (X) for all levels below this one across by Node-Width. (to allow for this one.
    }
  else if (Node is parent)
    {
    /*
            This ensures that a Parent Node is positioned centre way above its sub-classes.
            Its sub-classes will already have been encountered in the list, as the list
            was created using a depth-first search.
    */

    Calculate its position as centre way above the previous nodes on the previous level.
    }
  else if (Node is in a Run (Not 1st Node in Run))
    {
    /*
            If the node is part of a run, it will be added to a list associated with this level,
            which is accessed when the parent node for this run is encountered.
    */

    Take note of it, and this will be used later in the calculation of the position of its
            super-class.
    }
  else if (This is 1st Node in run) and (Current Level is bottom level)
    {
    /*
            The processing for this condition is essentially the same as that for the last one,
            except that a note needs to be kept of this node as being the first in the run.
    */

    Take note of the position of this node as the 1st in a run.
    }
 } while (there are more Nodes in the list);
```

The result of the execution of this algorithm, working through the list of node details returned from the search function, is the X_value filled in, in the Node object for each of the classes, in the hierarchy.

Similarly, calculating the positions of the lines between the nodes posed a problem. A series of stacks were used, one for each level, to keep track of the X positions of the nodes on each level of the hierarchy, as the list of nodes was being processed. This made the process of calculating the line positions from super-classes to sub-classes much easier.

Counting the number of instances of each class involved checking with the database the number of objects for each class in the list. The function which does this uses another iterator provided by ONTOS - ***InstanceIterator***, to loop through the list of instances for a given class. The C++ function implemented is :

```
short CountInstances(Type* classPointer)
{
 InstanceIterator   *instIter;  // The Iterator, moves through the objects of class ClassPointer.
 Object             *obj;       // Holds a pointer to the objects as they are returned.
 short              count;      // Used to keep count of the number of objects.

 instIter = new InstanceIterator(classPointer);

 count = 0;

 while(instIter->moreData())   // Loop through the objects of this class
   {
    obj = instIter->operator()();
    count++;
   }

 delete instIter;
 return(count);                 // return the total number of objects.
}
```

This function is applied to all the classes in the hierarchy and the value of the **no_instances** attribute of class Node is set according to the number of instances for each class, returned by the function.

Once this is done, all the relevant information is in the Hierarchy and Node objects to facilitate the display of the class-inheritance and instance hierarchies on the screen.

As the class-composition hierarchy is determined by the class which the user selects using the mouse, this is calculated when that representation is to be displayed. The problems and procedures involved in this are discussed later.

5.3.2.2 Creating a new class

The procedure for creating a new class, from a user's viewpoint is outlined in the previous section. The information is entered concerning the names and domains of the attributes for the new class. This information is used, in addition to the class selected from the hierarchy to create the new class in the database. As this class is created dynamically, a number of instructions need to be issued to the database to ensure that the class is created correctly.

As the names and domains of the variables are entered, a property is created in memory for that attribute. So, for example, if the attribute was **Name**, the domain was **OC_charPtr**, the ONTOS type for a character string, and the class which this is a property of is **Employee**, then the following **PropertyType** object would be created :

```
PropertyType *proptype = new PropertyType((char*) Name, "Employee", OC_charPtr);
```

This type of line would be repeated for each property or attribute being created for this class. Similarly, an object of class **Type*** needs to be constructed, to associate these properties with. If the new class being created was **Employee**, then the line to construct the database type (class) would be :

```
Type *cltype = new Type("Employee",SuperClass);
```

SuperClass is a pointer to the superclass for this new class.

When the class is to be written to the database, along with its properties, it is compiled, associating the properties with the class and then all the data is written to the database. The following lines would do this :

```
cltype->Compile();

cltype->putObject();
proptype->putObject();
```

This will create a new class in the database called Employee, with one property or attribute.

The classes **Type** and **PropertyType** are ONTOS-specific classes provided in the ONTOS library to facilitate the creation of classes dynamically, at run-time.

5.3.2.3 The Instantiation of Objects

When a class is constructed, a list of the attributes to be used for instantiation, both defined on this class and inherited from super-classes, is decided upon. This list is maintained in an external sequential file associated with this database. This file is accessed now to find out what attributes are to be used in the instantiation of a new object of the selected class.

A form is displayed on which the appropriate attribute values are to be entered.

The fields into which the details are entered are objects of one of three possible classes - **IntegerField, RealField,** and **CharField.** These are all sub-classes of a class called **EditField.** These three classes inherit the properties of a standard single-line edit field but each of them applies different restrictions on the possible values or characters which can be entered. For example, an object of class RealField will only allow the characters 0 to 9, the decimal point, and the plus or minus signs to be entered.

When the data has been entered, the values of the fields are copied into argument variables associated with the database which are written to it.

For example, setting the first argument to the name *John* could be done by

```
arglist->setElement((long) 0, (char*)"John");
```

The line :

```
Object *the_obj = (Object*) ClassPointer->Instantiate(arglist);
```

returns a new object, in memory, of the class pointed to by ClassPointer. The values this object will have are the values contained in the list **arglist**. The **Instantiate** function will expect the same number of parameters as the number of attributes created for this class, when the class was initially defined. Any of the attributes which are to be used in the instantiation are set to the values entered, the others are set to 0, the decision as to which attributes to use, being determined by the list in the associated sequential database file, as described above.

5.3.2.4 Getting Information from the Database

When the structure of a particular class is to be displayed, the class is selected from the hierarchy. An iterator - **PropertyIterator** is constructed to facilitate the return of the properties or attributes of this class :

```
PropertyIterator *pit = new PropertyIterator(classPointer);
```

where classPointer is a pointer to the selected class. Calling the function **operator()()** on the iterator, as with all iterators, will return a pointer to the next appropriate object, in this case, a pointer to the next property of the class will be returned. These can subsequently have their names and domains checked using the **typeName()** and **vSpec()** functions. So, the following lines of code will return a property from the class and print its name and domain:

```
PropertyType *ptype = pit->operator()();

printf("Property Name -> %s\n", ptype->typeName());
printf("Property Domain -> %s\n", ptype->vSpec()->typeName());
```

As regards the display of the information on individual objects in the database, this is done using an object identifier. When the user entered the information to construct an instance of the selected class, he also entered a value, which must be unique, to act as an identifier for this object. This identifier can be a number or a string, but its use facilitates simple data retrieval later. It allows GRIFON to access an object based on its unique identifier. OSQL is also provided to allow access to data but, this is over complicated if the user can sift through a list of objects and choose one to elaborate on.

An *InstanceIterator* is used again to loop through all the instances for a particular class, as shown above. When the appropriate object is returned, the value of it can be got using the *getValue* function of class Object. So, if **ptype** points at a property of the class and **the_object** points at the current instance of the class, whose value property value we wish to interrogate, the the line:

```
printf("Property Value --> %s\n",(char*)ptype->getValue(the_object));
```

will print out the value of the property, presuming that it is text.

5.3.3 How the hierarchy is displayed ?

As regards the display of the class-inheritance and class-instance hierarchies in GRIFON, all of the information required for their screen representation has been calculated and is included in the Hierarchy and Node objects. The display procedure loops through the list of nodes in the hierarchy, the **list** variable in class Hierarchy, and uses the co-ordinate values and the rectangle and line objects in each of the Node objects to display the hierarchy.

Although CommonView facilitates the display of rectangles and lines on the screen, its representation of text is limited to a single font, that provided by OS/2. For the display of the names of the classes in the node representations, calls are made to some of the PM API functions to display the information in a more aesthetically pleasing font and size.

If the hierarchy display is moved using the scrollbars, then the window needs to be cleared and the portion of the hierarchy now to be displayed in the window to be drawn. As the co-ordinates of the class nodes in the representation are all relative to the origin position of the hierarchy, the repositioning of the hierarchy only involves the incrementing of the origin co-ordinates by the amount of the movement due to the scrolling operation.

The windowing system takes care of the management of the display of the hierarchy, displaying only that which fits inside the window. The complete hierarchy is re-drawn, but only a certain section of it will be displayed in the window. This is due to the fact that the PM co-ordinate space is much larger than the window space. The PM co-ordinates may run from -32000 to +32000 whereas the physical co-ordinates of the window, in pixels, may only run from 0 to a few hundred.

The rectangle object which is drawn to represent each of the nodes in the hierarchy is contained in the Node class representation. There is a two-fold reason for this. Obviously, its inclusion in the node object, facilitates its display simply and quickly, as the rectangle is always available. Secondly, objects of the class RectangleObject are provided with a number of utility functions which, in addition to drawing the rectangle in a window, allow checks to be made with respect to the rectangle. One check which can be made is to see if the screen mouse pointer is inside a rectangle. As the nodes each contain a RectangleObject, checking to see if a class node has been selected from the hierarchy only involves looping through the list of Nodes applying the **PtInRect()** function to each of the RectangleObjects in turn. If the mouse button was pressed while the pointer was inside one of the node rectangles, then a value of TRUE will be returned from this function when applied to the RectangleObject of the selected Node.

The display of the class-inheritance and class-instance hierarchies is relatively straight forward in that it makes use of data which is available, having been pre-calculated either when the database was opened or when it was modified. However, the display of the class-composition of a selected class is slightly more complicated. It is not clear which nodes will be displayed until the required class-node has been selected, and this creates a problem.

5.3.3.1 Displaying the Class-Composition Hierarchy

When the user selects a node, whose composition is to be displayed, the structure of this class is checked in the database to determine what other classes act as domains for attributes of this class. Once this has been determined, the nodes associated with these classes in **list** are displayed, in an appropriate colour to indicate that they are acting as domains for the selected class.

This procedure is relatively time consuming as GRIFON needs to access the database, and subsequently cross-check with all the nodes in the database representation. However, the procedure is straightforward.

To make the class-composition representation more informative, it is necessary to display arcs which run from the selected class node to the domain class nodes, labelled with the appropriate attribute name. So, for example, if class **Vehicle** has an attribute **Manufacturer** which is an object of class **Company**, an arc will be drawn from the Vehicle class node to the Company class node and will be labelled Manufacturer. This is shown in Figure 4.3 in the previous chapter.

The drawing of these arcs posed another serious problem, because the positioning of the nodes is dynamic, the arc positioning needs to be done dynamically also. The procedure involves establishing what arcs are to be drawn, ie. to what nodes, and then determining where these nodes are on the hierarchy. There is no certainty about where they are - they may be on a different level in the hierarchy and, indeed, may not be displayed in the window at all.

The function for calculating these arcs is included in Appendix L. This function is applied to the selected class node and every domain node of this class. Even with the power of the computer, the procedure for displaying the labelled arcs takes a number of seconds. However, the resulting display is good with arcs being displayed intelligently between nodes on the same levels or different levels.

The function aims to minimise arcs crossing, and generally make the resulting diagram as pleasing as possible to view.

5.3.4 How queries are processed in GRIFON.

Query processing in GRIFON involves three stages -

- Allowing the user to construct the query.
- Actually executing the query.
- Displaying the results of the query.

When the user enters the query through the interactive query construction facility, the system limits the actions which the user can take. A restriction is placed on which buttons on the window can be selected, depending on the current stage of the query construction. CommonView facilitates the disabling of buttons, allowing screen buttons to be displayed but preventing the system from processing them.

GRIFON splits the query construction into five distinct stages. At each of these stages there is only a certain number of possible operations which may be carried out. For example, at stage 1, the initial stage, the user must select attributes from the list of available attributes. All the buttons on the screen are disabled with the exception of that associated with the list selection. At some later stage when the user is expected to select a logical operator (AND or OR), selecting an attribute from the list would not be a valid operation as GRIFON is expecting one of two buttons to be pressed, so the processing of the list selection does not take place.

The result of the interactive query construction is a textual OSQL query which can be processed by the database. The processing of the query is done using another ***iterator***, the ***QueryIterator***. This is passed the text of the query, and it returns data which matches the query conditions. The data is returned line-by-line, one line each time the **operator()()** function is invoked on the QueryIterator.

Consider, the query **SELECT e.Name FROM Employee e;**. If this query was stored in the string variable *buffer*, the following lines of code would pass this query to the database, through the iterator, and the results would be returned, one name on each line :

```
OC_startQuerySession();

QueryIterator* iter = new QueryIterator(buffer);

while(iter->moreData())
  {
   iter->yieldRowString(outbuffer, 200);
   printf("%s\n", outbuffer);
  }

