An Object-Oriented Database Methodology

for application development with extended relational or object-oriented DBMS

A thesis presented in partial fulfilment of the requirements

for the degree of

Master of Science in Computer Science at Massey University

Benny Liew

1992

Acknowledgement

Firstly, I would like to thank Mr Roger Tagg, my thesis supervisor, for helping me to draft out the contents and using the two case study examples from his 57.ODB(Object-Oriented Database) paper. His advice and guidance throughout is greatly appreciated. Many books and articles were borrowed from him for use in this thesis.

Next, I would like to express my appreciation to Dr. Daniela Mehandjiska-Stavreva, my alternate supervisor for her helpful comments on the structure, written style and presentation of this thesis.

I would like to thank Massey University Library for using the facilities for my literature search, especially the Library Interloan staff for their excellent services. A total of eight books and more than a dozen of journal articles were requested within New Zealand and overseas. Some of these articles came as far as Canadian universities' libraries.

Thanks go to Mr. Colin Eagle and Mr. Richard Rayner for their excellent support of Postgres on our Sun Microstations network; and also to Mr Todd Cochrane, our PhD student of Computer Science Department, for his past assistance. I wish Todd every success for his PhD research work.

Lastly, I would like to thank Mum and Dad for so many years of upbringing.

Benny Liew, DipSc(CompSc), TechDip(Elect Engg), MSIET

Master of Science(Computer Science) candidate,
Massey University,
Dept. of Computer Science,
Palmerston North,
New Zealand.

3/12/92

Abstract

Recently development methodologies have been proposed which describe themselves as "Object Oriented". While all of them offer approaches to extended data and behavioural modelling, none of them seem fully adequate to address the total concept of object-oriented development. They often do not provide constructs which lead to the use of databases, nor do they always recognise the shift from sequential to prototyping style which is inherent in much object-oriented technology.

The objective of this thesis is to lay a framework for an object-oriented methodology suitable for OODBMS. Details of conventional methods for developing database applications, and of the recent OO methods, have been examined and compared in order to propose a coherent set of tasks and deliverables. Account has also been taken of designing for re-use, which has been one of the main selling points of the OO approach.

The proposed methodology attempts to address related side issues, with particular focus on object concurrency, which seems particularly thinly covered in many of the current proposals. Many other side issues are also mentioned, but due to time constraints, they are not given any further discussion. The topic is an extremely multi-disciplinary one, and a very wide range of expertise would be necessary to do justice to all these aspects.

Mapping of the new methodology has been tried on two case study examples using Postgres and Ontos. Postgres is an extended relational DBMS developed as a research prototype at University of California, Berkeley. Ontos is the commercial object-oriented DBMS marketed by Ontos Incorporated, Burlington, Massachusetts. Some details of these implementation examples are included.

Rationale for the Research

Object-oriented technology has gained much popularity recently, but methodologies for its use are still at an immature stage. There are many proposed developments of the OO paradigm by pioneers in this area. Examples are Booch[4], Coad & Yourdon[7,8], Shlaer & Mellor[55], and Meyer[19]. These methodologies are often fairly general in nature and do not specifically address the needs of the OO paradigm to some special areas, such as databases.

On the other hand, pioneers in OODBMS like Zdonik and Maier[30], Stonebraker[56-58], Won Kim[15] and Lochovsky[14] and Rolland & Brunet[52] concentrate more on the requirements and implementation of a specific kind of OODBMS.

The concepts of Object Repository and reusability of software have also been subjects of discussion lately. There are many advantages associated with OO prototyping[20].

So far, there has not been an OO paradigm that covers the whole development cycle of an OODBMS, although there exists many OODBMS tools. This thesis aims to propose a total, unified paradigm applicable to OODBMS from feasibility through analysis and design to implementation stage. It emphases particularly on prototyping and reusability through the use of class libraries and repositories so as to support modern practices.

One way of doing this is to review all the currently proposed OO methodologies to gain an understanding of each in terms of techniques and diagrams used. Sometimes, different conventions and terms are used by different authors to represent the same semantic meaning. It is necessary to understand why such individual approaches are used.

An OO methodology should also have stages of development just like conventional software development using the functional approach. In addition, steps for each phase of development is prescribed.

Extended relational and object-oriented databases are examined, and their common features extracted. This is necessary for the formulation of an OODBMS methodology of general applicability.

The topic of object-oriented prototyping as applied to application development in OODBMS is also discussed. OO prototyping enables quick development of OO database applications and this technique should be used.

Thesis structure

This thesis is made up of eight chapters.

The first chapter of the thesis takes a look at past methodologies for software development and the evolution of present ones. It briefly describes the existing methodologies that are well accepted and practised by current software houses. It then describes the emerging methodologies of the 1990s such as RAD and the IBM AD/Cycle-Repository. Finally, some of the better-known OO approaches are briefly introduced and summarised.

Chapter 2 discusses the required features of an OODBMS methodology. These concepts are taken from various sources and each one is given a brief description. Later, in Chapter 4, some of them are selected to be applied in the proposed methodology.

Chapter 3 gives a brief description of existing methodologies using the object-oriented paradigm. It is important to note that not all of them are equally suitable for all types of implementation. For instance, Rolland & Brunet's O* Model is particularly suitable for OODBMS because it supports a lot of database concepts. A comparison is made on the methodologies covered in the literature search. The similarities, differences, strength and deficiency of each is pointed out in a matrix.

Chapter 4 is the proposal of a new methodology for OODBMS. The new proposal stresses 4 stages of development and the exploitation of object-oriented prototyping for object iteration. The techniques and diagrams adopted in each step have been described in Chapter 3.

Chapter 5 examines the application of the proposed paradigm as applied to extended relational database. Postgres is chosen as the extended relational database used to illustrate a case study example.

Chapter 6 examines the application of the proposed paradigm as applied to OODBMS. Ontos is used as the object-oriented database to illustrate a case study example.

Chapter 7 offers some conclusions. It also comments on the application of the proposed methodology to the two different types of DBMS. Further possible work on the enhancement of the new methodology is also suggested.

Five sections are included in the Appendices.

Section A gives a brief description of existing fourth generation languages(4GL) for OODBMS. Samples of the user interfaces of O2, GemStone, and GOOSE are shown. GOOSE is a graphical interface for an OO database schema environment created at Georgia Institute of Technology.

Section B discusses concurrency control protocols in OODBMS.

Implementation details of Postgres Case Study example are provided in Appendix C. Implementation details of Ontos Case Study example are provided in Appendix D.

Finally in Appendix E, current research areas relating to both types of DBMS are discussed.

The bibliography contains all the books and journal articles used in the formulation of the proposed methodology.

Table of Contents

List of Figures

Chapter 1 : Review of Software Development Methodologies

1.1 Introduction

Early 1960s' information systems were not built according to any formal methodology[1,25,26]. Analysis work was limited and the emphasis was towards programming. Implementation of information systems was mainly restricted to programming and was based on fixed file structures.

In the late 1960s and 1970s, software development was based largely on function-oriented design, whereby the design is decomposed into a set of interacting units, each having a clearly defined function. Large software systems have been built using this technique and thus it has stood the test of practice. However, the need to develop and maintain large complex software systems using advanced techniques such as databases in a competitive and dynamic environment drove interest in better approaches to software design and development. In the 1980s, this led to a batch of formal "methodologies", which have incorporated some blend of function-oriented and data-oriented approaches.

1.2 Mainstream Methodologies Description

Some of the well-known methodologies that have gained widespread acceptance for information systems development today are introduced below :

1.2.1 STRADIS : Structured Analysis, Design and Implementation of Information Systems

This is based on the work of Gane & Sarson. The development of this structured systems approach to analysis came as a result of the earlier development of a structured approach to design. The structured design concepts were first proposed in 1974 by Stevens, Myers and Constantine (1974) and were later developed and refined by Yourdon and Constantine (1978), and Myers(1975, 1978). Data flow diagrams are constructed to represent the existing system and its interfaces.

1.2.2 Information Engineering

The term Information Engineering[17,18] originates from Clive Finkelstein who described a data modelling methodology he developed in Australia in the late 1970s, although the details have developed from a variety of sources including Ian Palmer of CACI in the UK, and James Martin in the USA. Information Engineering is now a comprehensive methodology covering all aspects of the software life cycle. It is evolving in the area of automated tools and the development of the methodology to support 4GL. The methodology is divided into four levels, within which there are seven stages, each with different objectives as shown in Fig. 1.1.

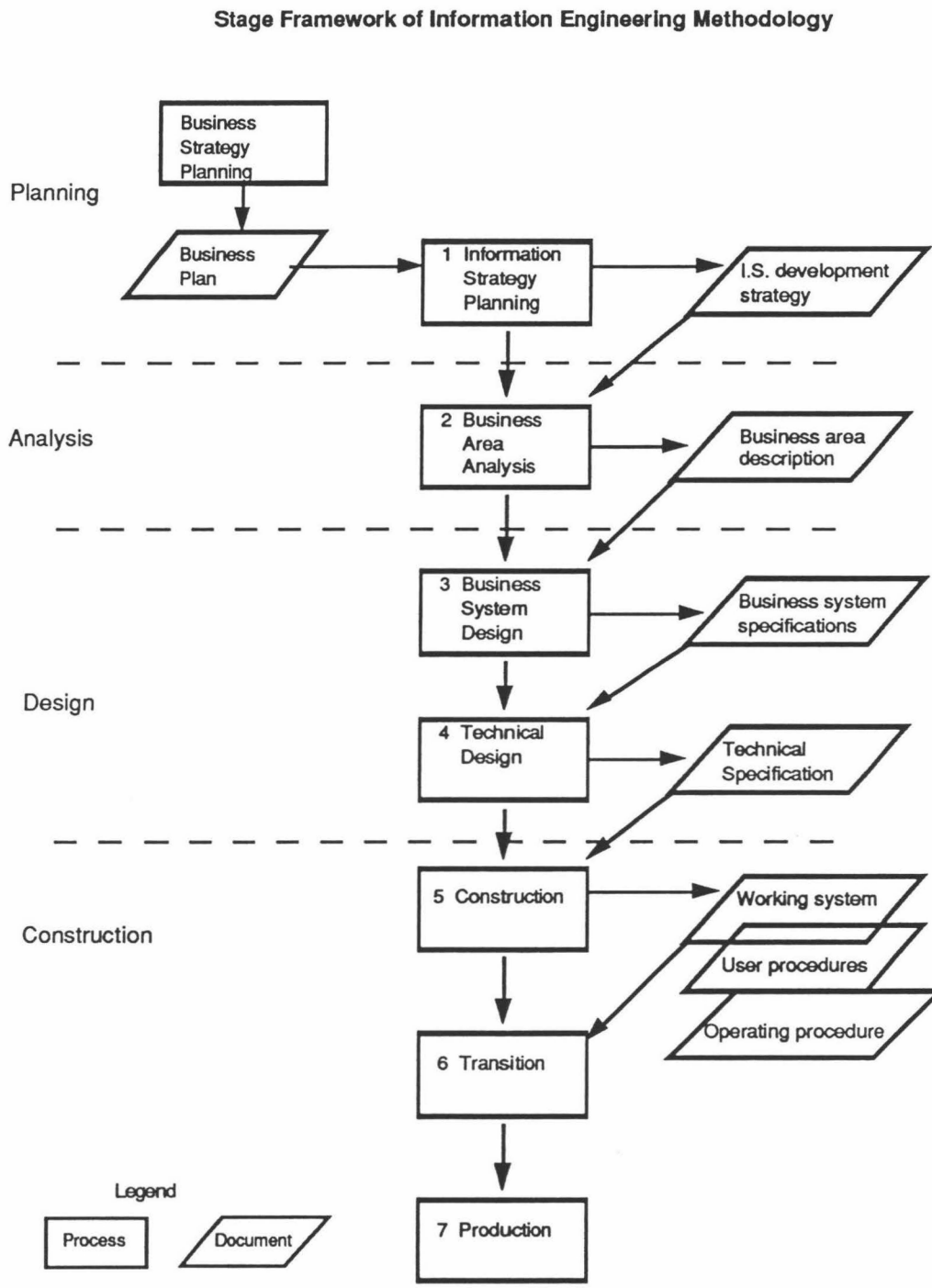


Fig. 1.1

1.2.3 Structured Systems Analysis and Design Methodology(SSADM)

SSADM[21] is a data-driven methodology developed originally by U.K. consultants, Learmonth and Burchett Management Systems and the U.K.Central Computing and Telecommunications Agency(CCTA). There are six phases in SSADM,

in which the first three phases are classified into systems analysis and the last three are systems design. They are :

(a) analysis of the current system,
(b) specification of the required system,
(c) user selection of Service Levels, including technical options,
(d) detailed data design,
(e) detailed procedure design,
(f) physical design control.

Data flow diagrams and entity models are needed to represent the static views of the system and a function/event matrix and an entity/event matrix are used to show the effects of time on the system.

1.2.4 Jackson Structured Design(JSD)

JSD[11] emphasises on the developing of maintainable software systems, and less on organisational need. Topics such as project selection, cost justification, requirements analysis, project management, user interface, procedure design or user participation are not addressed. JSD does not deal in detail with database design or file design. The major phases of JSD are :

(a) entity step action,
(b) entity structure step,
(c) initial model step,
(d) function step,
(e) system timing step,
(f) implementation step.

1.2.5 MERISE

MERISE[21] supports four stages of information system development. It combines an entity-relationship approach for data and a Petri-net based approach for processes.

1.2.6 Structured Systems Analysis(SSA)

SSA[21] was developed by Exxon in 1978, combining functional decomposition, data flow, relational data modelling and Jackson Structured Programming(JSP) techniques. Some information systems planning capability is also included.

1.2.7 Deficiency of mainstream methodologies

The 1980s have witnessed a growth in the number and variety of information systems methodologies. This increase in number of methodologies has caused much confusion. Many are the same(or very similar) and yet they have different 'brand names'. Some of them emphase in the techniques, the role of the computer, the documentation or the role of the people using the system. Some methodologies emphasize the importance of data and the development of a database. Some concentrate on analysis, others on design or implementation.

The classical waterfall software development life cycle, which is extensively used, is sometimes treated as a process in which work proceeds from one phase to another. It would be more difficult to return to the previous phase when the specification changes in comparison with OO development. Reasons why the traditional life cycle is inadequate for software development are :

(a) It assumes a relatively uniform progression of discrete steps, which includes little or no iteration,
(b) Due to the low cohesion and high coupling nature of program modules, it is difficult for the software to accomodate change which is a very desirable factor because each system is built from scratch and maintenance costs account for a large share of development cost,
(c) It does not accomodate the sort of evolutionary development made possible by rapid prototyping tools and 4GL,
(d) It does not allow future modes of software development like automatic code generation, module code transformation and 'knowledge-based' software development assistance,
(e) There is no emphasis on re-use of the software developed.

1.3 Current trends

In the early 1990's, there have been two new developments in the marketplace. One is Rapid Applications Development(RAD); the other is the IBM AD/Cycle applications development framework.

1.3.1 Rapid Application Development(RAD)

RAD[18] may be defined as the process of building and refining a working model or prototype of the final software system during the development process. The main purpose of prototyping is to refine functions, inputs and outputs during the design phase without having to wait for development to be completed. However, prototyping is not a

substitute for good analysis and design, but rather it is another way of producing results. If used properly, prototyping can be an effective tool and an aid in developing systems that allow closer user participation in the process, leading to information systems that meet the needs of the business.

Prototyping has been an informal methodology for quite some time. However, over the years, more experiences are gained in this area, and now it is possible to come up with some form of requirements or standards. The reason for prototyping is that the formal lifecycle is actually delaying the delivery of the final product. It is becoming the major cause of the application backlog. Moreover, the elapsed time between requirements and a delivered product erodes a customer's confidence. Perhaps, people are more impatient and pragmatic these days and would like to see some form of results earlier on. Gladden[18] suggests delivering any form of a prototype as quickly as possible. This approach is typified by Gilb[9] and Martin[17,18].

An approach to making prototyping successful was developed by Du Pont in 1985, called RIPP[3]. The approach was developed around the use of a CASE tool - CorVision from Cortex. A proposal and definition report was drafted between 10 to 15 days before proceeding to prototyping. The timebox is basically an iteration development process of the prototype limited to a maximum of 90 days before being evaluated again. Du Pont's first project using RIPP was completed in 5 man months compared with the 28 to 36 months using traditional approaches. This approach has saved them $2.3 million over 3 years, in 15 systems at 9 sites.

The RAD lifecycle has 4 phases[3] as applied in RIPP :

(a) Requirements Planning
(b) User Design,
(c) Rapid Construction,
(d) Transition.

During the first phase, developers create an outline model of the chosen area and define the scope of the planned system. Business executives, users, and developers take part in workshops(called the Joint Requirements Planning Workshop - JRP) that progress through a structured set of steps. All the results of the workshops are recorded using an integrated CASE(I-CASE) tool. The I-CASE tool is a repository for requirements and specifications. This stage usually takes one to three weeks.

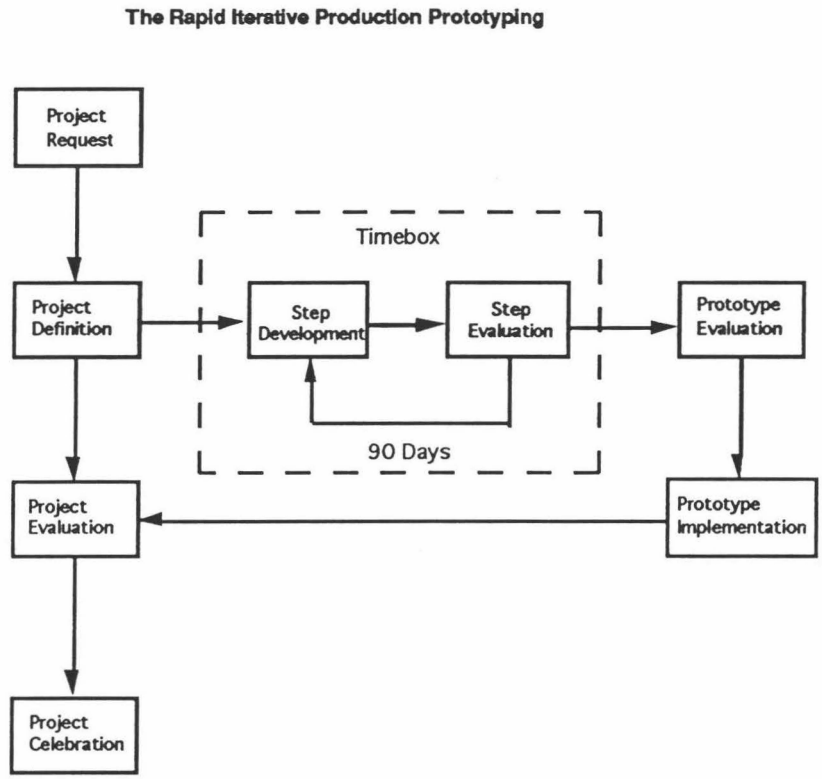


Fig. 1.2

The User Design stage requires that end-users participate strongly in the nontechnical design of the system under the guidance of an IS developer. User Design is done in a Joint Applications Design(JAD) workshop, which completes the detailed analysis of business activities and develops the outline design of the system. The information recorded in the I-CASE tool is used as input and is further refined. This stage usually lasts three to five weeks.

The third stage involves the design and implementation of the proposed system, which was outline in the previous stage. The software is constructed using an iterative technique. Finally this stage includes activities needed to prepare for cut over to production status. The I-CASE tool is used to generate the application code from database definitions.

When the system is cut over in the last stage, a variety of actions is needed, including comprehensive testing, end-user training, organisationanal changes and operation in parallel with the previous system until the new system settles in.

Prototyping approaches have the following advantages:

(a) improved developer user communications
(b) increased developer productivity
(c) working model versus a paper model
(d) model iterations

(e) user specification is changeable at any stage
(f) reduction in user training due to early participation
(g) production of error-free applications

However, the disadvantages are :

(a) configuration management and version control of prototypes is more difficult than with conventional development. Prototyping can result in many trial systems. It is possible to get versions mixed or to be unable to recover an earlier prototype version. Configuration management software can reduce this problem
(b) keeping documentation up to date may be difficult because of its rapidly changing and iterative nature
(c) maintaining discipline and objectives in the development team is difficult because it is possible to become distracted from the legitimate goals of the prototype due to the fluid nature and constant demands of prototyping
(d) Planning and allocating resource is difficult in an environment dealing with uncertainty and unknown
(e) ultimate testing may be neglected and left to the users.

Incidentially, a RAD approach has also been integrated into Information Engineering by Texas Instruments (James Martin Associates).

1.3.2 IBM AD/Cycle-Repository

In Sep 1989, IBM became a standard bearer for the computer-aided software engineering(CASE) industry by laying out its plan for the software development process. AD/Cycle-Repository[38,53,54] is an integrated framework intended for a CASE environment, and compatible with a range of development tools and techniques from many vendors. The goal is to vastly improve productivity in the applications development process. The only way to achieve this is to automate code generation through the use of models rather than conventional programming. Also it standardises repository storage of development objects. All CASE tools from other vendors, in order to link to AD/Cycle, must comply with certain IBM standards. However, no attempt has been made to create a standard in the methodologies themselves.

The primary benefit of the open repository-based environment is that users should be able to plug tools developed by CASE vendors complying with the repository standard into the environment and then use them together. CASE tools supporting various methodologies use the services of the Repository Manager to store user-defined application knowledge. The information contained in these models is stored in standard

format within the Repository Manager, from which it will be ultimately used to drive a code generator.

However, until now, it has not become popular due to a number of reasons. The MVS Repository Manager is not a stable product. Only a few CASE tools are compatible and it is difficult for other vendors to plug their CASE tools into the Repository. There is also problem with LAN configuration which is a important desired feature because today's CASE tool is geographically dispersed. Vendors with CASE tools running under MS-DOS and Unix have to rewrite them for OS/2EE for IBM PC and SAA compliance. One problem is that until now OS/2EE has not been popular.

While IBM is promoting integrated CASE in a mainframe environment, Digital Equipment Corp is following a more distributed path[53]. DEC's integrated CASE standard is known as ATIS(A Tool Integration Standard) and CDD/Repository in the VAX/VMS and Ultrix enviroments.

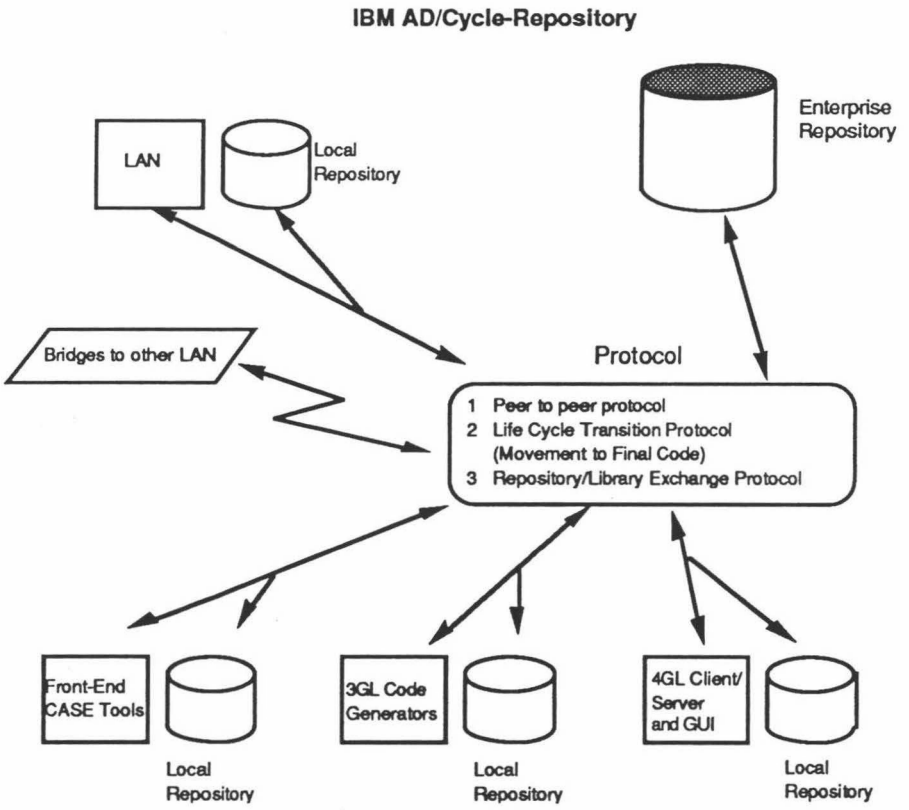


Fig. 1.3

1.4 Object-Oriented Methodologies

Recent suggestions[27] have been made that methods based on the paradigm of functions acting on data should be superseded by object-oriented approach. Object-

oriented methodology is defined as an application development strategy that models both requirements and software solutions as collections of objects that contain both data structure and behaviour.

However, many software organisations have developed standards and methods based on the functional approach and are understandably reluctant to embark on some design techniques that are still immature and unproven. Hence, any migration to new methods is likely to be a gradual one.

Current application of the OO paradigm has been limited to Design and Implementation due to the widespread use of C++ and Smalltalk in a small scale environment. Less has been done on the Analysis, although this is crucial for the construction of large and complex OO Information Systems.

The Object-Oriented development cycle is covered, in particular, by Booch[4], Budd[5], Henderson-Sellers[10,40], Korson[43], Jacobson[12,41], Bailin[31] and Coad & Yourdon[7,8].

1.4.1 Booch Methodology

Early versions of the methodology, proposed by Grady Booch were centered around Ada. In his most recent book, Booch introduces four models to capture OO semantics, which are then mappable to several target OO software environments.

1.4.2 Rolland & Brunet O* Model

This metholodogy[52], by the two authors at the University of Paris, concentrates on development for OODBMS, particularly the O2 system.

1.4.3 Coad & Yourdon OOA and OOD

This methodology[7,8] has been widely published through two books, one each on Analysis and Design, and a CASE tool has been developed.

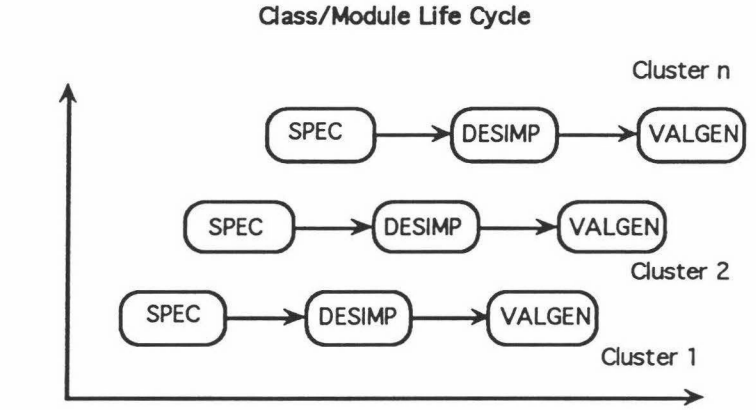
1.4.4 GE Labs Object Modelling Technique(OMT)

This technique[24] is developed by Rumbaugh, Blaha, Premerlani, Eddy and Lorensen at General Electric R&D Center, Schenectady, New York. Originally, this technique[34] was meant for use with relational database but has been modified to suit the object-oriented one.

1.4.5 Bertrand Meyer OO Methodology

Meyer's object-oriented methodology is centered around his OOPL, Eiffel. Not much is discussed about OOA. However, he claims that Eiffel language can both handle OOD and implementation[19]. The reason being the items of interest in each phase are

the same : objects. Objects and relationships between objects are identified in both the analysis and design phases. The cluster model has been proposed by Meyer as a life cycle for a tightly related group of classes, or cluster, in which three phases are identified.



Cluster model proposed by Meyer.
SPEC = specification; DESIMP = design+implementation; VALGEN=validation+ generalisation

Fig. 1.4

First, a specification is written by the systems designer(SPEC), then this is designed and implemented(DESIMP)(one process in a language like Eiffel) and finally it is validated and generalised(VALGEN). This life cycle occurs for different clusters of classes at different times. For example, a window cluster and a graphics cluster of classes could be specified, designed and implemented and then validated and generalised at different times. These phases are also iterative with refinements added.

1.4.6 Ivar Jacobson Object-Oriented Development

Ivar Jacobson come out with an early version of OO systems development in 1987. This technique originates from his work at Ericsson Telecom and since then has been used extensively within the whole Swedish telecommunication industry.

Basically, this paradigm describes a system as a set of properly interconnected blocks - each building block representing a packaged service of the system. A block may itself be made up of other, low-level blocks or by components. Components are standard modules which can be used for many different applications. The lowest-level blocks are made up of components only. Blocks as well as components are naturally implemented as classes using object-oriented programming. The designers are consequently provided with a set of components when building applications by means of blocks.

There is one interesting assumption made in this paradigm with regards to object concurrency. It is assumed that there is an infinite processor capacity, the execution speed is immensely high and endless storage volume. In this way, parallelism can be

disregarded and the course of events may be serialised. This assumption may not be adequate for the general case.

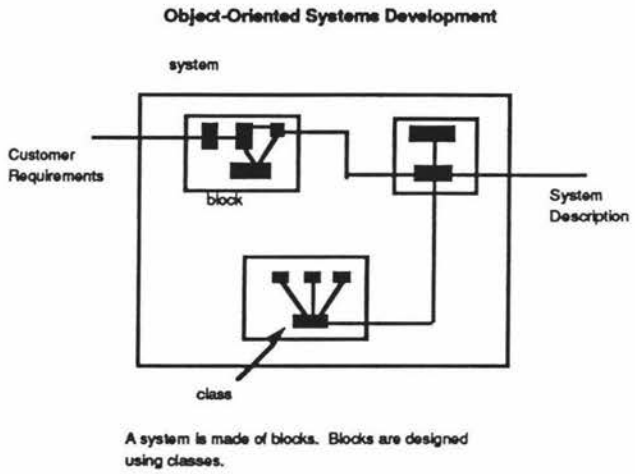


Fig 1.5

1.4.7 Henderson-Sellers OO Life Cycle

This methodology is developed at the University of New South Wales, Australia which describes the life cycle of OO systems devleopment. It focuses more on the front-end and high-level analysis. There are seven proposed steps to follow and earlier efforts made by Coad & Yourdon, Shlaer & Mellor, Bailin, and Wirfs-Brock are applied in these stages :

(a) Undertake object-oriented system requirements specification,
(b) Identify the objects and the services each can provide(interface),
(c) Establish interactions between objects in terms of services required and services rendered,
(d) Analyse stage merges into design stage : use of lower-level entity data flow diagrams/Information flow diagrams,
(e) Consider the bottom-up concerns and use of library classes,
(f) Introduce hierarchical inheritance relationships as required,
(g) Aggregate and/or generalise of classes.

