CLUSTERING STRATEGIES FOR OBJECT DATABASES

ANN LOUISE MEADS

Doctor of Philosophy

ASTON UNIVERSITY

November 1997

ASTON UNIVERSITY

CLUSTERING STRATEGIES FOR OBJECT DATABASES

ANN LOUISE MEADS

Doctor of Philosophy

November 1997

When object databases arrived on the scene some ten years ago, they provided database capabilities for previously neglected, complex applications, such as CAD, but were burdened with one inherent teething problem, poor performance. Physical database design is one tool that can provide performance improvements and it is the general area of concern for this thesis.

Clustering is one fruitful design technique which can provide improvements in performance. However, clustering in object databases has not been explored in depth and so has not been truly exploited. Further, clustering, although a physical concern, can be determined from the logical model. The object model is richer than previous models, notably the relational model, and so it is anticipated that the opportunities with respect to clustering are greater. This thesis provides a thorough analysis of object clustering strategies with a view to highlighting any links between the object logical and physical model and improving performance. This is achieved by considering all possible types of object logical model construct and the implementation of those constructs in terms of theoretical clustering strategies to produce actual clustering arrangements. This analysis results in a greater understanding of object clustering strategies, aiding designers in the development process and providing some valuable rules of thumb to support the design process.

OBJECT DATABASES; PHYSICAL DATABASE DESIGN; CLUSTERING STRATEGIES; PERFORMANCE

To my parents

Acknowledgement

I would like to express my gratitude to a number of people.

To my supervisor, Paul Golder, for his patience, encouragement and support.

To Dr. Hanifa Shah, who provided much-needed motivation at a very crucial time.

To my family and friends, especially Lea and Lynne, for having faith in my abilities and providing some light relief from studying when required.

And finally, to Ryan, for helping me to keep everything in perspective.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

This thesis is primarily concerned with object-oriented databases (object databases). More specifically, the main focus of this thesis is the performance of object databases. This chapter will explore these facts and present the context and content of the rest of the thesis.

1.1 MOTIVATION

Object databases are a relatively new and exciting phenomena mainly because they offer the potential to model the real world more closely than previous databases, most notably the relational database. Relational databases will continue to be very popular because of their inherent simplicity but object databases based on a consistent model of data from the user to the physical perspective signify the instinctively correct future direction towards seamless systems. It is felt that object databases will play a significant role in future computing.

In my initial readings on the subject, it was clear that the performance of object databases is an issue. It represents a crucial area of research because of the take-up of the product. It would be disappointing to find object databases being under utilised in the future simply because of poor performance when they have so much to offer in other respects. In fact, despite the relational model's simplicity, pre-relational products such as hierarchical and network databases continue to be heavily used in the business community purely because of their transaction processing speed. In addition, it appears to be a significant area of research because relational databases were also burdened with poor performance initially.

A number of areas play a role in the performance of databases; query processing/optimisation, buffer management, physical database design etc. This thesis explores performance in terms of physical database design, in particular storage. Although it does not represent the most obvious choice, from a personal point of view, I am interested in the very "bones" of a system and low level mechanisms. From my initial readings, it became clear that clustering is one physical database design technique which is under exploited. As suggested in the literature, the object model with more semantic expressiveness than previous models, should lead to more opportunities with respect to clustering at the physical level. In particular, the literature points to the main structural elements as pointers to physical database design in terms of clustering. However, there exists a lack of analysis and understanding in this area, specifically the translations required from

logical model structures to physical level clustering of those structures. This thesis therefore is interested in the links between the object logical model and physical model, in particular how the structural elements of the model can provide pointers to physical clustering. This takes the research into the realms of static based clustering as opposed to dynamic based clustering.

1.2 BACKGROUND

An object database is first and foremost a database and databases have been around for about three decades. A database is an organised collection of (logically) related data. Although this is a very general definition, it is frequently cited (Avison, 1997; Clifton, 1978; Elmasri and Navathe, 1989; Loomis, 1983; Oxborrow, 1989). For the purposes of this discussion, it should strictly include the assumptions that the collection of data:

- is generated and maintained by a machine i.e. computer;
- exhibits persistence i.e. the data survives outside the execution of programs (applications and database management modules).