OC_endQuerySession();
```

In this code sample, the results from the query would be displayed on a textual screen using the standard C printf() function, however, in GRIFON, the output is written to a dialog window which is used to display the results in a clearer, nicer manner.

The procedure for displaying the results in the output window, is just a matter is directing the contents of outbuffer to the window.

Appendix D contains a fuller sample program which creates a query, executes and deals with displaying the result, displaying the results.

5.4 Conclusion

GRIFON, has proved an important development, not only because of its novelty in the representation of an underlying database structure, but also because of its development using all object-oriented tools. The windowing environment used is PM, although not object-oriented in itself, through the programming of it through the CommonView class library has helped add a very strong structure to windowing programming. The addition of ONTOS to this environment is a natural progression, extending C++ to include persistence.

Although C++, CommonView and PM provide all the tools for the user-friendly representation of an ONTOS database, much work was required, both in arriving at the manner in which the database structures were to be represented and in the development of algorithms and methods to improve the displaying of these structures.

Chapter 6
Conclusions and Future Work

6.1 Conclusions

This thesis has endeavoured to examine the realm of object oriented databases and the application of graphical user interfaces to them. Through this examination, a better picture has been achieved of what form of interface can be applied to OODBs. Certain features of the database make it appropriate to a particular interface style. GRIFON was developed based on this research, to implement some of the ideas encountered. It was also developed to get a clearer picture of the issues involved in the development of a GUI, and the problems associated with its linking with a database. GRIFON does present a simple manner by which the user can interact with the database. Through GRIFON's representation of the underlying database, complex information can be gleaned from the database representation passively, without the construction of queries.

The thesis thoroughly examines the issues associated with both OODBs and user interface techniques, to give a background to the development of GRIFON.

Chapter 2 outlined the philosophy of OODBs, covering the main features which should be included in any forthcoming OO data model. Previous database research has contributed significantly to the development of OODBs, with semantic databases being the primary contributor. The principles implemented in these, and their contribution to current OODB developements is mentioned. Although no standard OO data model has been decided upon, a number of software developers have come up with their own OODB systems. Of those commercially available, four very different products are described.

Chapter 3 proceeded to examine the realm of user interface technology. The developments in human computer interaction have been widespread moving in many different areas. Systems have developed from simple textual interaction techniques to exciting visual direct manipulation techniques. Research has illustrated the increased productivity which can be achieved and the increased user-acceptance which is exhibited by these types of interfaces.

In the area of human-database interaction, textual interfaces have predominated. With the advent of more powerful computers providing enhanced graphics capabilities, added to the advances in user-interface techniques, users are offered the chance to interact with databases through pictorial representations of the data, with the underlying database structure being manipulable through the representation, implementing the ideas of direct manipulation [SHN82][SHN83]. Chapter 3 presented a number of database systems which have provided the user with graphical interfaces, through which the database can be, not only, examined and viewed, but also manipulated and modified. The systems examined deal with both object based databases, implementing the semantic data model, and also relational database.

Chapter 4 outlined the features of GRIFON in detail, highlighting the benefits accrued through this form of interaction with the database. The information in the database is presented in a simple manner, which the user can easily and simply relate to, however the development of GRIFON was not such a simple manner.

Chapter 5 described the development process. The various software tools and hardware are described. In the development of any interface, the manner in which the underlying data structures are represented is of the utmost importance to the performance of the system. GRIFON represented the features of the underlying database important for its representation in an object oriented manner. This proved very successful. Graphical display systems need to display results in a clear manner. In GRIFON's case, the representation of the class hierarchies presented a problem. The hierarchy can become very complicated creating problems in drawing the hierarchy on the screen, with the minimum of line crossings in the representation. The development of a number of algorithms became a necessity to facilitate the best possible representation of the database schema. These algorithms were outlined. ONTOS, the database on which GRIFON was based, provides programming language level functions through which an application can interact with it. To get the appropriate information from ONTOS, GRIFON applied a number of different access techniques. Some short coded examples were described to give a clear view of the issues associated with interacting with the database, from an application's viewpoint.

From the development of GRIFON it is clear that the creation of a GUI to an OODB is possible, and provides a new and clearer method of representing database structures and information, as well as facilitating interaction with the database, in a simple manner, by the user. GRIFON is the proof of this.

However, through the development of GRIFON, a number of issues arose. OODBs do provide a number of important benefits, but in their current incarnations, they contain a number of problems. The same can be said about the development of graphical interfaces.

Object-oriented databases are new. Few of them are commercially available, and as a result those which are tend to provide only basic OO features. The databases which are available, in particular ONTOS, is not very stable, as opposed to RDBs. No doubt, this will develop in time. Since the development of GRIFON, ONTOS Release 2.01 has become available. From initial investigation of this product, it would appear to add the required stability to ONTOS, while providing new extra facilities, such as multiple inheritance, good database distribution over a network and recovery.

Due to the lack of a standard for OODBs, developers tend to add the functionality to database systems which they feel are important. This may be good, in that extra functionality over what a model would provide might be included. However, often important elements are omitted and because of this personal approach to the development of the system, two databases may result in having very little in common, with inter-portability of code being virtually impossible.

This problem of a lack of a standard data model provides problems for interfaces designers and developers. Interface designers are slow to develop complex interaction systems to underlying database systems, which may prove to be obselete and useless if a subsequent data model appears totally different from the underlying database system on which they were developed. With current relational databases, such as INGRES, DB2 and ORACLE, interfaces developed for one database can simply be ported to work with another, because of the standard data model and the standards associated with SQL. With current OODBs like ONTOS and GemStone, ONTOS provides a limited object-SQL type query language, while GemStone provides a totally diverse retrieval language OPAL. Cross-portability would prove a major problem.

Dealing with the implementation of GRIFON, ONTOS, as it stands, provides an object database which implements most of the features outlined in chapter 2. It presents application developers with a clear view of OODB development issues. However, its lack of provision for the dynamic creation of functions which the user can access, illustrates its incompleteness. This is probably due to the fact that it is a new product, and as such is experiencing many of the teething problems associated with one. It is also probable that the developers of ONTOS, Ontologic Inc., are wary about committing themselves to too many features, until some standard is established.

As outlined in Chapter 5, the development of windowed applications using the API provided by the windowing environment can prove tedious, difficult and error-prone.

The development of GRIFON encompassed a C++ with the CommonView class library. Through the use of these, developers of windowed applications can construct their interfaces in a system independent manner. As CommonView is available for a large number of development plaforms, portability of windows applications is now a real possibility. Through the provision of this class of windows, as CommonView is, the development of windowed applications has been brought to a new level. Chapter 5 indicated the difference between the development of windowed applications using the PM API with C and using the CommonView class library with C++. The stark differences in the code only illustrate how simple the development of windows applications can be.

Although CommonView can help simplify the development of windowed applications, it cannot change the requirements placed on the system hardware. GRIFON was developed on a powerful personal computer, with large quantities of primary and secondary storage. An empty ONTOS database takes up 680Kbytes of diskspace. The construction of the class hierarchy and subsequent addition of data quickly increases this. For the adequate representation of a class hierarchy, high resolution graphics facilities are a necessity. Even with these enhancements, the speed of execution of the system is slow. GUIs and OODBs place large pressures on a computer system. After all, a windowed environment like PM is really only an application running on the computer. Subsequently executing a system like GRIFON, places large demands on the system, in addition to those provided by the windowed environment.

To make the display of the graphical representation of the database possible and plausible, the development of a number of hierarchy drawing algorithms was required. The development of these was enhanced by the structuring power of C++. These were described in chapter 5. Their development just illustrated again the major problems associated with the creation of a graphical interface. From the user's viewpoint, the representation of the database is taken for granted, the amount of processing and calculation to represent the database in a clear hierarchy is unknown. Judgement of the system by the user would be affected by the manner in which the database represents itself. This is the way it should be, and the implementation of the graphing algorithms illustrates their importance.

GRIFON illustrates a number of points preached by a number of experts in the field of human computer interaction. The provision of the mouse with most personal computers makes the simplification of applications a possibility. GRIFON requires the minimum of keyboard entry by the user. The keyboard can prove to be a problem both in use and a pyschologically worrying device to users familiar with it. The use of the mouse facilitated the implementation of direct manipulation [SHN82][SHN83] techniques. The idea of a user being able to interact with the database, and modify the structure of it, through a graphical representation of it, is intuitively pleasing.

6.2 Future Work

OODBs are new. Any development which has gone on in the area has been concentrated in the research area. The few exceptions to this are the few commercial OODBs available. Very little work has been carried out into the design of interfaces for them.

GRIFON in itself is very much a prototype GUI to a database. It has fulfilled its purpose of indicating the type of interface possible to a GUI, however, through its development a number of possible extensions to it, as well as a number of general projects, related to the area, appeared.

Initially for any substantial work to be carried out on OODBs, it is essential that a standard OODB data model is established. This will ensure that any applications developed for one OODB will be generally compatible, in the sense of relatively easy to port. In addition to the lack of a standard OODB data model, little or no work has been carried out in the area of developing a query language to OODBs. Commercial systems like ONTOS have come up with their own, object-oriented versions of SQL. The *Project/Select/Join* operators associated with relational databases and SQL have not the same relevance when applied to OODBs, so it would be useful to establish the form and structure which a query language should take to an OODB.

For those OODBs which do provide query languages, albeit SQL variants, these queries are not optimised in the same manner in which they are in relational systems. In the current commercially available OODB systems, this does not prove too important as the current applications using them tend to be experimental in nature. However, as OODBs grow in popularity, and are applied to large complex systems, distributed over large areas, the optimisation of queries will be an important performance factor. The development of an optimiser for such queries on OODBs will be an important future project.

GRIFON caters for the data parts of classes only. The methods associated with classes are not dealt with due to the limitations in ONTOS. In later releases which support the creation of methods and functions at run-time, and their addition to the database during the execution of an application, GRIFON will need to be extended to deal with this. Future projects could look at how an interface system would deal with methods, in particular, how they are created in the interface system, how the management of the functions is managed by the interface system, how they are presented etc., and importantly how they are executed inside the environment.

For a complete system, provision needs to be made for the serious modification of the database. The deletion of classes in the database can have serious repercusions for other classes in the database. Question arise about the validity of certain operations - can class X be deleted if it has sub-classes ? can class X be deleted if it is composed of other classes as domains for its attributes ? Modification to a single class may mean substantial modification to other sections of the database. These are all issues which would need to be dealt with to facilitate the extension of GRIFON.

In the recently released version of ONTOS - Ver.2.01, support is included for both multiple inheritance and database distribution. To an interface designer, these two issues can cause problems. The algorithms developed to draw the class hierarchies with the minimum of line crossings had to deal with classes having at most a single super class. In multiple inheritance, the problems associated with more than one super class appear. New algorithms will need to be developed to deal with this. This establishment of graph drawing algorithms to surmount this problem is definitely an area worthy of future research.

With the distribution of the database, the issues of locking of the data, concurrent access to database structures and how the data is represented, need to be addressed. Would it be wise to let the user know the location of the data in a network or should it be transparent to them? These are all issues which would need to be dealt with in future.

Whatever the future developments are in the areas of OODBs and GUIs, GRIFON, has illustrated that the development of a simple windowed interface representing the underlying database in a graphical manner is possible, and indeed favourable. It has shown that databases no longer need to be regarded as masses of textual information, but can and should be visualised. The abundance of windowed environments available currently for a number of different hardware platforms will, ultimately result in many more databases, both OO based and otherwise, being represented in totally new manners. A prime example of this is the appearance of CASE*Designer from Oracle Inc [ORA90], which, as described in chapter 2, combines a graphical user interface with X-Windows to present the underlying Oracle based CASE data in a graphical manner.

Although this is one of the first such applications, I am in no doubt that it will not be the last.

Bibliography

[ALA89a] : Suad Alagic, *Object Oriented Database Programming*, Texts & Monographs in Computer Science, Addison-Wesley, 1989.

[ALA89b] : A. M. Alashqur, S. Su, H. Lam, *OQL: A Query Language for Manipulating Object-Oriented Databases*, Proceedings of 15th Int. Conf. on V.Large Data Bases, 1989.

[AND88] : T. Andrews, C. Harris, *Combining Language and Database Advances in an Object-Oriented development Environment*, Proceedings of OOPSLA, 1987.

[AND90] : Tim Andrews, Craig Harris, Kiril Sinkel, *The ONTOS Object Database*, Ontologic Inc., Burlington, MA., 1990.

[ASH84] : Ashton-Tate Inc., *dBase III Version 1.0 User Manual*, Ashton-Tate Publications, 1985.

[ATK87] : M.Atkinson, O.P.Buneman, *Types and Persistence in Database Programming Languages*, ACM Comp. Surveys, June 1987.

[ATK90] : M. Atkinson, F. Baucilliou, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, *The Object Oriented Database System Manifesto*, Proc. OOPSLA, 1990.

[BAN87a] : J. Banerjee, W. Kim, H. Kim, H. Korth, *Semantics and Implementation of Schema Evolution in OO Databases*, Proc. ACM SIGMOD Int. Conf. Management Data, 1987.

[BAN87b] : J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballon, *Data Model Issues for Object Oriented Applications*, Readings in Object-Oriented Databases, San Mateo, California, Morgan Kaufmann, 1990

[BAR88] : Phil J. Barnard, M.R.C. Applies Psychology Unit, Cambridge, U.K., J. Grudin, M.C.C. Austin, Texas, USA, 1988, *Command Names*, Handbook of Human-Computer Interaction, Elsevier Science Publ., 1988.

[BAT86] : D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, T. Wise, *GENESIS: An Extensible Database Management System,* IEEE March 1987, Readings in Object-Oriented Databases, San Mateo, California, Morgan Kaufmann, 1990.

[BER88] : J.Berry, *The Waite Group's C++ Programming,* Howard Sams and Company Publishers, 1988.

[BIL88] : P. Billingsley, Human Factors Department, DEC, Maynard, MA., *Taking Panes: Issues in the Design of Windowing Systems,* Handbook of Human-Computer Interaction, Elsevier Science Publ., 1988.

[BRY90] : D. Bryce, R. Hull, *SNAP: A Graphics-based Schema Manager,* Comp. Science Dept., Univ. of South California, LA, California 1990.

[BUR85] : K.F. Bury, S.E. Davies, M.J. Darnell, *Window Management: A review of issues & some results from user testing, (IBM REP HFC-53),* San Jose California, IBM Human Factors, June 1985

[CAR85] : S.K. Card, *Windows: why they were invented, how they help.* The Office, March 1985.

[CAR86a]: M.Carey, D.DeWitt, D.Frank, G.Graefe, J.Richardson, E.Shekita, M.Muralikrishna, *The Architecture of the EXODUS Extensible DBMS* Proc. of the Int'l Workshop on OODB Systems, Pacific Grove, CA, Sept. 1986.

[CAR86b]: M. Carey, *Object and File Management in the EXODUS Extensible Database System,* Proc.of the 1986 VLDB Conf., Kyoto, Japan, Aug. 1986

[CAR88] : M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, S. Vandenberg, *The EXODUS Extensible DBMS Project: An Overview,* Computer Science Dept., Univ. of Wisconsin, Madison, Nov. 1988.

[CAR90] : L. Cardelli, *Semantics of Multiple Inheritance,* AT&T, Bell Labs. Murray Hill, NJ., 1990

[CHA76] : D.D.Chamberlain, *Relational Database Management Systems*, Computing Surveys, 1976

[COD71] : E.F. Codd, *A Relational model of data for large shared databases*, Communications of the ACM, Feb. 1971.

[COD74] : E.F.Codd, *Seven steps to RENDEZVOUS with the casual user*, IBM Research Report J1333, San Jose Research Centre, San Jose, CA., 1974.

[COD79] : E.F.Codd, *Extending the Relational Database Model to Capture More Meaning*, ACM TODS, Vol.4, No.4, December 1979

[COP84] : G.Copeland, D.Maier, *Making Smalltalk a Database System*, Proc.1984 ACM-SIGMOD Int.Conf.on the Mgt. of Data, June 1984

[CRE89] : P. Creasy, *ENIAM: A More Complete Conceptual Schema Language*, Proceedings of 15th Int. Conf. on V.Large Data Bases, 1989

[DAH68] : O.Dahl, *Simula 67 Common Base Language*, Norwegian Computing Center, Oslo, Norway, 1968.

[DAT86] : C. J. Date, *An Introduction to Database Systems, Volume I, 4th Edition*, Addison-Wesley Systems Programming Series, 1986

[DAT83] : C. J. Date, *An Introduction to Database Systems, Volume II*, Addison-Wesley Systems Programming Series Publ., 1983

[DEW89] : S.C. Dewhurst, K.T. Stark, *Programming in C++*, Prentice-Hall Software Series, 1989

[GED87] : D. Gedye, R. Katz, *Browsing the Chip Design Database*, Computer Science Division, UC, Berkeley, 1987

[GLO89a] : Glockenspiel Ltd., *Glockenspiel C++ 1.2 El Manual*, Dublin, Ireland., 1989

[GLO89b] : Glockenspiel Ltd., *Glockenspiel Commonview V1.1 User Manual*, Dublin, Ireland., 1989

[GOL83] : A. Goldberg, D. Robson, *Smalltalk-80: The language and it's implementation*, Addison-Wesley Publ., 1983

[GOL85] : K. Goldman, S. Goldman, P. Kanellakis, S. Zdonik, *ISIS: Interface for a Semantic Information System*, Dept. of Computer Science, Brown University, MIT, 1985

[GRE88] : J. Greenstein, *Input Devices*, Clemson University, Clemson, South Carolina, Handbook of Human-Computer Interaction, Elsevier Science Publ., 1988.

[HAM81] : M. Hammer, D. McLeod, *Database Description with SDM: A Semantic Database Model*, ACM 1981, Readings in Object-Oriented Databases, San Mateo, California, Morgan Kaufmann, 1990.

[HUD90] : S. Hudson, R. King, *CACTIS: A Database System for Specifying Functionally Defined Data*, Dept. of Computer Science, University of Colorado, Colorado, 1990.

[JAC86] : P. Jackson, *Introduction to Expert Systems*, University of Edinburgh, Addison-Wesley Publ., 1986.

[KEL84] : A. Kelley, I. Pohl, *A Book on C*, The Benjamins/Cumming Publishing Co. Inc., 1984.

[KER78] : B. Kernigan, D.Richie, *The C Programming Language*, Prentice-Hall 1978.

[KIM89] : Won Kim, *A Model of Queries for Object-Oriented Databases*, Proc. of 15th Int. Conf. on V.Large Data Bases, 1989.

[KIM90] : Won Kim, *Object-Oriented Databases: Definition & Research Directions*, IEEE Trans. on Knowledge & Data Engineering, Vol. 2, No. 3, September 1990.

[KIN84] : R. King, *Sembase: A Semantic DBMS*, Proc. of 1st Int. Workshop on Expert Database Systems, 1984.

[KIN90] : R. King, M. Novak, *FaceKit: A Database Interface Design Toolkit*, Proc. of 15th Int. Conf. on V.Large Data Bases, 1989.

[KUN89] : M. Kuntz, R. Melchert, *Pasta-3's Graphical Query Language: Direct Manipulation, CoOperative Queries, Full Expressive Power*, Proc. of 15th Int. Conf. on V.Large Data Bases, 1989

[LAE88] E. Laenens, *A Language for Object-Orented Database Programming*, Journal of Object Oriented Programming, Vol. 1, No. 5 1988

[LAE89] : E. Laenens, F. Staes, D. Vermeir, *Browsing à la carte in Object-Oriented Databases*, The Computer Journal, Vol. 32, No. 4, 1989

[MAI86a]: D. Maier, *Development of an Object-Oriented DBMS*, Proceedings of 1st Int. Conference of O.O. Programming Systems, Languages and Applications, Portland, Oregon, 1986.

[MAI86b]: D. Maier, *Indexing in an Object-Oriented DBMS*, Proc. Int'l Workshop on O.O. Database Systems, September 1986

[MAI87] : D. Maier, P. Nordquist, M. Grossman, *Displaying Database Objects*, Expert Database Systems, McGraw-Hill Publ., 1987

[MAI90] : D. Maier, J. Stein, *Development and Implementation of an Object-Oriented DBMS*, Readings in Object-Oriented Databases, San Mateo, California, Morgan Kaufmann, 1990.

[MIC88a] : Microsoft Inc., *OS/2 Programmers Reference Manual Volume 1*, Microsoft Press Publication, 1988

[MIC88b] : Microsoft Inc., *OS/2 Programmers Reference Manual Volume 2*, Microsoft Press Publication, 1988

[MOR81] : T.Moran, *The command language grammar, a representation for the user interface of interactive computer systems*, Int'l Journal of Man-Machine studies, June 1981.

[MUL89] : Mark Mullin, *Object Oriented Program Design with Examples in C++*, Addison-Wesley Publications, 1989

[MYE83] : B. A. Myers, *INCENSE: A System for Displaying Data Structures*, Xerox Palo Alto Research Center, California, 1983

[NYE90] : A. Nye, T. O'Reilly, *X Toolkit - Intrinsic Programming Manual*, O'Reilly and associates Inc. publications, 1990.

[OGD88] : William C. Ogden, *Using Natural Language Interfaces*, IBM San Jose Research Centre, San Jose, CA.(from Handbook of H.C.I., M.Helander)

[ONT90] : Ontologic Inc., *ONTOS Object Database Documentation, Release 1.42 (OS/2 Version)*, Ontologic Inc., Three Burlington Woods, Burlington, MA, 1990

[ORA90] : ORACLE Corp. U.K. Ltd., *CASE*Designer Ver 1.1, User Guide and Tutorial*, Surrey, England, United Kingdom, 1990.

[PAA88] : K. Paap, Comp. Research Lab., New Mexico State University, J. Roske-Hofstrand, Aero. Human Fact. Div., NASA-Ames Res. Cen., *Design of Menus*, (from Handbook of H.C.I., M.Helander)

[PET89] : Charles Petzold, *Programming the OS/2 Presentation Manager*, Microsoft Press publication, 1989

[REI81] : Phyllis Reisner, *Query Languages*, IBM Almaden Research Centre, San Jose, CA, USA. ACM Computing Surveys, March 1981

[ROW87a] : L. Rowe, C. Williams, *An Object-Oriented Database Design for Integrated Circuit Fabrication*, Electronic Research Laboratory, College of Engineering, UC, Berkeley, 1987

[ROW87b] : L. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, A. Tuan, *A Browser for Directed Graphs*, Electronics Research Lab., College of Engineering, UC,Berkeley, 1987.

[ROW87c] : L. Rowe, M. Stonebraker, *The POSTGRES Data Model*, Proceedings of Int. Conference on Very Large Databases, September 1987.

[SHN82] : B. Shneiderman, *The Future of Interactive Systems and the Emergence of Direct Manipulation*, Behaviour and Information Technology, 1, 1982

[SHN83] : B. Shneiderman, *Direct Manipulation: A Step beyond Programming Languages*, IEEE Computer, No. 16, 1983

[SHI81] : D. Shipman, Computer Corporation of America, *The Functional Data Model and the Data Language DAPLEX*, ACM Trans. in Database Systems, March 1981.

[SME88] : Dr. A. Smeaton, *CA4 Information Systems Course Notes*, School of Computer Applications, N.I.H.E., Dublin, 1988.

[SME91] : Dr. Alan F. Smeaton, *MSc. in Computer Applications, Information Systems Course Notes, Part 2 of 2*, School of Computer Applications, Dublin City University, Dublin, Ireland, 1991.

[SNY86] : A. Snyder, *Encapsulation and Inheritance in Object-Oriented Programming Language*, Proc. 1st Int. Conf. OO Programming Sys., Lang. & Apps, 1986.

[STO76] : M. Stonebraker et al., *The Design and Implementation of INGRES*, ACM trans. on database systems, September 1976.

[STO84] : M.R. Stonebraker, *QUEL as a Data Type*, Proc. 1984 ACM-SIGMOD Conf.on the Mgt. of Data, May 1984

[STO86a]: M. Stonebraker, *Inclusion of New Types in Relational Database Systems*, Proc. 2nd Int'l. Conf. on D.B. Engineering, Los Angeles, February 1986.

[STO86b]: M. Stonebraker, L. Rowe, *The Design of POSTGRES*, Sigmond Record, Vol. 15, No. 2, Association for Computing Machinery, June 1986

[STR86] : Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Publications, 1986

[TUL88] : T. Tullis, *Screen Design*, McDonnell-Douglas Astronautics Company, Huntington Beach, California, (from Handbook of H.C.I., M Helander)

[TYR90] : Pasi Tyrväinen, *Use of Object-Oriented Databases for the Domain and Task Models*, SIMPR, ESPRIT Project 2083, August 1990.

[VER88] : W. Verplank, *Graphic Challenges in Designing Object-Oriented User Interfaces*, ID TWO, San Francisco, California, Handbook of Human-Computer Interaction, Elsevier Science Publ., 1988.

[WOO88] : D.D.Woods, E.M.Roth, *Cognitive Systems Engineering*, Westinghouse R.& D. Centre, Pittsburgh, Pennsylvania, 1988.

[ZAN83] : Carlo Zaniolo, *The Database Language GEM*, ACM 1983, Readings in Object-Oriented Databases, San Mateo, California, Morgan Kaufmann, 1990.

[ZDO90] : S. Zdonik, D. Maier, *Object-Oriented Fundamentals*, Readings in Object-Oriented Databases, San Mateo, California, Morgan Kaufmann, 1990.

[ZIE88] : J. Ziegler, K. Fähnrich, Fraunhofer Institute IAO, Stuttgart, West Germany, *Direct Manipulation*, Handbook of Human-Computer Interaction, Elsevier Science Publ., 1988.

[ZLO75] : N.M.Zloof, *Query-by-Example*, Proceedings of the National Computer Conference, Arlington, VA, USA, 1975.

Appendix A

The following program is a C++ program which declares a class called Employee. Then an object of this class is created.