The last step is illustrated using a fountain model rather than the traditional waterfall model.

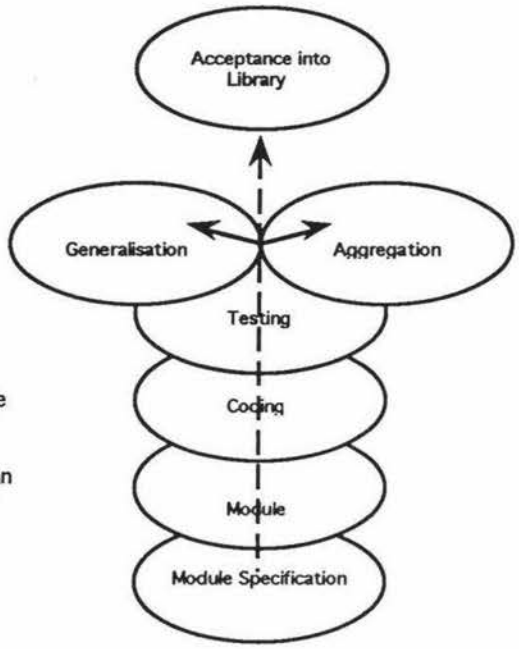


Fig. 1.6 Fountain Model

1.4.8 Summary of OO Methodologies

Booch and Henderson-Sellers have agreed that object-oriented design embodies an incremental, iterative process in between successive stages. Both Jacobson and Henderson-Sellers' ideas are particularly suitable for developing very large object-oriented software systems(>10 man-years). Another interesting point to note is that boundaries of analysis, design and implementation stages in object-oriented software development are blurred. Examples are Coad & Yourdon, who overlap between object-oriented analysis and design. Booch combines design with implementation.

Henderson-Sellers has identified the iterative process and comes out with the fountain model to replace the classical waterfall model. Development reaches a high level only to fall back to a previous level if so needed, to begin the climb once again. This is a better model of reality then the traditional waterfall model. Firstly, it provides a diagrammatic version of the stages present in an software life cycle and a clearer representation of the iteration and overlap made possible by object-oriented technology. Secondly, since the foundation of a successful software development is its requirements analysis and specifications, this stage has been placed at the base of the diagram. The fountain model can also be extended to the life cycle of a module, as outlined in Section 1.4.5.

Some of the advantages of object-oriented paradigm at the Analysis level[7,43] are:

(a) It can handle more complicated problem domains; emphasing more on the understanding of problem domains since it is based on objects, and not just functions or processes alone,

(b) It can improve interaction between analyst and client since it organises analysis and specification using the methods of organisation that pervade people's thinking,

(c) It can increase the internal consistency of analysis results. Object-oriented analysis introduced by several authors have consistent diagramming,

(d) The results obtained in Analysis can be reused on some similar projects.

Some of the advantages of object-oriented paradigm at the Design level[8,27,43] are :

(a) Object-oriented design is actually a continuation of the efforts made at the Analysis stage,

(b) Results and experiences gained during the Analysis stage can be reused,

(c) Object-oriented prototyping is used which increases productivity,

(d) Low life cycle cost,

(e) Modularity,

(f) Maintenability.

1.5 Conclusions

Most of the well-known OO methodologies have been given a brief introduction. While all of them offer approaches to extended data and behavioural modelling, none of them seem fully adequate to address the issues specifically related to OO database applications development. They have also not mentioned the guidelines and the steps involved in the prototyping process.

Chapter 2 : Required Features of an OODBMS Methodology

This chapter suggests and discusses what ingredients are necessary in an object-oriented database methodology and how different each component needs to be from the conventional counterpart. These ideas draw on many sources of existing OO methodologies and the existing OODBMS methodology[52] covered during the literature search.

2.1 Support for development in stages

An OODBMS methodology should be developed in stages. In conventional methodologies used with both functional and database approaches, there are discrete steps that progress from one to another. In OO development, the phases may be blurred because the same objects are under study throughout. The only difference between stages is the enrichment of information as the development progresses towards the deliverable.

Nevertheless, it seems desirable that any OO methodology should recognise not only Analysis, Design and Implementation stages, but also a preliminary Feasibility stage as included in most conventional methodologies.

2.2 Class Identification

An OODBMS should have provisions for class identification. The inclusion of a recommended approach to identifying suitable object classes is an important requirement. The parallel stages in conventional methodologies are identification of entity types and functions. The identification process needs to be defined in the form of guidelines as to groups of object classes which should be looked for in some description of the problem domain.

2.3 Relationships Identification

Just as in a conventional database-oriented methodology, the next requirement is to identify relationships, "is-a", "part-of", and "instance of". This process would be similar to the relationships identification in most OO methodologies. Because of the importance of inheritance in target OO software environments, and because of the relationships support many OODBMS give for complex data types, the "is-a" and "part-of" relationships assume a greater role than in conventional methodologies. There is often also a need to identify "instance of" relationships as individual object behaviour may be related to class behaviour.

However, consideration of other types of relationships ought not to be discarded as they represent an important part of the semantics of the problem domain.

A modelling structure is then required to cover the identified relationships (the "static" part of the Analysis. The structure includes a range of diagrams and templates for designer-user, designer-designer and designer-CASE tool interaction. Ideally, these facilities should represent an evolutionary development of existing ones rather than totally new concepts.

2.4 Behaviour modelling

Behaviour modelling should be supported in the OODBMS methodology. This is the area which appears to involve the most differences from a conventional approach, since "procedures" are no longer independent elements in the modelling system, and must be encapsulated with objects or object classes.

"CRUD" - Create, Read, Update, Delete - as used in many current methodologies - represents the easy part of the problem, as these functions can be encapsulated relatively simply to the objects(simple or complex) that they act upon.

The more critical requirement is for approaches to cover the following :

(a) multi-object transactions, where a unit of work can be identified in the problem domain which requires all subtasks to be completed successfully
(b) asynchronous triggering and messsage passing, where actions on one object lead over time to actions on other objects
(c) synchronisation of parallel threads of activity.

Modelling concepts in addition to the main class structure are needed, involving such things as intra-object behaviour, inter-object message passing, state transition and event dependency. Again it is desirable that existing concepts should be evolved whenever possible.

2.5 User Interface Development

In database-oriented methodologies, the user interface has tended to be peripheral to the main analysis model, and is brought in at the design stage in a fairly separated fashion. With an OO approach, the user interface can be expressed as a set of objects and classes at a number of levels of abstraction, in both the Analysis and Design.

The methodology should provide for such objects to be included in the main static and behavioural models.

2.6 Diagramming conventions

A recommended diagramming notation covering all the features suggested for object-oriented systems development should be enforced throughout to model both the static and dynamic aspects of the system, i.e. the analysis and design results.

2.7 Object-Oriented CASE Tools

It is desirable that the methodology is supported by an OO CASE tool. Existing tools can be classified into front-end and back-end. These are used to construct the model which would at the same time checked for the semantics. At the present moment, very few are widely marketed.

2.7.1 Tools for analysis and design(front-end)

One of the desirable characteristics of tools at the front-end is a graphics-based system supporting object-oriented design notation. This will enforce the notational conventions of OOA and OOD, and maintain control over the design products, and co-ordinate activities of a team of developers. It can be used throughout the life cycle as the design evolves into a production implementation. Such a tool is also useful in system maintenance.

2.7.2 Tools for implementation(back-end)

Implementation normally requires the use of some language, usually but not necessary an OOPL. A number of languages are accompanied by toolkits which support the low level design and implementation process. Some of the desirable characteristics of tools at the back-end are as follows :

(a) An object browser that knows about the class structure and module architecture of a system. Class hierarchies can become so complex that it is difficult even to find all of the abstractions that are part of the design or are candidates for reuse.

(b) An incremental compiler to handle minor changes rather than to recompile the whole program again which may be very time consuming for a large development.

(c) Debuggers that know about class and object semantics and support for multiple threads of control processes. The tool should permit the developer to exert control over the individual threads of control.

(d) Configuration management and versioning tools for large projects.

(e) Class library browser that allow developers to locate classes and modules in the library as they are developed. This is essential as a project matures,

and the library grows as domain-specific reusable software components are added over time.

2.8 Object-Oriented Prototyping

The technique of OO prototyping should also be incorporated as part of the OODBMS development methodology because it is an inexpensive and quick way of demonstrating the likely functionality of a final system[20]. This approach is especially suitable with OO paradigm. The prototyping tools therefore must be capable of generating a demonstratable system before all system classes are fully defined.

A number of OODBMS prototyping tools are described in the Appendix A.

2.9 Object Repository

An OODBMS methodology should support the concept of Object Repository. The Object Repository[29] should be the foundation of any application development and is vital for OO prototyping.

It is a centrally controlled data store which contains all object components built-up from both current Analysis and Design work, and past projects developed using different prototyping tools across various platforms. Early in the lifecycle, information already in the Repository is extracted and put to use. Information developed during planning is stored in the Repository for later use in Design. Information produced in design is stored in the Repository for use in the construction of the application. This technique enables fast development by reusing existing templates, structures, models and designs.

The Repository ensures consistency among diagrams and helps to enforce technical quality. It enables the integration of prototyping tools, code reuse and the automation of OO software development process.

As more and more objects are added, a classification utility should be provided to group similar objects together so that searching is made easier.

Local Repositories are called the Class Libraries and are built into the prototyping tools used.

2.10 Support for Reusability

Reusability can only be supported by utilising Class Libraries and Repositories. The object-oriented paradigm combines design techniques and language features to provide strong support for reuse of software modules[61]. A schematic view of this process is shown in Fig 2.1.

This process, quite unlike traditional methodologies, involves reference to objects created in other problem domains. There are two aspects :

(a) Designing objects in one domain with a view to their being reusable by other domains. This typically involves the inclusion of higher-level generic objects, which are not a strict requirement of this domain, but introduce a level of generality which makes re-use possible.

(b) Finding suitable generic objects in the Object Repository which can be specialised for use in the new problem domain.

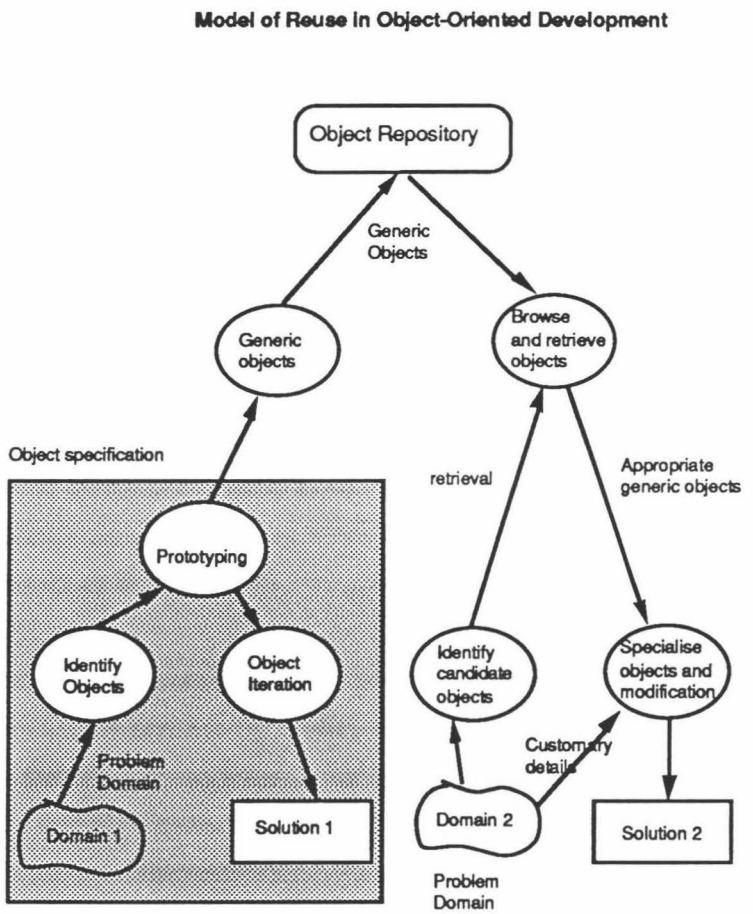


Fig 2.1

To enable reuse the new application has to be analysed to establish the differences between the generic objects in the Repository and the new applications, so that the appropriate specialisation can be carried out. Furthermore, some parts of the generic objects may be inappropriate for the new application. In this case, attributes and methods of the generic objects would need to be modified to suit the specialisation.

The advantages of re-use are discussed in more detail in [43]. The requirements on an OO methodology can be summarised under two headings :

(a) the use of an Object Repository as a central feature, with classification and searching facilities (Section 2.9).
(b) the inclusion of sub-tasks, both to search for off-the-shelf objects, and to create generic objects for later re-use.

2.11 Support for use of OOPL

In order to support OOPL, the methodology must be able to map the concepts it deals with at the Analysis and Design stages into concepts used by the implementation language which is normally an OOPL.

Although iteration, exception handling and parameterised classes have not yet been implemented in some OOPL languages, an OO methodology should support such features that might be implemented in later versions of the languages.

2.12 Support for use of OODBMS features

OODBMS, extended relational DBMS and other target environments are not uniform in the concepts they directly support. Examples are :

(a) limited levels of encapsulation,
(b) explicit rule and trigger systems,
(c) explicit versioning.

These non-standard features occur primarily in the extended relational databases like Postgres[56,57,58] and Starburst[46].

There are good arguments for including explicit rules and version relationships in the OO Analysis and Design model. However, if targeting an OODBMS, these concepts have to be "mapped out" into methods or supporting classes.

Likewise, encapsulation that cannot be supported in a target DBMS needs to be "mapped out" into an embedding software structure that could be regarded as belonging to an aggregate "whole system" controller object.

Chapter 3 : Introduction to Object-Oriented Methodologies

No single approach to Object-Oriented Methodology has yet reached widespread acceptance[40]. A number of approaches can be considered as possible "leading contenders", and this chapter provides a brief introduction to some of these :

(a) Grady Booch[4]
(b) The O* Model by Collete Rolland and Joel Brunet[52]
(c) Coad and Yourdon[7,8]
(d) GE Labs OMT[24]

Other proposals, not discussed in detail here, include Meyer[19], Shlaer & Mellor[55], Jacobson[12,41], Wybolt[63], and Hayes & Coleman[39]. Also the Object-Oriented System Development methodology proposed by Henderson-Sellers[10,40] has already been mentioned in Chapter 1. Some of the above still favour having a functional design as part of their object-oriented methodology[55,63,39,24]. A comparison of (a) through (d) will be provided later in the chapter.

3.1 Booch Methodology

Booch[4] defines a class as a set of objects that share a common structure and a common behaviour while an object is an instance of a class. A method is an operation upon an object, defined as part of the declaration of a class.

Booch does not say a great deal about object-oriented analysis, but concentrates more on design and implementation issues. Booch introduces four diagrams which form the basic notation of object-oriented design out of which the first two are most frequently used and important. They are the (i) class diagram, (ii) object diagram, (iii) module diagram, (iv) process diagram. The first two forms the logical view of a system while the last two are used to describe the physical structure of the system and the software and hardware implementation. The static and dynamic semantics for the class and object diagram are also represented.

In the class diagram, Booch uses chained cloud icons to represent abstractions of a real-world class entities. There are also interface and implementation classes, characteristic of OOPL concepts. Metaclass relationship is another type of relationship which is used to represent a class of a class relationship. In the below example, timer belongs to the metaclass of clocks. The clock metaclass is shown in shaded cloud. Although this class is usually not explicitly mentioned in most problem domains, it is

required in real-time applications. Others include utility class which represents a single free subprogram or a collection of such free subprograms.

The two important relationships, aggregation("is-part-of") and generalisation ("is-a") are supported. Generalisation is represented by inheritance while aggregation is represented in the attributes of classes.

For a simple example, fig 3.1 shows a security system which uses the class monitoring interface for the interface part and classes sensor, camera, and alarm for implementation. Infra-red sensor is a subclass of class sensor which inherits attributes and behaviour from the class Sensor and ACamera is an instantiation of the class Camera. Cardinality of relationships is also shown.

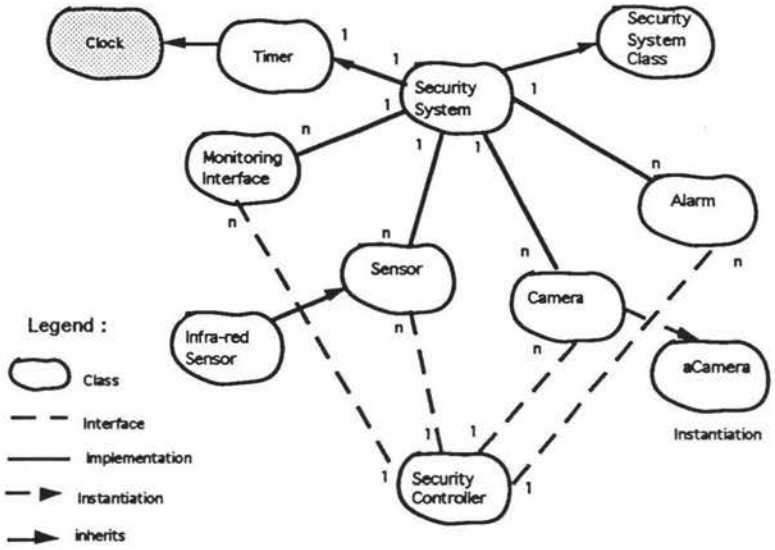


Fig. 3.1

Here, security controller is viewed as a transaction object for the purpose of separating direct visibility between classes and also to coordinate activities.

Notice that the attributes and operations are not demonstrated in the cloud icons. Details are left to be defined in the class template to avoid untidiness. Below is an example of templates used to define the classes.

Name : Alarm
Cardinality : n
Hierarchy :
Superclass : Object
Public Interface :
Uses : Security Controller
Operations : respondtoAlarmFault
activateAlarm
resetAlarm
Implementation :
Uses :
Attributes : alarm_id

alarm_location

Operations :

Concurrency : active

Fig 3.2 Template for the class Alarm

A good approach is to review the list of key abstractions and select only those that represent the largest conceptual chunks, that is elements at the highest level of abstractions. Each of the cloud icon may be further decomposed to review more details of the system structure if necessary. This is analogous to functional decomposition in data flow diagrams.

State transition diagrams are used adjunct to the class diagram to model the dynamic aspect of the system.

State Transition Diagram for the Class Alarm

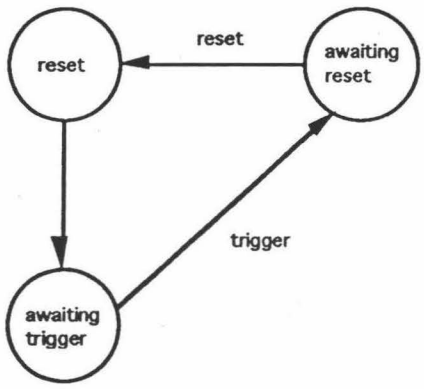


Fig 3.3

The object diagram is used to show existence of objects and their relationships in the logical design of a system, representing a time-lapse snapshot in time of an otherwise transitory event. Object visibility and synchronisation are also indicated inthe object diagram. Object visibility is about how two objects communicate with one another in their interface fields, lexical scope and passing parameters. Messages sent may be classified as simple, asynchronous, synchronous, balking and timeout. Objects are drawn in plain lines instead of chained lines as in class diagram. Nesting of object icons within another to support aggregation is another supporting feature for OOPL to utilise information hiding of operations belonging to a higher class. Individual object may be further decomposed to show the structure of the constituent objects and the breaking up of messages. This is analogous to data flow diagrams, but messages are used instead of data.

Booch also uses timing diagrams to show the duration and sequence of the operations of objects against time. It would not be elaborated here.

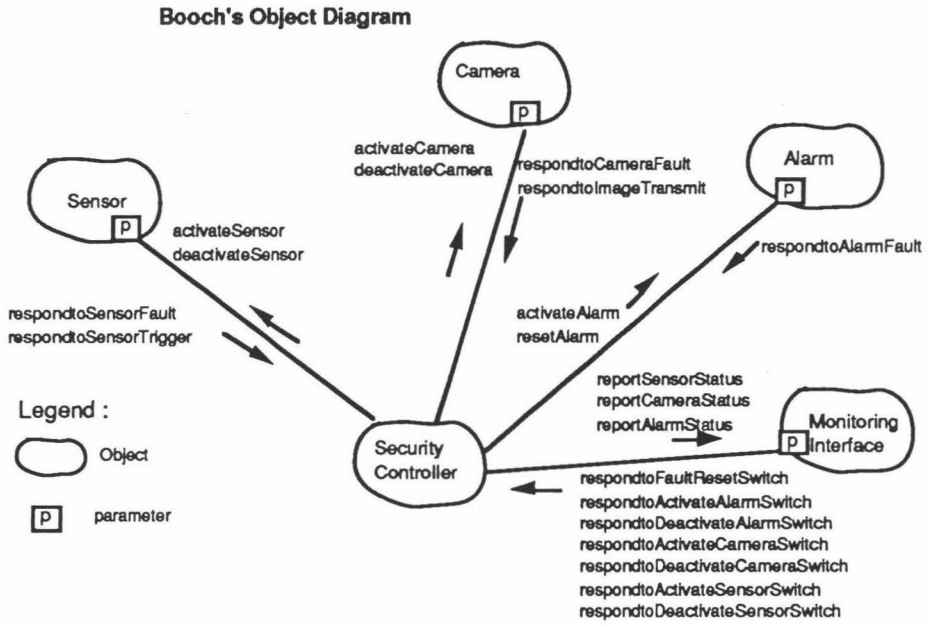


Fig. 3.4

A module diagram is used to show the allocation of classes and objects to modules in the physical design of a system. Some object-based and object-oriented programming languages support the concept of a module as separate from a class or object. This construct may be as simple as separately compiled files in C++ or as sophisticated as the idea of packages in Ada. It is the responsibility of the designer to decide how to allocate classes, objects, and other declarations to physical modules. Languages that do not support modules clearly do not require this notation. The two most important elements of a module architecture are modules and module visibility.

The process diagram is used to visualise and then reason about the problem of allocating processes to processors in the physical design of a system. The three most important element of process architecture are processors, devices and connections. A processor is a piece of hardware capable of executing programs; a device has no such computing power. Processors and devices communicate with one another. Details of each kind of element are again described in templates. The process diagram is more about showing hardware configuration of a system.

3.2 The Database Object Model by Rolland & Brunet

Collette Rolland and Joel Brunet's methodology for object database design is presented in the Object model. In their definition, an object is fully described by an object scheme, its life-cycle and its identity.

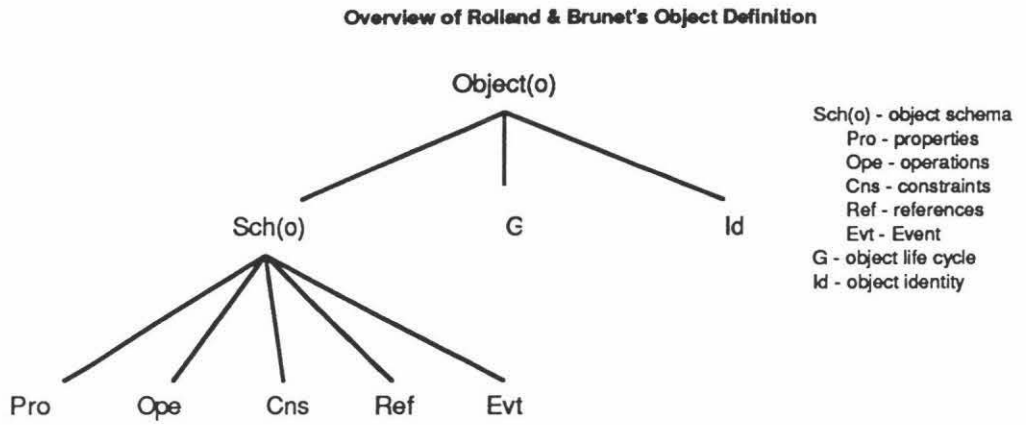


Fig. 3.5

The object scheme of an object is further described by its properties, a set of possible operations on the object, constraints to be verified by the object, static links consisting of a set of references of the object with other objects, a set of events which may be stimulated by particular state changes of the object. The first three items, **Properties, Operations,** and **Constraints,** characterise the local static aspect of the object while **Reference** and **Event** specify the dynamic aspect of the object. Properties, operations, constraints, reference are shown using in a **static interrelations diagram.** This is not very sophisticated as compared to OMT or Coad & Yourdon's class model for analysis and hence it would not be illustrated here. The last item is shown using their notation of a dynamic interrelations diagram. It shows to a certain extent of object visibility but is not very helpful on message passing compared with Booch's object diagram. An example of a dynamic interrelations diagram is shown in Appendix B.

A complete description of the object scheme is done on the template. An example of a class descriptor for a generic object is shown in Fig 3.6. The state of the object is the concatenation of all its properties and reference values at a given time. State changes of an object are triggered by an internal or external event. This behaviour may be represented by state transition diagrams and concurrency by Petri-nets although they are not explicitly shown.

Every object is given an identifier. This is similar to the concept of a surrogate key for a relation in extended relational databases.

Rolland & Brunet also provide some guidelines on how to reach the object schemes specification for the analyst. However, these guidelines are not elaborated in their paper. These refer to an inventory of initial objects, identification of final objects, identification of operations and identification of events.

A sample of the O* model textual description of Objects

object Object name

properties
Contain the names of all the attributes of the object and the definition of the data type

operations
Contain the names of all the methods

constraints
These may be local constraints, a limiting value on the fields or inter-object constraints

references
Refers to other objects

Fig. 3.6

Rolland & Brunet's methodology is more suitable for object database design particularly because it proposes a textual description of objects which contains triggers(rules in Postgres) and operations(functions in Postgres). The concept of references is also used, which is typical of OODBMS.

3.3 Coad & Yourdon's Methodology

Coad & Yourdon's OO Methodology is divided into Analysis and Design. They have introduced five steps in real-world abstraction. They are :

(a) Classes
(b) Structures
(c) Subjects
(d) Attributes
(e) Services

Some guidelines are used for identifying classes. The potential classes in a problem domain are structures, other systems, devices, things or events remembered, roles played, operational procedures, sites and organisational units. The first four are more important.

"Structures" are commonly used in data modelling. They are formed by the "is-a" and/or "has-a" relationships among objects in a hierarchical manner.

"Other systems and devices" include a related system or single entity that the system under present consideration interacts with. Examples are the user-interface system or a sensor that forms part of a data logging system.

"Things or events remembered" over time are recorded in storage objects. These storage objects may also be called data objects, and are usually persistent.

"Roles played" by objects are termed as temporal objects which exist as a result of relationships.

In their book, Object-Oriented Analysis, the above five steps are applied in the problem domain component. Besides the problem domain component, which is known as the schema in database terms, there are also other components that need to be examined which make up an object-oriented systems. The other three components human interaction, task management and data management are further discussed in their book, Object-Oriented Design. The book also demonstrates the implementation of all the four components using OOPL.

3.3.1 Object-Oriented Analysis

Much work on OOA has been carried out by Coad and Yourdon. In fact, the authors acknowledge that OOA is a relatively new method of managing real-world complexity, and that their recommendations should be tailored whenever to suit the organisation or project needs.

It is essential to identify objects and their classes within the problem domain. Some useful hints of finding objects occur in the client's summary like singular or adjective noun, things that have a structure, things that interact with other systems, devices itself, human or things that played a role, systems that require remembrance of particular operations or excercise sequencing, physical location or sites, and organisation units. Recognition of objects will become familiar with practice and a fair understanding of the environment under consideration. Besides these, one would also need to consider the behaviour, attributes, services offered by the identified objects. An object class contains instances whereas an abstract class does not.

Three important concepts of identification of structures are Generalisation, Specialisation and Whole-Part Structures. The directional notation for demonstrating Generalisation-Specialisation is shown in the below example. The superclass is drawn with a line outward from a semicircle midpoint to point to the subclass.

Consider a certain class of motor as an example. Assuming that there are two types of motors, the standard motors(class A and B) and the submersible ones(Class E). For the submersible ones, the operations are more critical and the standards more stringent. The superclass is assumed to provide only the common attributes without having instances itself. Hence, the class Submersible besides having all common attributes and

behaviour of the standard motor, will have other critical attributes of its own, shown in Fig 3.7. This illustration may further be redrawn to Fig. 3.8.

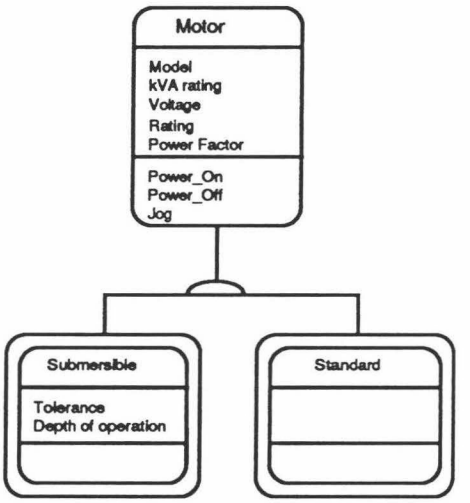


Fig 3.7 Using a Class as a generalisation

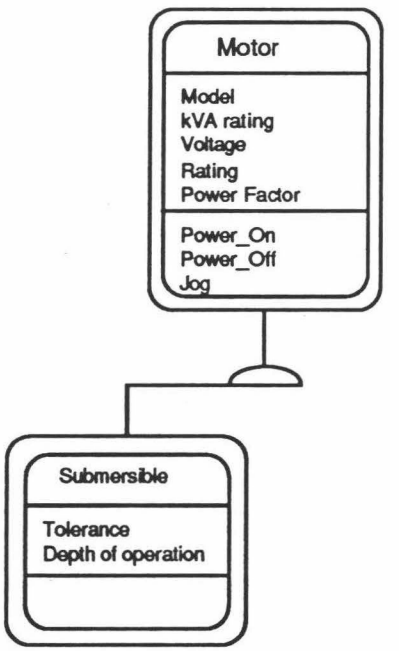


Fig 3.8. Using a Class Object as a generalisation

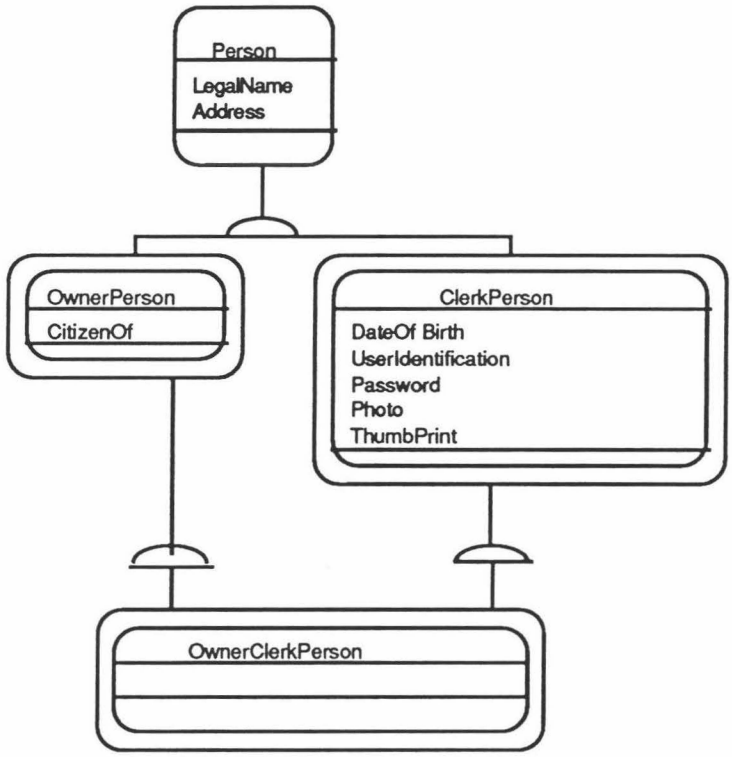


Fig 3.9. Person Gen-Spec structure, as a lattice

To consider whether an Object Class needs Generalisation or Specialisation check whether it is in the problem domain, and if there is inheritance between different object classes. In practice, the most common form of Gen-Spec structure is a lattice as shown in fig. 3.9.