Databases have evolved over three generations, up to the fourth generation today, that of object databases (Loomis, 1990a). The deficiencies of the file system approach (namely inconsistencies and difficult management of redundant data) provided the impetus for the introduction of databases, specifically hierarchical and network databases. The concept of a database management system (DBMS) was introduced as a means of allowing the collection of data to be shared among a number of users. A DBMS is the large piece of software that manages the database. The database and the software together are called a database system (Elmasri and Navathe, 1989) as shown in figure 1.1. The physical database includes not only the actual data content but also a system catalogue or dictionary which contains information about the data (meta-data). The DBMS provided by second generation products presented users with an abstract view of the database. It allowed users to work in terms of a given data model. A data model consists of inter-related elements with constraints on them (Kim, 1990a). For example, users of a network database consider their data to be arranged in a network (an extension of a hierarchy). A database is described by a database schema using the elements of a particular data model. The database schema (the intension) is a static description of the database whereas the actual physical database content (the extension) may change. The physical data at any particular point in time is known as an instance, occurrence or state of the database. The DBMS in network and hierarchical databases maps between accesses according to the particular data model and accesses to the physically stored database. However, this generation did not

possess what is today a notable characteristic of databases, physical data independence. Physical data independence is the separation of how a user logically views the data to how it is actually stored.

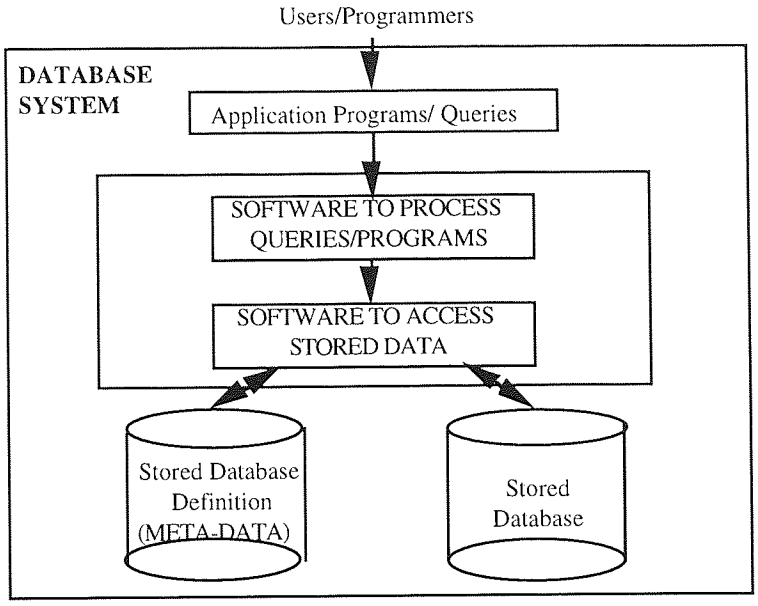


Figure 1.1 A database system *in* Elmasri and Navathe (1989).

The third generation of databases, relational databases, provided physical data independence. Users only have to consider a conceptual representation (i.e. data formatted into tables) of the database without any concern as to how the data is actually stored in the database. Physical storage details (e.g., the file organisation, record lengths and access paths) are of no concern to the user and are hidden by the DBMS which performs a mapping between the two viewpoints.

It is not necessary that the users access the database at the level of abstraction of the chosen logical model. As systems became more sophisticated, users were allowed to view only a subset of the database and/or view the database at a higher level or different level of abstraction (e.g., menus, graphical interfaces). The number of levels of abstraction is irrelevant; the DBMS needs to understand the mapping between the various levels. The DBMS may work with a number of user views which in turn are mapped to a collective conceptual model. The conceptual model therefore represents the total database or the amalgamation of all the user views.

The use of different levels of abstraction became standardised into the ANSI/SPARC (American National Standards Committee on Computers and Information Processing/Standards Planning and Requirements Committee) architecture as shown in figure 1.2 below.