```
/*****************************************************************************************************
 *
 *          Program Name :  CxxDEMO.cxx
 *          Date :          April 1991
 *          Description  :  This program creates a class called employee.  It
 *          is not inheritted from any other class.  It includes details on
 *          Employees.  The program simply creates the class and requests the
 *          the user to enter the details of an Employee.
 *
 *****************************************************************************************************/


class Employee
{
  long     Employee_ID;                /* The Employee's Number.   */
  char*    Employee_Name;              /* The Employee's Name.     */
  char*    Employee_Sex;               /* The Employee's Sex.      */
  float    Employee_Salary;            /* The Employee's Salary.   */
 public:
  Employee(long, char*, char*, float);               /* Method to create an Employee. */
  long     ID();       {return(Employee_ID);}        /* Get the ID No.    */
  char*    Name();     {return(Employee_Name);}      /* Get the Name.*/
  char*    Sex();      {return(Employee_Sex);}       /* Get the Sex.*/
  float    Salary();   {return(Employee_Salary);}    /* Get the Salary. */
};



/*****************************************************************************************************
          THE CODE OF THE EMPLOYEE CONSTRUCTOR METHOD.
 *****************************************************************************************************/

Employee :: Employee (long EmpID,
                      char* EmpName,
                      char* EmpSex,
                      float EmpSalary)
{
  Employee_ID = EmpID;                              /* Assign the Employee ID to the Object.          */

  Employee_Name = new char[strlen(EmpName)+1];      /* Allocate Space for        */
  strcpy(Employee_Name,Emp_Name);                   /* the Name.                 */

  Employee_Sex = new char[strlen(EmpSex)+1];        /* Allocate Space for        */
  strcpy(Employee_Sex,EmpSex);                      /* the Employee's Sex.       */

  Employee_Salary = EmpSalary;                      /* Assign the new Employee's Salary.*/
}
```

```
/*****************************************************************************************************
          THE CODE OF THE MAIN FUNCTION OF THE PROGRAM.
*****************************************************************************************************/

main()
{
  Employee  *the_emp;  /* A Variable for creating the new Employee Object. */

  /* Create a new instance of the Employee Class. */

  the_emp = new Employee ( "12345",
                           "Padraig Moran",
                           "Male",
                           20000);

  /*  Now Print out the individuals of the New Object. */

  printf(" ID : %s\n", the_emp->ID());
  printf(" Name : %s\n", the_emp->Name());
  printf(" Sex : %s\n", the_emp->Sex());
  printf(" Salary : %s\n", the_emp->Salary());

}
```

Appendix B

This is the modification of the Employee class program given in Appendix A to cater for ONTOS. The object created is added to the database, so adding persistence to the program.

```
/*************************************************************************************************
 *
 *        Program Name :  CxxDEMO.cxx
 *   Date :         April 1991
 *   Description  :  This program creates a class called employee.  It
 *    is not inheritted from any other class.  It includes details on
 *    Employees.  The program simply creates the class and requests the
 *    the user to enter the details of an Employee.  The newly class would
 *    be compiled to the database.  In addition, the new object is written
 *    to the database also.
 *
 *************************************************************************************************/


class Employee
{
  long      Employee_ID;                 /* The Employee's Number.   */
  char*     Employee_Name;               /* The Employee's Name.     */
  char*     Employee_Sex;                /* The Employee's Sex.      */
  float     Employee_Salary;             /* The Employee's Salary.   */
 public:
  Employee(long, char*, char*, float);              /* Method to create an Employee. */
  long    ID();       {return(Employee_ID);}       /* Get the ID No.   */
  char*   Name();  {return(Employee_Name);}    /* Get the Name.*/
  char*   Sex();     {return(Employee_Sex);}      /* Get the Sex.*/
  float   Salary();  {return(Employee_Salary);}   /* Get the Salary. */
};



/*************************************************************************************************
          THE CODE OF THE EMPLOYEE CONSTRUCTOR METHOD.
 *************************************************************************************************/

Employee :: Employee (long EmpID,
                       char* EmpName,
                       char* EmpSex,
                       float EmpSalary)
{
 Employee_ID = EmpID;                            /* Assign the Employee ID to the Object.  */

 Employee_Name = new char[strlen(EmpName)+1];    /* Allocate Space for         */
 strcpy(Employee_Name,Emp_Name);                 /* the Name.                  */

 Employee_Sex = new char[strlen(EmpSex)+1];      /* Allocate Space for         */
 strcpy(Employee_Sex,EmpSex);                    /* the Employee's Sex.        */

 Employee_Salary = EmpSalary;                    /* Assign the new Employee's Salary.*/
}

/*************************************************************************************************
          THE CODE OF THE MAIN FUNCTION OF THE PROGRAM.
 *************************************************************************************************/

main()
{
```

```
/* Open the Database into which Employee Class is to be inserted. */

OC_open("EmployeeDB");            /* Database is called EmployeeDB    */
OC_startTransaction();            /* Start the transaction.           */

Employee  *the_emp;  /* A Variable for creating the new Employee Object. */

/* Create a new instance of the Employee Class. */

the_emp = new Employee ( "12345",
                         "Padraig Moran",
                         "Male",
                         20000);

/* Now compile the new Class to the database &        */
/* write out the class (deactivate).                  */

the_emp->Compile();
the_emp->putObject();


/* Now Print out the individuals of the New Object. */

printf(" ID : %s\n", the_emp->ID());
printf(" Name : %s\n", the_emp->Name());
printf(" Sex : %s\n", the_emp->Sex());
printf(" Salary : %s\n", the_emp->Salary());

/* Commit the Transaction to the database & close it. */
OC_transactionCommit();
OC_close();

}
```

Appendix C

This program creates an Employee Class, but it is done at run-time. The class is created dynamically and unlike the programs in Appendices A and B, there is no C++ definitions of the classes.