Whole-Part Structure is shown with a whole Object at the top, and then a part Object below, with a line drawn between them. A triangle marking shown in Fig 3.10 and Fig 3.11 distinguishes Objects as forming a Whole-Part Structure. Each end of a Whole-Part structure line is marked with an amount or range, indicating cardinality, at any given moment in time.

In Fig 3.10, an aircraft is an assembly of possibly no engines (a glider) or at most four engines(Boeing 747) and an engine is part of possibly no aircraft or at most one aircraft. In Fig 3.11, an organisation is a collection of possibly no clerks or at most many clerks, and a clerk is a member of exactly one organisation.

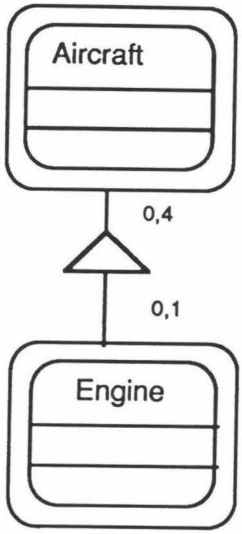


Fig 3.10

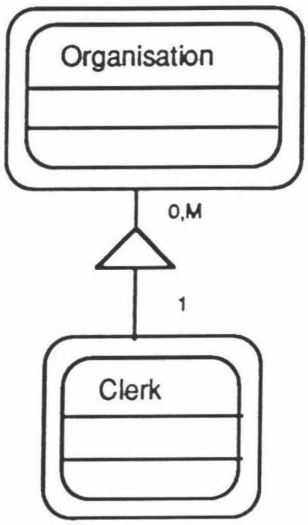


Fig 3.11

A subject is a mechanism for guiding a reader(analyst, problem domain expert, manager, client) through a large, complex model. Having discovered so many objects, there is a need to guide the analysts through the documentation of the project under consideration.

An attribute is data(state information) for which each Object in a Class has its own value.

A Service is a specific behaviour that an Object is responsible for exhibiting. Services are called methods in Object-Oriented Programming Languages. It is best to describe the services of each Object by a state diagram. This is in the form of conditions, procedures, and loops for which algorithms for the program may be worked out.

Message connection models the processing dependency of an Object, indicating a need for Services in order to fulfill its responsibities.

As mentioned above, Booch's modelling technique supports most important features provided by conventional extended Entity-Relationship diagram.

3.3.2 Object-Oriented Design

Besides, the problem domain component, Coad & Yourdon also recognises that there are other components. The five layers of real-world abstractions, subject, class, structure, attribute and service are again applied to them.

Some of the concepts in the proposed methodology in Chapter four are analogous to these three components. The human interaction component is similar to user-interface object, the task management component is similar to the controller object, and the data management component is similar to classes for data storage, which could be a container class.

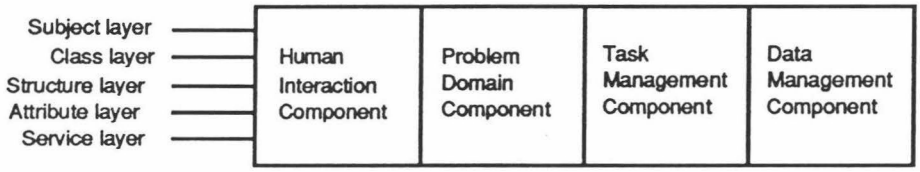


Fig 3.12 Four components and five layers

3.4 Object-Modelling Technique(OMT)

This software engineering methodology has been developed by staff at the General Electric Research and Development Center and has proven its effectiveness[9]. One of the interesting thing is that OMT covers all three stages of Object-Oriented Analysis, Design and Implementation. In addition, there is also a System Design stage in between the OOA and OOD stages.

Three models are needed to fully describe a system. They are the Object Model, the Dynamic Model and the Functional Model. These models are built initially in the Analysis stage and are made use of again in the Design and Implementation stages.

The Object Model of OMT is fairly similar to Coad & Yourdon's OOA, the same ideas being applied but at different places. However, OMT includes a Functional Model which consists of data flow diagrams.

Like every methodology, the Analysis stage begins by drafting an initial description of the problem. An Object Model is then built, together with the data dictionary for each defined object. The idea of a data dictionary, which can be called class descriptor, is present also in Booch. It is however, not explicity mentioned in Coad & Yourdon. The steps involved in building the Object Model are :

a. Identify Object Classes,
b. Create a data dictionary describing classes, attributes, and associations,
c. Add associations between classes,
d. Add attributes for objects and links,
e. Organise and simplify object classes using inheritance,
f. Test access paths using scenarios and iterate the above steps as necessary, and populate with some data,
g. Group classes into modules, based on close coupling and related function.

The notations used for semantic data modelling is slightly different from that of Coad and Yourdon. The Dynamic Model describes the dynamic behaviour of the objects once the Object Model is completed. The steps involved are :

a. Prepare scenarios of typical interaction sequences,
b. Identify events between objects and prepare an event trace for each scenario,
c. Prepare an event flow diagram for the system,
d. Develop a state diagrams for each class that has important dynamic behaviour,
e. Check for consistency and completeness of events shared among the state diagrams.

The only feature that is worth a special comment is the preparation of a scenario and the event trace diagram. This technique and convention is useful for modelling real-time object-oriented systems in which the complete cycle of the system can be pre-determined. This concept of scenario and event trace diagram is also brought up by Jacobson.

The steps involved in constructing the Functional Model are :

a. Identify input and output values,
b. Use data flow diagrams as needed to show functional dependencies,
c. Describe what each function does,
d. Identify constraints,
e. Specify optimisation criteria.

Data flow diagrams may be constructed using Teamwork, Excelerator or Information Engineering Workbench(IEW).

After obtaining these three models, further iteration may be done on the attributes, operations and developing scenarios to further verify these three models. The Analysis stage ends with preparing the document that describes the whole problem domain, the proposed solution in terms of the three models.

After a problem is analysed, it must be decided how to approach the design. System is the high-level strategy for solving the problem and building a solution. System design includes decisions about the organisation of the system into subsystems, the allocation of subsystems to hardware and software components, and major conceptual and policy decisions that form the framework for detailed design. The System Design is similar to Booch's Process Diagram. This stage should discuss :

(a) the type of processors to be used,
(b) the selection of the different types of hardware or software,
(c) load balancing in a mult-processors environment,
(d) the type of user-interface like Sunview or X-Windows,
(e) hardware configuration,
(f) systems management.

Once the Analysis stage and the Systems Design is planned out, it is almost ready to implement the application. The OMT adds another stage called Object Design whereby the analysis model is elaborated and provide a detailed basis for implementation. Object design starts a shift away from the real-world orientation of the analysis model towards the computer orientation required for a practical implementation. From the functional model, every process name is identified with an operation. The exact definition of the processes is obtained from the dynamic model through the algorithms. Data structures are selected which are most appropriate to the algorithms. Classes and associations are then packaged into modules which may be stored in the Class Library. Finally, the completed Design Document is prepared which consists of the detailed Object, Dynamic and Functional Model.

The tail end of software development discusses the specific details for implementing a system using OOPL, object-base languages, and DBMS. Writing code is an extension of the design process. Once all difficult decisions have been made during the design stage, writing code would be quite mechanical. The code should be a simple translation of the design decisions into the peculiarities of a particular language.

Implementation of an object-oriented design is easiest using an object-oriented language, but even object-oriented languages vary in their degree of support for object-oriented concepts. Each language represents a compromise among conceptual power, efficiency and compatibility with previous work.

When implementing using a database system, the main concern is access to persistent data, rather than the operations on the data, a database is often the appropriate form of implementation. Database operations are much less procedural than conventional OOPL statements.

3.5 Comparison of Methodologies

The matrix provided below is for the comparison of the existing methodologies examined in this chapter against the requirements defined in Chapter two.

Almost all methodologies described in this chapter recognise the need for stage development just like conventional software development. However, the feasiblity phase

was not mentioned in any of them. This phase is still worthwhile to consider because object-oriented requirements need to be examined which might be different from the traditional ones.

Class identification and class relationships are given very thorough treatment. Not all of them agree with the type of diagrams that should be included in each stage of their analysis with reference to their class identification, class relationship and behaviour modelling. For instance, Coad & Yourdon and OMT suggest the use of data flow diagrams. This concept is absent in Booch and Rolland & Brunet. The notations used in the class diagram of OMT and Coad & Yourdon is recommended because these are more familiar to database designers and closely resemble E-R diagrams.

Some OO authors also support the use of data flow diagrams in their methodologies. Shlaer, Mellor and Wybolt suggest that data flow diagrams are constructed for each state in the state transition diagrams for each object. Hayes & Coleman[39] propose that data flow diagrams are used to show the system behaviour rather than the behaviour of individual objects. All the authors have agreed that data flow diagrams are of less importance as compared with other diagrams.

The only time where data flow diagrams could be applied in the proposed methodology might be in the overall function requirements represented at a high-level.

User-interface issues are not paricularly discussed by Rolland & Brunet because their methodology concentrates on addressing the schematic aspects of OODBMS.

The idea of a controller object is essential in real-time OO systems, particularly those that are implemented using OOPL(although such concept can equally be applied in real-time OODBMS). Without it, the system would be static. For database transaction processing under normal circumstances, there might not be a need for a controller object. This is taken over by the user instead. Hence, Rolland & Brunet do not mention such an object in their OO methodology.

Object Concurrency is given a thorough treatment by Booch. However, this is from the OOPL point of view. Rolland & Brunet do provide hints on enforcing object synchronisation on events where triggering occur at an extended relaitonal or OODBMS perspective.

OMT comes with a prototyping tool that supports their methodology called OMTool. Coad & Yourdon also have similar tools that supports each stage of OO developments. They are OOWorkbench, OOATool, OODTool and OOCodeGen.

Very little is mentioned by the authors about using an Object Repository at this stage. The same is true of the idea of a Class Library which entirely depends on the tool used to support it.

Prototyping requires a CASE tool. Although Booch does not specifically have a tool to support his methodology, the idea of OO prototyping was supported. A number

of design tools are available which facilitates some part of the OOA or OOD processes. These tools can be divided into 2 categories : those that provide high-level analysis and design tools and those that purely assist with low-level design.

Tools such as OMTool(used by GE Labs), OOATool(Coad & Yourdon), TurboCase 4.0(StructSoft Inc.), MacAnalyst(Excel Software) provide some form of support at the high-level stage.

Some methodologies, due to their features, are more suitable for using in OOPL or OODBMS as compared with the others.

Matrix for the comparison of the methodologies

	Booch	C & Y	OMT	Rolland & Brunet
Stage development	X	X	X	
Class identification	X	X	X	X
Class relationship	X	X	X	X
Behaviour Modelling	X	X	X	X
User Interface	X	X		
Controller object	X	X		
Diagramming	X	X	X	X
Concurrency Issue	X	← Little →		
Tools, CASE		X	X	
Repository(Class Library)	X	X	X	
Prototyping	X	X	X	
Reusability	X	X	X	X
Support for OOPL concepts	X	X	X	
Support for OODBMS	X		X	X

Fig 3.13

Chapter 4 : A Proposed Object-Oriented Methodology for OODBMS

This chapter introduces a proposed methodology for OODBMS, which draws on those features and characteristics described in the previous chapter but includes a number of new items.

So far, most OO authors[4,5,6,7,8,24,27,39,40,41,55] have only discussed the definition of OO terms, suggest diagrams accompanying each stage of their methodologies and provide implementation examples. None has yet explained the steps involved in each stages, especially with regards to the use of Class Library and Repository. This Chapter attempts to address these issues. However, it is not intended to formalise conventions or notations of diagrams used in each stages.

Most OO authors[7,8,40] would agree that OO development should consist at least three stages : Analysis, Design, and Implementation that undergo iterative processes. The proposed methodology must therefore consist of a number of stages. Each stage has also a number of steps. Most OO methodologies claim that the boundary between OO Analysis and Design is blurred. However, in this proposal, an attempt is made to give a clearer separation of the steps involved in Analysis and Design.

1. Feasibility Study
   (a) Overall application purpose
   (b) Statement of interactions
   (c) Performance requirements
   (d) Failure conditions
   (e) Cost/Benefit analysis

2. Object-Oriented Analysis
   (a) Generating a description of the problem domain
   (b) Constructing the Analysis Model
   (c) Object-Oriented Prototyping

3. Object-Oriented Design
   (a) Identification of supporting classes
   (b) Identification of reusable library classes
   (c) Tailoring the class structure for reusability
   (d) Choosing a concurrency control protocol
   (e) Iteration of classes
   (f) Systems Design

4. Implementation
   (a) Mapping to the target language
   (b) Implementing the application
   (c) Querying the database

A graphical representation of the developed methodology is given in Fig 4.1.

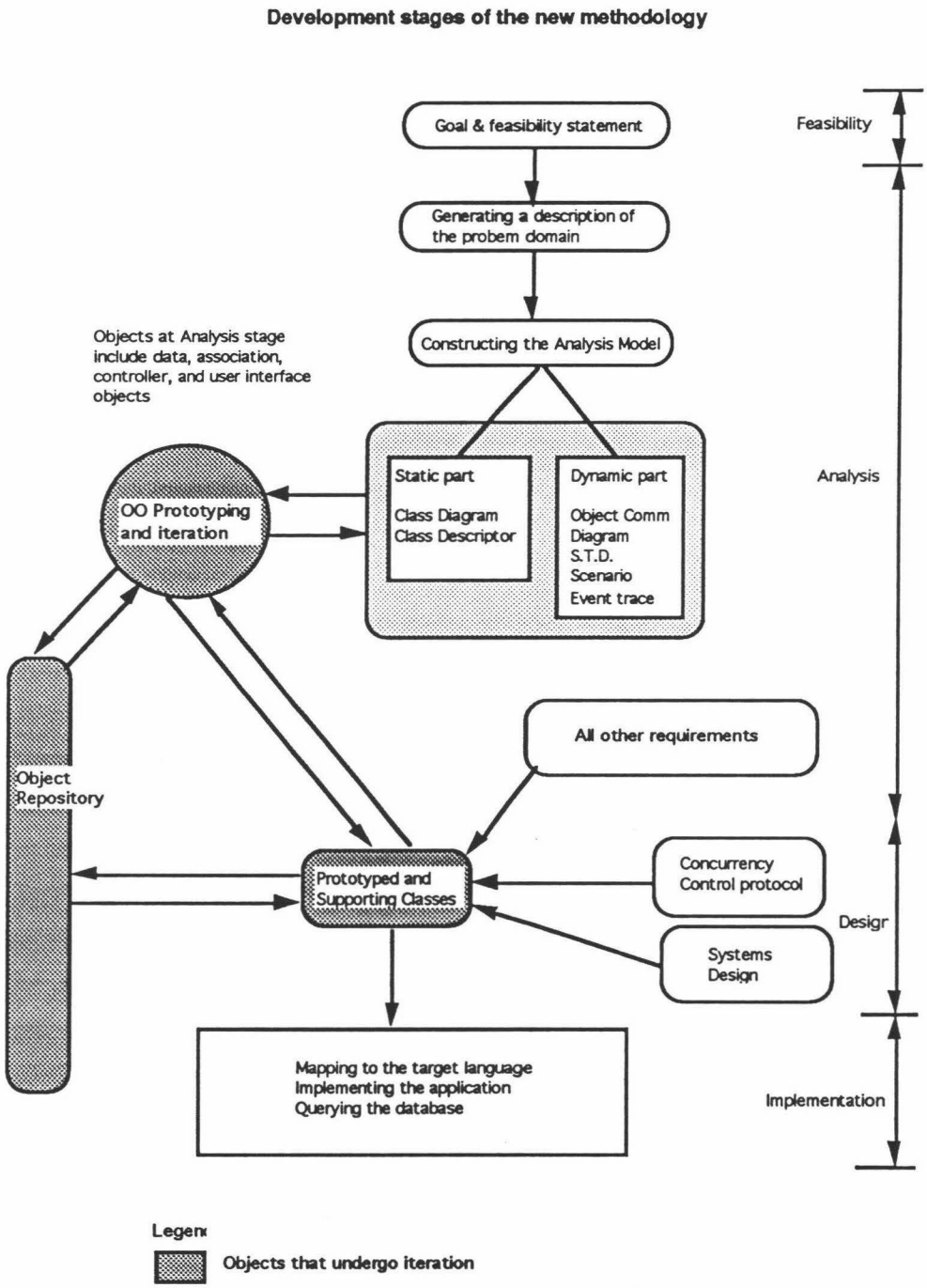


Fig 4.1

4.1 Feasibility Study

This is the first phase of the proposed methodology. It includes formulating the requirements and specifications for the system to be built. People involve in this stage are end-users, domain experts, business and system managers.

In most OO methodologies, feasibility study is not mentioned. There are several reasons why a feasibility stage is essential for developing OO systems :

(a) system complexity,
(b) prototyping used in the analysis and design stages.

For small projects, feasibility study may not be needed. However, for larger applications, a feasibility study is essential.

In the proposed methodology, prototyping is used thoroughout the Analysis and Design stage. In this regard, the original goal of constructing the application can be easily lost through the process.

Feasibility study results can lead to a conclusion that it is not necessary to follow an OO approach in analysis, design and implementation stages. It will greatly depend on the problem domain characteristics derived from the feasibility stage which approach (OO or non-OO) is more suitable, appropriate and efficient.

4.1.1 Overall application purpose

User requirements are documented and should state precisely the goals of the application, what overall functions it is to perform, the nature of its working context, and how it is to interact with other applications at a high level.

The application functions are services which are expected by the user of the system. In general, the user is uninterested in how these services are implemented so that the analyst should avoid introducing implementation concepts in describing these functions. The description for these application functions must also be consistent and complete. All services required of the application should be fully specified and no one requirement should contradict any others. The description can be expressed using natural language, tables and non-formal diagrams in the document.

Normally, the purpose of building such an application or system is to automate some or all of the tasks performed by an existing system. Present system responsibilities should be recognised because these may be taken over by the proposed new system, along with any new responsibilties that may be identified and incorporated into the new system.

4.1.2 Statement of interactions

This describes how the application would be made used of by users, other associated systems and the outside world. If the application is intended to be multi-user, then precautions must be taken to safeguard against concurrency conflicts. The authorised groups which will use the system are identified.

If the application is a development of a database, which will be used by other applications, i.e. the database has to export data, specifications of the data transfer should be laid down also. Other physical subsystems like sensors, robots, terminals, I/O devices, processors etc, that interact with the application should also be noted.

4.1.3 Performance requirements

Performance requirements set out the restrictions under which the proposed system must operate and the standards that must be met. At this stage, the performance measures will be at the application or user-perceived level, for example :

(a) Response times - average and peak,
(b) Throughput or overall transaction capacity,
(c) Availability, MTBF/MTTR,
(d) Error rates.

4.1.4 Failure conditions

From stages 4.1.1 and 4.1.2, the functions of the proposed application and its interacting environment have been identified. These functions are assumed to execute under normal situations. The failure conditions describe the exception conditons of these functions, which it is a primary task of the system to handle.

4.1.5 Cost/Benefit analysis

The methods used for calculating the tangible benefits in monetary terms would depend on the nature of the OO systems. These methods are relatively unknown at present and will not be further elaborated.

The intangible benefits for constructing such an application should also be identified which might sometimes outweigh the tangible ones. The intangible benefits are the conveniences which will arise as a result of having the proposed application that cannot be measured in monetary terms.

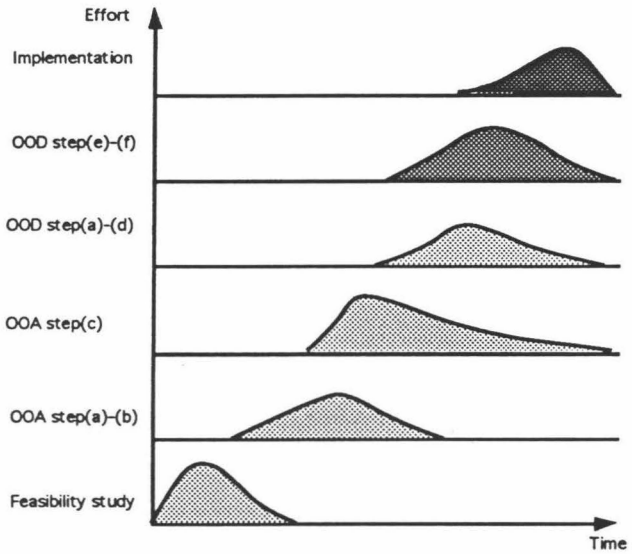
4.2 Object-Oriented Analysis

OOA is the first stage of the object-oriented paradigm. A typical feature of OOA is that much effort is channelled into this stage to get a better understanding of the problem domain, as outlined by Coad & Yourdon[7]. The proposed time and effort distribution

has been derived based on the developments applying the proposed methodology. If the amount of effort is plotted against time, then the graph might look liked Fig 4.2.

4.2.1 Generating a description of the problem domain

After deciding that a system should be built, the next step is to generate a formal description of the problem domain. This should contain details regarding the structure and behaviour of the domain. These details will typically be in the form of text descriptions or transcripts derived from a brainstorming session. Diagrams showing the flow of data and control etc would be used.



Effort as a function of time for various stages in the newly proposed methodology

Fig 4.2

4.2.2 Constructing the Analysis Model

The Analysis Model[30] represents both the static and dynamic aspects of the application. These two aspects are applicable to all the objects identified during Analysis. In order to model the static aspect of the Analysis Model, the classes and their relationships should be identified in the problem domain description and structured first. The dynamic aspect can then be modelled after the static models are completed.

(a) Identify Classes

The first step in constructing any model for Analysis is to identify relevant object classes from the problem domain. The purpose of grouping object instances into classes is to match the technical representation of a system more closely to the conceptual view of the real-world.

Class Identification involves the extraction of nouns from the problem domain. The potential object classes candidates are :

(a) "things or events remembered",
(b) "roles played" ,
(c) "user-interface",
(d) "physical subsystems",
(e) "logical subsystems".

"Things or events remembered" refer to the records that need to be stored. These are data objects that would appear in the traditional schema, and are generally passive.

"Roles played" refers to objects formed from associations between objects. An associative object may be viewed as a temporal relationship[52] between referred objects that play within this relationship a specific role. In other words, this object arises out of a binary relationship and might cease to exist at some point of time.

Fig 4.3 Association Object

An example is shown in Fig 4.3. For instance, the object Assignment is an associative object and it refers to the object Employee and the object Project. The Assignment object would only be created when the first employee is involved in a given project and ends when there are no more employees involved in projects.

"User-interface" refers to those objects that the users interact with the system. A terminal is an example of user-interface object. In this way, the user-interface object may be regarded as an active object in the system. The functionality of user-interface objects can be conceptualised at two levels : one level using dummy procedures and other level dealing with implementation details. The analysis should concentrate first on the information flow and control. Output can be simulated with dummy procedures. Details of the method functionality are left to the Design phase.

"Physical subsystems" include those entities that could interact with the system other than the users. Examples are sensors, alarms, "required" physical processors and robots. Normally, these are physical entities explicitly mentioned in the problem domain.

"Logical subsystems" are normally not explicitly mentioned in the problem domain. Examples are servers, controllers, coordinators and manager objects. In OO paradigm, real-world entities are organised into classes. However, by themselves they are rather static. For many real-time and other applications, one(or more) controller objects[4,22] are needed to coordinate the activities between all objects. This object arises purely out of the need to support behavioural modelling.

The above mentioned are categories of class identification which are directly related to the problem domain. Other classes which could be identified later for implementation include iterators and exception handlers which are not explicitly related to the problem domain.

An iterator is a temporal object[6,22] of its own, providing a method to produce the next element of a collection, a method to close the iteration when complete, and some internal state used to keep track of the current position in the collection.

The filtering, trapping and handling of errors is handled by a non-persistent object called Exception Handler[6,22].

(b) Identify Relationships

After identifying the potential classes, the next step is to identify relationships. Relationships often correspond to verbs or verb phrases. Apart from generalisation and aggregation, types of relationships may be identified under a number of broad groups. The following are suggested as guidelines :

(a) physical location or ownership(e.g. next to, part of, contained in),
(b) directed action(e.g. drives),
(c) communication(e.g. talks to),
(d) satisfying some conditions(e.g. works-for, manages).

In the proposed approach, relationships are divided into three categories, specialisation(is-a), aggregation(has-a), and association. Relationship (a) mentioned above represents either "is-a" or "has-a". Relationships (a), (b), (c) and (d) all represent associations.

It is sometimes worthwhile to represent an "instance-of" relationship between a particular object instance and its class, so that special attributes or behaviour may be modelled explicitly in the class diagram.

(c) Structure the Static Aspect

The static part consists of the **Class Diagram** and **Class Descriptors**, which forms the structure of all object classes modelled.

After initial completion of the statement of the problem, and identification of the object classes and relationships, the analyst builds a class diagram of the real-world situation showing its important properties. The class diagram is thought of as the application's schema in object-oriented database and would show :

(1) classes,
(2) inheritance,
(3) other relationships.

No special graphical notation is necessary.

An example of a Class Diagram is shown in Fig 4.4 which resembles an extended form of E-R diagram. In this example, object classes in an airport and their relationships are identified. The class Airport is an aggregation of the classes Hangar and Control Tower. The class Pilot is a specialisation of class Employee.

Although the full structure of the model would be stored in the CASE tool or Repository, it is likely that, for usability reasons, the diagram would be presented and interacted within a number of different "aspects", e.g.

(a) "is-a" hierarchy of a specific generalisation structure,
(b) "part-of" hierarchy of a specific aggregation structure,
(c) basic E-R with generalisations and aggregations,
(d) subject area diagrams[reference : James Martin IE, IEF, IEW etc].

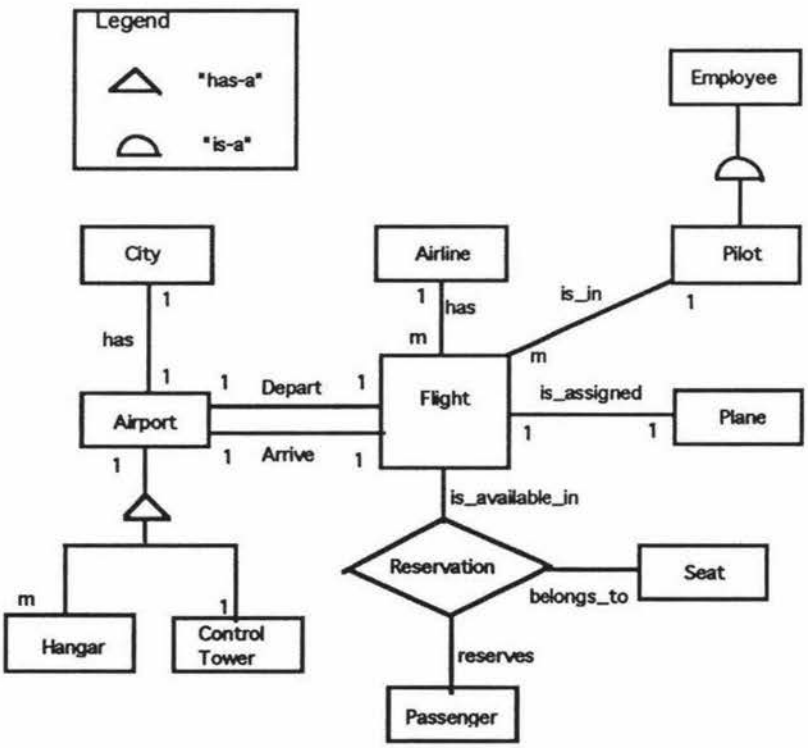


Fig 4.4 Extended E-R diagram

The Class Descriptor of Rolland & Brunet is used in the proposed methodology because it is more relevant for database application as compared with Booch's template given in Fig 3.2. Each object is given a class descriptor which is a textual description of each class. The Class Descriptor would show :

(1) class name,
(2) name of its superclass,
(3) names of properties and their data structures,
(4) names of services(operations) provided or required. A brief comment may be added,
(5) constraints applicable to local fields or triggers that affect the state of other objects,
(6) references to other objects if any.

Data objects can be implemented using container classes(collections). Container classes[24] include arrays, lists, queues, stacks, and binary trees. In the case where an object has collection attributes, the type of data structures to be chosen would be decided in the Design stage. For example, an ordered list may be required for the diagram elements in a picture to be drawn in the screen in some specific order because the ones drawn last may overlap the ones drawn first.

The class descriptor is used to supplement additional details which are not shown in the class diagram. The most outstanding rules and local constraints, if any, are defined in the class descriptor and tested in initial prototyping if possible. All secondary ones will be included during the Design phase. An example of a class descriptor for the associative object, Reservation is shown in Fig 4.5. At the Analysis, only the most essential information is provided in the Class Descriptor. This will facilitate information for prototyping.

object Reservation
superclass : nil
properties
flight no : integer
booking_date : DATE
flight_date : DATE
passenger_name : long word
seat_location : string
fare : floating number
operations
confirm % confirms a reservation made by passenger
browse % search for passenger name

constraints
booking_date is not less than five days from flight_date

references
passenger
flight
seat

Fig 4.5 Class Descriptor for the Class Reservation

(d) Structure the Dynamic Aspect

In the object-oriented paradigm, external stimuli, state change of objects, message passing and concurrency control, constitute the behavioural aspects of individual components and the system as a whole. Therefore, diagrams should be drawn to capture all these semantics where necessary.