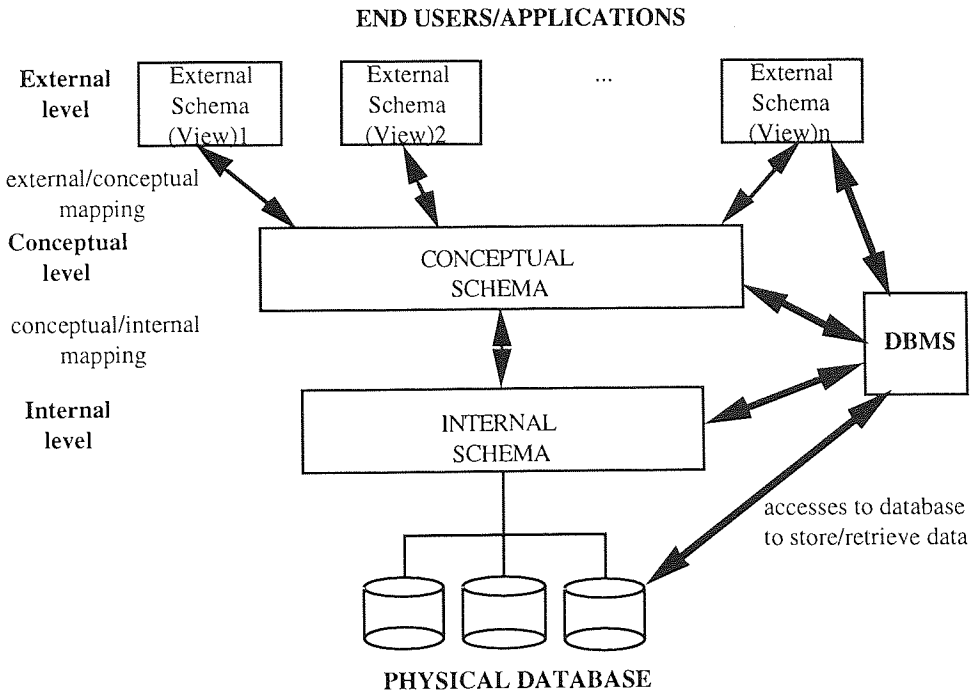


Figure 1.2 The ANSI/SPARC architecture.

The conceptual schema is a logical implementation-independent description of the total data in the database whereas the internal schema specifies physical details. Each external view is a subset of the entire data and is relevant to a particular user sub-group. Applications are also at this level. The diagram also shows that the DBMS is in control, performing the various mappings in order to access the physical database.

Relational databases rapidly became a very popular choice (both for business and for personal use) due to the simplicity with which users are required to perceive their data. However, the relational model's simplicity is also the basis of its deficiencies. With an ever increasing complex world, not all domains can be modelled with the simple constructs that the relational model provides. For some applications, it is desirable to include more semantic meaning in the model of the data. Hence in the mid-1980s, object databases were introduced based on a more powerful logical model.

Over the last decade, object databases have steadily come onto the market as an answer to the above and other problems. Databases were previously considered inappropriate for design applications such as CAD and CAM (computer-aided design and manufacturing). However, object databases provided the means of modelling such domains and their implementation. Indeed, some may argue that this problem provided the impetus for the introduction of object databases.

Although a number of commercial products have been introduced recently, object databases have not been without a typical introductory problem, poor performance. Despite the popularity of the relational model, they too were initially burdened with poor performance. Hence this appears an inevitable obstacle that must be overcome when a new idea is introduced. It is an important area of concern because of its effect on the "take-up" of the product in the market. Despite the powerful modelling capabilities of the object data model it is futile without adequate performance of its products. Hence, performance of object databases is the general area of concern for this thesis.

This thesis will concentrate on one measure of performance, response time, although other measures (main and secondary storage cost and reorganisation cost) are often taken into account alongside time delays. Response time is made up of a number of components as shown in figure 1.3. However, Teorey and Fry (1982) conclude that a database designer can exert significant control over only a single component of response time, input/output (I/O) service time.

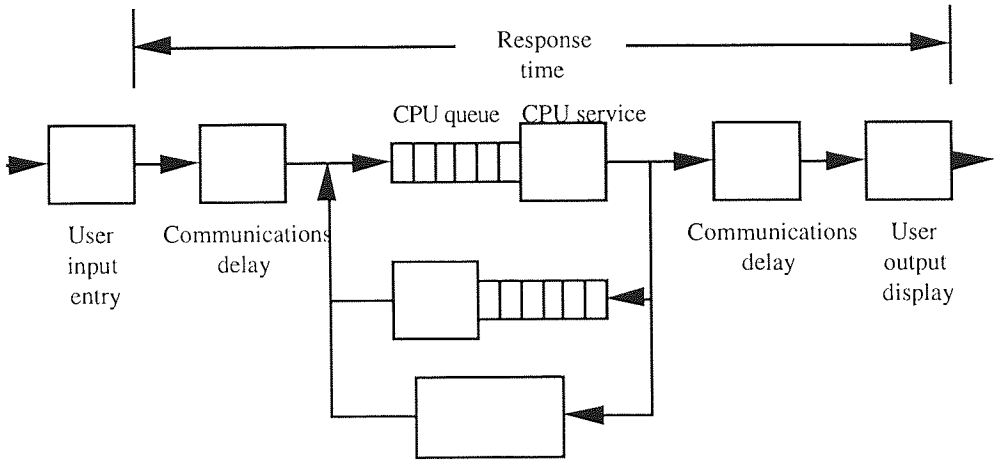


Figure 1.3 Components that make up response time *after* Teorey and Fry (1982).