```
/**********************************************************************************************************
 *
 *         Program Name :  CxxDEMO.cxx
 *   Date :          April 1991
 *   Description  :  This program creates a class called employee.  The
 *    class is created dynamically.
 *
 **********************************************************************************************************/

/**********************************************************************************************************
           THE CODE OF THE MAIN FUNCTION OF THE PROGRAM.
 **********************************************************************************************************/

main()
{

 /*  Open the Database into which Employee Class is to be inserted.  */

 OC_open("EmployeeDB");  /* Database is called EmployeeDB      */
 OC_startTransaction();      /* Start the transaction.                 */

 /* Create the new Employee Type. */

 Type*  pEmp_type = new Type("Employee",       /* The name of the new Class              */
                                          "Object");          /* The SuperClass.               */

 PropertyType* pEmpID = new PropertyType("Employee_ID",           /* Prop. Name   */
                                                      OC_long,                    /* Prop. Domain */
                                                      pEmp_type);                 /* Class Ptr.   */
 PropertyType* pEmpName = new PropertyType("Employee_Name",
                                                           OC_charPtr,
                                                           pEmp_type);
 PropertyType* pEmpSex = new PropertyType("Employee_Sex",
                                                         OC_charPtr,
                                                         pEmp_type);
 PropertyType* pEmpSalary = new PropertyType("Employee_Salary",
                                                           OC_float,
                                                           pEmp_type);

 /* Compile the new Datatype to the Database. */

 pEmp_type -> Compile();

 /* Write out the New Class & it's associated properties to the database. */

 pEmp_type -> putObject();
 pEmpID -> putObject();
 pEmpName -> putObject();
 pEmpSex -> putObject();
 pEmpSalary -> putObject();

 /*  Commit the Transaction to the database & close it. */

 OC_transactionCommit();
 OC_close();

}
```

Appendix D

The following is a sample C++ program which creates an ONTOS query, executes against the TESTDB database and prints out the results.

```
/*****************************************************************************************************
*
*         Program : Sample.CXX
*         Descr.  : This program creates a simple ONTOS OSQL query.  This is executed
*                   against the TESTDB database.  The results of the database are
*                   returned and displayed.
*
*****************************************************************************************************/

#include <stdio.h>            // Standard I/O functions.
#include <Exceptio.h>         // ONTOS Exception class declarations.
#include <Object.h>           // General ONTOS header files.
#include <Database.h>
#include <QueryIte.h>         // Header file for OSQL functions.
#include <Director.h>         // Header file for directory management.
#include <string.h>           // String manipulation functions.


main()

{
  char  query[200];
  char  output[500];

  OC_open("TESTDB");        // Open the database & start the
  OC_transactionStart();    // transaction.

  //
  //  Place the query text into the query variable.
  //
  strcpy(query,"SELECT e.Name, e.Age FROM Employee e WHERE e.Age > 25;");

  OC_startQuerySession();  // Start the Query process rolling.

  QueryIterator     *iter;                 // Object to process the query.
  ExceptionHandler  ahandler;              // Create Exception handler for SQL parsing.

  if (ahandler.doesNotOccur())
       iter = new QueryIterator(query);              // execute the query.
  else
          {
           printf(" Error occurred during parsing\n");
           exit(0);
          }

  //
  // Now return the results from the query.
  //

  while (iter->moreData())
          {
           iter->yieldRowString(output,499);         // extract the rows as strings
           printf("%s\n",output);                    // and print them.
          }


  //
  // Now return to the beginning of the list and count the number of rows returned.
  //
```

```
    iter->Reset();
    while (iter->moreData())
              {
               ArgumentList *arglist = iter->yieldRow();
               k++;
              }

    printf("Cardinality is %d\n",k);
    delete iter;