Documentation of the dynamic part is made up of two main diagrams, the Object Communications Diagram and State Transition Diagram.

The **Object Communications Diagram** is used to capture the following semantics :

(1) the controller object(s) if any,
(2) message interaction among all objects defined in the schema,
(3) message interaction of the above objects with user-interface objects,
(4) indication of synchronous messages.

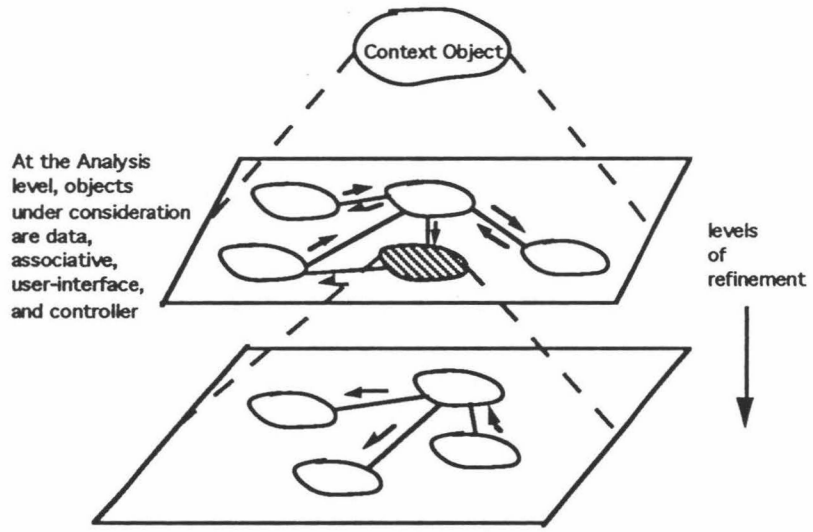


Fig 4.6 Object Communication Diagram

The controller object(s) is added as part of the necessity of behavioural modelling. Indication of synchronous messages in the diagram is optional if the OODBMS does not have an implementation construct to support it.

There are many objects in the problem domain for a large system. Therefore, it must have a top-down decomposition of objects to handle the complexity of the problem domain, as in Booch's object diagram[4](the equivalent of Object Communications Diagram in Shlaer & Mellor[55] and Wybolt[63]).

A **Context Object** as in Fig 4.6 is needed to represent the highest view of the problem domain and is analogous to the idea of a Context Diagram in Data Flow Modelling. The shaded cloud is further decomposed to review finer details of object communications at a lower level. The Context Object is decomposed into controller object(s), objects belonging to the schema, user-interface objects and other objects belonging to the problem domain. The level of refinement would be decided by the designer of the system.

Object decomposition in this manner is helpful for the Design stage because a clear picture of object interactions and concurrency can be built up early. The Object Communications Diagram would follow a top-down decomposition as in Booch starting from a Context Object. An example of the Object Communications Diagram has already been shown in Fig 3.1.

State Transition Diagrams are be used when there are many possible states that need to be shown for an object. It must show the following :

(1) start and end state of an object,

(2) state transition, represented by arrows in between object state, with the event and action labels.

However, there is no strict requirement that every class should be accompanied by a state transition diagram. This is because some classes may be inactive. If one is needed, it would be helpful in formulating the algorithm of the procedures later on in the Design stage.

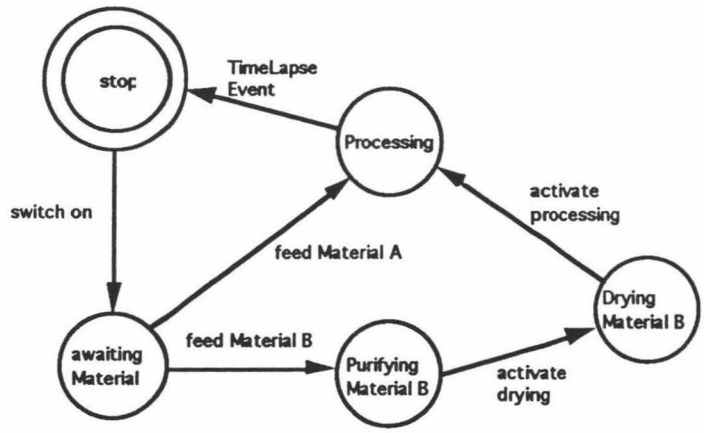


Fig 4.7 State Transition Diagram

An example of a state transition diagram is shown in Fig 4.7. Here, a chemical processor can accept two types of material, A or B. If Material A is accepted, it would process the material straight away. However, if Material B is accepted, it needs to undergo purification and drying processes before proceeding to actual processing. There is a duration or time lapse before the chemical processor returns to its dormant state from the processing state.

The Object Communications Diagram and State Transition Diagram mentioned so far represent the dynamic behaviour of objects without the dimension of time scale. The dynamic behaviour of the object-oriented system across time might be illustrated using the event trace diagram derived from a **scenario.**

A scenario is a sequence of events that occurs during one particular execution of a system[24]. The scope of a scenario varies; it may include all events in the system, or it may include only those events impinging on or generated by certain objects in the system. Exceptions are left out.

An **event trace diagram** would be used to show the interaction of objects under normal error-free conditions in sequence. This diagram is only used in real-time automation processes where a complete life cycle of the application can be pre-determined and fixed, such as an ATM. It is also used in robotics, computer integrated manufacturing(CIM) and similar areas involving routine tasks. Modelling of object behaviour across time may not be required in non real-time applications.

Consider a simple robotics manufacturing environment consisting of an operator, controller and robot as an example. A scenario may be prepared as follows :

The operator initialise the controller(software system).
The controller sends back an initialise OK message.
The operator starts the operations through the controller.
The controller sends operation instructions to the robot.
The robot executes these instructions and then sends back a complete signal to the controller.
The controller displays the results to the operator.

Fig 4.8 shows the event trace diagram for the above scenario.

Concurrency among objects, may be recognised at the Analysis stage. Some means of identifying object concurrency on diagrams at an early stage should be sufficient as given in Appendix B. Deciding what concurrency control protocol to use is a Design issue because its enforcement would directly depend on the target computing environment and their concurrency control protocol available.

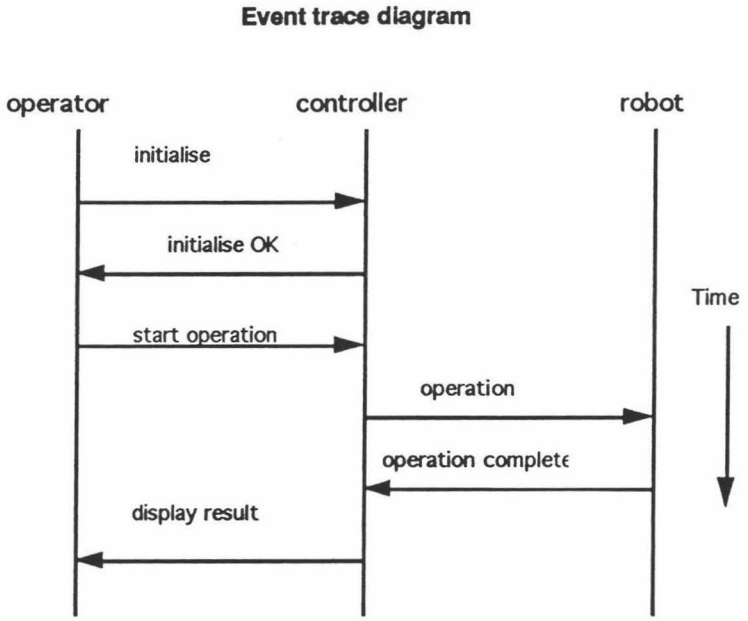


Fig 4.8

(e) Structure the Static/Dynamic Interaction

In conventional methodologies, and in some of the proposed OO ones, the use of data flow diagrams is included, presumbly as a means of showing procedure-data interactions. However, it is felt that the diagrams proposed above already capture the required semantics.

In this proposed methodology, data flow diagrams are not introduced because the intention is to move away from the conventional functional design into a more object-oriented one so as to observe the principle of data encapsulation in OO paradigm.

4.2.3 Object-Oriented Prototyping

All objects that have been used in constructing the Analysis Model should now be prototyped using an Object-Oriented prototyping tool which normally consists of a GUI toolkit and Schema Form Designer that normally comes with most OODBMS. The objects that can be prototyped using the existing OODBMS prototyping tools include the data, controller, associative and user-interface objects created during Analysis.

This step can involve looking at the exisiting objects stored in the Class Library or Repository for possible reuse in the prototyping process. The classes and their methods derived in the earlier steps have to be mapped into a specific OODBMS environment. Depending on the tool available, not all the details identified at the Analysis stage can be implemented directly. The default mapping principles for OODBMS and ERDBMS are as follows :

	OODBMS	ERDBMS
Classes	Classes or Types	Relations or Classes
Properties	Private Properties	Attributes or Columns
Operations 1. CRUD	Generated Public Member Functions	Query Language only
2. Asynchronous message passing	Generated Public Member Functions	Query Language with Rules
3. Synchronised concurrent operations	Not implemented at this stage	Not implemented at this stage
Constraints	Generated or dummy Member Functions in Data or Controller Classes	Rules or Triggers
References (relationships)	Private Properties of type "reference"	Attributes of type defined by the linked object class or relation
Versions	Not considered at this stage	Not considered at this stage

Fig 4.9 Mapping Principles for Analysis

Most Schema Form Designers of OODBMS will have entries for classes, properties, operations and references. However, most of them do not have entries for rules which otherwise, is an important feature to include in the prototyping process. All mappings that could not be implemented now will be left to the Design stage. The prototyping process should not involve any encoding so that a prototype can be developed quickly.

Binary Large Objects(BLOBS) will only be incorporated as part of the properties of the objects if they are essential to the prototyping process. This is because it is time-consuming to construct and retrieve them during prototyping.

The analyst will ignore all failure modes and assume the correct condition of the application.

User-interface objects are built using the GUI toolkit available. Their functionality need not be implemented during this stage.

Results obtained during Analysis will be placed in the Repository and/or Class Library. Objects prototyped here will undergo iterations and refinement through further prototyping using the same tool in the Design stage.

After Analysis, it would be possible to decide whether the development will proceed using the current OODBMS.

4.3 Object-Oriented Design

This section attempts to explain the Design stage. The Design stage consists of identification of supporting classes, identification of reusable library classes, tailoring the class structure for reusability, choosing a concurrency control protocol, iteration of classes and systems design. There are six aspects of the Design stage :

(a) Other objects that does not appear earlier must be added for efficiency and completeness. Examples are exception handler and iterator objects.

(b) Not all objects involve in prototyping during Analysis would be reused. Identification is needed for those that would be reused in the Design stage.

(c) Objects identified for possible reuse in future projects are then tailored as generic classes and stored in the Class Library.

(d) An object concurrency control protocol is selected for the application.

(e) Details or refinements would be further carried out to objects identified at the Analysis stage.

(f) Lastly, the application performance and its environment are considered in systems design.

A primary distinction between Analysis and Design is that Analysis is only concerned with exploring the problem domain, while Design is concerned with finding a good solution. This means that one Design decision could be to choose the target software environment, including OODBMS/ERDBMS and coding language. This environment could be different from(or the same as) that used for OOA prototyping.

4.3.1 Identification of supporting classes

The objects identified at the Analysis stage will not function fully on their own. In addition, other classes of objects associated with the application may be needed. Coad & Yourdon recognise three other types of supporting components during OO Design stage besides the schema. These are the task management, human interaction, and data management components. One or more controller objects are used to coordinate the activities of the other objects for task management component. Further objects like container and iterator objects, are used to satisfy the data management component. User-interface objects like windows and buttons are necessary for human interaction component.

In the proposed methodology, all the above three components have been considered to some extent at both the Analysis stage, rather than left entirely to Design. This is because they are also essential to the construction of a prototype. The remaining supporting objects which are be added in Design are the exception handler and iterator which are not explicitly described in the problem domain.

4.3.2 Identification of reusable library classes

Once the full set of objects at the design level is defined, a search is then made of existing class libraries and repositories. No formal procedure is proposed here. The facilities provided with such libraries or repositories will determine the pattern of tasks.

4.3.3 Tailoring the class structure for reusability

If new classes have been designed for the current application which offer possibilities of reuse elsewhere, then adjustments should be made to class hierarchies or other structural aspects at this stage, to create suitable generic library objects. Again, a formal procedure for this is not proposed here.

4.3.4 Choosing a concurrency control protocol

A conflict response specifies how the application is to behave in the presence of other agents which concurrently access some of the same objects. It is necessary to examine in the application whether objects being read could at the same time, be updated by some other agent - or vice versa. This is assumed to be completed in Analysis.

During design, it is now necessary to select a policy whereby readers and writers of an object do not conflict. Concurrency may be enforced by both the user and the system. If it is enforced by the user, then there must be methods in the requestor objects that enforce the concurrency control. Another possible way is to introduce a separate controller object that would serialise the execution of a certain group of messages.

The Reservation object in Fig 4.4 is a good example. Using the locking scheme, a group of users that perform a global task can synchronise their view of data. A notify lock held by one user can make him or her aware that other users are accessing the locked data.

At the systems level, there are default mechanisms in OODBMS that handle concurrency during normal transaction processing.

4.3.5 Iteration of Classes

Depending on whether the target environment is the same as or different from the OOA prototyping environment, the prototype is transferred or converted into a new role as the initial iteration of the working design of the system.

All objects prototyped during Analysis can now be further refined and their implementation details considered. These include the data, user-interface, controller objects brought down from Analysis; and exception handlers and iterators introduced during Design. More than one pass may be required for the results obtained in the Analysis.

The Analysis Model forms a basis for modification and improvement. However, it may show inconsistencies and some constructs may be awkward and may not seem to fit in. These inconsistencies are examined in the Design stage. It would be better to rectify these inconsistencies then to let them carry through the Implementation.

For the data objects, all other rules, field constraints and versions are now added in the class descriptor of each object. Additional properties or operations can be added if left out during Analysis. If the Schema Form Designer allows these rules and contraints to be added, they will be done so at this stage. If not, then they will have to be implemented using the provided database programming language of the OODBMS at the next stage.

The method functionality of the user-interface and controller objects is now fully defined and will be implemented at the next stage. Concurrency control methods with asynchronous message passing or synchronous concurrent operations as in Fig 4.9 are now defined for the controller objects. These will be implemented in the next phase. Fig 4.10 shows the list of items that needs iteration.

	OODBMS	ERDBMS
Classes	More Classes	More Relations or Classes
Properties	More Private Properties	More Attributes
Operations 1. CRUD	Modification only	Modification only
2. Asynchronous message passing	Additional generated Public Member Functions	Modification only
3. Synchronised concurrent operations	Generated Member Functions in Controller Classes	Left to implementation
Constraints	Additional generated methods	Secondary Rules or Triggers added
References (relationships)	To be refined	To be refined
Versions	Container Classes with generated methods	Versioning facilities if available

Fig 4.10 Mapping Principles for Design

The model is then further iterated using objects in the Class Library or Repository wherever possible, obviously within the limitations of Repository conversion if a new environment is being used.

4.3.6 Systems Design

The Systems Design stage would only be considered when building an entire system from scratch. The hardware, software configuration and the operating system in which the application resides would need to be considered. Another factor to consider is the kernel configuration of the OODBMS which holds the schema. If an application is to be built in an existing environment, these factors might be given less consideration.

Earlier in the Analysis phase, the subjects of overall function requirements, failure requirements, statement of interactions, performance requirements are briefly discussed. In the Design stage, they must now be closely examined, bearing in mind the implementation details which can be supported by the chosen target OODBMS.

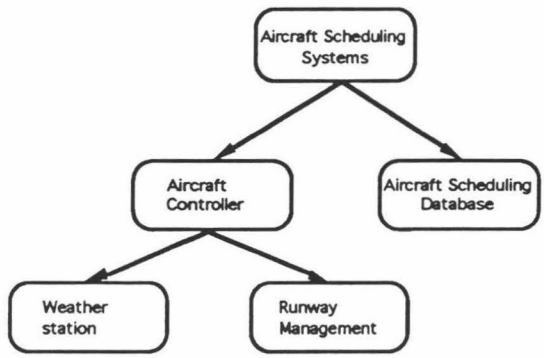


Fig 4.11 Module Diagram

Diagrams for Systems Design are not as crucial as those in the Analysis. They can still be drawn to show the interconnection of various software components that made up the OO systems. This diagram would facilitate easy comprehension of the logical design of the system. The notations can be adopted from Booch's Module Diagram.

An example is given in Fig 4.11. The modules Weather station and Runway Management provide information to the Aircraft Controller module which will ultimately determine its output. The Aircraft Controller output is then processed by the Aircraft Scheduling Systems module and stored in the Aircraft Scheduling Database module.

4.4 Implementation

The Implementation stage involves mapping to the target software environment, implementing the application and querying the database. If the Analysis and Design stages have been thorough study, then implemetation would be carried out with less difficulty.

4.4.1 Mapping to the target language

Today's Schema Form Designers in most OODBMS do not necessarily support all the default mapping principles. Some of the objects created during Analysis and Design, therefore have to be coded in a host language in order that the remaining details can be added. For example, rules are added into the codes. Implementation of functionality using the host language is also carried out for controller and user-interface objects. Fig 4.12 provides the guidelines for the implementation.

	OODBMS	ERDBMS
Classes Properties Operations 1. CRUD	Completed earlier	
2. Asynchronous message passing	Tailored Public Member Functions using host language	Code modules invoked by a main program *
3. Synchronised concurrent operations	Tailored Member Functions in Controller Classes using host language	Code modules invoked by a main program *
Constraints	Completed earlier	
References (relationships)	Completed earlier	
Versions	Tailored Controller Classes using host language	Completed earlier perhaps some code in host language

* If the ERDBMS works with a non-persistent OOPL, associate temporary OOPL classes with each relation, and invoke the ERDBMS from the methods of these classes.

Fig 4.12 Mapping Principles for Implementation

4.4.2 Implementing the application

This is merely defining a new physical and logical database before loading the schema. The exact method of configuration varies under different platform of operating systems that the OODBMS resides in. Most of the time, a database name must be created first before loading of the schema can be carried out. The appropriate operating system directories should be chosen to hold the database's area files.

The database is populated with some test values and the object SQL is performed to verify them. In the case of extended relational databases, it would depend on what the built-in query language is.

4.4.3 Querying the database

The proposed methodology suggests building query objects to handle the query processing of the application. Most extended relational DBMS and OODBMS should have this facility for implementation. Concurrency must also be verified especially for those objects that have triggers embedded in them.

4.5 Maintenance of the application

This stage is not intended to be part of the proposed methodology. From time to time, enhancements in terms of the functionality, user interface, systems performance, tuning etc, will be required of the application. Users' inputs and requests for further improvement are considered.

The nature of the OO paradigm enables the application to be built resilient to changes. This is because extra object classes may be added quite easily to the existing program which has a low coupling.

4.6 Summary

The basic steps of the proposed methodology have been described. This methodology serves as a guideline to the software development of OODBMS applications. In the following two chapters, the proposed methodology is applied to two Case Study examples, targeted for Postgres and Ontos respectively.

Chapter 5 : Application of the proposed methodology to Postgres Case Study

This chapter takes a look at applying the proposed methodology to the Case Study example using Postgres, the extended relational database[6,23,56,57,58]. The features of Postgres are described in the first section. The example given in this chapter is based on a Case Study on a floor layout design. Complete description of the example is given in Appendix C. Application of the proposed methodolgy to this Case Study is then discussed.

5.1 Features of Postgres

Postgres incorporates data, object and knowledge management in its modelling constructs. Data management covers traditional transaction management and query facilities. In Postgres, transaction are carried out by using Postquel or C commands, whereas queries are carried out using Postquel. The main postquel commands used for transaction are **create** for a relation; **retrieve, append, replace** and **delete** for a tuple.

Object management entails efficiently storing and manipulating non-traditional data types such as point, box, and polygon. These are built-in data types, but data types may also be user-defined. Object management features arise out of the need to support application areas such as CAD/CAM.

Knowledge management entails the ability to store and enforce a collection of rules that are part of the semantics of an application. This feature enables the definition of integrity constraints about the application, as well as allowing the derivation of data that are not directly stored in the database. Definition of rules is expressed in Postquel.

Behavioural aspects can be implemented using C, CLOS or Postquel. Postquel can be used for standalone queries, or for defining functions, or can be embedded in host language programs. Functions can also be implemented using C language and queries can be embedded in a main.c program.

Further details about Postgres can be found in the manual[23].

5.2 Feasibility Study

Before this application is constructed, a preliminary feasibility study was carried out to determine the objective, requirements and cost/benefit before Analysis. This stage consisted of the following :

5.2.1 Overall application purpose

The purpose of constructing the application is to assist the designer of an office layout to evaluate alternatives. It is also advantageous to have a graphical display of the floor layout, provide space management, and the maintenance of records of employees within the organisation. For this Case Study example, the purpose is :

(a) to evaluate versions of the floor plan design according to some measures,
(b) to check proposed variations for acceptability against certain rules,
(c) to provide a user interface to enable the designed layout to be varied and feedback about it to be presented.

5.2.2 Statement of interaction

Basically, a standalone application is sufficient for this Case Study. There is no need for multiple users at different locations or geographical sites.

The user will have to interact with the application through a screen, keyboard and a mouse. The screen should display of a number of windows. These windows are for the demonstration of the floor plan, performing queries and other commands etc.

The authorised users of the applications are mentioned here. There might also be other applications sharing data from this application. For instance, one department might "data-grab" and incorporate the acquired data in their spreadsheet.

5.2.3 Performance requirements

Response time on failure due to a broken rule is required at less than two seconds. The interactive response time for the notification of other unacceptability for a design floor change or for the evaluation of a design should not exceed ten seconds. The availability of the application should be as high as possible, preferably close to 99% based on past experiences. In terms of robustness, the application should be able to recover quickly when there is a systems failure and the stored data must not be corrupted.

5.2.4 Failure conditions

This section is not required for this Case Study, as there is no critical real-time operation.

5.2.5 Cost/Benefit analysis

The tangible benefits might be the automatic maintenance of consistent records, time savings on planning the floor space for each department(converted to cost) and portability of the developed software to other subsidiaries or marketed as a product to other organisations.

5.3 Object-Oriented Analysis

Object-Oriented Analysis consists of generating a description of the problem domain, constructing the Analysis Model, and OO Prototyping. Although Postgres is only an extended relational DBMS, all the steps in the OOA of the proposed methodology can still be followed.

5.3.1 Generating a description of the problem domain

The Case Study is one that involves planning of a rectangular floor layout planning. The floor has two long sides where the windows are located. It also has double door entrance with a central aisle running through them. It is partitioned into open and enclosed work spaces of various sizes for the tenants depending on their status. Spaces are also allocated for computer and communication equipment, kitchen, reception and interview rooms. The database content is required to contain a chain of versions of the planned layout and the company's organisation structure.

However, certain conditions are imposed for the planning of floor space. For instance, the directors' offices must be adjacent to windows, no work spaces are to overlap or to extend beyond the limits of the floor area.

The operations of the system will involve making changes to the layout while checking feasibility, and calculation of average distances from one office to another and the average distance from the fax and printer to each office.

For more details, refer to Appendix C for the Case Study description.

5.3.2 Constructing the Analysis Model

Firstly, all visible object classes of interests in the problem description were identified. These objects would later form the relations of Postgres. In the exercise, there were at least three subclasses of **working_area.** The floor plan includes **open_area** for ordinary staff up to section head level, **closed_area** for directors, and **group_area** for public. No user-interface and controller objects were identified in this Case Study.

The organisation structure is relatively simple. The relation **staff** should have an identifier staff_id, employee name, age, salary and manager. All people other than directors would have a manager. It was assumed that directors do not report to anyone.

After identifying all objects, the attributes were identified and extracted :

working_area(location)
open_area(staff_id, dept, location)
group_area(purpose, location)
closed_area(name, staff_id, dept, location)
staff(staff_id, name, age, salary, manager)

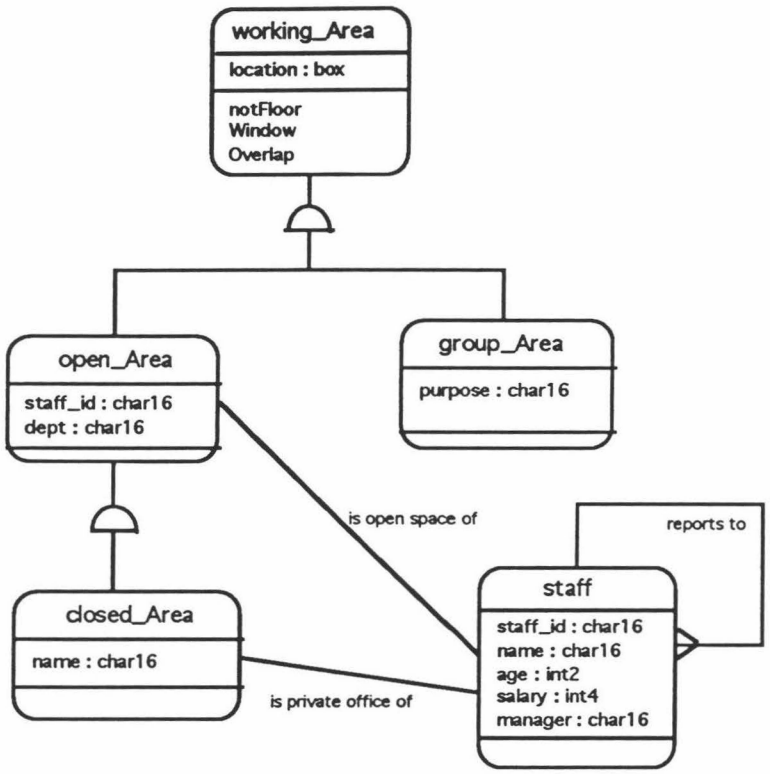


Fig 5.1 Class Diagram for Postgres Case Study

Note that location is common to all classes and is unique to **working_area.** This provides a hint that **working_area** would be the base class. **Closed_area** has some attributes in common with **open_area** apart from having its unique attribute, name. This implies that **closed_area** is a subclass of **open_area. Group_area** has only location in common with the base class and nothing in common with both open and closed_area.

Some inheritable functions that were performed on the **working_area** and its subclasses would check that the values of the instances were within the constraints. The function called **Window** would check for offices adjacent to the long side of the wall. The function **notFloor** would check for offices that were outside the floor area. **Overlap** would check for overlapping offices. Other functions were used for the calculation of average distances as required in the Case Study.

After determining the object classes, attributes and functions, relationships between classes were now identified. The classes were arranged in a supertype-subtype relationship if any.

Note that in this exercise, the Coad & Yourdon notation[7] for OOA had been used because there are few classes in the problem domain. If there are many classes, it might be advisable that attributes and functions be hidden. The class diagram for the Case Study is shown in Fig 5.1.

Details of attributes(and the type of data structures), methods, and constraints were represented using a class descriptor. An example of the class descriptor in Rolland & Brunet's O* Model, for the class open_area is shown in Fig 5.2 below :

object : open_area
superclass : working_area
references : nil
attributes :
staff_id : char16
dept : char16
functions : nil
constraints :
locations must not overlap group area or closed_area location

Fig 5.2 Class descriptor for Postgres Case Study

The dynamic aspect of the system is modelled using the state transition diagram, object communications diagram, scenario and event trace diagram at a high level of abstraction. However, since this is not a real-time application, the dynamic model is probably not important. This is because only the user is an agent and the rest of the entities are static data objects.

No preliminary object concurrency was identified for this Case Study.

5.3.3 Object-Oriented Prototyping

At this stage, all visible objects identified would be implemented first. In Postgres, functions and rules are not encapsulated within the schema object(relation) and are defined externally. This is also true for other types of extended RDBMS. The most essential rules were used for prototyping. The less important ones would be considered in the Design stage. However, since this is only a small exercise, the rules and functions were totally prototyped during Analysis.

Strictly speaking, prototyping at both the Analysis and Design stage should not involve coding. Since Postgres does not come with a Schema Form Designer, prototyping was achieved by using Postquel which mapped the contents of the class descriptor. Relations for **working_area, open_area, closed_area, group_area** and **staff** were now created together with their attributes. Rules and constraints were implemented using either Postquel or external C functions. For example, **notFloor, Window** and **Overlap** were rules implemented using C functions instead of Postquel.

From the recommended mapping list in Fig 4.9, versions and synchronised message passing were omitted during prototyping. References relationship was not used in this Case Study but should be looked at if there was a requirement.

There is also an external GUI toolkit for Postgres called PICASSO. Unfortunately, it was not available to be used in this exercise. Otherwise, user-interfaces could be built rapidly.

A small number of instances were populated to test if the rules and functions worked. Instructions on implementation details are fully explained in the Postgres Manual[23].

5.4 Object-Oriented Design

The Design stage includes identification of supporting classes, identification of reusable library classes, tailoring the class structure for reusability, choosing a concurrency control protocol, iteration of classes and systems design.

5.4.1 Identification of supporting classes

Supporting classes such as iterators and exception handlers were not implemented in this Case Study. Postgres has yet to support exception handlers.

5.4.2 Identification of reusable library classes

Built-in abstract data types in Postgres class library such as box and point are examples of types/classes defined previously and used in this Case Study.

5.4.3 Tailoring the class structure for reusability

There is a possibility that the classes **staff** and **working_area** could be reused by similar projects in the future. They could have been tailored into generic classes but it was not attempted here.

5.4.4 Choosing a concurrency control protocol

Concurrency was not explored in this exercise since the application is not real-time.

5.4.5 Iteration of classes

Additional attributes or functions of the above classes might be added if necessary. Since there was no user interface and controller objects, only the data objects would need iteration. However, since most of the implementation was carried out during prototyping, there is little that can be said about this step.

Versioning of working areas such as **open_area** could be implemented at this stage.

5.4.6 Systems Design

Since this exercise is a simple one, no special attention is needed for the systems design.

5.5 Implementation

Implementation includes mapping to the target software environment, implementing the application, querying the database, improving the application, and maintenance of the application.

5.5.1 Mapping to the target language

This step had already been performed during prototyping during Analysis.

5.5.2 Implementing the application

The database should be properly configured to sit in the appropriate Unix directory.

Once the correct results for rules and functions had been verified in Analysis, all the instances were now populated. The original prototype database would need to be destroyed and a new one created. This is because Postgres does not check if an instance has already existed in the database.

5.5.3 Querying the application

The relations created in the Case Study were populated with all the instances needed in the application. Queries on the application were then performed again to ensure that the results obtained were consistent with the ones obtained in the earlier stages.