I/O service time, as used by Teorey and Fry (1982) is the time it takes to perform a number of logical record accesses. This is dependent on the number of disk accesses required to achieve the request. Teorey and Fry (1982) show that disk access time is significant compared to memory access time. Hence, to improve the overall performance of a system, the amount of total disk accesses or their duration should be reduced. Indeed, an objective of many systems is the minimisation of the number disk accesses.

A reduction of the number of physical disk accesses can be achieved by spending time on the physical database design stage ensuring that a good (not necessarily

optimum) physical model is produced. A good arrangement of data on disk and appropriate access paths improve the performance of multiple requests.

Physical database design is concerned with choosing a particular data organisation technique for each file of the database from a variety of:

- file organisations;
- indexes;
- clustering strategies;
- pointer/link chains.

The database designer must choose a particular technique from those available options provided by the specific DBMS. This choice is affected by the nature of the applications the database must serve. In addition, the logical model which the database supports provides a basis for the physical design stage. Although, in theory, the applications that use the database and the chosen logical model should be separate and independent to the physical model, in reality they provide pointers to the implementation of the database. For example, a relational model is usually implemented by making each separate relation a separate physical file.

File organisations have become fairly standard and stable over the four generations of databases. Therefore, the physical model compared with the logical model has changed the least over the years. The logical model, on the other hand, being closer to the users, has changed as needs have changed.

1.3 PERFORMANCE OF OBJECT DATABASES

Clustering is one storage management technique which aims to improve performance by reducing the duration and the amount of accesses to the physical medium. Clustering means intelligently placing data in close proximity to data that is likely to be accessed at the same time. Implicit in this definition is the fact that semantic knowledge about the particular application domain may be used to influence the physical arrangement of data on storage. Although, when modelling the conceptual/logical schema, care is taken to produce a view which is independent of physical concerns, semantic knowledge may be used to influence physical design decisions so as to produce an efficient system. Knowledge should therefore be filtered down the mapping process or along the database development lifecycle. This thesis intends to explore the link.

Optimisation techniques for relational databases are well established and much effort has been focussed on the improvement of various file organisation techniques. However, it is noted that with the simple structures of the relational model, physical

design decisions based on semantic structural knowledge have been sacrificed. In fact, the relational model, with its implicit relationships decided at the logical design stage, reduces physical design decisions. The most popular option when optimising relational databases is simply to add another index. Therefore, process knowledge and not structural knowledge is exploited in the implementation of relational databases. The object data model, on the other hand, provides us with an opportunity to exploit structural semantics. The greater complexity of the object model gives rise to more ways to arrange the data on its physical medium (i.e. more clustering strategies) compared to the relational model. The same reasoning behind the shift of emphasis in systems development from process-oriented to data-oriented techniques and consequently the push for databases can be applied here. Relationships between entities are less open to change than processes and so a data-oriented approach should produce a model which is more resilient. In fact, the area of object clustering strategies has not yet been significantly explored to determine the potential of clustering. It is therefore the purpose of this thesis to explore the different clustering strategies of object databases with a view to improving their performance. Further, in the process it is expected that any links between the semantic knowledge available at the conceptual analysis stage and the decisions made at the physical design stage can be exposed.

It is anticipated that this thesis will lead to an improvement in the ability to make explicit decisions with respect to the physical model, specifically where on storage to place records, tuples or objects. It does not immediately translate to the development of new, radical file structures. The traditional file organisation techniques are well established and will remain for some time. In fact, a database is a database whether it is based on a relational, hierarchical or object data model. Traditional techniques will continue to be employed in object databases as any logical model can be mapped to any physical model. This is not to say that new physical models will not be available to improve the implementation of object databases, but will co-exist with traditional techniques.

1.4 OBJECTIVES

A necessary objective of this thesis is to establish the key features of the object model, through a synthesis of ideas in the literature, in order to dispel the uncertainties and vagueness in the area. Following on from this, it is necessary to provide more understanding on object clustering in general by reviewing the various proposed clustering strategies in the literature. An offshoot of this objective is to

provide a consolidation of the ideas surrounding object clustering strategies, possibly cross-referencing and organising them.

The primary objective of this thesis is to uncover any links between the structural elements of the object logical model and physical model in terms of clustering with a view to improving performance. Through this process, it is intended to provide more information to the physical database designer possibly uncovering some heuristics to aid the design process. It is acknowledged here that real performance does depend on the query flow for a particular system (dynamic based optimisation) and a technique based purely on structure (static based optimisation) can only go some way into arming the designer with information. Further, it is not expected that the research translates to a new clustering strategy but to at least provide a degree of help to the designer.