    //
    //  End Query session, transaction and close database.
    //
    OC_endQuerySession();
    OC_transactionCommit();
    OC_close();
}
```

Appendix E

IBM OS/2 & Presentation Manager

Introduction

Developed by IBM and Microsoft as a successor to PC-DOS (MS-DOS), OS/2 is an operating system for small computers based on the Intel 80286, 80386 and subsequent microprocessors. OS/2 uses the unprotected mode of the 80286 microprocessor to unleash the 16MB address space of the 80286 and implement efficient and safe multitasking.

Version 1.0 of OS/2 was launched in December 1987. This consisted of just the OS/2 kernel. The kernel is a traditional environment for both users and programmers. The command line interface and most internal and external commands were inherited from DOS. However, the similarity ended there. From a programmers viewpoint, much of the functionality of the OS/2 kernel resembled DOS, UNIX and traditional minicomputer operating systems. The kernel handled file I/O, memory management and multi-tasking. The programmer's interface (API) included facilities for keyboard and mouse input and a fast full-screen character mode video I/O (VIO) system.

The OS/2 kernel supports multiple full-screen sessions. Each session runs one or more processes that use the video display in either a teletype or full-screen fashion.
In the end of 1988 when OS/2 Version 1.1 was released, it had Presentation Manager added to it, to provide a graphical environment to the operating system's power. In OS/2 1.1, one session is devoted solely to the Presentation Manager, and many different processes can be executed in this session.

Features of OS/2

OS/2 like any other operating system deals with managing the file structure of the system. It provides the standard facilities for interaction with the computer's hardware, while managing it. However, OS/2 offers a number of features which give it it's increase in power over DOS and other single-user operating systems.

Multi-tasking

Multi-tasking, one of the principal features of OS/2, is the ability of the system to manage the execution of more than one program at a time. This ability helps to optimise use of the computer, since time is normally spent by a program waiting for user input distributed to other programs that may be printing a document or recalculating a spreadsheet. OS/2 supports up to 16 concurrent sessions running concurrently. It also permits a single program to run more than one copy of itself, at the same time.

Every program that has been loaded into memory and is running is called a ***process***. Each copy of a process is called a ***thread***. A ***process*** owns resources such as file handles, queues, semaphores, threads and it's own memory map. A process always has at least one thread, called the main thread, and can create more threads. These additional threads are useful for carrying out tasks unrelated to the processing of the main thread. For example, a process may create a thread to read data in from a disk file. This frees the main thread so that it can continue to process user input.

The scheduling of the different threads is controlled by the OS/2 scheduler. This operates in the *Round-Robin* manner. A thread will execute until the scheduler preempts it, the next thread is then resumed, with the state of the preempted thread being saved.

Dynamic Linking

In DOS and other operating systems, when applications are developed, libraries of system functions used by the application are linked into the final application when it is being developed. This results in large applications taking up much of the computer's primary storage with code which may not be required at all during the current execution.
In contrast to this approach, most OS/2 programs use dynamic link libraries. Dynamic linking lets a program gain access at run-time to functions that are not part of it's executable code. These functions are contained in dynamic-link libraries. These are special programs modules which contain executable code but cannot be run as programs. Instead, programs load the appropriate dynamic-link libraries and execute the code in the libraries by linking to them dynamically.

Most of the OS/2 operating system is implemented in this manner, with just a small code kernel linking to the appropriate libraries as required.

The chief advantages of dynamic-linking is the reduction in the amount of memory required to execute a program. If a particular library is needed by a number of threads simultaneously, only one copy of the library will be loaded, thus cutting down on memory usage.

From a maintenance viewpoint, updates to a system can be distributed through new dynamic-link libraries replacing old ones, without the need for the updated application to be re-linked.

Memory Management

Programs can at any time allocate additional memory for their own. Once the system allocates the memory, a *selector* is passed to the program indicating the size of the segment of memory received. This selector is then used to access the memory.

The process that allocates memory owns it, and no other process can access it. Any attempts to do so by other processes will result in a *protection violation* error and terminate that process. Processes can share memory, through two possible methods. One process can pass the selector of the memory segment to the sharing process, or pass the name of the shared segment to that process. In this situation, the *protection violation* error cannot occur for the sharing processes, but all other processes are locked out from use of this memory.

OS/2 implements *Virtual Memory*. The system has a large possible logical address space as mentioned above but regularly the physical memory will not be as large. OS/2 provides *staging* to allow secondary storage to be used as an extension of main memory. This adds to the power of the system, allowing large applications to execute on systems which have less primary storage than is required for execution of the program. However, the use of virtual memory may result in a substantial degradation of the performance of the system, as data needs to be loaded into and out of memory from the hard disk.

Interprocess Communication

With the provision of multi-tasking in OS/2, a number of methods are also provided to allow interprocess communication :

Semaphores
Pipes
Signals
Queues

Semaphores

A semaphore is a special variable that a process can use to signal the beginning and ending of a given operation and to prevent more than one thread within the process from accessing a given operation and to prevent more than one thread within the process from accessing a specific resource at the same time.

Pipes
A pipe os a special file that two processes can use to transfer data. A pipe is not actually a disk file but is maintained by the system. The two processes using the pipe get handles. One gets the read handle, the other the write handle. One process uses it's write handle to write data to the pipe, the other uses it's read handle to read the data in the pipe.

Signals
A signal is a special interrupt that is sent to a process by the system or by another process. The signal temporarily stops normal execution of the process and causes the process to execute a signal handler. This handler can be used to determine the execution of a particular operation when the signal is raised.

Queues
A queue is a special buffer that a process creates and shares with other processes. A queue is a convenient way for one process to channel data from two or more related processes into a single buffer. Any thread in the system can write to the queue, but only the creator can read from it, or carry out other operations on it (purge/delete).

The File System
Disk files in OS/2 are treated in the same manner as other devices. The same functions can be executed on files as on, for example, the serial staff. Each open file or device is assigned a handle, by which it is accessed. OS/2 programs can create, delete, open, move, and delete files and directories in the file system. When a file is opened, a process must specify if it is to be shared. This sharing, also applies to devices being used. Version 1.2 of OS/2 provided two types of file systems from which the user can choose one when the operating system is being installed.

File Allocation Table System (FAT)
This is the filing system implemented by DOS also. The file allocation table is a map of how space is utilised in the files area of a disk. The organisation of the FAT is simple: There is

one entry in the FAT for each cluster in the files area on the disk. If a FAT entry is not is not marked as unused, reserved or defective, then it corresponds to a cluster which is part of a file, and the value in the FAT entry itself indicates the next cluster in the file. The space on the disk that corresponds to a file is mapped by a chain of FAT entries.

From a programming or usage viewpoint, the file structure consists of a directory structure emanating from the root directory (\). Files can have an 8 character name with a 3 character type descriptor.

High Performance File Storage System (HPFS)

HPFS is an installable file system (IFS) designed to provide better performance than the existing file allocation table (FAT) based file system. HPFS is designed to provide extremely fast access to very large disk volumes. The structure of the system is such that is can support the coexistence of multiple, active file systems on a single personal computer, with the capability of multiple and different storage devices. Both FAT and HPFS support the same naming conventions and the existing logical file and directory structure. Features of HPFS include:

- File names up to 254 characters in length
- Fast access to very large disk volumes
- Strategic allocation of directory structures
- Extended attribute support for files. Information on the file can be attached to a file, including originator, icon, description, etc.
- Caching of directories, data, and file system structures
- Large file support

The FAT table structure has the advantage that the disk drive can be shared between DOS and OS/2 applications, while the implementation of the HPFS system on a partition of the hard disk restricts it's use of that partition to OS/2 only.

Input-Output Facilities

For full-screen programs, not using Presentation Manager, OS/2 provides access to the keyboard, the mouse, and the video display. As mentioned above, these devices are treated very much in the way files are. Output of data to the full screen is a lot simpler than in Presentation Manager where the output is concentrated towards windows, and this involves it's own problems. The level of I/O support in the full-screen environment is more like that supported in DOS, providing basic operations with little restriction on their usage.

With the introduction of OS/2 Version 1.1, the Presentation Manager offered a new interface by which the features of the operating system could be accessed.

The Presentation Manager

The Presentation Manager is part of the OS/2 operating system, version 1.1 and higher. It provides a high-level interface to the underlying multi-tasking operating system. One of the principal goals of OS/2 is to provide visual access to most, if not all, applications at the same time [MIC89a]. Presentation Manager (PM) provides a friendly, simple and efficient of fulfilling this aim. When OS/2 boots up initially, it creates a PM session. This session controls the screen. All applications that run in this session share the same screen and are known as Presentation Manager Applications. These applications will tend to be executed in windows. However if the application requires it, it can create a new session of it's own which will allow it to use a separate full screen for it's execution. This type of application is called a Full-Screen Application.

A PM application shares the display with other applications by using a "*window*" for interaction with the user. In keeping with the ideas outlined in Chapter 3, a window in PM terms, is a rectangular portion of the system display that the system grants to the application. However, a window is also a combination of visual control devices, such as menus, controls, scroll-bars, with which the user directs the actions of the application. Once an application has created one or more windows, OS/2 provides the application with detailed information about what the user is doing with the window and automatically carries out many of the tasks the user requests, such as moving and sizing the window. In addition, as many applications may have many different windows displayed at the same time, the system also needs to manage the placement of windows, ensuring that two applications do not attempt to access the same portion of the screen at the same time. Because of the nature of OS/2 applications in different windows can be multi-tasked, therefore strict management of these windows is essential.

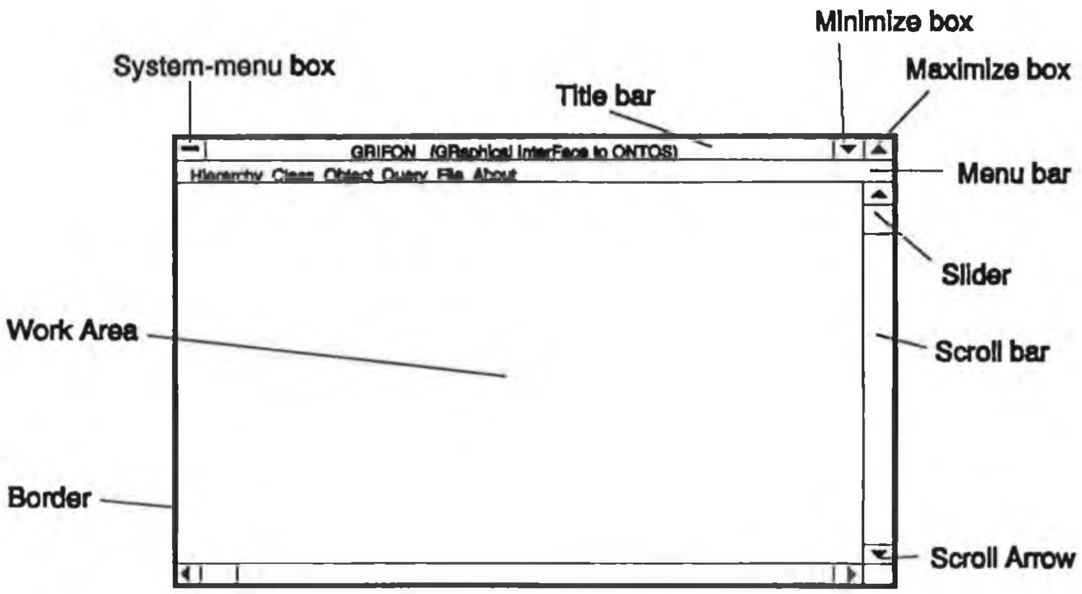
For the application programmer, OS/2 provides facilities to allow the creation and management of windows and related elements. These functions are carried out by the ***Window Manager***.

The Window Manager

The OS/2 Window Manager consists of system functions that let applications create and manage windows and related elements. These related elements are primarily menus, dialog windows (for input/output), controls, and the window management facilities. The window manager provides the elements that your applications need to construct a graphical user interface.

Windows

Windows are the primary input and output device of any PM application. It is the application's only access to the system display, so, since nearly all PM applications interact with the user in some way through the system, these applications must use windows. A typical window is composed of a title bar, a menu-bar, scroll bars, borders, and other features You list the features you want for a window when you create the window. Although an application creates a window and technically "owns" it, the management of the window is actually a collaborative effort between the application and the system. The system maintains the position and appearance of the window, manages the standard window features such as the border, scroll-bars, and title, and carries out many tasks initiated by the user that directly affect the window. The application maintains everything else about the window - like what is to be displayed. A sample window is displayed below highlighting it's various basic components.



A Typical Presentation Manager Window.

Menus

Menus are the principle means of user input for a Presentation Manager application. A menu is a list of commands that the user can view and choose from. When the application is developed, the programmer supplies menu and command names. OS/2 manages the menu itself. When a selection is made, control is directed to a procedure to deal with executing the appropriate application procedure.

Dialog Windows

A dialog window is a temporary window that can be created to let the user supply more information for a command. Dialog windows contain one or more controls. A control is a small window that has a very simple input or output function. The controls in a dialog window give the user a means of supplying filenames, choosing options, and otherwise directing the action of the command. Buttons, list-boxes and entry-fields are all types of controls. For example, and entry-field control lets the user enter and edit text.

Window Management Facilities

The window manager provides all of these components for application interface creation. In addition it also provides control mechanisms for detecting and responding to changes in the window configuration on the screen. For example, if a window is moved and partially covers another window, the window manager responds to this by preserving the contents of the underlying window and restoring them when necessary. Similarly, it reacts automatically to buttons being pressed, items being selected from lists or data being entered into edit-lines in dialog boxes by issuing an appropriate message to the application, allowing it to respond accordingly.

From a programmers viewpoint PM is extremely powerful through the library of program accessible functions for accessing all facets of presentation manager. In particular the provision of the ***graphics programming interface (GPI)*** is very useful.

The Graphics Programming Interface

The graphics programming interface consists of the OS/2 system functions that let you create device-independent graphics for your applications. The GPI functions are used in conjunction with the window manager to draw lines, shapes, and text in windows. Mathematical functions are provided to facilitate the drawing of complex shapes and diagrams with the minimum of effort in windows. The functions provided by the PM API toolkit as the GPI are listed in Appendix K.

Presentation Manager, and it's inclusion on top of a powerful multi-tasking operating system like OS/2 is important from a number of points of view.

- **Choice of Interface**

With the inclusion of a graphical interface onto a powerful text-based operating system, programmers now have a choice of developing applications for either the OS/2 kernel or the OS/2 Presentation Manager. Each environment has distinct advantages and disadvantages. For purely text-based applications or converted PC-DOS applications the OS/2 kernel is preferable. It will generally be simpler to develop applications for the kernel and if they are converted from PC-DOS, the conversion will be a lot more straight forward with concern for the presentation manager. In addition, as the OS/2 kernel is text-based, applications will tend to execute faster than they would if developed in PM as it is solely graphics based.

However, for many sophisticated applications, particularly those that use graphics, the

Presentation Manager is clearly the better environment.

■ **The Graphical Environment**

The graphical environment of the Presentation Manager is rich in functionality. Programs can use graphics and formatted text to convey a high density of information to the user. A traditional program gets user input from the keyboard and displays output to the screen. But with the addition of a mouse, the screen itself becomes a potential source of user input. Logic within the Presentation Manager assists the application in obtaining user input from various controls on the screen, such as menus, scroll bars, buttons, and dialog boxes. The interaction between the mouse and the screen narrows the gap between user and program.

■ **The Consistent User Interface**

Many different applications have appeared in PC-DOS, sporting fancy windowing interfaces. One major problem has existed with these in their lack of consistency from application to application, and indeed from version to version of the same application. Because the menu and dialog box interface is built into the Presentation Manager rather than into each individual application, the interface is consistent across applications. This means that a user with experience with one PM application can easily learn a new PM program.

Fears have been expressed that such uniformity of application appearance will lead to program designer's creativity being inhibited. However, the only restriction being put on the designer is facilities being provided with which to interact with the user. This restriction is more than balanced out by the extent of the range of functionality provided by the PM programmers interface.

■ **Device Independent Graphics**

With the advent of different graphics standards like CGA, EGA, VGA, Hercules etc., conventional applications have regularly needed to know the graphical environment for which they were being designed. This no-longer exists with Presentation Manager. The programmer can develop an application without ever considering the video technology employed on the machine. If the application is subsequently executed on a computer sporting different graphics capabilities, then Presentation Manager on this machine will deal with executing the application correctly.

- **Systems Application Architecture**

Systems Application Architecture (SAA) is an ambitious plan by IBM to set user interface and Applications Programming Interface (API) standards across much of their line of computers. The Presentation Manager is one of the first products to be a part of SAA. If the goal of SAA comes to pass, then the Presentation Manager user interface will become a common sight on IBM minicomputer and mainframe terminals. Just as important for the program developer, it may one day be possible to write a Presentation Manager program in a high-level language and compile it to run on a variety of computers from the IBM PS/2 to the IBM 370.

Conclusions

OS/2 as an operating system is powerful, offering the facilities and features outlined above. The inclusion of a user-friendly interface to it through Presentation Manager makes applications simpler to use. Yet Presentation Manager is more than just a *pretty face*. It offers the user simple and easy access to the power of OS/2. To the programmer, it offers a rich library of hundreds of functions to deal with controlling the windowed environment and creating impressive graphical interfaces simply, without concern for the hardware being used. The PM style of interface has recently been adopted for Microsoft Windows, a PC-DOS based environment, which has proved a huge success, indicating it's favour with users.

IBM's choice of Presentation Manager as the interface for it's SAA plans indicates the potential importance of it in the future of personal computers and the not-quite-personal computers as well.

Appendix F

C++ Programming Language & CommonView Class Library

Introduction

The programming language C was developed in 1972 by Dennis Richie of Bell Laboratories. It was initially designed to be a systems language for the UNIX operating system. The initial version of UNIX was developed by Ken Thompson in assembler and the B programming language. B was a programming language based on BCPL[1]. C evolved from B and BCPL and incorporated typing [KEL84]. C, as it is now, is a mature general purpose language, having evolved from these roots. As software complexity has developed over the past number of years, a number of problems associated with C appeared. C does not carry out very tight type-checking. In addition, it encourages the use of pointers to data structures in memory. These prove efficent when used correctly, but in large applications, their use can tend to get out of control [GLO89a]. In addition to these, C like most other third generation programming languages is not very semantically powerful. It does not readily support the modelling of the problem space easily.

These and other features of C spurred Bjarne Stroustrup, at AT&T, to develop a successor to C, he called this C++, applying the C increment operator, '++', to name of the original language. C++, provides solutions to many of the problems associated with C, and in addition adds object-oriented concepts and structures to the language [STR86].

In the remainder of this section, I will take a brief look at C, outlining it's main features. I will then outline the C++ programming language, describing how it builds on C's features. C++ implements a number of new programming concepts. These will be examined, with reference being made to the way they are used to represent the problem space in a more realistic manner.

Windowed applications tend to be extremely difficult to develop, due to the element of

[1] BCPL was developed in 1967 by Martin Richards. It was a typeless systems programming language, with it's basic data type being the machine word. It made heavy use of pointers and address arithmetic.

uncertainty about the sequence of steps the user will take. C++, due to it's structure, can help make the development of such applications easier and the resulting systems more structured. I will look at how C++ can be used in these situations.

C - The Programming Language

C is a small language with fewer keywords that Pascal, where they are known as reserved words, yet it is arguably the more powerful. C is a small language and a compiler for it can be coded in under 10000 lines of C code.

C is the native language of UNIX, possibly the most used, multi-user interactive operating system available. A language does not gain popularity on its own merit. It is the system environment that is the hidden secret of a language's success. For example, C does not need to have embedded input/output constructs or complicated interrupt handlers, but instead relies on library routines for these functions.

C is portable. This is by virtue of being small and initially being developed on a small machine, a Digital (DEC) PDP-11. The code of C is readily tailored to a new host machine. Due its size and construction, a C compiler can be booted to a new system in a matter of months [KEL84]. In addition, its construction, with system utilities and the preprocessor allow the programmer to isolate possible machine dependencies outside of the main code. This makes for easy redefinition from one to another C system.

C is terse. It has a powerful set of operators. Many of these indicate the personal taste of its designers and what was available on its original environment. The increment operator, ++, has a direct analogue in PDP-11 machine language, the original development environment. Indirection and address arithmetic can be combined within expressions to accomplish in one statement or expression what would require many statements in another language. For some this is elegant, for others it is obscure. Software productivity studies show that programmers can produce, on average, a small amount of working code a day. A language that is terse explicitly magnifies the underlying productivity of its programmer. [KEL84]

C is modular. C supports one style of routine, the external function, which calls parameters by value. It does not allow function nesting. C does allow limited forms of privacy by using the storage class static within files. These features, along with the typical UNIX environment, readily support user-defined libraries of functions and mocular programming.

On the negative side, C is not without its criticism. It is not as strongly typed as other recent programming languages. It allows the compiler to reorder evaluation within expressions and parameter lists. It has no automatic array bounds checking. It makes multiple use of such symbols as * and =. For example, a commonn programming mistake is using the operator = in place of the operator ==.
Even taking this into account, C is an elegant language. It places no straitjacket on the programmers access to the machine. The imperfections it has are easier to live with than a perfected restrictiveness, as is more evident in other languages.

C contains the following features :

- **Modularity**
  C provides a user-defined function capability. Many of the features provided by C for input/output are themselves system provided functions. Parameter passing is pass-by value, but if variable parameters are to be passed, this can be done by passing the address of the variable concerned. Any change to this variable, in the function will obviously be reflected in the actual variable.

- **Iteration Constructs**
  Four iteration constructs are provided in C -
  - repeat .. until,
  - while do ..,
  - do .. while,
  - for ..

  In addition, resursion[2] can be used.

- **Basic Datatypes.**
  C provides only three basic datatypes on which extra ones can be built. These are **char, int** and **float**. Qualifiers are applicable to these to allow extra features. For example, **long** is used to qualify **int** to store long integers. Similarly numbers can be

[2] Recursion means defining a problem in terms of itself. In C this involves getting a function to call itself, splitting the problem into smaller identical problems.

signed or **unsigned**. Strings of text can be represented as characters pointers - **char ***.

Structured datatypes can be constructed, as a combination of other types - in record format. These new structures can be defined as new types and used in the same place as other primitive datatypes. Arrays can also be created of all available datatypes.

- **Dynamic data structures.**
  C is powerful in its ability to allocate memory to variables. Dynamic data structures can be created with pointers being used to reference them. The advantages of this are clear - better use of memory and better control over resources. However, very tight control needs to be kept on the memory and the pointers to ensure that the correct areas are being accessed or modified. Operators are provided allowing the program to access the memory contents, specifying the address.

- **Operations**
  C provides a minimal amount of operators itself - basic arithmetic operations, incremental operations, relational and logical operators. Trigonometric or other functions are provided by external libraries. Text operators are also included in libraries, to be used if required.

- **Extensibility.**
  Much of C's power comes from its extensibility. The kernel of the language consists of 32 keywords. The functionality of the language comes from its facility to extend itself through the inclusion of library functions to carry out most tasks. This makes C very customisable to an individual application's needs, being appropriate for the development of a wide variety of appliations through the provision of a wide variety of libraries.

It is true to say that much of C++'s power comes from its close association with C. C++ provides all the facilities of C, but attempts to allow the user to access them in a more structured manner. It provides for the structuring of the program to better represent the problem space.

C++ - Object-Oriented C !

C++ is an enhanced version of C. It tries to make up for the inadequacies of its predecessor. In doing this it also adds some new functionality. Some of the features are :

- **Variable declarations where used.** Variables no longer need to be declared at the top of the code block, but can be declared where required.

- **A *constant* datatype is provided.** Constants no longer need to be declared using the **#define** preprocessor instruction[3], but can now be declared in the program code, in the same manner as variables. They are scoped[4] like variables, unlike preprocessor constants.

- **Function argument checking and conversion.** Checks are made on the number of parameters actually passed to a function, to ensure that there are the same number as the function expects. In addition, an attempt is made to convert parameters to their correct type. For example, if the function expects a long int, and an int is passed to it, then this is converted automatically. These two things would have caused an error in C.

- **In-line functions.** Functions can be declared as in-line, to increase performance, this replaces the #define of one-line functions in C, which is still available.

- **Structures can have functions.** In C structures can be composed of variables. In C++ functions can also have functions as their members. In addition, once a structure is defined, it automatically becomes a new data-type in the program. This would have to be done explicitly in C.

- **Classes provide data-hiding.** Classes are special types of structures in C++. The members (variables) of the class can be made private, preventing access to them by outside modules. Functions can be associated with the class which are the only

[3] Preprocessor constants are defined external to the program. They associate a value or operation with a global identifier.

[4] The scope refers to the area in a program where a variable has meaning. For example, a constant declared inside a function is only accessible inside that function.

method of accessing or manipulating the private members - hiding the data.

For example in the following class, the variables are only accessible through the provided functions, which are declared under the public section :