5.6 Summary

The proposed methodology had been applied to a Postgres Case Study example as outlined above.

It is noticeable that much of this Case Study could be implemented during OOA. The OOA stage of the proposed methodology involves a great deal of capturing the most essential semantics for the schema of the Case Study. A more complex example would be required to fully test the methodology where host language coding(other than for implementation of functions) was required.

Chapter 6 : Application of the proposed methodology to Ontos Case Study

This chapter examines the application of the proposed methodology in Chapter Four to a Case Study targeted to use Ontos, the object-oriented DBMS.

6.1 Features of Ontos

Ontos is a multi-user, distributed object DBMS with a C++ class library interface. Its purpose is to provide a reliable persistent storage facilities for C++ objects. It also allows C++ programs to retrieve these persistent objects(objects created by other programs).

Ontos also provides other facilities like controlling concurrent access by multiple users, providing ad-hoc query capabilities and allowing independent control over the physical location of data by other Ontos database applications.

Ontos includes a class library of useful data and control abstractions, which supports different kinds of aggregates, iterators, exception handling, and user interface construction. These are important supporting features for an OODBMS.

In addition, Ontos comes with utilities that help in the production of a database application. One of these is DB Designer, which is a visual schema browser and designer. Others include the Ontos Studio and Shorthand which may be regarded as 4GLs. Further utilities include DBATool which helps in creating, configuring, performance tuning, and database administration; the Classify schema compiler and the Cplus preprocessor tool.

An application can be written using C++. Encapsulation is better enforced here as compared to Postgres. However, definition of rules is not explicit and is usually embedded in the body code of the classes.

Unfortunately, the Ontos software was not available for use in this Case Study and hence some conclusions depend on assumptions based on the manuals.

6.2 Feasibility Study

This stage consists of the following steps :

6.2.1 Overall application purpose

The main goal for constructing this application is to provide early warning for low water levels that might subsequently interrrupt hydroelectric power supplies. It is also

advantageous to have features in the application that provide the display and querying of geographical data connected to the problem.

The application must be able to provide early warning to the users when the water levels in the reservoirs reach critically low levels. The management should also be able to simulate the critical condition by extrapolating graphs showing lake levels before these fall to a given minimum.

6.2.2 Statement of interactions

The application can either be used as a simulation or as a real-time database used by many users in different locations. The users are to interact with the system through terminals in a WIMP environment. This interactive environment should include graphical displays and maps.

The sensors located at the lakes and rivers can be polled periodically to obtain data which is stored in the database.

6.2.3 Performance requirements

The system could be unduly slow due to the handling of graphics. Therefore a guideline should be made regarding the response time of the application software, data acquisition time, sensitivity specifications of sensors, compilation and run time. For instance, the data sampling time of the sensors should be less than 1 millisecond.

Some of the performance requirements from the users' point of view are :

(a) Graphical display and interactions should give response time within 10 seconds,
(b) Other response times must not exceed a certain duration say, 3 seconds,
(c) Modifications or enhancement of the application must be provided for within 1 hour of user request.

6.2.4 Failure conditions

The failure conditions are :

(a) system power failure,
(b) system crash,
(c) for each condition there should be user indication and automatic recovery where possible,
(d) malfunction of the associated hardware, like sensors and other equipment,
(e) unsuccessful execution of system functions.

6.2.5 Cost/Benefit analysis

Some of the tangible benefits in implementing such a system would be :

(a) saving of time in maintenance of facilities,

(b) saving of time in administration.

The intangible benefits are :

(a) more information, hence more effective resource management,

(b) better analysis with less field collection of data needed,

(c) ability to perform simulations, hence giving advance warning of problems,

(d) higher standards and accuracy of information,

(e) quicker access to information,

(f) better understanding and analysis of a complex system, leading to better decisons and planning.

6.3 Object-Oriented Analysis

The analysis for the exercise includes generating a description of the problem domain, constructing the Analysis Model, and OO prototyping.

6.3.1 Generating a description of the problem domain

The application domain consists of lakes, reservoirs, rivers, streams, measuring points, generating stations, and weather stations. Attributes of lakes include the lake's name, the normal, actual and minimum water levels and the surface area, together with a polygonal boundary. Procedures are used to calculate the lake level rise per mm of rainfall on the catchment. Rivers and streams are to be plotted as a series of time segments, allowing flows to be aggregated.

Sensors located at designated measuring points along rivers and lakes are used to detect flow rate and lake levels respectively.

The generating stations are required to provide methods for calculating flow of water required per one percent of generating capacity and flow of water actually reaching the station.

Weather stations provide forecasting services such as rainfall, snowfall, and temperature.

The application should be able to display maps and plot graphs in several forms.

A complete description of the Case Study is given in Appendix D.

6.3.2 Constructing the Analysis Model

The following diagrams were constructed for the Analysis Model of the Case Study:

(a) class diagram,
(b) class descriptor,
(c) state transition diagram,
(d) object communication diagram,

Types, properties, inheritance relationships, member functions and Ontos free functions were identified in the class diagram. The candidate objects and their attributes in this problem domain are :

The types of classes and their attributes are :
Lakes [Lakename, normal_level, minimum_level, actual_level, surface_area]
Rivers [name, length]
Catchment [name, location]
Measuring_point [id, flow_rates]
Gen_Stations [name, capacity]
Weather_Stations [name, rainfall, snowfall, temp, windspeed]
City [name, location, population, daily power demand]
Clock[min, day, month, year]
Hydrographic_Object
User_Interface_Object

The Hydrographic_Object is a controller object. It receives messages from remote sensors at measuring points. These messages will be passed on to other objects which will cause the desired methods to execute at threshold values. The User_Interface_Object is made up of other smaller user interface objects that the user can interact with the system.

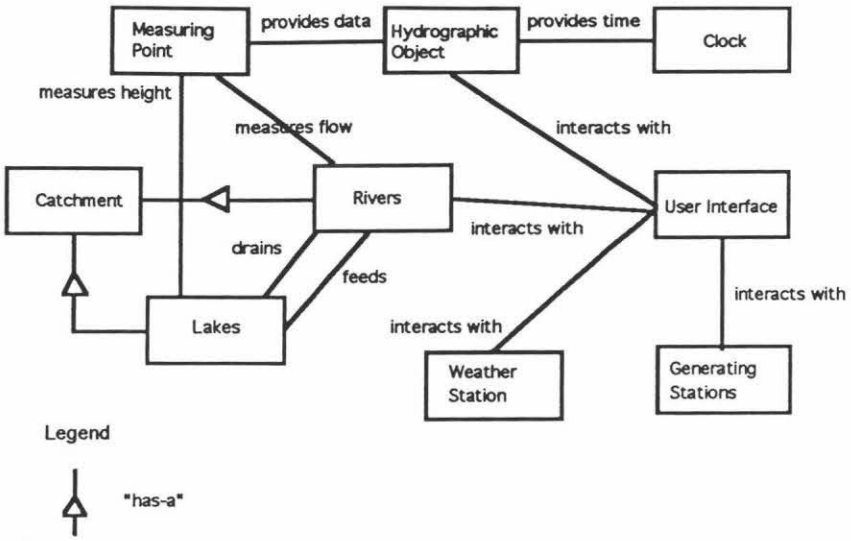


Fig 6.1 Class diagram for Ontos Case Study

A detail description for each class and their methods are described in Appendix D. An example of the class descriptor for Lakes is shown in Fig 6.2.

object Lakes

properties

Lakename : char
priv_normal_level : integer
priv_actual_level : integer
priv_min_level : integer
priv_surface_area : integer

operations

%catchment

constraints

priv_normal_level > priv_min_level

references

Rivers

events

low water level : predicate
priv_normal_level >= priv_min_level

triggers

extrapolate_graph on user interface object

Fig 6.2 Class descriptor for Lakes

For the dynamic aspect, the State Transition Diagram and Object Communications Diagram were shown. An example of a State Transition Diagram, for the class Measuring_Point, is shown below.

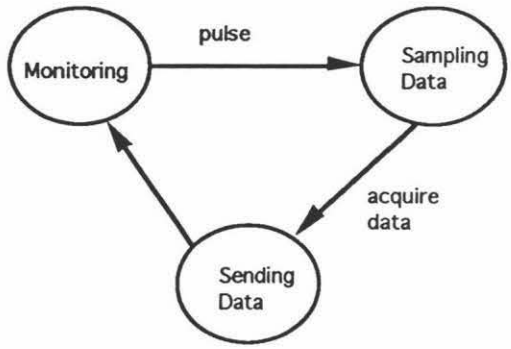


Fig 6.3 State Transition Diagram for class Measuring_point.

An example of an Object Communications Diagram, centred on User_Interface_Object is shown above in Fig 6.4.

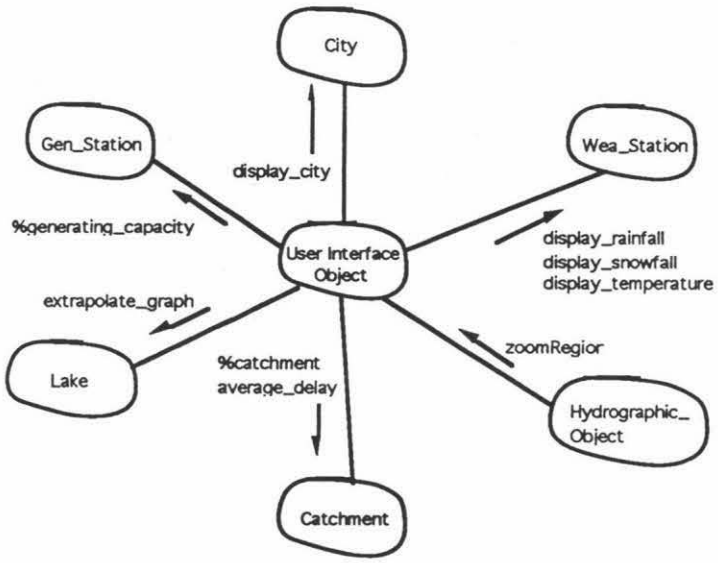


Fig 6.4 Object Communications Diagram for Ontos Case Study

6.3.3 Object-Oriented Prototyping

If Ontos Studio and Shorthand had been available, prototyping of the results obtained during Analysis could have been carried out easily. The data objects include Lakes, Rivers, Catchment areas, Measuring points, Generating stations, Weather stations, City and Clock. The class descriptors of all the above objects constructed earlier would be helpful. The names of the classes, values of attributes and names of functions

could be entered in the fields of the DB Designer. Take for instance, the operation %catchment of the class Lake. A dummy procedure for this operation might be implemented through Studio so that it could be invoked during Analysis.

Studio could also have provided for the construction of a "mocked-up" interface. Otherwise, implementation of method functionality for User_Interface_Object would be deferred. The group of messages interacting between the User_Interface_Object and the other objects which need synchronisation were identified at this stage, and would be included in the prototype.

Since rules are not explicitly defined in Ontos, and can be implemented only in embedded functions using C++, they were not considered until the Design stage.

6.4 Object-Oriented Design

This stage includes :

6.4.1 Identification of supporting classes

Additional classes like exception handler and query iterator were now introduced.

Failing conditions of the data, controller and user interface objects were identified. This would be handled by the class HydroException as in Appendix D, derived from the class called Failure in the Ontos class library, defined by the user. It was not necessary to consider its error handling functionality at this stage.

The query iterator object, HydroIterator, as in Appendix D, was required to retrieve instances stored in data classes. Again, the functionality was not considered.

Both the HydroException and HydroIterator objects were incorporated into the model for further prototyping.

6.4.2 Identification of reusable library classes

This was not attempted in the Case Study, but would have looked for previously created objects suitable for modification. This might include graphical and map elements, and simulation controller objects.

6.4.3 Tailoring the class structure for reusability

This step was not attempted. However, the class Gen_Stations might be tailored into a generic object and placed in the Ontos Class Library. They might be reused by future projects such as the construction of another application to simulate power generation.

6.4.4 Choosing a concurrency control protocol

The synchronisation issues was addressed at this step. For instance, the type of concurrency control protocol needed to enforce synchronisation between message passing between User_Interface_Objects and the others, was chosen from the different locking protocol available in Ontos.

NotifyNoLock was selected so that the DB Server could be directed to track changes to all objects concerned revolving the User_Interface_Objects.

6.4.5 Iteration of classes

At this stage, most of the classes and properties of the Case Study were completed, given in the mapping guidelines in Fig 4.10. The User_Interface_Objects were decided to be made up of buttons, windows and icons in the Ontos Class Library.

The behaviour of the two supporting classes, HydroIterator and HydroException could now be determined.

In the class Lakes, there was a constraint that if the number of days before the lake level reached their minimum level drop to less than a month, it would trigger off some alarm. A dummy procedure was used to test out this condition.

6.4.6 System Design

This step was not examined in this application. Packaging into modules was not required.

6.5 Implementation

This example was not implemented because the Ontos software is not available. Therefore, it is only possible to describe the steps of implementation on paper.

The schema, control and main.c files were separated so that they could be compiled using the DBATool.

6.5.1 Mapping to the target language

The reusable objects in the Design were written into C++ source code. The class descriptors refined during Design were now helpful in adding final details to the source code for implementation of the application. However, only a small number of instances were populated for querying and retrieval purposes.

The control files would contain the implementation code of methods as in Appendix D. Rules were coded in the public section. For instance, in the class Lakes, if the number of days before lakes reach their minimum level drops to less than 1 month, it would trigger the method extrapolate_graph.

Methods enforcing concurrency would now be implemented using C++ for User_Interface_Object and Hydrographic_Object.

6.5.2 Implementing the application

Details of C++ code which could be used for the implementation of the Case Study is given in Appendix D.

6.5.3 Querying the database

A simple query using OSQL was set up in the main.c program to retrieve all the instances belonging to the class Lakes which have a surface area of less than 50 square metres. Details are given in Appendix D.

6.6 Summary

The proposed methodology is better suited to mapping problem domains into Ontos than Postgres.

Firstly, many failing conditions can be identified during Feasibility stage since this is a real-time application. More diagrams in the Analysis stage of the proposed methodology can also be drawn for this Case Study. OO prototyping is facilated by using DB Designer and Studio which enables rapid development without writting code.

Objects created can also be stored in Ontos Class Library for possible reuse in the future. Concurrency is also better supported by Ontos in this Case Study.

Chapter 7 : Conclusions

This concluding chapter provides comments on the results of the newly proposed methodology as applied to Postgres and Ontos Case Study.

7.1 Author's comment on the proposed methodology

The literature search has covered a very broad base. So far, there have not been many journal articles written on the Analysis and Design of OODBMS[33,42,47]. The proposed methodology has laid a general framework for further research towards a more unified and coherent OO methodology for object-oriented databases. It is not intended for this thesis to provide detailed formats or for each stage/step. Rather, it aims to incorporate all the relevant OO techniques and extracts the essential ingredients that might make up a general OO methodology for the Analysis and Design.

Mapping has been tried on Postgres and Ontos to illustrate the new paradigm's applicability. Object-Oriented Prototyping is also introduced as part of the main paradigm to emphase on the reuse of objects throughout the whole development cycle and a reduction of development time.

Of course, the new methodology may not be the only way of designing an OO database application. However, it could provide object-oriented database designers with a sound basic approach, as this is demonstrated in the two case study examples.

It should be noted that it is quite impossible to come out with one methodology that totally supports all the features of both extended relational and object-oriented DBMS.

Summarising, the application of the proposed methodology for the development of OODBMS has proved that :

(a) the use of Class Library and Repository introduced in Analysis and Design stages make the development more efficient and less time-consuming,

(b) OO prototyping introduced in Analysis stage helps to decide on the applicability of the target DBMS,

(c) specific OODBMS features and DBMS techniques have been observed, considered and applied in conjunction with OO paradigm,

(d) the proposed methodology can be used as a general framework for application development and utilisation of the existing ERDBMS and OODBMS.

Extensions could still be carried out on this new methodology, particularly in the Analysis and Design phase. Examples are :

(a) further proposals at the feasibility and analysis stages to support business modelling and objectives.
(b) additonal object-oriented conventions to enforce semantic integrity,
(c) extending this paradigm to support parallelism,
(d) advanced object concurrency modelling,
(e) sub-methodology for OO prototyping,
(f) search techniques for Repository and Class Library.

Subjects of current research[57] carried out internationally on object-oriented databases are discussed in Appendix E.

7.2 Comparison of its application to ERDBMS and OODBMS

As shown in the previous two Case Studies, the proposed methodology is more applicable to true OODBMS. The Analysis and Design stages support two very important concepts : prototyping and reusability. Unfortunately, most extended RDBMS have yet to realise these features at present. It may be a possibility that in the future, extended RDBMS will be provided with class libraries and a Schema Form Designer so that the proposed methodology could fully support it.

On the other hand, OODBMS are weak in the prototyping of rules at the Analysis stage. In most OODBMS, they are usually coded as embedded functions rather explicitly defined in the Schema Form Designer. Rules remain as an important feature and it is desirable that Schema Form Designers will incorporate it in the future.

Appendix A : OO Prototyping Tools

A.1 Smalltalk

Some authors have suggested using Smalltalk for prototyping[8,20]. However, most users would need to overcome a steep learning curve before able to prototype efficiently. Smalltalk is an OOPL which is tightly integrated with its environment which is a WIMP interface[27]. Smalltalk is a good prototyping language for four reasons :

(a) The object-oriented nature of the language means that systems are resilient to change.

(b) The Smalltalk system and environment is an inherent part of the language and all the objects defined there are available to the Smalltalk programmer. Thus a large number of reusable components are available which may be incorporated in the prototype under development.

(c) Objects created under the Smalltalk environment may be exported to the Gemstone database[49].

(d) The Smalltalk browser allows a user to browse through the class inheritance hierarchy, display instance variables and methods, and determine which classes send or receive a given message.

A.2 Ingres windowing version

In the case of OODBMS environment, particularly those in commercial data processing, a prototyping approach may supplant the conventional development model as so-called 4GL, are used for system development. However, object-oriented 4GL is a very new tool as compared to a great variety of 4GL for relational databases already in existence.

Ingres is a popular relational DBMS. It comes with a 4GL with prototyping capabilities. The latest version has windowing facilities. The current version of the DBMS(6.4) includes methods and triggers, which makes the existing INGRES DBMS start to resemble Postgres. However, this does not mean that it would support object-oriented features like encapsulation, polymorphism and complex objects.

Although prototyping clearly reduce systems development costs, the effect of 4GL on overall life-cycle for large OODBMS is not yet clear. They are obviously to be recommended for prototyping but the lack of standardisation may result in long term maintenance problems.

A.3 GemStone

Recently, Servio Corp, the GemStone OODBMS vendor based in Alameda, California, has come up with GeODE[50], a 4GL for rapid object-oriented prototype applications development. The idea is to let end-users build full object-oriented applications-complete with GUI and multimedia without the need

to master an OOPL. It is made up of four components : the Form Designer, the Visual Program Designer, the Application Designer, and Systems Programming Tools.

GeODE is originally developed for the use with GemStone. However, by using a gateway, it could access Sybase Inc.'s SQL Server RDBMS. Soon, this facility would be extended to DEC's Rdb, Informix Software's RDMS, Ingres and Oracle. It is now running on SunOS or OpenLook and will also be extending to other platforms like MacIntosh, Microsoft Windows and OS/2 Presentation Manager by the middle of this year. GeODE runs on a client/server distributed architecture. Many organisations are happy with their existing relational database environment which is sufficient to do most of their work. GeODE is designed to co-exist with relational database systems so that it could benefit organisations which would like to have object-oriented development without replacing the old investments. Some of the organisations experimenting with GeODE are the National Oceanic and Atmospheric Administration(NOAA), and Texas Instruments Inc. in Dallas. GeODE could find a niche in the 4GL business - especially in CAD and places where there is a need to deal with complex objects.

A.4 O2

Another OODBMS which has an integrated application development is O2[48]. The environment includes a query language(O2query), a user interface generator(O2 look), an object 4GL(O2C), a graphic programming environment including a debugger and a schema and database browser. Moreover, classes in C++ can be created and imported in the database.

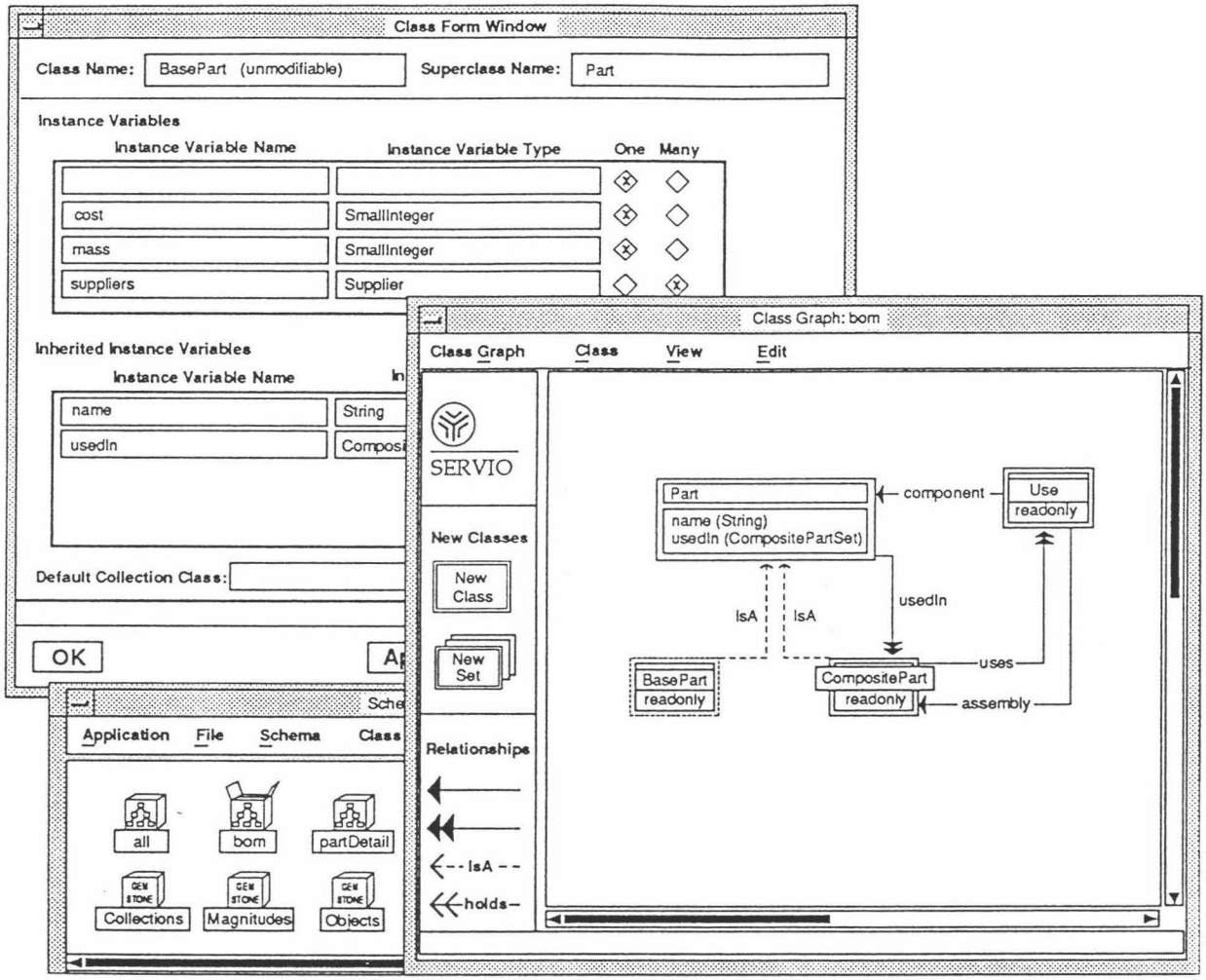
A.5 Ontos

Ontos DBDesigner, Studio and Shorthand[22] provide the users with rapid prototype development due to their GUI in X windows where the application with its interface may be quickly built.

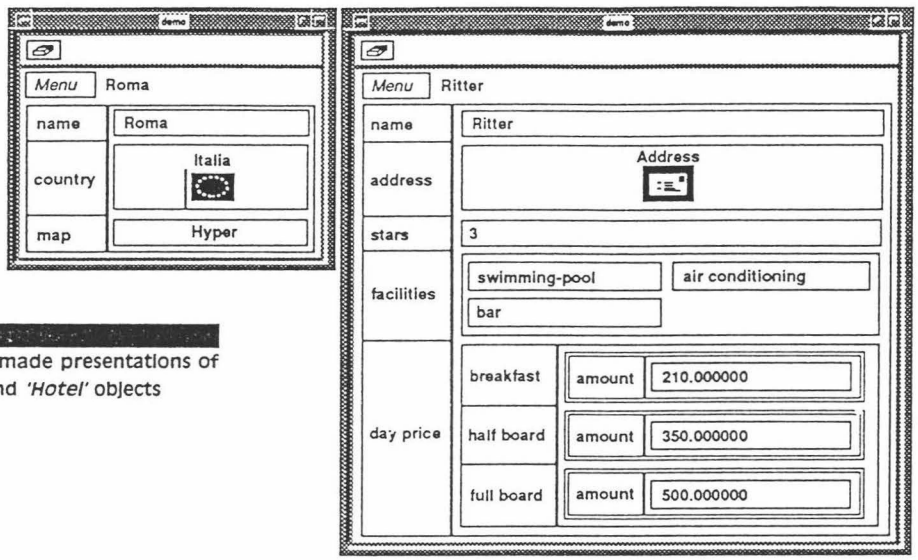
A.6 GOOSE

Another recent Schema Management and Prototyping Interface tool for an OODBMS has been developed at Georgia Institute of Technology. It is called GOOSE (Graphical Interface for an Object-Oriented database Schema Environment) and can support schema evolution and schema versioning.

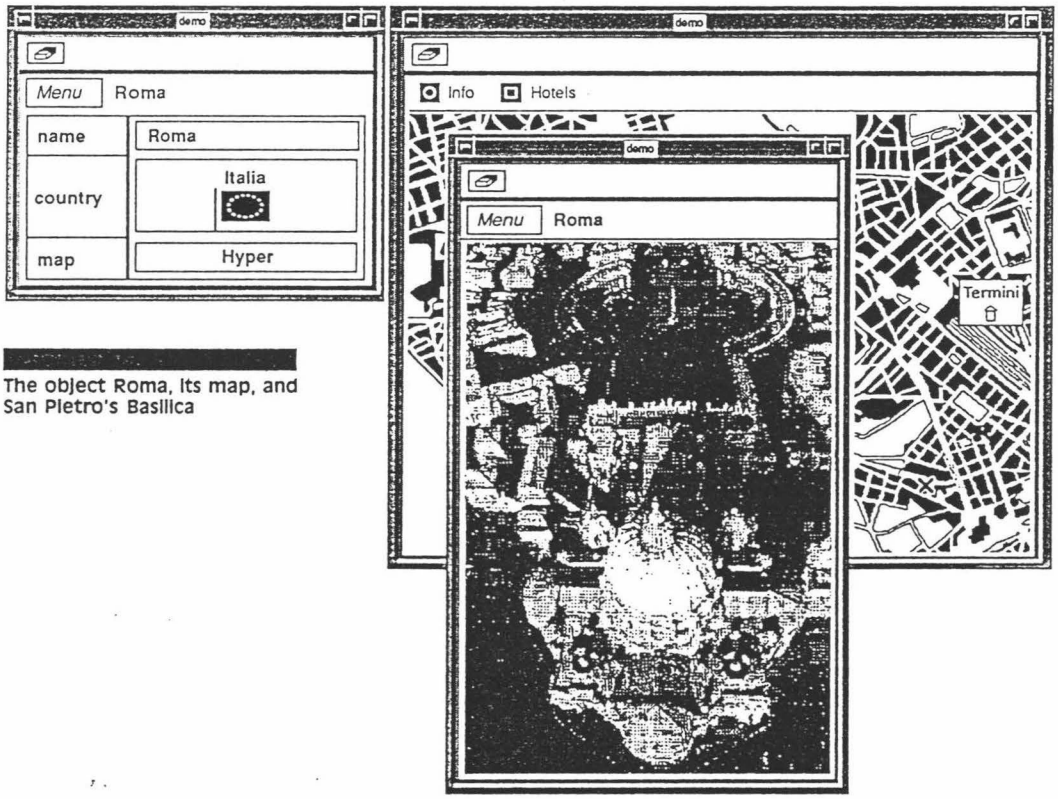
Examples of windows in GS Designer in GemStone



Objects created in O2 schema designer

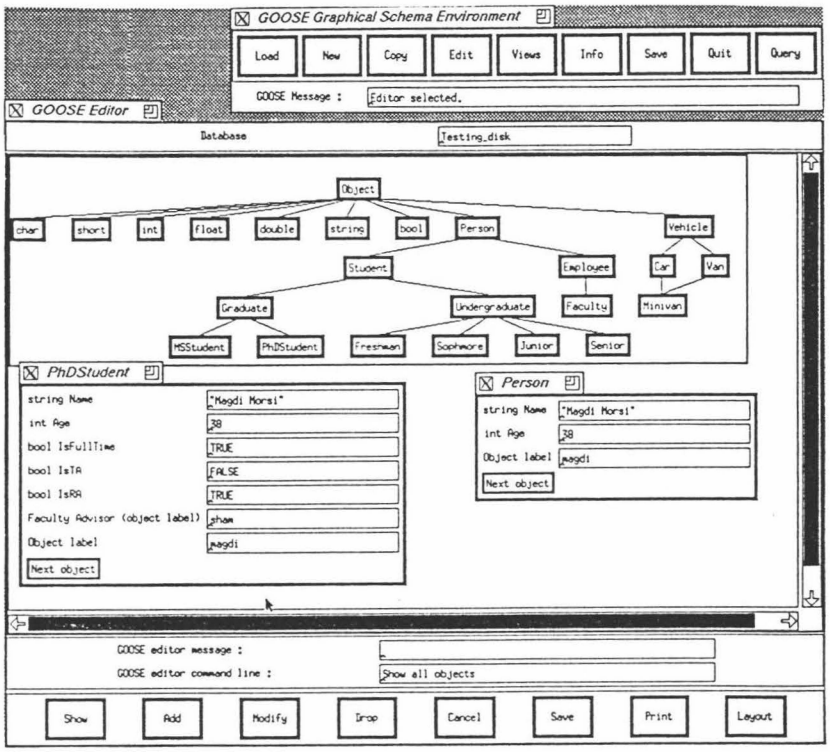


Ready-made presentations of *'City'* and *'Hotel'* objects

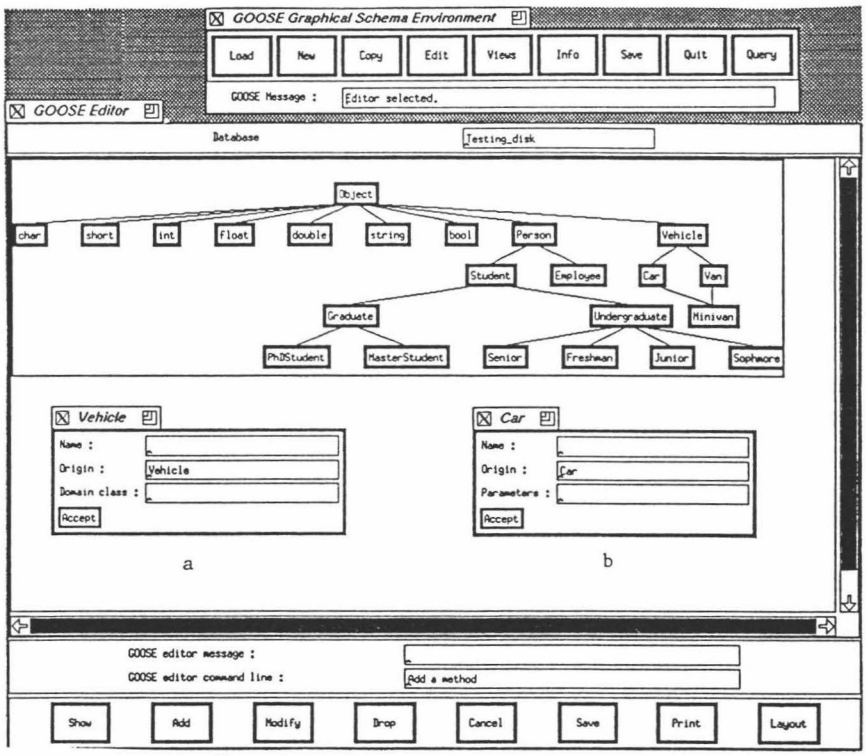


The object Roma, its map, and San Pietro's Basilica

GOOSE Schema browser showing an object through its home class and its superclass



GOOSE templates showing (a) instance variable (b) method



Appendix B : Concurrency Control in OODBMS

B.1 Object Concurrency Identification at OOA

This section addresses the side issue of concurrency control at a high level during an early stage. A diagrammatic convention to indicate message syncronisation is also required.