1.5 METHOD OF RESEARCH

Although there does not exist a single prescribed approach for the research process, there are a number of advocated stages that any research should at least include. Phillips and Pugh (1987) identify four elements of the research process producing a "PhD form"; background theory, focal theory, data theory and contribution. This research has adhered to popular approaches to support some elements of this form. In particular, an extensive literature review was undertaken to provide a professional level of understanding of the field (background theory) and to establish any weaknesses to reveal potential research issues (focal theory).

Dingley (1996) cites the following research methods to support the data theory element of the form:

- laboratory experiment;
- field experiment;
- survey;
- case study;
- forecasting/futures research;
- simulation/role playing;
- subjective/augmentative;
- action research.

Laboratory experiments and their associated field experiments can be instantly dismissed as being valid methods in information systems research (Dingley, 1996). The objective nature of scientific experiments cannot be applied in a soft systems arena.

This research can be placed in the exploratory or testing out arena as opposed to the problem solving arena (Phillips and Pugh, 1987). To some extent, this research is exploratory because the field is not well established and there is still a general vagueness in the area. Surveys, case studies and action research assume the availability of real world systems. These methods would be better suited to a more mature area where the research is more problem solving.

Forecasting and futures research are very specific methods and are not considered applicable to this research.

At first sight, simulation was considered to be a possible method. However, it was decided that a simulation would yield results for a particular environment and would require specific strategies to be decided upon and put in place. It was felt that there did not exist a more general, analytical look at clustering. Therefore, the subjective/augmentative method best describes the approach adopted for this research. This thesis presents a list of logical arguments/premises which lead naturally to a conclusion. It may be likened to a mathematical logic proof in that, if the series of arguments are accepted, the final conclusion can also be accepted. Although the arguments will require a degree of subjectivity, the process is heavily supported with in-depth analyses, extending the existing knowledge in the area and pushing the boundaries of the field.

1.6 OVERVIEW OF THE THESIS

This thesis intends to contribute to the physical design stage for object databases by providing the designer with a better understanding of object clustering strategies and their impact on performance. Prior to considering the specifics of any logical model, chapter two will review the physical model because it is both relevant to any database and the most predictable component of a database. This chapter will also provide us with the basis to understanding how the performance of databases can be improved. Continuing to the less predictable, logical model, many lessons have been gained in the development of relational databases and so chapter three will look specifically at the relational model. The final part of chapter three will bring together physical and logical aspects by taking a look at the implementation and performance aspects of relational databases. The whole of chapter four is devoted to the object data model. Chapter five then considers the differences the object model brings to the physical design of databases and the alternative clustering strategies available. In an attempt to understand how the various structural semantics can be taken on board and implemented as particular clustering strategies, chapter six presents a thorough and exhaustive analysis of object structures.

Chapter seven then goes on to explore how specific object structures of chapter six can be implemented by mapping them to the theoretical clustering strategies discussed in chapter five. Chapter eight draws the conclusions about this thesis.

CHAPTER TWO

THE DATABASE MANAGER AND PHYSICAL MODEL

This chapter will explore the physical model of databases because of its impact on performance, the general area of concern for this thesis. The physical model, as its name suggests is concerned with physical characteristics of a database such as storage media and the arrangement of data on such storage media. Prior to reviewing these characteristics, the first section considers how the physical model fits into the scheme of a database.

2.1 ACCESSING THE PHYSICAL MODEL

A DBMS accepts queries in terms of a particular data model upon which the database is based. These queries must be mapped into low level requests to read data from or write data to the particular storage medium. The DBMS includes a number of components that work together to perform this function and include,

- a storage manager;
- a run time database processor;
- a query processor;
- a Data Definition Language (DDL) compiler.

The DDL compiler translates schema definitions and stores them in the DBMS catalogue. The query processor accepts high level requests and transforms them into a number of low level sub-queries typically in an 'internal' query language. It is the query optimiser that determines the optimum (not necessarily the best) order in which the sub-queries should be processed. The run time database processor carries out the operations generated by the query processor using the services of the storage manager which in turn may rely on the operating system to perform low level I/O operations. Before the storage manager is explored, it is necessary to look at the physical details of storage media and the way that raw data is typically stored on such media.

2.2 STORAGE MEDIA

From our definition of a database, it is clear that storage on some form of computer medium is required. Computer storage media can be classified into an hierarchy and includes primary memory and secondary storage. Primary memory or main store is the highest level of storage and is that which can be directly operated on by the central processing unit or CPU (Elmasri and Navathe, 1989). It is fast but of limited capacity (being relatively expensive) and volatile. On the other hand, secondary storage is comparatively slow (as it cannot be accessed directly) but of

large capacity and non-volatile. In fact, main memory is some five magnitudes faster than secondary storage (Date, 1994) for accessing a particular piece of data. Despite this fact, a database is typically stored on secondary storage because:

- it is often too large to fit into main memory;
- persistence is required.