```
class Employee
{
        char*          Name;          // Employee's Name (Text)
        char           Sex;           // Employee's Sex (Char.)
        short          Age;           // Employee's Age (short int)
        float          Salary;        // Employee's Salary (float)
 public:
        Employee(char*, char, short, float);  // Constructor
        char* Name();                         // return Name
        void IncreaseSalary(float);           // Inc. Salary
        void DisplayDetails();                // Disp.Emp.Details
};
```

The Constructor function is used when a new object of this class is created. It must have the same name as the class.

- **Classes can be inherited.** Classes can be defined which inherit the members and functions from another class. These can be overridden by new members or functions of the same name. For example, class Manager is defined as a sub-class of Employee:

```
class Manager : public Employee
{
        short          officeNo;      // Manager's Office Number
        long           telephone;     // Manager's telephone number
  public:
        Manager(char*, char, short, float, short, long);
        void DisplayDetails();
};
```

Effectively class Manager has six member variables, the two new ones and the four inherited ones. The member function DisplayDetails() defined in this class over-rides the one in Employee.

Of the six parameters in the Manager constructor function, four of these will be passed to the constructor for the Employee class and the last two will be applied to the Manager class. The four parameters are passed back up to Employee through the following declaration of the

Manager constructor function :

```
Manager :: Manager      (char* nam,
                         char sex,
                         short age,
                         float sal,
                         short off,
                         long tel) : (nam, sex, age, sal)
```

The four parameters in the brackets after the ':' are those which are passed up to the constructor of the super-class.

The piece **Manager :: Manager** indicates that the function called **Manager** (the 2nd Manager) is a member function of the class **Manager** (1st Manager).
Similarly the first line of the function DisplayDetails() of the Manager class would be written as :

```
void Manager :: DisplayDetails()
```

- **Free Storage Management is provided.**
  C provides library functions like *malloc()* and *free()* to allocate and deallocate memory in programs. Pointers are used to manage the memory allocated. However freeing memory which is not allocated or has already been deallocated can cause very unusual effects to occur.

  C++ provides new operations to allocate and deallocate memory - ***new*** and ***delete***. The function ***new*** allocates memory for an object of the class and calls the constructor to initialise the new object's member variables. The function ***delete*** calls the destructor[5] function for the class and then deallocates the memory associated with the object. An object of class Manager might be created as follows :

```
Manager   *mgr = new Manager ("Padraig Moran",
                              'M',
                              24,
                              14000.00,
                              218,
                              5363);
```

[5] A destructor function is the opposite to a constructor. Where a constructor initialises the new object, the destructor can be used to clean up the member variables before the memory is deallocated.

The pointer **mgr** is declared and initialised on the same line. The **new** function calls allocates the memory to hold a Manager object and assigns the member variables of this object the parameters given. Finally **mgr** is set to point to this object in memory. The power of C++ allows all of this to be done in a single line. To a similar operation in C would take substantially more.

- **Virtual functions implement *Polymorphism.***
  In the inheritance hierarchy from class to class, member functions can override ones higher up in the hierarchy. If the function in the highest level class is declared with the modifier **virtual** then it facilitates the idea of polymorphism being applied to this function. This is easily seen from the following example:

  Imagine a program with a base class called **Polygon** and a couple of derived classes called **Square** and **Triangle.** Each type of polygon knows how to **Draw** itself, so a **Square** will draw a square and a **Triangle** will draw a triangle and so on. The following code :

```
void DrawShapes()
{
        Polygon  *p[4];

        p[0] = new Square;
        p[1] = new Triangle;
        p[2] = new Square;
        p[3] = new Triangle;
        for (int j=0; j<4; ++j)
                p[j]->Draw();
}
```

  has the effect that each polygon knows its own type and draws itself correctly. This is only true if **Draw** is declared **virtual** in **Polygon.** Otherwise **Polygon::Draw** is called in each case, because **p[j]** is of type **Polygon *.**

Through the above features C++ successfully implements the main object oriented features.

Programming using objects can result in large quantities of data in the system. To manage objects, C++ provides the idea of containers. Most programming languages implement the idea of containers to group similar objects together. BASIC supplies arrays, C and Pascal offer arrays and pointers by which linked lists can be constructed. C++ provides the Container class. In true object oriented spirit, access to a container object is through the member functions provided by it.

C++ with its underlying C programming language and its object oriented facilities offers a new development environment for all types of applications. It provides a cross between current procedural programming development techniques and object-oriented techniques and offers a facility to realistically represent the problem space.

One area where C++ can be applied to both accurately the application domain and in the process simplify the application is in the development of windowed environment applications.

C++ and Windows

The development of windowed applications presents a number of problems for programmers:

(i) Windowed programs are inherently more difficult to develop than old style file based applications. No longer is the system just concerned with the user's input to the application. It also needs to consider the user's interaction with the windowed interface itself.

(ii) Windowed systems are more difficult environments in which to debug software, because of the interaction between the observer and the observed. In other words, the system never knows what the user will do next. Windowed applications are event driven, with the user determining what happens rather than the system offering the user a small set number of options to choose from, in a set sequence.

(iii) Products which use windows are naturally more volatile in their use of the system's resources, such as memory, CPU and disk. To work efficiently, windows environments require radically improved heap management.

(iv) Windowing software tends to spawn large numbers of inter-communicating processes, requiring message management and control of shared resources.

Although the facilities provided by the operating system may help in solving some of the problems outlined above, C++ seems to be suited to windows applications due to the following reasons :

(i) If you implement using strong data types, C++ finds a much higher propertion of errors at compile time than C does. Put another way, if your program passes C++ syntax checking there is a far greater chance that it will executely correctly.

(ii) Object-oriented programming fits naturally into a windows environment. Objects in the human interface suggest the functionality of software objects which implement them.

(iii) The aspects of windows applications which are difficult for C progammers can be implemented as services[6] in C++. Re-expressing the management of complex resources in terms of services contains the complexity within small areas of your design.

CommonView

Before the arrival of window systems, printing a message on an output device, such as a screen, was a simple matter. Functions in C's standard input/output library made it possible to access the hardware directly. Programs employing these functions were command driven where the application followed a predetermined sequence, pausing in execution to accept input from the user.

Window environments change all that. Multiple screens (windows) are possible, each of which can differ in size and location relative to the physical screen. Each window has its oen canvas area, where it displays its output. The window is responsible for its own canvas area, preventing output from going outside it, facilitating size changes of this area, and allowing the contents of the window to be scrolled.

In addition to these changes, the structure of a typical program also changes. Window programs are event driven. It is the user who directs the course of the program. Multiple

[6] To design a service in C++, you simply decide on the functionality of each object used to provide the service. You then express the functionality in the very concise struct declaration syntax of C++.

windows may be opened, menu options may be selected and data entered in a sequence that cannot be predicted by the program.

Traditional C library functions are not enough to cope with the complexities involved. Thus the API was developed to provide a new set of tools to create programs in a windows environment.

However, the number and complexity of functions available, compared to traditional C libraries, is enormous. These functions, in turn, can take a large number of arguments, many of them structures with perhaps 10 members needing to be initialised.

All of this adds to the complexity of windowing programming and places a heavy burden on the application developer - to the extent that more time is spent on coding than on design. To make matters worse, window environments differ across machines and operating systems, each requiring its own API for application development.

This lack of standardisation prevents portability and encourages the application developer to develop the code to work on one platform only, rather than seek a standardised user interface for the program.

CommonView, using object-oriented techniques, solves the twin problems of portability and code complexity. Its class library encapsulates all that is necessary to produce applications that are portable across different presentation systems.

From a developer's persepctive it means it is no longer necessary to understand the underlying mechanism of different window environments to produce a portable application. Once the principles and the practice of CommonView have been grasped, the developer can write an application without having to take account of the environment in which the application will ultimately run.

Windowing environments are all made up of a similar set of components, although they may appear different in the individual environments. CommonView exploits this commonality to produce the implementation independent class libraries. Most WIMP interfaces consist of Windows, Controls like buttons, lists, scrollbars, etc., Dialog Boxes, Edit fields, and Menus.

These are the primary components of a windowing environment, and consequently the primary classes in CommonView. The following is a list of the classes from the CommonView V1.1 class library :

- **Accel**
- **App**
- **Bitmap**
- **Brush**
- **Button**
  - **CheckBox**
  - **PushButton**
  - **RadioButton**
- **Color**
- **Control**
  - **FixedIcon**
  - **ScrollBar**
    - **HorizScrollBar**
      - **WndHorzScrollBar**
      - **WndVertScrollBar**
    - **VertScrollBar**
  - **TextControl**
    - **Edit**
    - **MultiLineEdit**
    - **SingleLineEdit**
    - **FixedText**
    - **ListBox**
      - **FileListBox**
- **Cursor**
- **DrawObject**
  - **LineObject**
  - **ShapeObject**
    - **EllipseObject**
    - **RectangleObject**
  - **TextObject**
- **Event**
  - **ControlEvt**
  - **ExposeEvt**
  - **FocusChangeEvt**
  - **KeyEvt**
  - **MenuCommandEvt**
  - **MenuInitEvt**
  - **MenuSelectEvt**
  - **MouseEvt**
  - **MoveEvt**
  - **ReSizeEvt**
  - **ScrollEvt**
- **EventContext**
  - **Window**
    - **AppWindow**
      - **ChildAppWindow**

```
            TopAppWindow
        ControlWindow
            EditWindow
        DialogWindow
            ModeLessDialog
Font
Icon
Menu
    SysMenu

MessBox
    ErrorBox
Pair
    Dimension
    Point
    Range
    Selection
Pen
Pointer
Rectangle
ResString
```

The provision of general classes such as Window, act as a springboard on which more specific classes can be constructed.
To deal with the event driven problems associated with windowing applications, an Event class is provided with methods which act automatically in response to the events in the application. The sub classes of Event, provide reactionary methods for all possible events - mouse movements, text entry, menu selections, keyboard entry and many other events.

The application of these classes in a C++ program is simple, as can be seen in the program included in Appendix I. The facilities provided by CommonView are complete for most windowing applications, however, some extra facilities which are provided by the underlying environment cannot be offered directly through the CommonView class library. These extra features can easily be incorporated into a C++/CommonView program because of CommonView's provision of a *Handle* attribute for all its windows and controls. This attribute is the link between the CommonView programs and the PM API. The handle allows the use of PM API features in CommonView programs. This can be seen in Chapter 5, where GPI functions of the PM API are used to draw objects on a window, created in a CommonView program.

CommonView has been developed for virtually all the available windowing environments, including : IBM/Microsoft OS/2 Presentation Manager

Microsoft Windows 3.0

and many X-Windows derivatives

As Glockenspiel C++ is fast becoming the defacto standard version of the AT&T C++ standard, it is becoming available on more and more platforms. Similarly where C++ is available, CommonView facilities are also being provided.

Appendix G

This program is written in C and makes calls to functions in the Presentation Manager API. It creates a window on the screen, and displays the message - Hello World in the window.

```
// ****************************************************************************************
// *
// *        This program displays a window and the message
// *                    Hello World.
// *
// ****************************************************************************************

#define INCL_WIN
#include <OS2.H>

MRESULT EXPENTRY ClientWndProc (HWND, USHORT, MPARAM, MPARAM);

int main(void)
    {
            static CHAR         szClientClass [] = "Example Program";
            static ULONG        flFrameFlags =       FCF_TITLEBAR     | FCF_SYSMENU |
                                                     FCF_SIZEBORDER| FCF_MINMAX |
                                                     FCF_SHELLPOSITION | FCF_TASKLIST;

            HAB      hab;
            HMQ      hmq;
            HWND     hwndFrame, hwndClient;

            hab      = WinInitialize(0);
            hmq      = WinCreateMsgQueue (hab,0);

            WinRegisterClass (    hab,
                                  szClientClass,
                                  ClientWndProc,
                                  CS_SIZEREDRAW,
                                  0);

            hwndFrame = WinCreateStdWindow (
                                 HWND_DESKTOP,
                                 WS_VISIBLE,
                                 &flFrameFlags,
                                 szClientClass,
                                 NULL,
                                 0L,
                                 NULL,
                                 0,
                                 &hwndClient);

            WinSendMsg( hwndFrame, WM_SETICON,
                          WinQuerySysPointer (HWND_DESKTOP, SPR_APPICON, FALSE),
                          NULL);

            while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
                      WinDispatchMsg(hab,&qmsg);

            WinDestroyWindow(hwndFrame);
            WinDestroyMsgQueue (hmq);
            WinTerminate (hab);

            return 0;
    }
```

```
MRESULT EXPENTRY ClientWndProc (HWND hwnd, USHORT msg, MPARAM mp1, MPARAM mp2)

    {
        static CHAR szText [] = "Hello World";
        HPS      hps;
        RECTL    rcl;

        switch(msg)
            {
                case WM_CREATE :
                        return 0;
                case WM_PAINT:
                        hps = WinBeginPaint (hwnd, NULL, NULL);

                        WinQueryWindowRect(hwnd, &rcl);

                        WinDrawText(hps, -1, szText, &rcl, CLR_NEUTRAL, CLR_BACKGROUND,
                                        DT_CENTER | DT_VCENTER | DT_ERASERECT);

                        WinEndPaint(hps);

                        return 0;
                case WM_DESTROY:
                        return 0;
            }
        return WinDefWindowProc(hwnd, msg, mp1, mp2);
    }
```

Appendix H

This program, in C++, makes use of the CommonView class library to construct a PM window and display the message - Hello World in it.

```
/*****************************************************************************************************
 *
 *        Program to display a window and the message hello world in CommonView.
 *
 *****************************************************************************************************/


#include <CommonVu.hxx>

//
// Define the ExampleWindow class.  It has not variables, just a
// constructor.
//
class ExampleWindow : public TopAppWindow
{
  public :
        ExampleWind();
};


//
//  This function starts event driven processing of the application.
//  Initially, here, it displays the sample window, the message and then
//  waits for events to happen.
//
void App :: far Start()   // This is the main function equivalent to main()

{
  ExampleWindow  *sample = new ExampleWindow();

  sample->TextPrint("Hello World",Point(100,100));

  Exec();  // Start polling the buttons etc.
}


//
// The constructor function for this class.
//
ExampleWindow :: ExampleWindow()

{
  EnableBorder();               // Display the window border.
  EnableSysMenu();              // enable system menu at top left of window.
  EnableMinBox();               // enable the minimisation box at top right.
  EnableMaxBox();               // enable the maximisation box at top right.