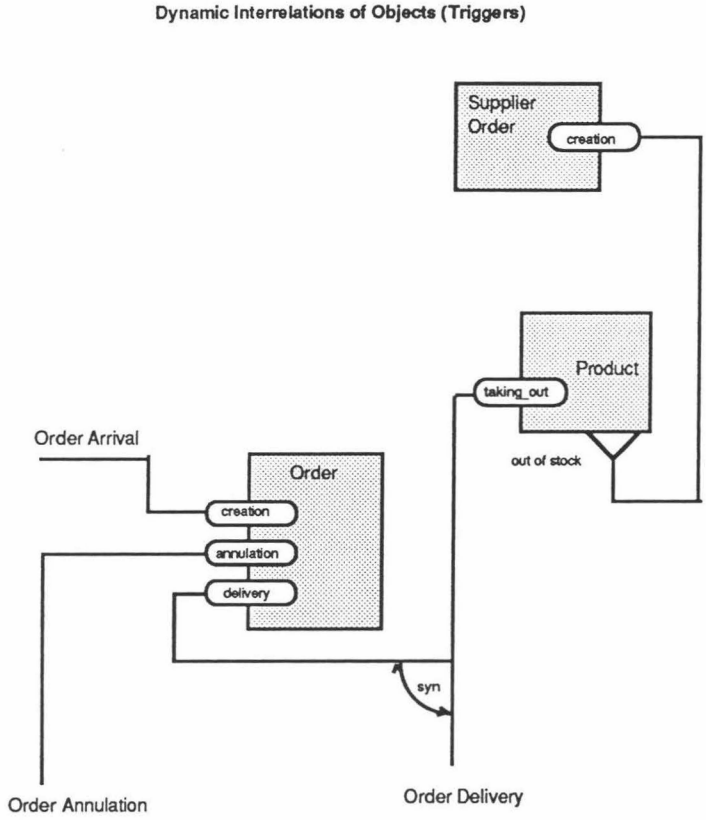


Fig B.1 Dynamic interrelations diagram

In extended relational database, such as Postgres(U. of California, Berkeley), Starburst(IBM), SABRE, EXODUS(U. of Wisconsin), GENESIS(U. of Texas, Austin), triggers that affect the internal state of the object or the state of other objects may be explicitly declared in the **class descriptor**(textual description). In object-oriented databases, rules are however, declared as embedded functions(C++ codes).

Operations of classes could change local values (intra-object triggers) and values of other objects(inter-object triggers) should certain circumstances arise. These operations are noted. Triggers are required to make the database "active" so as to give the user some form of early warning so that actions can be taken. It is because of these triggers that further contribute to the concurrency problem. Assume a situation where several users enter values into the extended RDBMS. All of them are accessing the same database. It could happen that while one user is modifying the values of a collection in object O, an external trigger takes place to modify the state of object O. Therefore, a concurrency control protocol would need to be worked out depending on the application. One way to specifically outline a group of

messages that could cause inter-object triggers is by drawing a **dynamic interrelations diagram.** This diagram should be used to demonstrate the trigger between objects only and is applicable for both extended relatioanl and object-oriented DBMS.

During the Analysis stage, groups of messages that can trigger its own state or the states of other objects are noted.

In the above example, the inter-object trigger is the out of stock condition in object class **Product.** Once the quantity value falls below the minimum set quantity value, it would send an order immediately to the class Supplier Order. This may be implemented by getting the operating system to send an electronic mail order from the client to the vendor. Hence, the object, Product would need concurrency attention at the Design and Implementation stage since it has inter-object triggering capabilities. Synchronisation of normal transaction processing would be handled by the OODBMS.

Also a response to the message Order Delivery could not be carried out until the taking_out function in Product and delivery function in Order has been sucessful. To indicate this, a **syn** symbol is placed at the junction when the message Order Delivery branches out to other objects. **Syn** implies that synchronisation is needed. Exact implementation would again depend on what facilities is provided by the tool used.

B.2 Concurrency Handling in OODBMS

The way concurrency is handled in OODBMS is similar to that of a DBMS, apart from the fact that it is now handling object instances rather than records or tuples. Much of the knowledge gained from concurrency control in traditonal DBMS has been applied and modified to suit the object-oriented ones. In a typical execution environment of a DBMS, transactions run concurrently. In other words, multiple transactions will be active at the same time. These can access and update the same persistent databases or the same persistent data objects. The DBMS would have to gurantee the consistency of the persistent database and the transaction results. Actions should have the following three properties :

(1) Serialisability. Multiple actions that execute concurrently should be scheduled in such a way that the overall effect is as if they were executed sequentially in some order.

(2) Atomicity. An action either successfully completes or has no effect.

(3) Permanence. The effects of an action that successfully completes is not lost, except in the effect of a catastrophic failure.

To guarantee database and transaction consistency, DBMS imposes a serialisable order of execution. Serialisable order means that the results of the transactions are the same as if the transactions were executed one after another (in a series), instead of being executed at the same time.

There are three ways of ensuring serialisability of transactions; time-stamping optimistic synchronisation and pessismistic synchronisation or locking algorithms. Gemstone[49], Ontos[22], and Postgres support either one or all of the below concurrency control strategies.

B.2.1 Time-stamping

With this strategy, each transaction is given a time-stamp (in most cases the transaction start time), and the system attempts to order the execution of transactions based on their time-stamp. For example, assume that there are two transactions on object O, denoted by T1 and T2 and T2 > T1(T2 is older than T1). The DBMS will try to impose T1 before T2 in a serial order. Assume that T2 happens to update object O first and then commits. Later on, T1 attempts to read O which has already been updated earlier by T2. The system detects a conflict and T1 is aborted. The system would then send a message to the user or changes made on O by T2 is undone and recovery is made prior to the transaction by T2. In Postgres, time-stamping are not used to force a strict transaction order but instead it uses normal two phase locking using memory lock table.

B.2.2 Optimistic synchronisation

These algorithms allow transactions to continue executing until they are done. An object does not take steps to prevent conflicts from occurring while transactions are being processed. The transactions update the persistent data in private workspaces. This approach is optimistic because if the transactions aborts (e.g. due to conflicts) all its work will be wasted. When the transaction is done with updating or retrieving from the persistent database, it enters a certification phase and attempts to commit. If the data it had read or updated does not conflict with reads and updates of other transactions, it is allowed to commit. Optimistic synchronisation are best in applications where there are no heavy transactions on data objects.

The major problem with the optimistic scheme is that some actions that successfully complete may still be forced to abort. Furthermore, multiple copies of each object must be maintained in memory to permit concurrency and the changes made by each committed action must be recorded in secondary storage to enable the commit procedure to test for serialisability.

B.2.3 Pessismistic synchronisation or locking algorithm

Locking algorithms assume the worst and acquire locks on every persistent object that a transaction accesses. Read/write locks are the most common mechanisms used by a pessimistic synchronisation scheme. If a transaction reads an object, it must acquire a read lock. If a transaction wants to write an object, it must first acquire a write lock on the object. If an object is already locked, the transactions must wait until the lock is released. Thus there is the potential of deadlocks and the system must use some mechanism either to detect deadlocks and abort a transaction or to prevent an imminent deadlock from occuring in the first place. Ontos supports this scheme of locking algorithms, whereas the GemStone is an add-on to the optimistic scheme.

It is possible to have hybrid schemes that combine the optimistic and pessimistic approaches. Gemstone, for instance, supports both pessimistic and optimistic concurrency control mechanisms. Ontos implements this hybrid scheme through the combination of types of lock and modes of transaction commit i.e. WriteIntent or ReadIntent.

B.2.4 Granularity Locking

Concurrency between objects can be achieved through locking. To what extent should the objects be locked? Should it be at the object level or instance record level? Different constructs of OODBMS will have different mechanisms that can operate at different granularity of objects, i.e. page, segment, object, record level. The granularity chosen for locking would affect the performance which might slow down the requests of users in a multi-user environment.

If locking is chosen at the object level, then requests should be made on individual OIDs. If the OODBMS supports object replication and that these replicated objects could reside in various segments, locking would only be done on these requested objects.

However, if locking at the segment level is chosen, then a large proportion of the OODBMS might be locked which includes the requested and non-requested objects. Locking purely at the object level allow clients to share segments which is more efficient from the user's point of view. However, facilities should be provided to lock the required objects easily. Locking at segment level is easier because only a single specification is needed.

There are many types of read/write locks. In Ontos, there are the read/write locks with or without readers and writers conflict, and null locks. In GemStone, there are read, write and exclusive locks. ENCORE has a very similar locking scheme as Ontos : restrictive and non-restrictive read/write locks and null locks. However, the actual meaning of each type is different as compared to Ontos.

Null locks are called soft locks and the others are called hard locks in the pessimistic scheme. In a client/server architecture supporting pessimistic segment locking scheme, all objects explicitly requested in the segment are hard locked and the non-requested ones uses soft lock. The client does not know exactly which objects would be successfully obtained. Hence, soft locks are viewed as a convenience rather than a necessity. If a hard lock cannot be granted, it is queued; soft locks are not queued. The client requesting a hard lock is notified whether the lock was granted or denied, but is only notified if the lock was granted for soft locks. Using soft locks, the size of the lock queue can be reduced and thus minimising the amount of information returned to the user.

B.2.5 Comparison of pessimistic and optimistic scheme

A pessimistic scheme avoids the overhead of undoing and redoing requests at the expense of reduced concurrency. For example, two actions that examine and modify different parts of the same object are not able to execute concurrently, even when there is no problem of conflict. However, some schemes in ENCORE and Ontos provides a modified non-restrictive read/write conflict which allows a client to read an earlier version of the object while it is currently being written by another client.

An optimistic scheme on the other hand, avoids the overhead of delaying requests at the expense of undoing and redoing requests. Therefore, a pessimistic scheme performs better than an optimistic scheme when conflicts are frequent resulting from heavy transactions from many users.

B.2.6 Synchronisation issues in existing methodologies

All the methodologies that have been examined so far : Booch, OMT, Rolland & Brunet and Coad & Yourdon, none has specifically mention techniques to tackle synchronisation between objects in their high level diagrams.

Synchronisation would be required if objects operate concurrently. Concurrency refers to the potentially parallel execution of parts of a computation. In a concurrent computation, the components of a program may be executed sequentially, or they may be executed in parallel. Two objects are inherently concurrent if they can receive events at the same time without interacting.

Booch has indicated synchronisation in his object diagram, by which messages can exist in five different forms. However, synchronous messages are most frequently applied in OOPL. A synchronous message is one resulting from an operation that commences only when the sender has initiated the action and the receiver is ready to accept the messages; the sender and receiver will wait indefinitely until both parties are ready to proceed. A message is said to be asynchronous if a sender can intiate an action regardless of whether the receiver is expecting the message. In his example in the Smalltalk-80, concurrency is handled by the component, semaphore, in the class library. For C++ running under Unix System V, there is a similar technique used, called shared memory. Again, semaphores are used to control the access to a resource. Apparently, during the design stage, the effects of concurrent events should be noted. However, the use of shared memory (leaving it to the operating system to handle concurrency of objects) is not part of the object-oriented paradigm.

Since the real-world entities is modelled in terms of objects with messages passing between them, it is possible that an object might receive two messages simultaneously. This is particularly true in object-oriented real-time systems. At the top level diagram, using Booch's notation in the object diagram to show the type of message passing between objects should be sufficient. However, at the implementation stage, the designer would need to consider the mechanisms for controlling synchronisation between objects in detail.

In OMT, it is acknowledged that OOPL such as Smalltalk and C++ is inadequate to support the concurrency inherent in objects. Concurrent objects are identified in the dynamic model. However, there are concurrent versions of Smalltalk, Extended Eiffel, and C++(ACTOR version 3.0) which are suitable for use in a parallel and distributed environment.

In Rolland & Brunet's example, they have also suggested that the set of operations to be triggered when a particular situation occurs should be grouped and examined for synchronisation. This should be done at the Analysis stage after all classes and their attributes have been identified.

Coad & Yourdon do not explicitly address this issue. However, in their specification of real-time analysis, they do mention allocating an overall thread budget across the participating Services and Message Connections, which is not explicit.

MASSEY UNIVERSITY

Department of Information Systems

57.ODB Object-Oriented Databases

First Assignment - Postgres

Case Study Description

Moa Insurance is a new venture offering a service to executives wishing to protect themselves in the case of early redundancy. The company collects premiums which are used to build up a fund to provide a supplementary income.

Due to expansion, the company has just acquired new office space, whch consists of part of a single floor of a half-empty tower block. The area is rectangular, 40 metres by 15 metres, with access through a double door in the centre of the short side nearest to the lifts, stairs and toilets.

Moa wishes to use a computer to assist in finding an optimum layout for staff desk space, equipment space and enclosed offices which ensures that staff are located near to the people and facilities they need to communicate with.

Organisation Details

The company has a board of 5 directors and managers, each of whom must have a closed-off area 5m x 3m adjacent to the windows (ie along the long sides). All other staff are to be allocated rectangular areas in the open plan. The 10 section heads are allowed 4m x 2.5m, and the 40 other staff 3m x 2m each. The receptionist's are must be adjacent to the entrance door.

Other areas to be planned are as follows:

Board Room (5m x 4m , enclosed)
2 small meeting/interview rooms (5m x 3m, enclosed)
Reception seating area (4m x 4m, open plan)
Kitchen (3m x 2m, enclosed, must be adjacent to short side nearest the door)
Central computer, file servers, communications boxes (4m x 4m, enclosed)
Line printer / fax room (3m x 2m, enclosed)
Central aisle (35m x 2m, open plan, running down length of floor area from door)

The organisation structure is as follows:

Finance Director - 2 section heads, each with 3 other staff
Sales Director- 3 section heads, each with 2 other staff
Policy Admin Director - 4 section heads, with 2,3, 5 and 7 staff respectively
Marketing and Planning Director - 2 staff
Company Secretary - 5 staff including receptionist,
also controls IT (1 section head with 4 other staff)

Database Content

a) A chain of versions of the planned layout, with identification of the staff member occupying each space. Include rules to ensure no space is placed in a position which overlaps another space, and that all spaces are fully within the overall floor area.
b) The company's organisation structure.

Queries to be Demonstrated

a) Place or move a space to a location on the floor area
b) Add, amend or delete a staff member within the organisation structure
c) Display the average distance apart of staff who work in the same section
d) Display, for each department, the average distance of staff from the printer and fax

Appendix C : Implementation details of Postgres Case Study

This section of Appendix describes in detail, the actual implementation techniques and problems encountered in the implementation of the sample database on a floor layout plan. Anyone who wishes to develop a database with Postgres should go through the demo first.

Postgres version 3.0 was used on the Sunsparc station platform running SunOS 4.1.1. There had been great difficulty at first trying to get Postgres to work on Sun3 machines running on SunOS 4.1.1. The C functions and inheritance worked but the rules did not. System crashes occurred very often. It is recommended that Postgres should be installed on DEC stations 3100 & 5000 running Ultrix 4.0 or higher, Sun4 and Sunsparc stations running OS4.0 or higher. Postgres version 3.0 has also numerous bugs.

Since Postgres is still a research prototype, updates can be obtained from University of California,Berkeley. The Internet address is **128.32.149.1**. There is a public folder in which anyone could login as anonymous and make a binary file transfer. Large files or executable files are normally in tar.Z form. These would need to be uncompressed and extracted from the storage device once a copy was successfully obtained :

uncompress filename.tar
tar -xvf filename.tar

C.1 Implementation of the sample Database

There were two steps involved in the Implementation stage once the initial Analysis stage had been completed.

Firstly, the database was created and populated using Postquel commands. Secondly, the database was tested to see if the functions respond and rules fire.

C.2 Creating and Populating the Database

Emacs files were required for

(a) creating objects(relations),

(b) definition of abstract data types,

(c) definition of functions, and

(d) populating the database with object instances.

First of all, instructions using Postquel for step (a), (b) and (c) was coded in the emacs file **officedb.** This file was tested at the asterisk prompt. If Postgres does not show a warning message, this means the above has been successfully constructed. A retrieval was then made. Since there were no object instances, only the tuple containing the object (relation) attribute names were shown.

Secondly, some sample instances were appended It would be unwise to populate the database with all the instances at this stage. The tracing, **officedboutput** shows the complete population of instances. Note the messages to indicate successful transaction.

C.3 Testing the Rules

It was necessary to test the formulated rules one by one. Three working windows were needed to carry this out in SunView or OpenWindows :

(a) the first one was for running postgres in a console window.

(b) the second one was an emacs window for defining the rule using Postquel commands, e.g. **floor.def.**

(c) the third one was also an emacs window containing test values that would and would not cause the rule to trigger.

After writing the rules in postquel in (b), it was tested out in the window running Postgres(a). If a warning sign appeared instead of a DEFINE sign, it would imply that the syntax of the postquel rule was incorrect and was rejected by Postgres. Once the correct syntax was cleared, testing of the semantics was then proceeded. The test values used in the third window were instances desired to test out the rules(those that would cause the rule to trigger and those that don't).

Upon appending the incorrect values to the database, the rules should trigger(c).

In this exercise, two rules overlap and contain were tested. Overlap was used to test whether two boxes intersect each other. The built-in postquel function **&&** had unexpected result. Results were shown for overlap1 and overlap2.

In **overlap1**, two boxes, (36,10,38,13) and (32,9,34,12) were appended that overlap with existing floor spaces. This successfully blocked that entry of the above two incorrect values into the database. However, attempting to append a correct value (37.5,10,39.5,13) that did not touch the side of existing boxes had been blocked also. See overlap1 tracings.

In **overlap2**, the values were appended again but the rule was removed. The correct value was appended. See overlap2 tracings.

Implementation of overlapping function using Postquel was not satisfactory as demonstrated in overlap1 and overlap2. In testoverlap, a C function was embedded within a rule which produced the desired results. See **overlapoutput.**

We may conclude that from the above listing that **&&** works fine for doing a retrieval and has a bug in it when incorporating it into the rules.

Postgres allows instances of the same identifier to be appended many times. As a result there would be many repetitive tuples due to iterative testing, making the database very untidy. The database would need to be destroyed and created occasionally to tidy up occasionally. This was a time consuming process.

Using rules would safeguard against appending incorrect values.

C.4 Testing the Functions

Another approach of ensuring correct values in the database was through the use of functions. Initially, incorrect values were allowed into the database. They were then retrieved by using these user-defined functions.

Just like testing the rules, functions should also be tested one by one. Preferably, four windows were required :

(a) the first one was for running postgres in a console window,

(b) the second one was an emacs window for editing C functions,

(c) the third one was for compiling the C file into an object file so that it could be loaded into the database,

(d) the fourth one was an emacs file containing the definition of the function, load command and the postquel retrieve command.

The first window would have the asterisk prompt for running postgres. Writing C functions would be done on the second window. C functions would be compiled in the third window. An object file could be obtained by using the unix command executed in the third window :

cc filename.c -c -o filename.o

In this case, it would be

cc window.c -c -o window.o

A loading path was then defined for the successfully compiled object file. It was important to note that the C filename must not be the same as the C function name. As an example, the function for retrieving those boxes along the longside of the wall would be Window which was contained in the C file named **window.c.**

To test out the C function, the function and the path on which the object file could be loaded was defined, presuming that the incorrect values had been populated into the database. This emacs test file called **window.def** should also contain a postquel query to retrieve all the incorrect values. Once the function was tested working, the definition could then be incorporated into part of the database **officedb.**

In the **testwindow** tracings, those open_area locations that were and not along the longside of the wall were retrieved. It was also tried on closed_area. In this exercise, assume that there were many windows along the longside of the wall to make it simple.

Note that C functions may be inherited down the hierarchy as shown in **testwindow** output. The same function could be applied throughout the subclasses (page 19 of the Postgres manual). This is however not possible through the use of rules.

C.5 Creating versions of the relations

The concept of versioning is also important in CAD/CAM. The example query showed creating a version out of the closed_area class.

create version ca1 from closed_area["now"]\g

"Now" is a representation of a timestamp on the newly created version. Right now, closed_area becomes the base class and cal becomes the working copy. Changes made to the version will not propogate to the base class as shown in versionoutput. The merge command is only supported in Postgres version 4.0(page 47 of the Postgres manual) and hence would not work in this case.

C.6 Loading a picture file in Postgres

PICASSO is a GUI toolkit for use with Postgres.

The object management feature of OODBMS should enable it to handle large objects like bitmap also. Bitmap can be defined as type map in Postgres. However, this feature is not implemented yet. It would be useful if a photo of an employee could be displayed by supplying the identifier of a Postquel retrieval command.

Since there was a choice to use Postgres in OpenWindows, X-Windows utilities could be used to load a picture file directly, bypassing Postgres. Loading of picture files would not work in SunView.

Firstly, sis-server machine had to treat ms-suna as a host which would then return a message. Assuming on machine sis-lab12 :

```
sis-lab12 BLiew > xhost ms-suna
ms-suna being added to control list                (return message)
```

Then login to ms-suna in the sis-server machine :

```
sis-lab12 BLiew > rlogin ms-suna
```

Once in ms-suna, set up the shared library correctly so that once the xv command is invoked, it will search the correct path. This is best appended in the .login file to save the trouble of typing it every time :

```
setenv LD_LIBRARY_PATH /usr/openwin/lib :
    /usr1/local/bin/X11/usr/lib
```

Instruct X-Window to load the picture file unto machine sis-lab12 when the xv command is executed. Then type in the name of the picture file.

```
ms-suna BLiew > setenv DISPLAY sis-lab12:0
ms-suna BLiew > xv filename
```

A photo of the employee would then be displayed at the top right hand corner of the screen.

Since, ms-suna and sis-server are different nodes, this further demonstrates the distribution capability of Postgres where you can retrieve information from a remote location.