Further, a database today is normally stored on magnetic disk because magnetic tape only provides sequential processing and optical storage has yet to become popular due to its relative expense. This takes the organisation of databases into the field of files and records for magnetic disk as opposed to the field of data structures for main memory. The following sections therefore explore magnetic disks, files and records.

2.3 THE CHARACTERISTICS OF MAGNETIC DISK

A magnetic disk is a metal platter coated in a magnetisable coating. Data is recorded in concentric tracks containing the same amount of data even though their circumferences differ and possibly divided into hard-coded sectors. The same tracks (radially equidistant) on different recording surfaces form a notional cylinder. The operating system also divides the tracks at formatting time into equal-sized blocks separated by fixed-sized inter-block gaps containing control information. Each block has a hardware address due to disks being random access addressable devices which is a combination of the surface number, the track number (within the surface) and the block number (within the track).

A number of disk platters may be packaged together in a stack of up to 20 disks (disk pack). These disks typically rotate continuously at constant speed on a central spindle and move under a read/write (R/W) head. Disks may have moveable or fixed R/W heads (Wiederhold, 1987). From a performance point of view, the length of time to read/write a block of data from/to disk is a critical factor and this depends upon the speed of rotation of the disk and the speed of movement of any moveable heads.

2.4 BLOCKS

A block, in addition to being the division of a track on a disk, is also typically the fixed unit of data that is transferred between disk and main memory for processing. Typically, a number of records are contained in a single block. This reduces the number of inter-block gaps on the storage device and so increases its storage capacity (Elmasri and Navathe, 1989). Transferring a block as opposed to a single

record aims to reduce the amount of disk accesses because it is anticipated that a request for a particular record will often be followed by a request for a record that is physically close on disk. In particular, blocking significantly reduces the amount of individual transfers of data units when a complete file must be transferred from secondary storage to main memory. Further, transferring fixed sized units as opposed to records that may be variable in length significantly reduces complexity.

To access a block, the hardware address of the block must be supplied to the disk I/O hardware. A block is not transferred to a working area in memory but to an area called a buffer. This is a contiguous, reserved area in memory equal to the size of one block (Elmasri and Navathe, 1989). Records contained in the buffer can then be manipulated and copied to a working area by the CPU. Records are also grouped into a block in a buffer before being written to disk.

2.5 DATABASE BUFFER MANAGEMENT

Typically, there will be more than one buffer for holding blocks of data from disk. This again can reduce the amount of disk accesses required for a number of requests for records. Although buffer management is an operating system concern, a DBMS will typically provide a buffer manager module because the facilities provided by the operating system are considered inadequate (Stonebraker, 1981). For example, System R provides a buffer manager module which interfaces with the index, recovery, and storage manager (Traiger, 1982). Buffer management is concerned with managing a number of buffers, particularly choosing which block to swap out when all buffers are full. When a request is made for page to be read, the buffer manager looks firstly to its set of buffers. If the page is already in the buffer, then no more actions are required. However, when the page is not in the buffer (known as a fault) the policies of the buffer manager must be utilised to decide which buffer is to contain the incoming page. Prior to reading the required page into the selected buffer, the page it contains must be written back to disk. Therefore, when a fault occurs both a disk read and write are required. It is an area of much research due to its effect on the overall performance of the system. A number of buffer management policies exist based on one or both of the following criteria:

- age;
- frequency.

For example, the FIFO (first-in-first-out) algorithm where the oldest page becomes the candidate for replacement is based on time/age whereas the LFU (least frequently used) algorithm where the least popular page (irrespective of the length of time it has been in the buffer) becomes the candidate for replacement is based on

frequency. Alternatively, algorithms such as LRU (least recently used) based on both criteria are superior (Effelsberg and Haerder, 1984).

When repeated faults occur due to competing resources, the situation is known as thrashing. Sacco and Schkolnick (1982) have devised the Hot Set Model to overcome the thrashing problem that arises particularly within database buffer managers.

2.6 THE STORAGE MANAGER

Data stored on disk is organised into files of stored records where a record is a collection of facts about entities (attributes and relationships). This is the view of the database as far as the DBMS is concerned. The DBMS must map requests for records of the conceptual model to the retrieval of records stored in files on disk. The storage manager, at the lowest level of a DBMS, supports operations on the basic storage objects of the DBMS (records) and maintains them in file objects. This allows the DBMS to build on these primitive operations to support higher level storage structures such as indexes. The storage manager is sometimes viewed as the lowest level of a DBMS or as an external component that the DBMS builds on. In either case, there typically exists a module that provides file management operations. In fact, the architecture of a DBMS is often given as consisting of essentially two layers (Wiederhold, 1987):

- a file system layer;
- a database system layer.