  SetCaption("Example Window");         // Put title on the window.
  Show();                               // Display the window.
}
```

Appendix I

A Technical description of ONTOS

Introduction

Ontos is a fully distributed object database managment system. It basically consists of a database, a programmatic interface and a set of tools. To the programmer, ONTOS consists of a set of classes which are incorporated in C++ applications. These C++ applications are used for accessing a database which has been previously created and registered with the database registry.

A number of tools are provided by ONTOS to deal with the registration of the database, and the simple conversion of existing C++ programs to be operational with ONTOS. In addition, to facilitating the alteration of existing programs, which declare C++ classes statically in the program code, ONTOS provides a number of classes and functions which provide for the dynamic creation of classes and objects in the database, during the execution of the program, without the need for the class declaration in the application code itself.

Tools provided by ONTOS

- **classify**
  This utility takes a standard C++ header file, which contains the class declaration code, and generates a database schema.
- **cplus**
  The cplus utility is a compiler front-end, a preprocessor, that prepares C++ constructors and member functions for use with ONTOS.
- **DBATool**
  DBATool provides an interactive interface for accessing the ONTOS database registry. DBATool provides support for registration and administration of ONTOS databases. It is used to define the mapping between logical databases and their physical locations and to define the agents that will manage these databases.

Programmatic interface

Programmatic interaction with ONTOS is through a class library. The class library consists of the C++ classes predefined by ONTOS. It is part of the Client Library, which also includes a library of free functions that are independent of any class (the Function Library).

The classes provided can be classed under a number of different headings.
This is a summary of the classes in the Client Library.

Persistence

Entity: An abstract class of all values that can be referenced in the database, including both primitive and persistent objects.
Object: Provides an interface for persistent storage of class instances.

General-purpose Aggregates

Aggregate: Base class for all general-purpose container classes.
Association: Base class for keyed Aggregates.
Array: An integer-indexed Association.
Dictionary: An Association that can be indexed by instances of arbitrary types, including objects and primitives (such as integers or strings). Dictionaries may be ordered or unordered, and are implemented by B*trees or hash-tables, respectively.
List: Ordered, unkeyed Aggregate. Represents linked lists, sequences, queues, or stacks.
Set: Unordered Aggregate, insensitive to duplicate insertions; represents the mathematical concept of sets.

General-purpose Iterators

Iterator: An abstract base class for classes used to return successive Entity values. Defines a common protocol supported by such classes.
AggregateIterator: Base class for iterators over Aggregate classes.
ArrayIterator: Returns successive values from Arrays, in either the forward or reverse limited to a specific key value, or if ordered, a specific range of key values.
DictionaryIterator: Returns successive elements or keys of Dictionary objects. May be limited to a specific key value, or if ordered, a specific range of key values.
ListIterator: Returns successive elements Lists. May be limited to a specific range of indices.
SetIterator: Returns successive elements of Sets.

Schema-Definition Classes

Type: Represents C++ class definitions in a runtime-usable form. It is used by the database to define schema information.

PropertyType: Represents a class's property (field or data member) definitions in a runtime-usable form, including each property's defining class or Type, the allowed Type of its values, and its default value. Used in abstract data member access and representation of data members in schema definitions.

Procedure: Represents a class's member functions as well as free functions in a runtime-usable form; contains a binding to function code. Used in abstract function invocation and representation of (member) functions in schema definitions.

ArgSpec: Represents a functions argument's data type and default value, and indicates whether it is passed by value or by reference.

ArgSpecList: List of a specific Procedure's ArgSpecs.

ArgumentList: List of argument values for passing to functions represented by Procedure objects.

FuncBinding: Maps a function's generated "C" function name to its code address. Used to bind Procedure objects to code.

Classes for C++ Primitives

Argument: A class allowing for consistent treatment of values of Entity-based classes and C++ data types like *int, double, char**, etc. Uses cast operators to allow conversion between these C++ data types and their Entity-based (primitive-based) analogs.

Primitive: Base class for all analog classes for C++ primitive data types.

Integer: Represents data of types *int, short, long,* and the corresponding *unsigned* version.

Pointer: Represents memory pointers.

Real: Represents data of types *float* and *double.*

String: Represents data of type *char** (character strings).

Iterators for Schema Definition Classes

ProcedureIterator: Sequentially returns all the Procedure objects defined for a Type (class definition).

PropertyIterator: Sequentially returns all the PropertyType objects defined for a

Type (class definition).

ConstructorIterator: Sequentially returns all the Constructor Procedure objects defined for a Type (class definition).

SubTypesIterator: Sequentially returns all the immediately-derived classes of a class.

InstanceIterator: Sequentially returns all instances of a particular class. Requires that the extension property of the Type had been enabled when the Type was created.

OffsetIterator: Sequetially returns the byte offsets of all references to persistent objects or primitive values in a specified object.

Exception-Handling Classes

CleanupObj: Root class for most Client Library classes. Provides cleanup functionality when exception handling results in an abort.

Failure: Root class for defining exception classes and passing error information to exception-handling functions.

ExceptionHandler: Defines execution scope for exception handling, and links a particular subclass of Failure to an exception-handling function for the duration of that scope.

Other

Directory: A container class for storing Object names and the mappings to an from Objects and their names.

DirectoryIterator: Returns successive names or objects from a Directory.

QueryIterator: Interprets SQL queries into ONTOS database operations.

TRef: Reference to a persistent object. It replaces a direct memory reference (or pointer), and may be used whether the object is in memory or not. Converts transparently between database reference form and memory pointers.

A number of other free functions are provided to allow the programmer to access data facilities, like transaction management, retrieval of objects, based on their unique identifiers, and database opening/closing.

To the programmer, ONTOS puts forward a very consistent appearance. Obviously, all most of its functionality is implemented through classes and methods. In addition to this consistenct, the manner in which data is returned from the database, whether it is class information, object data or query results, is done through iterators. This consistenct makes the usage of the class library relatively simple.

Release 2.01

Since the development of GRIFON, Ontologic have release a new version of ONTOS. In addition to its provision of improved stability of the features of version 1.42, it also offers the programmer a number of extra features. The latest standard in C++ - Version 2.0, supports multiple-inheritance. ONTOS reflects this extension supportin persistence for multiple inheritance. In keeping with the manner in which features are provided by ONTOS, a new iterator class is provided to facilitate the return of the super types or classes for a specified class. Therefore the extra learning on the part of the programmer, to be able to use the extra features of ONTOS 2.01 are minimal. In addition to some new features, some of the problems associated with the earlier version, as mentioned in chapter 5, have been fixed.

Some new utilities have been provided with the new version of ONTOS.

DBDesigner is an interactive, visually-oriented tool for looking at and putting together the structure, relationships and content of an ONTOS database. DBDesigner, users can design the database schema "by eye" and then immediately generate the corresponding C++ include files directly from the schema.

DBRecover is a utility provided to facilitate the recovery of a corrupted database area to its previous state. An area could become corrupted for any of the following reasons:

- CPU failure
  if the host machine that runs the area's server process crashes.
- program error
  if the area's server process is abnormally terminated due to programming error.
- human intervention
  if the area's server process is forcefully terminated by a system administrator.

Appendix J

The PM API provides a number of different groups of functions for accessing the features of the underlying hardware. The Graphics Programming Interface (GPI) set of functions provides the programmer with a large selection of functions for the development of substantial graphical applications. The GPI functions provided by the OS/2 1.1 Software development toolkit are listed below.

GpiAssociate	Associates pres. space with a device context
GpiBeginArea	Starts an area bracket
GpiBeginElement	Starts an element bracket
GpiBeginPath	Starts a path bracket
GpiBitBlt	Copies bitmaps
GpiBox	Draws a rectangular box
GpiCallSegmentMatrix	Draws a segment using an instance matrix
GpiCharString	Draws a character string at current position
GpiCharStringAt	Draws character string at specified position
GpiCharStringPos	Draws a character string with formatting
GpiCharStringPosAt	Draws a character string with formatting
GpiCloseFigure	Closes a figure
GpiCloseSegment	Closes the current segment
GpiCombineRegion	Combines two regions
GpiComment	Adds a comment to a segment
GpiConvert	Converts an array of coordinate pairs
GpiCopyMetaFile	Copies a metafile
GpiCorrelateChain	Correlates a chain
GpiCorrelateFrom	Performs a correlation operation
GpiCorrelateSegment	Correlates a segment
GpiCreateBitmap	Creates a bitmap
GpiCreateLogColorTable	Creates a logical color table
GpiCreateLogFont	Creates a logical font
GpiCreatePS	Creates a presentation space
GpiCreateRegion	Creates a region
GpiDeleteBitmap	Deletes a bitmap
GpiDeleteElement	Deletes an element
GpiDeleteElementRange	Deletes an element range
GpiDeleteElementsBetweenLabels	Deletes the elements between two labels
GpiDeleteMetaFile	Deletes a metafile
GpiDeleteSegment	Deletes a retained segment
GpiDeleteSegments	Deletes all segments
GpiDeleteSetId	Deletes a logical font or bitmap tag
GpiDestroyPS	Destroys a presentation space
GpiDestroyRegion	Destroys a region
GpiDrawChain	Draws a picture chain
GpiDrawDynamics	Redraws dynamic segments
GpiDrawFrom	Draws a section of a picture chain
GpiDrawSegment	Draws a specified segment
GpiElement	Draws an element
GpiEndArea	Ends an area bracket
GpiEndElement	Ends an element bracket
GpiEndPath	Ends a path bracket

GpiEqualRegion	Checks two regions for equality
GpiErase	Clears the output display
GpiErrorSegmentData	Returns an error location in a segment
GpiExcludeClipRectangle	Excludes a rectangle from the clip region
GpiFillPath	Draws the interior of a path
GpiFullArc	Creates a full arc
GpiGetData	Get graphics order data from a segment
GpiImage	Draws an image
GpiIntersectClipRectangle	Sets a clip region from an intersection
GpiLabel	Creates a label element
GpiLine	Draws a line
GpiLoadBitmap	Loads a bitmap from a resource
GpiLoadFonts	Loads fonts from a resource file
GpiLoadMetaFile	Loads data from a file into a metafile
GpiMarker	Draws a marker
GpiModifyPath	Modifies a path
GpiMove	Moves current position to a specified point
GpiOffsetClipRegion	Moves the clip region
GpiOffsetElementPointer	Sets the element pointer
GpiOffsetRegion	Moves a region
GpiOpenSegment	Opens a segment
GpiPaintRegion	Paints a region
GpiPartialArc	Draws a partial arc
GpiPlayMetaFile	Plays a metafile
GpiPointArc	Draws an arc through three points
GpiPolyFillet	Draws a curve
GpiPolyFilletSharp	Draws a fillet
GpiPolyLine	Draws straight lines
GpiPolyMarker	Draws a marker
GpiPolySpline	Draws Bezier splines
GpiPop	Restores one or more primitive attributes
GpiPtInRegion	Determines whether a point is in a region
GpiPtVisible	Determines whether a point is visible
GpiPutData	Draws graphics orders from a buffer
GpiQueryArcParams	Retrieves the current arc parameters
GpiQueryAttrMode	Retrieves the current attribute mode
GpiQueryAttrs	Retrieves attributes for a primitive type
GpiQueryBackColor	Retrieves the current background color
GpiQueryBackMix	Retrieves the current background mix mode
GpiQueryBitmapBits	Copies bitmap image data to a buffer
GpiQueryBitmapDimension	Retrieves the dimensions of a bitmap
GpiQueryBitmapHandle	Retrieves the handle to a bitmap
GpiQueryBitmapParameters	Retrieves bitmap parameters
GpiQueryBoundaryData	Retrieves boundary data
GpiQueryCharAngle	Retrieves the character-angle attribute
GpiQueryCharBox	Retrieves the character-box attribute
GpiQueryCharDirection	Retrieves the character-direction attribute
GpiQueryCharMode	Retrieves the character-mode attribute
GpiQueryCharSet	Retrieves the character-set identifier
GpiQueryCharShear	Retrieves the character-shear angle
GpiQueryCharStringPos	Retrieves the positions of characters

GpiQueryCharStringPosAt	Retrieves the character positions at a point
GpiQueryClipBox	Retrieves a clip-box rectangle
GpiQueryClipRegion	Retrieves the handle to a clip region
GpiQueryColor	Retrieves the line-color attribute
GpiQueryColorData	Retrieves color-table data
GpiQueryColorIndex	Retrieves the color index
GpiQueryCp	Retrieves the code-page identifier
GpiQueryCurrentPosition	Retrieves the current position
GpiQueryDefaultViewMatrix	Retrieves the default viewing matrix
GpiQueryDefCharBox	Retrieves the size of the default char box
GpiQueryDevice	Retrieves a device context from a PS
GpiQueryDeviceBitmapFormats	Retrieves bitmap formats
GpiQueryDrawControl	Checks for a drawing control
GpiQueryDrawingMode	Retrieves the drawing mode
GpiQueryEditMode	Retrieves the current editing mode
GpiQueryElement	Retrieves element content data
GpiQueryElementPointer	Retrieves the current element pointer
GpiQueryElementType	Retrieves information about an element type
GpiQueryFontFileDescriptions	Retrieves font-file descriptions
GpiQueryFontMetrics	Retrieves information about font metrics
GpiQueryFonts	Retrieves font information
GpiQueryGraphicsField	Retrieves coordinates of a graphics field
GpiQueryInitialSegmentAttrs	Retrieves an initial segment attribute
GpiQueryKerningPairs	Retrieves kerning-pair information
GpiQueryLineEnd	Retrieves the line-end attribute
GpiQueryLineJoin	Retrieves the line-join attribute
GpiQueryLineType	Retrieves the cosmetic line-type attribute
GpiQueryLineWidth	Retrieves cosmetic line-width attribute
GpiQueryLineWidthGeom	Retrieves the geometric line-width attribute
GpiQueryLogColorTable	Retrieves the logical color table
GpiQueryMarker	Retrieves the marker-symbol attribute
GpiQueryMarkerBox	Retrieves the marker-box attribute
GpiQueryMarkerSet	Retrieves the marker-set attribute
GpiQueryMetaFileBits	Transfers a metafile to application storage
GpiQueryMetaFileLength	Retrieves the length of a memory metafile
GpiQueryMix	Retrieves the foreground mix mode
GpiQueryModelTransformMatrix	Retrieves the model-transformation matrix
GpiQueryNearestColor	Retrieves the nearest available color
GpiQueryNumberSetIds	Retrieves the number of lcids in use
GpiQueryPageViewport	Retrieves the coordinates of a page viewport
GpiQueryPattern	Retrieves the shading-pattern attribute
GpiQueryPatternRefPoint	Retrieves the pattern reference point
GpiQueryPatternSet	Retrieves the pattern-set identifier
GpiQueryPel	Retrieves the color of a specified pel
GpiQueryPickAperturePosition	Retrieves the center of a pick aperture
GpiQueryPickApertureSize	Retrieves the size of a pick aperture
GpiQueryPS	Retrieves the page parameters for a PS
GpiQueryRealColors	Retrieves RGB values
GpiQueryRegionBox	Retrieves a region-box rectangle
GpiQueryRegionRects	Retrieves region rectangles
GpiQueryRGBColor	Retrieves an RGB color

GpiQuerySegmentAttrs	Checks for a segment attribute
GpiQuerySegmentNames	Retrieves the segment identifiers
GpiQuerySegmentPriority	Retrieves the segment priority
GpiQuerySegmentTransformMatrix	Retrieves a segment-transformation matrix
GpiQuerySetIds	Retrieves information about fonts
GpiQueryStopDraw	Retrieves the stop/draw condition
GpiQueryTag	Retrieves a tag identifier
GpiQueryTextBox	Retrieves the coordinates of a text box
GpiQueryViewingLimits	Retrieves coordinates of the viewing limits
GpiQueryViewingTransformMatrix	Retrieves a viewing-transformation matrix
GpiQueryWidthTable	Retrieves font-width-table information
GpiRealizeColorTable	Realizes a logical color table
GpiRectInRegion	Determines whether rectangle is in a region
GpiRectVisible	Determines whether a rectangle is visible
GpiRemoveDynamics	Removes dynamic segments
GpiResetBoundaryData	Resets boundary data
GpiResetPS	Resets a presentation space
GpiRestorePS	Restores a presentation space
GpiSaveMetaFile	Saves a metafile
GpiSavePS	Saves a presentation space
GpiSetArcParams	Sets the current arc parameters
GpiSetAttrMode	Sets the current attribute mode
GpiSetAttrs	Sets the attributes for a primitive type
GpiSetBackColor	Sets the current background color
GpiSetBackMix	Sets the current background mix mode
GpiSetBitmap	Sets a bitmap
GpiSetBitmapBits	Sets the bits of a bitmap
GpiSetBitmapDimension	Sets the dimensions of a bitmap
GpiSetBitmapId	Sets a bitmap identifier
GpiSetCharAngle	Sets the character-angle attribute
GpiSetCharBox	Sets the character-box attribute
GpiSetCharDirection	Sets the character-direction attribute
GpiSetCharMode	Sets the character-mode attribute
GpiSetCharSet	Sets the character-set identifier
GpiSetCharShear	Sets the character-shear attribute
GpiSetClipPath	Sets a clip path
GpiSetClipRegion	Sets a clip region
GpiSetColor	Sets the line-color attribute
GpiSetCp	Sets the graphics code-page identifier
GpiSetCurrentPosition	Sets the current position
GpiSetDefaultViewMatrix	Sets default viewing transformation
GpiSetDrawControl	Sets a draw control
GpiSetDrawingMode	Sets the drawing mode
GpiSetEditMode	Sets the editing mode
GpiSetElementPointer	Sets the element pointer
GpiSetElementPointerAtLabel	Sets the element pointer at a label
GpiSetGraphicsField	Sets the coordinates of a graphics field
GpiSetInitialSegmentAttrs	Sets the initial segment attributes
GpiSetLineEnd	Sets the line-end attribute
GpiSetLineJoin	Sets the line-join attribute
GpiSetLineType	Sets the line-type attribute

GpiSetLineWidth	Sets the cosmetic line-width attribute
GpiSetLineWidthGeom	Sets the geometric line-width attribute
GpiSetMarker	Sets the marker attribute
GpiSetMarkerBox	Sets the marker-box attribute
GpiSetMarkerSet	Sets the marker-set attribute
GpiSetMetaFileBits	Copies data from a buffer to a metafile
GpiSetMix	Sets the foreground mix mode
GpiSetModelTransformMatrix	Sets the model-transformation matrix
GpiSetPageViewport	Sets the coordinates of a page viewport
GpiSetPattern	Sets the shading-pattern attribute
GpiSetPatternRefPoint	Sets the pattern reference point
GpiSetPatternSet	Sets the pattern-set attribute
GpiSetPel	Sets the color of a pel
GpiSetPickAperturePosition	Sets the center of the pick aperture
GpiSetPickApertureSize	Sets the size of the pick aperture
GpiSetPS	Sets the page parameters of the PS
GpiSetRegion	Sets a region
GpiSetSegmentAttrs	Sets an attribute for a retained segment
GpiSetSegmentPriority	Sets the segment priority
GpiSetSegmentTransformMatrix	Sets a segment-transformation matrix
GpiSetStopDraw	Sets the stop-draw condition
GpiSetTag	Sets a tag for a primitive
GpiSetViewingLimits	Sets the coordinates of the viewing limits
GpiSetViewingTransformMatrix	Sets the viewing-transformation matrix
GpiStrokePath	Strokes a path
GpiUnloadFonts	Unloads font definitions
GpiUnrealizeColorTable	Unrealizes the logical color table
GpiWCBitBlt	Copies a bitmap to a presentation space

Appendix K

The C++ function to draw the arc between class nodes in the display of the class-composition hierarchy. The calculation of the arcs, ensures their best display on the screen.