The following pages were the tracings of this exercise.

```
/*  officedboutput  */
ms-suna BLiew : createdb OFFICE
ms-suna BLiew : monitor OFFICE
Welcome to the C POSTGRES terminal monitor

Go
* \i  officedb  \g

Query sent to backend is "      define function boxarea(language = "c", returntype=int4) arg is
(box)  as  "/home/ms-suna/postgres/o"  "
LOAD
Query sent to backend
Query  sent  to  backend  is "load "/home/ms-suna/BLiew/window.o" "
LOAD
Query sent to backend is "    create working_area(location = box)  "
CREATE
Query sent to backend is "create open_a (working_area)"
CREATE
Query sent to backend is "create closed_area(name=char16)inherits(open_area,workcreate
staff(staff_id=char16,  name=char16,  age=int4,  salary=int4,manager=char16)"
CREATE
Query sent to backend is "  append group_area(location="(25,3,30,7)",purpose="BoardRm")"
APPEND
Query sent to backend is "append group_area(location="(14,9,19,12)", purpose=
"InterviewRm")"
APPEND
Query sent to backend is "append group_area(location="(19,9,24,12)",purpose=
"InterviewRm")"
APPEND
Query sent to backend is "append group_area(location="(10,9,14,13)",purpose="CenCom")"
APPEND
Query sent to backend is "append group_area(location="(12,13,15,15)", purpose=
"LinePrint")"
APPEND
Query sent to backend is "append group_area(location="(0,4,2,7)",purpose="Kitchen")"
APPEND
Query sent to backend is "  append closed_area(location="(20,0,25,3)",staff_id="PA",
dept="PolicyAdmin",name="Cattel")"
APPEND
Query sent to backend is "append closed_area(location="(25,0,30,3)",staff_id="MP1",
dept="Market&Plan",name="Zdonik")  "
APPEND
Query sent to backend is "append closed_area(location="(30,0,35,3)",staff_id="SA",
dept="Sales",name="Maier")  "
APPEND
Query sent to backend is "append closed_area(location="(15,12,20,15)",staff_id="CS",
dept="CompanySec",name="Kim")"
APPEND
Query sent to backend is "append closed_area(location="(20,12,25,15)",staff_id="FI",
dept="Finance",name="Lochovsky")"
APPEND
Query sent to backend is "append open_area(location="(0,2,3,4)",staff_id="pa1",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(0,0,3,2)",staff_id="pa2",dept=
"PolicyAdmin")"
APPEND
```

```
Query sent to backend is "append open_area(location="(3,2,6,4)",staff_id="pa3",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(3,0,6,2)",staff_id="pa4",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(6,2,9,4)",staff_id="pa5",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(6,0,9,2)",staff_id="pa6",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(9,2,12,4)",staff_id="pa7",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(9,0,12,2)",staff_id="pa8",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(12,2,15,4)",staff_id="pa9",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(12,0,15,2)",staff_id="pa10",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(2,4,4,7)",staff_id="pa11",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(4,4,6,7)",staff_id="pa12",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(6,4,8,7)",staff_id="pa13",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(8,4,10,7)",staff_id="pa14",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(12,4,14,7)",staff_id="pa15",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(12,4,14,7)",staff_id="pa16",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(14,4,16,7)",staff_id="pa17",dept=
"PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(15,0,17.5,4)",staff_id="pash1",
dept= "PolicyAdmin")"
APPEND
Query sent to backend is "append open_area(location="(17.5,0,20,4)",staff_id="pash2",
dept= "PolicyAdmin")"
APPEND
Query sent to backend is "   append open_area(location="(16,4,18,7)",staff_id="mp1",dept=
"MarPolicy")"
APPEND
Query sent to backend is "append open_area(location="(35,0,37,3)",staff_id="s1",dept=
"Sales")"
APPEND
```

```
Query sent to backend is "append open_area(location="(37,0,39,3)",staff_id="s2",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(38,3,40,6)",staff_id="s3",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(35,7,38,9)",staff_id="s4",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(38,7,40,10)",staff_id="s5",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(38,10,40,13)",staff_id="s6",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(30,3,32.5,7)",staff_id="ssh1",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(32.5,3,35,7)",staff_id="ssh2",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(35,3,37.5,7)",staff_id="ssh3",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(0,9,4,13)",staff_id="Recep",dept=
"ComSec")"
APPEND
Query sent to backend is "append open_area(location="(0,13,3,15)",staff_id="cs1",dept=
"ComSec")"
APPEND
Query sent to backend is "append open_area(location="(3,13,6,15)",staff_id="cs2",dept=
"ComSec")"
APPEND
Query sent to backend is "append open_area(location="(6,13,9,15)",staff_id="cs3",dept=
"ComSec")"
APPEND
Query sent to backend is "append open_area(location="(4,11,7,13)",staff_id="cs4",dept=
"ComSec")"
APPEND
Query sent to backend is "append open_area(location="(4,9,7,11)",staff_id="it1",dept=
"infotech")"
APPEND
Query sent to backend is "append open_area(location="(7,11,10,13)",staff_id="it2",dept=
"infotech")"
APPEND
Query sent to backend is "append open_area(location="(7,9,10,11)",staff_id="it3",dept=
"infotech")"
APPEND
Query sent to backend is "append open_area(location="(9,13,12,15)",staff_id="it4",dept=
"infotech")"
APPEND
Query sent to backend is "append open_area(location="(25,9,27.5,13)",staff_id="itsh",
dept= "infotech")"
APPEND
Query sent to backend is "append open_area(location="(34,9,37,11)",staff_id="f1",dept=
"fin")"
APPEND
```

```
Query sent to backend is "append open_area(location="(34,11,37,13)",staff_id="f2",dept=
"fin")"
APPEND
Query sent to backend is "append open_area(location="(34,13,37,15)",staff_id="f3",dept=
"fin")"
APPEND
Query sent to backend is "append open_area(location="(25,15,28,15)",staff_id="f4",dept=
"fin")"
APPEND
Query sent to backend is "append open_area(location="(28,13,31,15)",staff_id="f5",dept=
"fin")"
APPEND
Query sent to backend is "append open_area(location="(31,13,34,15)",staff_id="f6",dept=
"fin")"
APPEND
Query sent to backend is "append open_area(location="(25,9,27.5,13)",staff_id="fsh1",
dept= "fin")"
APPEND
Query sent to backend is "append open_area(location="(27.5,9,30,13)",staff_id="fsh2",
dept= "fin")"
APPEND
Query sent to backend is "append open_area(location="(30,9,32.5,13)",staff_id="fsh3",
dept= "fin")"
APPEND
Query sent to backend is "append staff(staff_id="f1", name="john", age=34, salary=4500,
manager="elliot")"
APPEND
Go
*
Query sent to backend is " "


/*     floor.def   */
define function notFloor( language = "c",
                    returntype = bool)
             arg is (box)
                as "/home/ms-suna/BLiew/floor.o"  \g
load  "/home/ms-suna/BLiew/floor.o"    \g



Go
* \i  notFloortest  \g

Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
-----------------------------------------
| staff_id   | dept       | location    |
-----------------------------------------
| s1         | Sales      | (37,3,35,0) |
-----------------------------------------
| s2         | Sales      | (39,3,37,0) |
-----------------------------------------
| s3         | Sales      | (40,6,38,3) |
-----------------------------------------
| s4         | Sales      | (38,9,35,7) |
-----------------------------------------
| s5         | Sales      | (40,10,38,7)|
-----------------------------------------
```

```
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
| s6         | Sales      | (40,13,38,10)|
----------------------------------------
Query sent to backend is "delete open_area where open_area.staff_id="s6""
DELETE
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is " append open_area(location="(40,0,43,2)",staff_id="s6",
dept="Sales")"
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
| s6         | Sales      | (43,2,40,0) |
----------------------------------------
Query sent to backend is "append open_area(location="(38,10,40,13)", staff_id="s6",dept=
"Sales")"
APPEND
```

```
Query sent to backend is "retrieve (open_area.all) where notFloor(open_area.location) and
open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s6         | Sales      | (43,2,40,0) |
----------------------------------------

Go
*
Query sent to backend is " "



/*  overlap1  */

retrieve (open_area.all) where open_area.dept="Sales"\g
delete open_area where open_area.staff_id="s6"\g
retrieve (open_area.all) where open_area.dept="Sales"\g

define rule overlap is
on append to open_area where new.location &&
open_area.location
do instead nothing \g

/* These are overlapping boxes */
append  open_area(location="(36,10,38,13)",staff_id="s6",dept="Sales")\g

append  open_area(location="(32,9,34,12)",staff_id="s6",dept="Sales")\g

retrieve (open_area.all) where open_area.dept="Sales"\g

/* This box does not touch the side of existing one */
append  open_area(location="(37.5,10,39.5,13)",staff_id="s6",dept="Sales")\g

retrieve (open_area.all) where open_area.dept="Sales"\g


\*  Output for overlap1  *\
* \i overlap1 \g

Query sent to backend is "  retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
```

```
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
| s6         | Sales      | (40,13,38,10)|
----------------------------------------
Query sent to backend is "delete open_area where open_area.staff_id="s6""
DELETE
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "define rule overlap is on append to open_area where new.location
&& open_area.location do instead nothing "
DEFINE
Query sent to backend is " append open_area(location="(36,10,38,13)",staff_id="s6",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(32,9,34,12)",staff_id="s6",dept=
"Sales")"
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "append open_area(location="(37.5,10,39.5,13)", staff_id="s6",
dept="Sales")"
```

```
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------

Go
*
Query sent to backend is " "


/*  overlap2  */

retrieve (open_area.all) where open_area.dept="Sales"\g
delete open_area where open_area.staff_id="s6"\g
retrieve (open_area.all) where open_area.dept="Sales"\g

define rule overlap is
on append to open_area where new.location &&
open_area.location
do instead nothing \g

/* These are overlapping box */
append  open_area(location="(36,10,38,13)",staff_id="s6",dept="Sales")\g

append  open_area(location="(32,9,34,12)",staff_id="s6",dept="Sales")\g
retrieve (open_area.all) where open_area.dept="Sales"\g
remove rule overlap\g

/* This box does not touch the side of existing one */
append  open_area(location="(37.5,10,39.5,13)",staff_id="s6",dept="Sales")\g

retrieve (open_area.all) where open_area.dept="Sales"\g


* \i overlap2 \g

Query sent to backend is "  retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
```

```
| s1        | Sales      | (37,3,35,0) |
----------------------------------------
| s2        | Sales      | (39,3,37,0) |
----------------------------------------
| s3        | Sales      | (40,6,38,3) |
----------------------------------------
| s4        | Sales      | (38,9,35,7) |
----------------------------------------
| s5        | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1      | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2      | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3      | Sales      | (37.5,7,35,3)|
----------------------------------------
| s6        | Sales      | (40,13,38,10)|
----------------------------------------
Query sent to backend is "delete open_area where open_area.staff_id="s6""
DELETE
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id  | dept       | location    |
----------------------------------------
| s1        | Sales      | (37,3,35,0) |
----------------------------------------
| s2        | Sales      | (39,3,37,0) |
----------------------------------------
| s3        | Sales      | (40,6,38,3) |
----------------------------------------
| s4        | Sales      | (38,9,35,7) |
----------------------------------------
| s5        | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1      | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2      | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3      | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "define rule overlap is on append to open_area where new.location
&& open_area.location do instead nothing "
DEFINE
Query sent to backend is " append open_area(location="(36,10,38,13)",staff_id="s6",dept=
"Sales")"
APPEND
Query sent to backend is "append open_area(location="(32,9,34,12)",staff_id="s6",dept=
"Sales")"
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id  | dept       | location    |
----------------------------------------
| s1        | Sales      | (37,3,35,0) |
----------------------------------------
| s2        | Sales      | (39,3,37,0) |
----------------------------------------
| s3        | Sales      | (40,6,38,3) |
```

```
----------------------------------------
| s4        | Sales      | (38,9,35,7) |
----------------------------------------
| s5        | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1      | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2      | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3      | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "remove rule overlap"
REMOVE
Query sent to backend is " append open_area(location="(37.5,10,39.5,13)", staff_id="s6",
dept="Sales")"
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id  | dept       | location    |
----------------------------------------
| s1        | Sales      | (37,3,35,0) |
----------------------------------------
| s2        | Sales      | (39,3,37,0) |
----------------------------------------
| s3        | Sales      | (40,6,38,3) |
----------------------------------------
| s4        | Sales      | (38,9,35,7) |
----------------------------------------
| s5        | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1      | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2      | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3      | Sales      | (37.5,7,35,3)|
----------------------------------------
| s6        | Sales      | (39.5,13,37.5,10)|
----------------------------------------
Go
*
Query sent to backend is " "
```

```
/* overlap.c  Test if two boxes overlap */

typedef struct {
        double  xh, yh, xl, yl;      /* high and low coords */
} BOX;

#define ABS(X) ((X) > 0 ? (X) : -(X))

int Overlap( box1, box2)
      BOX *box1, *box2;
{
int  result;

result =  (
```

```
( box1->xl <= box2->xl && box1->xl < box2->xh &&
  box1->xh > box2->xl && box1->xh <= box2->xh &&
  box1->yl <= box2->yl && box1->yl < box2->yh  &&
  box1->yh > box2->yl && box1->yh <= box2->yh ) ||

(box2->xl <= box1->xl && box2->xl < box1->xh &&
  box2->xh > box1->xl && box2->xh <= box1->xh &&
  box2->yl <= box1->yl && box2->yl < box1->yh  &&
  box2->yh > box1->yl && box2->yh <= box1->yh ));

return  result;

}/*  overlap    */


* \i  testoverlap \g

Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| s6         | Sales      | (40,13,38,10)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "delete open_area where open_area.staff_id="s6""
DELETE
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
```

```
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "define rule overlapping is on append to open_area where
Overlap(new.location, open_area.location) do instead nothing"
DEFINE
Query sent to backend is "append open_area(location="(36,10,38,13)",staff_id="s6", dept=
"Sales")"
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
Query sent to backend is "append open_area(location="(37.5,10,39.5,13)",staff_id="s6",
dept="Sales")"
APPEND
Query sent to backend is "retrieve (open_area.all) where open_area.dept="Sales""
----------------------------------------
| staff_id   | dept       | location    |
----------------------------------------
| s1         | Sales      | (37,3,35,0) |
----------------------------------------
| s2         | Sales      | (39,3,37,0) |
----------------------------------------
| s3         | Sales      | (40,6,38,3) |
----------------------------------------
| s4         | Sales      | (38,9,35,7) |
----------------------------------------
| s5         | Sales      | (40,10,38,7)|
----------------------------------------
| ssh1       | Sales      | (32.5,7,30,3)|
----------------------------------------
| ssh2       | Sales      | (35,7,32.5,3)|
----------------------------------------
| ssh3       | Sales      | (37.5,7,35,3)|
----------------------------------------
| s6         | Sales      | (39.5,13,37.5,10)|
----------------------------------------
Go
*
Query sent to backend is " "
```

```
/*window.def  */
define function Window(language = "c", returntype=bool)
arg is (box)
as  "/home/ms-suna/BLiew/window.o"  \g
load  "/home/ms-suna/BLiew/window.o"  \g


/*window.c*/

typedef struct {
        double  xh, yh, xl, yl;      /* high and low coords */
} BOX;

#define ABS(X) ((X) > 0 ? (X) : -(X))
#define XMAX 40
#define YMAX 15
#define XMIN 0
#define YMIN 0

int Window(box)
BOX *box;
{
int result= ( (box->yh == YMAX) || (box->yl == YMIN) );

return  result;

}    /*Window*/


/*  testwindow  */
retrieve (open_area.all) where Window(open_area.location)
and  open_area.dept="fin"\g

retrieve (open_area.all) where not Window(open_area.location) and
  open_area.dept="fin"\g

retrieve (closed_area.all) where not Window(closed_area.location)\g

/*  windowoutput  */
* \i  testwindow  \g

Query sent to backend is "retrieve (open_area.all) where Window(open_area.location) and
open_area.dept="fin""
-----------------------------------------
| staff_id   | dept       | location    |
-----------------------------------------
| f3         | fin        | (37,15,34,13)|
-----------------------------------------
| f4         | fin        | (28,15,25,15)|
-----------------------------------------
| f5         | fin        | (31,15,28,13)|
-----------------------------------------
| f6         | fin        | (34,15,31,13)|
-----------------------------------------

Query sent to backend is "retrieve (open_area.all) where not Window(open_area.location) and
open_area.dept="fin""
```

```
-----------------------------------------
| staff_id   | dept       | location    |
-----------------------------------------
| f1         | fin        | (37,11,34,9)|
-----------------------------------------
| f2         | fin        | (37,13,34,11)|
-----------------------------------------
| fsh1       | fin        | (27.5,13,25,9)|
-----------------------------------------
| fsh2       | fin        | (30,13,27.5,9)|
-----------------------------------------
| fsh3       | fin        | (32.5,13,30,9)|
-----------------------------------------
```

Query sent to backend is "retrieve (closed_area.all) where not Window (closed_area.location)"

```
-----------------------------------------------------------
| name       | location   | dept       | staff_id    |
-----------------------------------------------------------
```

MASSEY UNIVERSITY

Department of Information Systems

57.ODB Object-Oriented Databases

Second Assignment - Alternative Ontos Case Study

Case Study Description

Due to problems with electricity generation caused by low water levels in reservoirs, a new group has been set up to keep a Hydrographic database of water volumes at generating stations, levels in reservoirs, and capacity in catchment areas. It is planned to have graphics displays to show the data and predictions in map format. You have been asked to advise on the design of an Object-Oriented Database (using Ontos in this case).

Data Content

a) Lakes and reservoirs - polygonal boundary and enclosed area. Attributes include levels: normal, actual, minimum allowed; surface area; %catchment - if 1 mm of rain falls on the catchment area, by how much does the lake level rise?; average delay - how many days before it does rise?
b) Rivers and streams - to be plotted as a series of line segments, allowing flows to be aggregated
c) Catchment areas - apply with respect to a lake or measuring point. Consist of a polygonal boundary and an enclosed area. Attributes include water "locked up" (eg in snow, ground water etc) by month.
d) Measuring points on rivers and streams - with flow rates for each day
h) Generating stations - flow of water required per 1% of generating capacity, flow of water actually reaching the station
i) Weather stations - points with daily rainfall, snowfall and temperature records and forecasts
e) Overall area of "The Mainland" as outlined by a polygonal boundary
f) Major topographical point features for map display orientation, ie cities, towns, peaks, each with name to be displayed in characteristic font for each type of feature
g) Daily power demand (forecast and actual) in terms of % capacity at which generating stations have to run

Behavioural Specification

The purpose of the system is to operate a model of the Hydro generation situation, so that better decisions can be made about use of other (thermal) power sources.

1 Precipitation in catchment areas is converted to "locked up" water, river flow or lake level increase (with appropriate delay).
2 Temperature triggers melting of snow, releasing "locked up" water
3 Power generation is converted to river flow requirement.
4 Actual river flow is subtracted and shortfall is taken from lakes by opening sluices
5 Lake level trends are extrapolated and management alerted if the number of days before lakes reach their minimum level drops to less than 1 month.
6 Weather forecasts are ignored.

Tasks - all students to complete a), b) and c), then EITHER d) OR e)

a) Define the structure of the database (Ontos Types, Properties and Procedures)
b) Outline the logic required for the methods (Procedures) (pseudo-code, not C++)
c) Write a sample OSQL query, and show how the Ontos Query Iterator would be used to handle the result
*** d) Outline the structure of further objects representing user interfaces to the system
* (eg a map of the area, a schematic diagram of water flows, graphs of lake levels etc)
*** e) Write a skeleton set of C++ classes to support the structure and behaviour of the system

Appendix D : Implementation details of Ontos Case Study

The Ontos Assignment will show :

(a) The organisation of real-world objects into Ontos schematic and other requirements.

(b) Skeleton set of C++ codes and brief description of each Type

(c) Application of concurrency control protocol in main.c program

(d) Population of some instances through the main.c program

(e) Perform a simple OSQL in the main.c program

D.1 Schematic Objects

Class Hydrographic_Object

The type Hydrography was the context object by which all other objects, like Lakes, Rivers, etc form an "is part of " inheritance relationship with it. One of the characteristics of the object-oriented paradigm was polymorphism. Therefore, it was possible to define virtual functions which would then determine the correct behaviour at run-time for each subclass. For example, the function display(argument) would display the correct picture depending on the argument.

In the private part, references were made to the other persistent data objects. These data objects were stored in the form of collections(lists, dictionary, array, etc) and addressed by the primary key. Imagine that these persistent objects were equivalent to relations in relational databases.

In the public part, there were standard functions required for a type. Every type must have a special activation constructor, APL which loaded the object from the server cache into the client cache and thus "activating" the object. They would also require a special member function called getDirectType( ) which was virtually defined by the Ontos class Entity. Since it had a destructor, it should also have a function called Destroy( ). Also since it was persistent, then the function putObject and deleteObject should be included.

```
class Hydrographic_Object : public Object {
    private :
        Reference priv_catchment;
        Reference priv_genstations;
        Reference priv_weatherstations;

    HydroIterator* getIterator( );
    public :
        Hydrographic_Object ( char* nameofregion);
        Hydrographic_Object ( APL* theAPL);
```

```
        ~ Hydrographic_Object( );
        virtual Type* getDirectType();
        virtual void Destroy ( OC_Boolean aborted = FALSE );
        virtual putObject ( OC_Boolean deallocate = FALSE );
        virtual deleteObject ( OC_Boolean deallocate = FALSE );
        void zoomRegion(nameofregion);
        //contains also procedures for coordinating message passing activities and signals
        // from remote sensors.
};
```

class Time

The class Time serves as an abstraction to data from the Measuring_Points at regular intervals. It may also be set to trigger the objects to perform some required function. To save space in the database, it is declared the class as non-persistent.

The function interval sends a pulse at regular interval to request for sampled data from the Measuring_Point class.

```
class Time {
    private :
        unsigned priv_minute;
        unsigned priv_day;
        unsigned priv_month;
        unsigned priv_year;
    public :
        Time( int min, int day, int month, int year );
        Time(char* DateTimeString);
        Time(APL* theAPL);

        interval( );
};
```

class Lakes

Class Lakes and Rivers were subclasses of Catchment. When there was rainfall and snowfall, water would go into the catchment area. Precipitation was converted into "locked-up" water, i.e. there was no immediate rise in level of the rivers and lakes. The class Lakes contained the private attributes. We could picture this as in the form of a relation and the constructor was used to instantiate objects. The object instance created was then stored in aggregate type Dictionary. Since the spatial attribute of the class Lake was in the form of polygonal boundary and enclosed area, it would have to be entered into the

database through STUDIO, the form designer of Ontos. It could be drawn or a picture scanned. Referencing in the persistent store Dictionary was done by the primary key priv_lakename.

Data logging was done periodically by the Type Time and entered into the persistent store of each respective data objects so that they might later be retrieved for statistical purposes. The member function comp_normal_lake_level compared the sampled data entered to the persistent store with the threshold value. Opening_sluices would regulate the flow of the river when the threshold level of the lake was reached. This function satisfy the behavioural specification no. 4 in the problem domain.

The function Lakelevel appeared as a function in the STUDIO window. When this function could also be invoked by clicking the mouse on the STUDIO double-click box in which a graphical display of the lake would be invoked.

The function extrapolate_graph satisfy behavioural specification no. 5.

```
class Lakes : public Lakes {
    private :          // Dictionary of hydrography objects indexed
                              // Primary key need not be integer
        char* Lakename;
        int priv_normal_level;
        int priv_min_level;
        int priv_surface_area;
        Reference priv_lakename;
    public :
        Lakes ( char*  Lakename, int normal_level, int min_level, int surface_area );
        Lakes ( APL*  theAPL );
        ~ Lakes( );
        virtual Type*  getDirectType( );
        virtual void Destroy ( OC_Boolean aborted = FALSE );
        virtual putObject ( OC_Boolean deallocate = FALSE );
        virtual deleteObject ( OC_Boolean deallocate = FALSE );
        void comp_normal_river_level( int nrl );
        void extrapolate_graph( );
        void display_lake( );
        void opening_sluices( );
        };
```

class Rivers

This class was a subclass of Catchment. It was also a persistent object type and contains the required virtual functions.

```
class Rivers : public Catchment {
```

```
        private :                                   // Dictionary of Hydrography objects
                Reference priv_rivername;
                char* RiverName;
                int priv_length;
        public :
                Rivers ( char*  RiverName, int length );
                Rivers ( APL* theAPL );
                ~Rivers( );
                virtual Type*  getDirectType( );
                virtual void Destroy ( OC_Boolean aborted = FALSE );
                virtual putObject ( OC_Boolean deallocate = FALSE );
                virtual deleteObject ( OC_Boolean deallocate = FALSE );
};
```

class Catchment

The class Catchment was made up of subclasses Rivers and Lakes. Member functions %catchment was used to calculate the percentage rise in water level when 1 mm of rain or snow fell on the catchment area. The function average_delay was for calculating the time taken for the water level in the catchment area to rise. This was dependent on the temperature. This function satisfied behavourial specification no. 1.

```
class Catchment : public Hydrographic_Object {
        private :                                   // Dictonary of Hydrographic_Objects
                Reference priv_catchmentname;
        public :
                Catchment ( char*  CatchmentName );
                River ( APL*   theAPL );
                ~River( );
                virtual Type*  getDirectType( );
                virtual void Destroy ( OC_Boolean aborted = FALSE );
                virtual putObject ( OC_Boolean deallocate = FALSE );
                virtual deleteObject ( OC_Boolean deallocate = FALSE );
                void %catchment( );
                void average_delay( );
};
```

class Measuring_Points

This was stored as a persistent object. The member function cal_flow_rate obtained a value from remote flow sensor on the river. Data logging was done periodically by the systems clock and the daily flow rate was entered into the persistent store. The function aggregate_flow summed up all the flow rate of the rivers.

```
class Measuring_Points : public Hydrographic_Object {
        private :
                Reference priv_mp;
                int priv_flow_rate;
        public :
                Measuring_Points ( int MeasuringPoint, flow_rate );
                Measuring_Points ( APL*  theAPL );
                ~Measuring_Points( );
                virtual Type*  getDirectType( );
                virtual void Destroy ( OC_Boolean aborted = FALSE );
                virtual putObject ( OC_Boolean deallocate = FALSE );
                virtual deleteObject ( OC_Boolean deallocate = FALSE );

                void cal_flow_rate( );
                void aggregate_flow( );
                void lakelevel( );
};
```

Class Generating_Stations

This was also a persistent object. The member function %generating_capacity was for calculating the flow of water needed for one percent of generating capacity (MegaWatt). This function satisfy behavioural specification no. 3.

```
class Generating_Stations :: public Hydrographic_Object {
        private :
                Reference priv_gen_name;
                int priv_capacity;
        public :
                Generating_Stations (int gen_id, char* gen_name, int capacity);
                Generating_Stations (APL* theAPL);
                ~Generating_Stations( );
                virtual Type* getDirectType( );
```

```
        virtual void Destroy ( OC_Boolean aborted = FALSE );
        virtual putObject( OC_Boolean deallocate = FALSE );
        virtual deleteObject( OC_Boolean deallocate = FALSE ):

        void %generating_capacity( );
};
```

class Weather_Stations

The member functions, display_rainfall, display_snowfall, and display_temperature would give a graphical display when the mouse was double clicked in the STUDIO window.

The function weather_forcast provided weather information for the next 24 hrs. However, this information was not to be taken seriously. It satisfied behavioural specification no. 6.

```
class Weather_Stations : public Hydrographic_Object {
    private :
        Reference priv_wea_name;
        int priv_rainfall;
        int priv_snowfall;
        int priv_temp;
        char* priv_forcast;
    public :
        Weather_Stations ( char* wea, int rainfall, int snowfall, int temp,
                                    char* forcast );
        Weather_Stations ( APL* theAPL );
        ~Weather_Stations( );
        virtual Type* getDirectType( );
        virtual void Destroy ( OC_Boolean aborted = FALSE );
        virtual putObject ( OC_Boolean deallocate = FALSE );
        virtual deleteObject ( OC_Boolean deallocate = FALSE );
        void display_rainfall( );
        void display_snowfall( );
        void display_temperature( );
        void weather_forcast( );
};

class City :: public Hydrographic_Object {
    private :
        Reference priv_cityname;
```

```
        public :
                City ( char* cityname );
                City ( APL*  theAPL );
                City( );
                virtual Type*  getDirectType( );
                virtual void Destroy ( OC_Boolean aborted = FALSE );
                virtual putObject ( OC_Boolean deallocate = FALSE );
                virtual deleteObject ( OC_Boolean deallocate = FALSE );

                void display_city( );
};

enum MRI_Status { onLakes, onRivers, on
class HydroIterator  {
private :
        Hydrographic_Object* priv_hydro;
        MRI_Status priv_status;
        AggregateIterator priv_iterator;
public :
        HydroIterator ( Hydrographic_Object* theHydrographic_Object );
        ~HydroIterator( );
        void Reset( );
        OC_Boolean moreData( );
        DataObject* operator( ) ( );        };

class HydroException : public Failure {
private :
        Hydrographic_Object*  priv_hydro;
public :
        Hydrographic_Object*  Hydrographic_Object( ) { return priv_hydro; }
        virtual void Report( ) = 0;
        OC_STANDARD_FAILURE_MEMBER_DECLS;          };
```

Control Files

```
// Implementation for class Time
Time :: Time(char* DateTimeString) {
        int min,  day,  month,  year;
```

```
        sscanf( DateTimeString, "%d / %d / %d / %d, &min, &day, &month, &day);
                priv_minute = min;
                priv_day = day;
                priv_month = month;
                priv_year = year;      }

Time :: Time(APL* theAPL) { }
Time :: interval( ) {
//      Sends regular pulse to sensors for datalogging     }

// Implementation for class Hydrographic_Object
Hydrographic_Object :: Hydrographic_Object ( APL* theAPL ) : (theAPL) { }
Type* Hydrographic_Object :: getDirectType( ) {
        return (Type*) OC_lookup ( "Hydrographic_Object");      }

Hydrographic_Object :: ~Hydrographic_Object( ) {
        Destroy ( FALSE );      }

void Hydrographic_Object ::  Destroy ( OC_Boolean aborted ) {
        Entity* ent;
        ent = priv_catchment.Binding(this); delete ent;
        ent = priv_genstations.Binding(this); delete ent;
        ent = priv_weatherstations.Binding(this); delete ent;
        if (aborted) Object :: Destroy (aborted);   }

Hydrographic_Object :: Hydrographic_Object( char* NameRegion) : (NameRegion) {
                        directType ( getDirectType( ) );
// Examples of how to initialise a Reference to an object

priv_catchment.Init( new List( (Type*) OC_lookup ("Catchment") ), this );
priv_genstations.Init( new List( (Type*) OC_lookup ("Generating_Stations") ), this );
priv_weatherstations.Init(new List( (Type*)OC_lookup("Weather_Stations") ), this );     };
void Hydrographic_Object :: putObject ( OC_Boolean deallocate ) {
// Saves structure of the Hydrographic_Object lists and arrays
( ( List* ) priv_genstations.Binding(this) ) -> putObject (FALSE);
( ( List* ) priv_weatherstations.Binding(this) ) -> putObject (FALSE);
( ( List* ) priv_catchment.Binding(this) ) -> putObject (FALSE);
// Saves Hydrographic_Object itself
```

```
        Object :: putObject ( deallocate );          }
void Hydrographic_Object :: deleteObject ( OC_Boolean deallocate ) {
// Deletes the list objects only, not contents of the lists
( ( List* ) priv_genstations.Binding(this) ) -> deleteObject (deallocate);
( ( List* ) priv_weatherstations.Binding(this) ) -> deleteObject (deallocate);
( ( List* ) priv_catchment.Binding(this) ) -> deleteObject (deallocate);
// Delete the Hydrographic_Object list itself
        Object :: deleteObject ( deallocate );          }

Hydrographic_Object :: void zoomRegion(nameofregion)  {
//      display the name of the region on the STUDIO window     }

// The Hydrographic_Object also contains other procedures for coordinating activities which in this
// case is not further elaborated.

HydroIterator :: HydroIterator ( Hydrographic_Object* theHydrographic_Object ) {
        priv_hydro = theHydrography;
        priv_iterator = (AggregateIterator*) NULL;
        // Iterator constructors usually call Reset
        Reset( );               }

HydroIterator :: ~HydroIterator ( ) {
        if (priv_iterator) delete priv_iterator;        }
void HydroIterator :: Reset( ) {
//      This resets the iterator after use }
OC_Boolean.HydroIterator :: moreData( ) {
//      This retrieves more elements from the List   }
//      Implementation for Type Lakes
//      Lakes, an ordered dictionary keyed by string.

Lakes :: Lakes ( char* Lakename, int normal_level, int min_level, int surface_area ) {
sscanf ( "%s / %d / %d / %d  ", &Lakename, normal_level, min_level, surface_area );
        priv_lakename = Lakename;
        priv_normal_level = normal_level;
        priv_min_level = min_level;
        priv_surface_area = surface_area;
priv_lakename.Init (new Dictionary (OC_String  (Type*) OC_lookup ("Lakes", WriteLock), IsOrdered ))
; }
```

```
// Activation constructor
Lakes :: Lakes (APL* theAPL ) : (theAPL) { }
Lakes :: ~Lakes ( ) {  Destroy( FALSE );  }
void Lakes :: Destroy ( OC_Boolean aborted ) {
// If the Lake object is already in memory
if ( priv_lakename.isActive(this))  {
        Entity* lke = ( Entity* )priv_lakename.Binding(this);
        //  Delete it
                delete lke;
        // Don't forget to call the base Destroy
                if  (aborted) Object :: Destroy (aborted);   }

Type*  Lakes :: getDirectType( ) {
                return (Type*) OC_lookup ("Lakes");  }
void Lakes :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_lakename.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);          }
void Lakes :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_lakename.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);                  }

int Lakes :: comp_normal_lake_level(arguments) {
//      The system would compare water level data coming from
//      Type Measuring_Point with the set value.  If it is lower than the set
//      value, it would call extrapolate_graph function and sound an alarm
//      in the user interface object  }
void Lakes :: extrapolate_graph(arguments) {
//      Calls a graph plotting function to extrapolate a graph on the STUDIO
//      window.  Display a warning sign if the number of days before lakes level
//      reach their minimum drops to less than 1 month.  This function can be
//      called by the user or when the minimum water level is reached.            }
void Lakes :: display_lake( ) {
//      Display the picture of Lakes on the STUDIO window          }

// Implementation for Type Catchment
// Catchment, an ordered dictionary keyed by string.
Catchment :: Catchment ( char* Catchmentname) {
```

```
sscanf ( "%s ", &Catchmentname );
        priv_catchmentname = Catchmentname;
        priv_catchmentname.Init(new Dictionary (OC_String(Type*) OC_lookup    ("Catchment"),
IsOrdered ) );                  }

// Activation constructor
Catchment :: Catchment (APL* theAPL ) : (theAPL) { }
Catchment :: ~Catchment ( ) {  Destroy( FALSE );  }
void Catchment :: Destroy ( OC_Boolean aborted ) {
// If the Catchment object is already in memory
if ( priv_catchmentname.isActive(this))  {
        Entity* catch = ( Entity* )priv_catchmentname.Binding(this);
        //  Delete it
                delete catch;
        // Don't forget to call the base Destroy
                if  (aborted) Object :: Destroy (aborted);   }

Type*  Catchment :: getDirectType( ) {
                return (Type*) OC_lookup ("Catchment");  }
void Catchment :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_catchmentname.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);        }
void Catchment :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_catchmentname.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);                  }

Catchment :: void %catchment( ) {
//      Calculate the percentage rise in water level when 1 mm of rain or snow falls on
//      the catchment area.                }
Catchment :: void average_delay( ) {
//      The function average_delay calculates the time taken for the water level in the
//      catchment area to rise.  This function must also access the temperature field
//      of the object type Weather_Stations to calculate the amount of released water
//      from the melting of snow       }

// Implementation for Type Rivers
// Rivers, an ordered dictionary keyed by string.
Rivers ::  Rivers ( char* river, int length ) {
```

```
sscanf ( "%s  / %d ", river, length );
        priv_rivername = river;
        priv_length = length;
        priv_rivername.Init(new Dictionary (OC_String(Type*) OC_lookup("Rivers"),  IsOrdered ) ) ;
                                }
// Activation constructor
Rivers :: Rivers(APL* theAPL ) : (theAPL) { }
Rivers :: ~Rivers( ) {  Destroy( FALSE );  }
void Rivers :: Destroy ( OC_Boolean aborted ) {
// If the Rivers object is already in memory
if ( priv_rivername.isActive(this))  {
        Entity* river = ( Entity* )priv_rivername.Binding(this);
        //  Delete it
                delete river;
        // Don't forget to call the base Destroy
                if  (aborted) Object :: Destroy (aborted);   }

Type*  Rivers :: getDirectType( ) {
                return (Type*) OC_lookup ("Rivers");  }
void Rivers :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_rivername.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);          }
void Rivers :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_rivername.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);                    }

//  Implementation for Type Measuring_Points
//  Measuring_Points, an ordered dictionary keyed by string.
Measuring_Points :: Measuring_Points ( char* mp) {
sscanf ( "%s ", mp );
        priv_mp = mp;
        priv_flow_rate = flow_rate;
        priv_mp.Init(newDictionary(OC_String(Type*)OC_lookup("Measuring_Point),      IsOrdered ) ) ;
}
// Activation constructor
Measuring_Points :: Measuring_Points (APL* theAPL ) : (theAPL) { }
Measuring_Points :: ~Measuring_Points ( ) {  Destroy( FALSE );  }
void Measuring_Points :: Destroy ( OC_Boolean aborted ) {
```

```
// If the Measuring_Points object is already in memory
if ( priv_mp.isActive(this)) {
        Entity* measuringpoint = ( Entity* )priv_mp.Binding(this);
        // Delete it
                delete measuringpoint;
        // Don't forget to call the base Destroy
                if (aborted) Object :: Destroy (aborted); }

Type* Measuring_Points :: getDirectType( ) {
                return (Type*) OC_lookup ("Measuring_Points"); }
void Measuring_Points :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_mp.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);       }
void Measuring_Points :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_mp.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);                 }

void Measuring_Points :: cal_flow_rate( ) {
//      Cal_flow_rate gets a value from remote flow sensor on the river. Data logging is done
periodically by the systems clock and the daily flow rate is entered into the persistent store.  }

void Measuring_Points :: aggregate_flow( ) {
//      aggregate_flow sums up all the flow rate of the rivers.  This  value is then
//       send to the Type Rivers  }

void Measuring_Points :: lakelevel( ) {
//      Sends the water level of the lake to the Type Lakes  }

// Implementation for Type Generating_Stations
// Generating_Stations, an ordered dictionary keyed by string.
Generating_Stations :: Generating_Stations ( char* gen) {
sscanf ( "%s / %d", gen, capacity );
        priv_gen_name = gen;
        priv_capacity = capacity;
        priv_gen_name.Init(newDictionary(OC_String(Type*)OC_lookup
("Generating_Stations"), IsOrdered ) ) ;                                    }
```

```
// Activation constructor
Generating_Stations :: Generating_Stations(APL* theAPL ) : (theAPL) { }
Generating_Stations :: ~Generating_Stations( ) {  Destroy( FALSE );  }
void Generating_Stations :: Destroy ( OC_Boolean aborted ) {
// If the Generating_Stations object is already in memory
if ( priv_gen_name.isActive(this))  {
        Entity* generatingstation = ( Entity* )priv_gen_name.Binding(this);
        //  Delete it
                delete  generatingstation;
        // Don't forget to call the base Destroy
                if  (aborted) Object :: Destroy (aborted);   }
Type*  Generating_Stations :: getDirectType( ) {
                return (Type*) OC_lookup ("Generating_Stations");  }
void Generating_Stations :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_gen_name.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);          }
void Generating_Stations :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_gen_name.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);                    }

float Generating_Stations :: %generating_capacity(argument) {
        // calculates the river flow to generate the required amount of Mega-Watt    }

// Implementation for Type Weather_Stations
// Weather_Stations, an ordered dictionary keyed by string.
Weather_Stations :: Weather_Stations ( char* wea) {
sscanf ( "%s  / %d / %d / %d /%s", wea, rainfall, snowfall, temp, forcast );
        priv_wea_name = wea;
        priv_rainfall = rainfall;
        priv_snowfall = snowfall;
        priv_temp = temp;
        priv_forcast = forcast;
        priv_wea_name.Init(newDictionary(OC_String(Type*)OC_lookup
("Weather_Stations"),  IsOrdered ) ) ;                                                  }

// Activation constructor
Weather_Stations :: Weather_Stations(APL* theAPL ) : (theAPL) { }
Weather_Stations :: ~Weather_Stations( ) {  Destroy( FALSE );  }
```

```
void Weather_Stations :: Destroy ( OC_Boolean aborted ) {
// If the Weather_Stations object is already in memory
if ( priv_wea_name.isActive(this)) {
        Entity* weatherstation = ( Entity* )priv_wea_name.Binding(this);
        // Delete it
                delete weatherstation;
        // Don't forget to call the base Destroy
                if (aborted) Object :: Destroy (aborted); }

Type* Weather_Stations :: getDirectType( ) {
                return (Type*) OC_lookup ("Weather_Stations"); }
void Weather_Stations :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_wea_name.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);       }
void Weather_Points :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_wea_name.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);                 }

void display_rainfall( ) {
        // show on the STUDIO window, the daily rainfall in quadtree graphics  }
void display_snowfall( ) {
        // show on the STUDIO window, the snowfall in quadtree graphics    }
void display_temperature( ) {
        // show on the STUDIO window, the snowfall in quadtree graphics    }
void weather_forcast( ) {
        // display the latest information on the weather for the next 24 hours  }
// Implementation for Type City
// City, an ordered dictionary keyed by string.
City :: City ( char* city) {
sscanf ( "%s ", city );
        priv_cityname = city;
        priv_cityname.Init(newDictionary(OC_String(Type*)OC_lookup("City"), IsOrdered ) ) ; }

// Activation constructor
City :: City(APL* theAPL ) : (theAPL) { }
City :: ~City( ) { Destroy( FALSE ); }
void City :: Destroy ( OC_Boolean aborted ) {
// If the City object is already in memory
```

```
if ( priv_cityname.isActive(this)) {
        Entity* city_nam = ( Entity* )priv_cityname.Binding(this);
        // Delete it
                delete city_nam;
        // Don't forget to call the base Destroy
                if (aborted) Object :: Destroy (aborted); }

Type* City :: getDirectType( ) {
                return (Type*) OC_lookup ("City"); }
void City :: putObject ( OC_Boolean deallocate) {
                ( ( Object* )priv_cityname.Binding(this) ) -> putObject (FALSE);
                Object :: putObject (deallocate);       }
void City :: deleteObject ( OC_Boolean deallocate ) {
                ( ( Object* )priv_cityname.Binding(this) ) -> deleteObject(FALSE);
                Object :: deleteObject(deallocate);             }
void City :: display_city( ) {
//      Display a map of city on the STUDIO window          }
```