The file system layer supports the view that a database is a collection of file objects and operates without concern of the data content of storage objects (i.e. records are merely byte strings and have no internal structure). It is the database system layer that makes use of the file management services to manipulate data based on its content, interpreting a byte string as a collection of fields with some order. The internal structure is used by the database system layer with higher level storage structures such as indexes. The purpose of the file manager as described above is still somewhat removed from low level physical details such as blocks and tracks. Therefore, to be more precise it is best to consider the layer below the DBMS as a storage manager that contains two components:

- the file manager;
- the disk manager.

The file manager supports the DBMS view and in turn views the database as a collection of logical pages, a view supported by the disk manager. The disk

manager views the database "as it really is" (Date, 1994). This architecture is typical because it is desirable to keep device specific information independent of the logical organisation of data into files. Date (1994) discusses the role of the disk and file manager and a summary is provided here.

2.6.1 The File Manager and Disk Manager

The file manager supports the view of the DBMS that the database is a collection of files containing records. Each file is identified by a file name or file identifier and each record is identified by a record identifier (RID) which is unique at least within its stored file but in practice is unique across the whole of the disk. The file manager uses the operations provided by the disk manager to manipulate logical collections of fixed size pages. These collections are known as page sets and one or more files are represented by a single page set with each file name or identifier being unique at least within the containing page set. Each page and each page set have a unique identifier across the disk. The file manager therefore makes requests in terms of logical page numbers. The disk manager is aware of physical addresses and must map logical page numbers into corresponding physical disk addresses to enable it to actually transfer a physical block to and from secondary storage. The disk manager must allocate and de-allocate fixed size pages on demand from the file manager. Pages are allocated from and de-allocated to a page set that contains all the available pages known as the free page set. Initially, all pages belong to the free space page set and are numbered sequentially from one.

There are a number of ways that files can be allocated to disk. In contiguous allocation, consecutive disk blocks are used for a file. In linked allocation, blocks constituting a file are linked by pointers so file blocks are not necessarily physically adjacent. Disadvantages can be attributed to both options; the former technique restricts the growth of the file whereas the latter technique makes reading the whole file a slow process and records that are logically adjacent are not necessarily physically adjacent on disk. A third technique offers a compromise between the above two techniques by allocating disk blocks to a file in groups (of, for example, 64 pages). The individual clusters are then linked by pointers. A cluster will have a fixed number of disk blocks and is alternatively known as a segment or an extent. Yet another option is the use of index blocks that point to the file blocks which may be distributed across the disk (indexed allocation). Finally, it is common to find a combination of the above options (Elmasri and Navathe, 1989).

Extent allocation is more frequently discussed in the literature (Astrahan et al., 1976; Date, 1994; Haberhauer, 1990; Stonebraker, 1981) and so for the purposes

of this thesis this technique is assumed. The disk manager is therefore responsible for maintaining the list of extents for a given page set. As far as the file manager is concerned, any particular page set has a logical collection of extents allocated to it. When another extent is requested the disk manager gets the next available extent in the free page set and maintains the correspondence of extents to page sets using pointers. After the system has been running for some time and extents have been continually allocated and de-allocated from and to the free page set the next available page can be physically anywhere on disk. For this reason, logically adjacent pages are not necessarily physically adjacent on disk. However, the sequence of extents constituting a file when pointers are followed is sometimes referred to as the 'physical' sequence. If extents are linked by pointers the disk manager only needs to know the location of the first page of the page set in order to access its pages. This information is maintained in a special page, page 0. A list of page sets with their corresponding address of the first page in the set is maintained by the disk manager. This is known amongst other things as the 'disk directory'. The file manager maintains the logical order of records within pages by shifting records upon insertion and deletion. Records are typically maintained at the top of the page with all the remaining free space at the bottom. Hence a file is sequentially accessed by following the page header pointers within a page set and the physical order of records within a page. Even if more than one type of record exists in a file, records can be sequentially accessed by skipping any records of a different type during a sequential scan.

Each record must have a unique record identifier at least within the containing page set. A typical implementation is found in System R (Astrahan et al., 1976), INGRES (Stonebraker et al., 1976) and POSTGRES (Stonebraker, 1987) where record identifiers are made up of two parts. The first part contains the page number of the page that contains the record. The second part contains the offset from the foot of the page that identifies some slot. This slot contains a byte offset from the top of the page that identifies the position of the record. This allows records to be moved within a page without requiring a change in its identifier. Record identifiers that do not change are advantageous because they are used elsewhere in, for example, indexes.