```
#include "CvDefs.hxx"
#include <CommonVu.hxx>
#include <DrawObj.hxx>
#include <stdio.h>

#include "node.hxx"

#define INCL_GPIPRIMITIVES
#include <OS2.H>


//*************************************************************************************************
//
// This Method is passed two nodes in the Tree and displays the
// Arc between them.  The first node is the Selected Node and
// the second node is the domain Node or the destination Node.
//
//
// Parameters:  Node*  selectedNode (the node chosen by the user.)
//              Node*  domainNode (the domain node. )
//
//
//  This method will also draw the ArrowHeads at the end of the Arcs
//  to indicate the direction of the Attribute/Domain relationship.
//
//*************************************************************************************************

void Draw_Arc(pWindow p,Node* selectedNode, Node* domainNode,int x,int y,char* propnam, HWND hwnd)

{

 char arrowhead=0, vline=0, samelevel=0;  // 1 is North, 2 is south, 3 is East, 4 is West
 long sx_val, sy_val;
 long dx_val, dy_val;
 char buffer[30];

 sx_val = selectedNode->get_x_value()+x;
 sy_val = selectedNode->get_y_value()+y;
 dx_val = domainNode->get_x_value()+x;
 dy_val = domainNode->get_y_value()+y;


 POINTL  arrPoint[5];

 HPS hps = WinGetPS(hwnd);
 /*
   If the Domain is at a Lower Level than the Selected Node, then
   Draw the Arc from the Side of the Selected Node to the Bottom
   of the Domain Node.
 */

 if (domainNode->get_level() < selectedNode->get_level())
   {

    /*
      The selected Node is to the Right of the Domain Node, so
      Draw the Line from the Left side of the selected Node to
      the bottom of the Domain Node.  If it is to the left, draw
      a line from the Right side of the Selected Node.
    */

    arrowhead = 1;  // ArrowHead is going to be UpWard.
vline = 0;

if (sx_val > dx_val)
  {
   // Point on the Left of the Selected Node.
   vline = 2;
   arrPoint[0].x = sx_val;
   arrPoint[0].y = sy_val + (NODE_HEIGHT/2);
  }
else if (sx_val < dx_val)
        {
         // Point on the Right of the Selected Node.
         vline = 3;
         arrPoint[0].x = sx_val + NODE_WIDTH;
         arrPoint[0].y = sy_val + (NODE_HEIGHT/2);
        }
      else
        {
         // Point on the Top of the Selected Node.
         vline = 1;
         arrPoint[0].x = sx_val + (NODE_WIDTH/2);
         arrPoint[0].y = sy_val + NODE_HEIGHT;
        }



//  Move to the initial position.
GpiMove(hps,&arrPoint[0]);


if (vline == 1)
  {
   // Draw a Vertical Line from One to the Other.
   //
   arrPoint[1].x = arrPoint[0].x;
   arrPoint[1].y = dx_val;
   GpiLine(hps,&arrPoint[1]);
   //
   // Now Write the Name of the Attribute in the Approp.
   // position. Name just below the ArrowHead.

   sprintf(buffer,"%s",propnam);
   p->TextPrint(buffer,Point((arrPoint[1].x-30),(arrPoint[1].y-30)));
  }
else
  {
  // Get the Point on the Bottom of the Domain Node.
  arrPoint[1].x = dx_val + (NODE_WIDTH/2);
  arrPoint[1].y = dy_val;

  // Calculate the intermediate Point.
  arrPoint[0].x = arrPoint[1].x;     // Same x Value as the Domain
  arrPoint[0].y = arrPoint[0].y;     // Same y Value as the Selected

  // Now draw the Arc using GPIPolyFillet
  GpiPolyFillet(hps,2L,arrPoint);
  sprintf(buffer,"%s",propnam);
  p->TextPrint(buffer,Point((arrPoint[1].x-30),(arrPoint[1].y-30)));
  }
}
else
  /*
     The Selected Node at a lower level than the Domain Node.
     Draw a line from the side of the Selected Node to the Top
     of the Domain Node.
  */
  if (domainNode->get_level() > selectedNode->get_level())
```

```
{
/*
    If the Selected Node is to the right of the Domain Node
    then draw from the left of the Selected Node.  If it is
    to the left, draw a line from the right side of the
    Selected Node.
*/

arrowhead = 2; // Arrowhead is pointing down.
vline = 0;

if (sx_val > dx_val)
  {
   // Point to the to the Left of Selected Node.
   vline = 2;
   arrPoint[0].x = sx_val;
   arrPoint[0].y = sy_val + (NODE_HEIGHT/2);
  }
else if (sx_val < dx_val)
       {
        // Point on the Right side of Selected Node.
        vline = 3;
        arrPoint[0].x = sx_val + NODE_WIDTH;
        arrPoint[0].y = sy_val + (NODE_HEIGHT/2);
       }
     else
       {
        // The Nodes atr directly under each other.
        vline = 1;
        arrPoint[0].x = sx_val + (NODE_WIDTH/2);
        arrPoint[0].y = sy_val;
       }

//  Move to the initial position.
GpiMove(hps,&arrPoint[0]);

if (vline == 1)
  {
   arrPoint[1].x = arrPoint[0].x;
   arrPoint[1].y = dy_val + NODE_HEIGHT;
   GpiLine(hps,&arrPoint[1]);
   sprintf(buffer,"%s",propnam);
   p->TextPrint(buffer,Point((arrPoint[1].x-30),(arrPoint[1].y+30)));
  }
else
  {
   // Get the Point on the Bottom of the Domain Node.
   arrPoint[1].x = dx_val + (NODE_WIDTH/2);
   arrPoint[1].y = dy_val + NODE_HEIGHT;

   // Calculate the intermediate Point.
   arrPoint[0].x = arrPoint[1].x;
   arrPoint[0].y = arrPoint[0].y;


   // Now draw the Arc using GPIPolyFillet
   GpiPolyFillet(hps,2L,arrPoint);
   sprintf(buffer,"%s",propnam);
   p->TextPrint(buffer,Point((arrPoint[1].x-30),(arrPoint[1].y+30)));

  }
}
else
  if (domainNode->get_level() == selectedNode->get_level())
    {

     /*
```

```
          If the Domain Node and the Selected Node are on the same
          level then draw an arc from the top of the Selected Node
          to the top of the Domain Node.
        */

        arrowhead = 2;
        samelevel = 1;
        // The point on the top of the Selected Node.
        arrPoint[0].x = sx_val + (NODE_WIDTH/2);
        arrPoint[0].y = sy_val + NODE_HEIGHT;

        //  Move to the Initial Point on the arc,
        GpiMove(hps,&arrPoint[0]);

        // Get the 2nd Middle Control Point
        arrPoint[0].x = arrPoint[0].x;
        arrPoint[0].y = arrPoint[0].y+50;

        // Get the Point on the Top of the Domain Node.
        arrPoint[3].x = dx_val + (NODE_WIDTH/2);
        arrPoint[3].y = dy_val + NODE_HEIGHT;

        //
        arrPoint[2].x = arrPoint[3].x;
        arrPoint[2].y = arrPoint[3].y+50;

        // Get the 1st Middle Control Point
        if (sx_val > dx_val)
           arrPoint[1].x = arrPoint[3].x+((arrPoint[0].x - arrPoint[3].x)/2);
        else
           arrPoint[1].x = arrPoint[0].x+((arrPoint[3].x - arrPoint[0].x)/2);

        arrPoint[1].y = arrPoint[0].y+50;

        GpiPolyFillet(hps,4L,arrPoint);

        sprintf(buffer,"%s",propnam);
        p->TextPrint(buffer,Point((arrPoint[3].x-30),(arrPoint[3].y+30)));
       }


//
// Now Draw the ArrowHead at point arrPoint[1]
//

POINTL rootPoint;

if (samelevel==1)
   {
    rootPoint.x = arrPoint[3].x;
    rootPoint.y = arrPoint[3].y;
   }
else
   {
    rootPoint.x = arrPoint[1].x;
    rootPoint.y = arrPoint[1].y;
   }
 samelevel = 0;



// Display an arrowHead facing Upward.
if (arrowhead == 1)
 {
  arrPoint[0].x = rootPoint.x - 8;
  arrPoint[0].y = rootPoint.y - 12;
  GpiMove(hps,&rootPoint);
```

```
   GpiLine(hps,&arrPoint[0]);
   arrPoint[0].x = rootPoint.x + 8;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
  }

 //  Display an arrowHead facing Downward.
 if (arrowhead == 2)
  {
   arrPoint[0].x = rootPoint.x - 8;
   arrPoint[0].y = rootPoint.y + 12;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
   arrPoint[0].x = rootPoint.x + 8;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
  }

 // Display an arrowhead facing east >
 if (arrowhead == 3)
  {
   arrPoint[0].x = rootPoint.x - 12;
   arrPoint[0].y = rootPoint.x + 8;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
   arrPoint[0].x = rootPoint.x - 8;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
  }


 // Display an arrowhead facing west <
 if (arrowhead == 4)
  {
   arrPoint[0].x = rootPoint.x + 12;
   arrPoint[0].y = rootPoint.x + 8;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
   arrPoint[0].x = rootPoint.x - 8;
   GpiMove(hps,&rootPoint);
   GpiLine(hps,&arrPoint[0]);
  }

 //
 // The Name of the Property also needs to be displayed
 // on the arc to the Domain Node.
 //


 WinReleasePS(hps);
}
```