Configuring the database

Firstly, an empty database called Hydrography was created. This could be done using DBATool. The Ontos kernel is the area supporting the schema of the application. It might also be shared with other different database applications. The OntosSchema is in the Unix directory, /usr/local/ONTOS/ONTOS DB/db.

Let's say, all Ontos applications would be put in the subdirectory, OODB already created. A copy of the Ontos Schema is needed and was renamed as HydroKern to be registered with the database Hydro in the subdirectory OODB. DBATool was then invoked and the prompt will change. Hydro was registered with HydroKern, which held the logical name of the kernel area. Type syn and then quit from DBATool. Once the above steps were done, the database Hydro could now be accessed.

```
> cd /home/ms-suna/OODB
> cp  /usr/local/ONTOS/ONTOSDB/db/OntosSchema  myKernel
> DBATool
>> register kernel HydroKern on ms-suna at /home/
        ms-suna/OODB/myKernel
>> register database Hydro with kernel HydroKern
>> syn
>> quit
>
```

Implementing the application

The schema was coded in the header file named "Hydrography.h". The definition of all methods of all the different classes was done in Hydrography.ctl file, which would direct classify to provide an explicit list of all other objects.

To classify the Hydrography schema in a UNIX environment, execute the command :

```
> classify  +cHydrography.ctl  +DHydrography_db
  \+dHydrographyDirectory  -I/usr/local/ONTOS/h  Hydrography.h
```

The +c switch identifies the control file; the +D switch identifies the database; the +d switch tells classify to create a name directory called HydrographyDirectory and to record the names of schema objects there; the -I switch locates the operating system directory containing ONTOS DB class definition files that were included by Hydrography.h.

All methods of the application were contained in the control file, Hydrography.ctl. To compile this file with cplus utility, type :

> cplus -c -g -I/usr/local/ONTOS/h Hydrography.ctl

As for the spatial characteristics of the application, they are displayed using STUDIO. Basically, graphics or maps might be electronically scanned and stored as a bitmap file. It could then be stored as persistent objects using the OC_XBitmap utility. These codes could be written in the control file :

```
OC_XBitmap *newXBitmap(char *bitmapfileName)  {
        OC_XBitmap *aBitmap = new OC_XBitmap( );
        aBitmap->construct(bitmapfilename);
        aBitmap->putObject( );
        return aBitmap;
}
```

In this case study example, several samples of the DBDesigner forms of object type Lakes and several output maps of STUDIO windows were illustrated.

Querying the database

Ordinary SQL is not sufficient to query the database[37]. Visual query language is needed as an alternative to query spatial attributes. This matter would not be examined further because it is beyond the scope of this exercise.

To access the database, the application opened the database and started a transaction. The intial transaction might involve populating the database with sample data. This piece of code was defined in the main.c file of the database. It would then be possible to access the objects in the database, made changes to them, and wrote them back. When the application had completed a consistent piece of work, it would commit the transaction, and actually caused the database to be updated with the changes it had

made. Alternatively, it might decide to back out of the changes by aborting the transaction. The general format of the main.c file is :

```
void main( ) {
        OC_open ("databasename");
        OC_transactionStart( );

    // do your transaction, i.e. add, delete, modify here
            if ( doSomething( ) )
                OC_transactionCommit( );
            else
                OC_transactionAbort( );
        // do your query here to verify the results
        OC_startQuerySession( );
        ExceptionHandler sql_handler ("SQLProblem");
        QueryIterator* myQIter;
            if (sql_handler.doesNotOccur( ) ) {
                // do your OSQL coding here     }
            else {
                // do your coding to handle OSQL query error    }
        delete myQIter;
        OC_endQuerySession( );
        OC_close( )          // for database
```

Ontos provides Object SQL to make queries over the database. A QueryIterator must also be set up to process all programmatic queries, which is the required context for the QueryIterator. The session begins with OC_startQuerySession() and ends with OC_endQuerySession( ). The session should always include an exception handler. The QueryIterator is deleted at the end of the session.

The main.c program

The main.c program should perform two tasks. One was to populate instances into the database and the other was to perform some queries to verify the results. The Ontos header files were enclosed in angle brackets whereas the one containing the schema of the Hydrography was in inverted commas. It was assumed earlier that the database had been created using the DBATool.

Transaction processing could only be done when the database was opened. The command OC_open connects the client application to the server of the database and the opens the database for access. As mentioned earlier, when the lake level fell to its minimum value, it would trigger an alarm. Our methodology earlier on recommends that the objects that had the potential to trigger itself and other

objects should have synchronisation mechanisms built into them. There could arise a situation when the type Lakes would trigger when a new instance is writting to it. To safeguard this, locking was applied to this type by specifying WriteLock as one of the arguments of OC_lookup. This was shown in the control file.

OC_transactionStart(XAConflictResponse = OC_waitOnConflict, BFP buffering = OC_defaultBuffering) was chosen as one of the available concurrency control protocol available in Ontos to handle conflict situation. Transaction would wait until the lock was relinguished by the locking process to complete the database operation. The buffering was to optimise the overhead incurred between the link communications between the client and the server. The OC_transactionCommit (OC_cacheDisposition = OC_cleancache) would delete all the objects activated during the transaction in the client cache once they were stored in the server. The last command OC_close closed the database and cleaned up the client cache completely in preparation for opening up another database. This command also killed all servers started by this client and the server cache was deallocated only if there are no other clients using the server.

The Exception Handler object handled all the failures during transaction and query processing. The comments associated with each step was fairly descriptive.

In the transaction part, aLake was a template of type Lake. It was used for the instantiation of values of Lakes and stored as persistent objects in the database.

In the query part, a Query Iterator was set up, myQIter. In this simple OSQL example, the surface area of lakes smaller than 50 square miles were retrieved. There were only two qualifying instances as shown, Tamaki and Awatea. The heading was copied into the title string using the member function in the Query Iterator object, yieldHeaderString. This member function would accept as its argument the title string and the size of the buffer that holds this title string. The title string is then displayed using the cout function specified in stream.h file. Next, all qualifying values were retrieved using the function yieldRowString and displayed. This function placed the results in buf and the size of buf is specified in maxlength.

```
// main program for Hydrography database
#include < Database.h >
#include < Directory.h >
#include < Exception.h >
#include < Object.h >
#include < QueryIterator.h >
#include < Type.h >
#include < stream.h >
#include "Hydrography.h"
```

```
main( int argc, char** argv ) {
        if (argc != 2 ) {
                printf ( "Usage : %s  <database name > \n", argv[0] );
                exit(1);                        }
// dbName is the first argument
        char* dbName = argv[1];
// Open the specified database
        if (! OC_open( dbName ) ) {
// Issued error message if unsuccessful
        printf ( " Could not open %s \n" dbName );
        exit(1);                        }

{       OC_Boolean shouldCommit;
        ExceptionHandler itFailed ( "Failure" );
// Trap all failures
if (itFailed.doesnotOccur( ) )  {
// Start a transaction
        OC_transactionStart( XAConflictResponse = OC_waitOnConflict,
        BFP buffering = OC_defaultBuffering) );

// Do application processing.  Populate with some instance of Type Lakes
Lakes *aLake;
// Create instances of Lake and save them in the database
        Lakes  aLake("Waitea", 50, 10, 60, 12:00);
        aLake->putObject;
        Lakes  aLake("Avalon", 90, 20, 100, 12:00);
        aLake->putObject;
        Lakes  aLake("Tamaki", 58, 22, 30, 12:00);
        aLake->putObject;
        Lakes  aLake("Awatea", 33, 10, 45, 12:00);
        aLake->putObject;

// Commit a transaction
        OC_transaction Commit( OC_cacheDisposition = OC_cleancache );   }
else  {
// Handle any failures, prevent recursive failures
        itFailed.removeFromStack ( );
        printf ( "\n \n \t !! Failure : %s : %s \n ",
```

```
        itFailed.handled_exception -> failureName ( );
        itFailed.handled_exception -> message ( )  );
// Rollback any changes
        OC_transaction Abort ( OC_doNothing );                }}
// do your query here to verify the results
                OC_startQuerySession( );
                ExceptionHandler sql_handler ("SQLProblem");
                char Titles[128];
                char buf[80];
                int maxlength = 128;
                QueryIterator* myQIter;
                        if (sql_handler.doesNotOccur( ) )  // handle all OSQL exceptions
{       // create the QueryIterator
                        myQIter = new QueryIterator("select name, priv_normal_level,
        priv_min_level from Lakes where priv_surface_area < 50 ");

        strcpy( Titles, " NameofLake, Norm Level, Min Level");
        myQIter->yieldHeaderString(Titles, maxlength);
        cout << Titles  << "\n";

        while (myQIter->moreData( ) )    // retrieve all values if not exhausted
        {       myQIter->yieldRowString( buf, maxlength );
        cout << buf << "\n";
        // get the results as one character string of text and print them
                        else {
        itFailed.handled_exception -> failureName ( );
        itFailed.handled_exception -> message ( )  );          }
                delete myQIter;
                OC_endQuerySession( );
        OC_close( dbName, cleanAll= FALSE );
        exit(0);
}
```

Appendix E : Future Directions of OODBMS

OODBMS is still a very new area. Until now, it is not known if any organisation in New Zealand uses OODBMS although in United States, they are highly marketed. The current research area into OODBMS may broadly be classified into :

(1) Formalisation of Object-Oriented Methodology
(2) Query Processing
(3) Architectures for Object Data Management
(4) Rule Management
(5) Distributed Extended RDBMS and OODBMS
(6) Visual Query Language
(7) Concurrency Control Techniques
(8) Indexing and Clustering
(9) Crash Recovery
(10) Access Methods

The above topics would be briefly discussed, the problems these research would address and the institutions involved.

E.1 Formalisation of Object-Oriented Methodology

As mentioned earlier in chapter 3, there is yet a universally accepted object-oriented data model. The existing OODBMS, including extended relational databases, places a lot of emphasis on the usage and programming part. Little has been described on the Analysis and Design. Our proposed methodology in chapter 5 attempts to address this issue.

What can be done to strengthen the connection between theory and system builders in object-oriented development? As OODBMS theory is relatively new, support will increase only if more object database theory is seen to impact actual systems. For example, tool builders would like the developers of new database design algorithms to examine how the algorithm will behave if users should supply erroneous or incomplete input.

There are also numerous research effort in this area coming from universities like Stanford and Texas and also software houses like Object Design Inc, Data Integration Inc, and the MITRE Corp.

E.2 Query Processing

The decreasing cost of computing hardware makes it economically viable to reduce the response time of today's commercial database systems by using parallel execution. Parallel execution is one source of high performance. This is done by optimising Select-Project-Join (SPJ) queries for parallel execution. The need to reduce response time is evident in decision support applications in which human beings pose complex queries and demand interactive responses. For example, a system for stock portfolio

managers is capable of running a query at the click of a button and getting a graphical results of many categories of stocks. Achieving a reasonable response time is vital. In some cases, it is so important that the response time is at most two seconds.

Many query languages for object-oriented database management systems have been proposed in the last few years. Currently, a serious problem exists in their optimisation. The implementation of a query optimiser is still an awkward and delicate operation. The essence of a query optimisation is to find an execution plan that minimises a cost function. An optimisation process traditionally involves two deeply connected levels that are qualified as logical and physical. The logical level uses the semantic properties of the language in order to find expressions equivalent to the one given by the programmer (query rewriting done by the DBMS). The physical level uses a cost model based on system informations to choose the best algorithm for the evaluation of a given expression. Since a lot of knowledge regarding this area has been derived from the relational database systems, much effort has been done to adapt the techniques developed for the RDBMS to the OODBMS environment. However, the characteristics of the object-oriented environment had lead to consider entirely new techniques. Current investigation is to integrate in a common simple framework the different techniques that have been proposed to support the object-oriented model. XSQL is one such proposal put forward by Michael Kifer, Won Kim, and Yehoshua Sagiv.

E.3 Architectures for Object Data Management

An OODBMS provides persistence and transaction management for objects. Inter-object references are typically represented by 32-bit pointers in programming language implementations. However, this is too restrictive for database systems. For this reason, OODBMS typically use objects oids of 64 to 96 bits. The choice of an architecture is highly relevant to the users, as the selected architecture influences the performance characteristics, ease of use in the application development, the size of an application's executable, and both source and compatibility of existing libraries. This area of research examines the problem of representing inter-object references using a greater number of addressing bits.

E.4 Rule Management

Designers of traditional database management systems have long wanted to transform databases from passive repositories for data into active systems that can respond immediately to a change in state of the data, an event, or a transition between states. Many problems need to look into such as :

(1) Design of a suitable language for expressing active rules

(2) Design of a condition-testing mechanism for rules that is efficient enough to keep track of fast transaction processing

(3) integration of rule condition testing and execution with the transaction processing system

Besides Postgres, there are also other efforts to provide extensions to relational database. Examples are Ingres and Ariel. The Ariel system is an implementation of a relational DBMS with a built-in rule system developed by the Database Systems Research and Development Center at the University of Florida.

E.5 Distributed Extended RDBMS and OODBMS

There have been growing research efforts in the area of parallel database systems during the past few years. XPRS is a multi-user parallel database management system that is currently under development at the University of California, Berkeley based on Postgres. It is implemented on a shared-memory multiprocessor and a disk array.

Objectivity/DB has delivered to production use a fully distributed OODBMS architecture, where full distribution includes distribution of data and control. Distribution of data allows object to reside and execute anywhere on the network, and to be used anywhere on the network, transparently, by users and applications that need not be aware of the object's actual location and without any server bottlenecks.

E.6 Visual Query Language

Besides be able to query the OODBMS by object SQL, there is current research on a new visual and declarative language. An example is DOODLE (Draw an Object-Oriented Database LanguagE). The main principle behind the language is that it is possible to display and query the database with arbitrary pictures. The users are allowed to tailor the display of the data to suit the application at hand or their preferences. The user-defined visualisations are to be stored in the database, and the language to express all kinds of visual manipulations. The semantics of the language is given by a deductive query language for object-oriented databases.

SNAP is a graphics-based Schema Manager developed at the University of Southern California where visual query language is supported.

E.7 Concurrency Control Techniques

Commercial OODBMS generally claim to have very fast data access. Some benchmarks have shown that OODBMS perform very well in certain single-user application areas. But many commercial database applications have challenging concurrent, multi-user performance requirements. Traditional concurrency control protocols can slow down the overall system performance of an OODBMS to the point where it has no real performance advantages over non-OODBMS products. New concurrency control protocol need to be develop which are appropriate to justify performance goals. Ontos Inc. is carrying out research in this area.

E.8 Indexing and Clustering

The support of the superclass-subclass concept in OODBMS makes an instance of a subclass also an instance of its superclass. As a result, the access scope of a query against a class in general includes

the access scope of all its subclasses. To support the hierarchical relationship efficiently, the index must achieve two objectives. Firstly, the index must support efficient retrieval of instances from a single class. Secondly, it must also support efficient retrieval of instances from classes in a hierarchy of classes. A new index called H-trees has been proposed at the National University of Singapore to support retrieval for the relationship. A performance analysis is conducted and both experimental and analytical results indicate that H-tree is an efficient indexing structure for OODBMS.

In recent years a number of clustering algorithms for OODBMS have appeared to improve the performance by placing on the same page related sets of objects, thus facilitating fast access. More of such clustering algorithms are under exploration from the Dept. of Computer Science, University of Wisconsin, Madison.

E.9 Crash Recovery

Networks of powerful workstations and servers have become the computing environment of choice in many applications. As a result, most recent commercial and experimental DBMS have been constructed to run in such environments e.g. Sybase. These systems are referred to as client-server DBMS. Recovery has long been studied in centralised and decentralised database systems, and more recently in architectures such as shared-disk systems and distributed transaction facilities. However, little has been published about recovery issues for client-server database systems especially object-oriented ones. Research is carried out by the Dept. of Computer Science, University of Wisconsin, Madison in this area using EXODUS.

E.10 Access Methods

One of the requirements for the object database is the ability to handle spatial data. Spatial data arise in many applications, including cartography, CAD, computer vision and robotics. The University of Maryland is currently researching into the use of parallelism to accelerate the performance of spatial data access methods. To do this, the hardware architecture would need to be re-arranged and the physical data structure stored in the form of multiplexed R-tree. A number of forms of R-tree are created and their performance studied in terms of response time and load balancing.

Bibliography

Books

1. Avison D.E., Fitzgerald G. *Information Systems Development* : Methodologies, Techniques and Tools. Blackwell Scientific Publications 1988.

2. Bancilhon, Francois & Delobel, Claude & Kanellakis, Paris. *Building an Object-Oriented Database System : The Story of O2*. Morgan Kaufmann Publishers 1992.

3. Batezael, Daniel. *Du Pont RIPP Presentation*. Information Engineering Associates 1988.

4. Booch, Grady. *Object-Oriented Design with Applications*. Benjamin/Cummings 1990.

5. Budd, Timothy. *An Introduction to Object-Oriented Programming*. Addison-Wesley 1991.

6. Cattell, Rick G. *Object Data Management*. Addison-Wesley 1991.

7. Coad, Peter & Yourdon, Edward. *Object-Oriented Analysis*. Prentice-Hall 1991.

8. Coad, Peter & Yourdon, Edward. *Object-Oriented Design*. Prentice-Hall 1991.

9. Gilb, Tom. *Principles of Software Engineering Management*. Addison-Wesley 1988.

10. Henderson-Sellers, Brian. *A Book of Object-Oriented Knowledge*. Prentice-Hall 1991.

11. Jackson Michael. *System Development*. Prentice-Hall 1983.

12. Jacobson, Ivar & Christensen, Magnus. *Object-Oriented Software Engineering*. Addison-Wesley 1992.

13. Kaehler, Ted & Patterson, Dave. A Taste of Smalltalk. W.W. Norton & Company 1986.

14. Kim, Won & Lochovsky, Frederick. *Object-Oriented Concepts, Databases, and Applications*. ACM Press 1989.

15. Kim, Won. An Introduction to Object Oriented Databases. MIT Press.

16. Lippman, Stanley. *The C++ Primer.* Addison-Wesley 1992.

17. Martin, James and Palmer, Ian. *An Introduction to Information Engineering Methodology.* James Martin & Company 1991.

18. Martin, James. *Rapid Application Development .* Macmillan 1991.

19. Meyer, Bertrand. *Object-Oriented Software Construction.* Prentice-Hall 1988.

20. Mullin, Mark. *Rapid Prototyping for OO Systems.* Addison-Wesley 1990.

21. Olle, T. William. *Information Systems Methodologies : A framework of understanding.* Addision-Wesley 1992.

22. *Ontos Manuals : Developer's Guide, First-Time Users Guide, OSQL Guide, and Tools & Utilities Guide.* Ontos Incorporated 1991.

23. *Postgres Manual.* University of California, Berkeley.

24. Rumbaugh, Blaha, Premerlani, Eddy, Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall 1991.

25. Silver, Gerald A., Silver, Myrna L. *Systems Analysis and Design.* Addison- Wesley 1989.

26. Skidmore, Steve, Wroe, Brenda. *Introducing Systems Design.* NCC Blackwell 1990.

27. Sommerville, Ian. *Software Engineering.* Addison-Wesley 1989.

28. Stroustrup, Bjarne. *The C++ programming language.* Addison-Wesley 1992.

29. Van Assche, F & Moulin, B & Rolland, C. *Object-Oriented Approach in Information Systems.* North-Holland 1991.

30. Zdonik, S & Maier, D. *Readings in object-oriented database systems.* Morgan Kaufmann(1990).

Journal Articles

31. Bailin, S.C. *An object-oriented requirements specification method.* Comm ACM, Vol 32 No 5, 1989, pp 608-623.

32. Barclay, P. J. & Kennedy, J. B. *Semantic integrity for persistent objects.* Journal of Information and Software Technology. Vol 34 No 8 Aug 1992. pp 533-541.

33. Beynon-Davies, P. *Entity models to object models : object-oriented analysis and database design.* Journal of Information and Software Technology. Vol 34 No 4 Apr 92.pp 225-262.

34. Blaha, Michael & Premerlani, William & Rumbaugh, James. *Relational Database Design using an Object-Oriented Methodology.* Comm of ACM, April 1988, Vol 31 No 4. pp 414-427.

35. Bloom, Toby & Zdonik, Stanley. *Issues in the Design of Object-Oriented Database Programming Languages.* OOPSLA '87. pp 441-450.

36. Dickenson, Holly & Calkins, Hugh. *The economic evaluation of implementing a GIS.* International Journal Geographical Information Systems, 1988 vol 2 no. 4. pp 307-327.

37. Egenhofer, Max. *Why not SQL ?* International Journal Geographical Information Systems. International Journal Geographical Information Systems, 1992 vol 6 no. 2. pp 71-85.

38. Fosdick, Howard. *Ten Steps to AD/Cycle.* Datamation. 1 Dec 1990. pp 59-64.

39. Hayes, Fiona & Coleman, Derek. *Coherent Models for Object-Oriented Analysis.* OOPSLA '91 pp 171-183.

40. Henderson-Sellers, Brian & Edwards, Julian. *The Object-Oriented Systems Lifecycle.* Comm ACM, Vol 33 No 9, 1990, pp 142-159.

41. Jacobson, Ivar. *Object-Oriented Development in an Industrial Environment.* OOPSLA '87 Proceedings pp 183-191.

42. Khoshafian, S. *Insight into object-oriented databases.* Journal of Information and Software Technolgy. Vol 32 No 4 May 1990. pp 274-289.

43. Korson, Tim & McGregor, John. *Understanding Object-Oriented : A Unifying Paradigm.* ACM, Vol 33 No 9, 1990, pp 41-60.

44. Lamb, Landis, Orenstein, Weinreb. *The ObjectStore Database System.* Comm of ACM Oct 1991 Vol 34 No 10. pp 51-63.

45. Lee, The. *Borland's Bridge to OOP.* Datamation, 15 Jan 1992.

46. Lohman, Lindsay, Pirahesh, Schiefer. *Extensions to Starburst : Objects, Types, Functions and Rules.* Comm of ACM Oct 1991 Vol 34 No 10. pp 95-109.

47. McLeod, D. *Perspective on object databases.* Journal of Information and Software Technology. Vol 33 No 1 Feb 1991. pp 13-21.

48. O. Deux et al . *The O2 System.* Comm ACM, Vol 34 No 9, Oct 1991, pp 35-48.

49. Otis, Allen & Stein, Jacob & Butterworth, Paul. *The Gemstone Object Database Management System.* Comm ACM, Vol 34 No 9, Oct 1991, pp 65-77.

50. Ricciuti, Mike. *The Easy Way to OOPS.* Datamation, 1 Feb 1992. pp 24-27.

51. Roberts, S.A. & Gahegan, M & Hogg, J. & Hoyle, B. *Application of object-oriented databases to geographical information systems.* Journal of information and software technology vol 33 no 1 Jan 1991. pp 38-46.

52. Rolland, Colette & Brunet, Joel. *Object Database Design.* Object Management, BCS Data Management Specialist Group, 1992. pp 97-108.

53. Schussel, George. *The Promise and the Reality of AD/Cycle.* Datamation. 15 Sep 1990. pp 69-73.

54. Semich, J. William. *Open CASE Emerges as AD/Cycle Lags.* Datamation. 1 Mar 1992. pp 30-38.

55. Shlaer, Sally & Mellor, Stephen. *An OO Approach to Domain Analysis.* ACM SIGSOFT Software Engineering Notes Vol 14 no 5, Jul 1989, pp 66-77.

56. Stonebraker, Michael & Hanson, Eric & Potamianos, Spyros. *The Postgres Rule Manager.* IEEE transactions on Knowledge and Data Engineering. Vol 14 No 7 July 1988, pp 897-906.

57. Stonebraker, Michael & Kemnitz, Greg. *The Postgres Next-Generation Database Management System.* Comm of ACM Oct 1991, Vol 34 No 10. pp79-92.

58. Stonebraker, Michael & Rowe, Lawrence & Hirohama, Michael. *The Implementation of Postgres.* IEEE transactions on Knowledge and Data Engineering. Vol 2 No 1, Mar 1990, pp 125-141.

59. Stonebraker, Michael. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 2-5 June 1992, San Diego, California. Vol 21, Issue 2.

60. Tagg, Roger. Integrating database systems - is 'object-oriented' the answer? Database Technology 1990 Vol 3 No2-4. pp 47-54.

61. Wirfs-Brock, Rebecca J & Johnson, Ralph E. *Surveying current research in object-oriented design.* Comm of ACM Sep 1990 Vol 33 No 9. pp 104-124.

62. Worboys, Michael & Shaw, Hilary & Maguire, David. *Object-Oriented Modelling for Spatial Database.* International Journal Geographical Information Systems, 1990 vol 4 no 4, pp 369-383.

63. Wybolt, Nicholas. *Experiences with C++ and Object-Oriented Software Development.* ACM SIGSOFT Software Engineering Notes vol 15 no 2, Apr 90 pp 31-39.