From the discussion above, it is clear that when the DBMS requests the retrieval of a specific stored record, a number of modules work together in mapping from logical to physical details. The layering of modules achieves isolation of these mappings along the different stages of access to the raw data. Each module supports a number of operations that the next higher layer need only be interested in. The

disk manager allows the file manager to think only in terms of page input and output (page management) and so supports operations that manipulate at the level of logical pages. The operations that a disk manager typically supports and consequently that the file manager can issue include:

- retrieve page p from page set s;
- replace page p within page set s;
- add a new page to page set s (i.e. acquire an empty page from the free page set);
- remove a page p from page set s (i.e. return page p to the free space page set).

In turn, the file manager allows the DBMS to only be concerned in stored files and records (stored record management) and supports operations that the DBMS can issue which allow manipulation at the stored file level. These include such operations as:

- retrieve a stored record r from stored file f;
- replace stored record r within stored file f;
- add a new stored record to stored file f and return the new record ID, r;
- remove stored record r from stored file f;
- create a new stored file f;
- destroy stored file f.

For the DBMS to request the retrieval of a record therefore, it calls the file manager to retrieve record r from stored file f. The file manager determines on which page the record exists and then calls the disk manager to retrieve the logical page p. The disk manager maps the logical page number of the requested page to a physical address so that it can instruct the transfer of that page into memory (if it is not already there).

The Storage Manager and The Operating System

A DBMS must make use of the operating system for system services but it is not clear which are typically part of the DBMS and which are provided by the operating system. A disk manager is a component of the underlying operating system and provides basic I/O services. File management may also be provided by the operating system as a general purpose function but it has been found to be inadequate for the special purpose of serving a DBMS. Consequently, sometimes a file manager will be provided that has been specially built for and packaged with the DBMS e.g., Oracle and System R (Astrahan et al., 1976). The designers of INGRES later found that using the services of UNIX instead of building a special purpose file manager was a wrong decision (Stonebraker, 1980).

2.7 FILES AND RECORDS

From the above discussion, the DBMS views the data of a database to be logically organised into files of records. The content and type of each record is of interest to the DBMS although it is of no concern to the storage manager. Typically, all records of a file will be of the same type, where a type is a collection of field names with their corresponding data type. Records of a file may be of fixed or variable length. Records will be of variable length if, for example, the records are of different types. Even if all records of a file are of the same type they may still be of variable length due to one or more optional fields, repeating groups or fields of varying size. However, it is typical to choose some mechanism to make records of the same type the same size for these cases. A record contains not only the actual descriptive data but there will also be some control information maintained at the front of the record in a record prefix (Date, 1994). This includes, for example, the identifier of the stored file, the record length (for variable length records), a delete flag (if records are not physically deleted at the time of the logical deletion), and pointers (for chained records).

The allocation of records to blocks may be spanned or unspanned. In unspanned allocation, records cannot cross block boundaries. Hence, there must be an integral number of records within a block. If records are of a fixed size then this number (which is equivalent to block size (B) divided by record size (R)) will also be fixed and is known as the blocking factor (Bfr). Further, the space that is wasted in each block will be fixed and known (B - (Bfr * R)). It is typical to use unspanned allocation for fixed length records because it makes processing somewhat simpler. Spanned allocation allows records to be split across a number of blocks and is typically used with variable length records or where records are relatively large. For blocks containing part of a record, a pointer will be required to the blocks containing the remainder of the corresponding record.

Each file on disk will have a file descriptor. This contains information necessary for accessing individual records. When a block is brought into a memory buffer the file descriptor information is used to search for a particular record. Typical information contained in the file descriptor or header is the disk addresses of the blocks containing the file and the file's record format.

The DBMS uses the primitive operations supported by the file manager to build higher level storage structures. A storage structure describes both the arrangement of data and how it can be accessed and updated. Therefore, at a high level, a

database is seen to be organised into a collection of files and associated with each is a particular file organisation and possibly a number of access methods.

2.7.1 File Organisation

A file organisation specifies how records are distributed on the storage medium and how they are interlinked. The fundamental file organisations are sequential, direct, and indexed files.

Sequential Files

A sequential file is a collection of records with a linear order. The records are organised and stored in this sequential order according to some key (an attribute or combination of attributes that uniquely identifies each record). There exists no reference within a record to any other record and so records can only be read or written in their stored order. Random access is time consuming and updating the file requires reading the old version of file and writing a new updated copy of the entire file using a transaction file (batched updates).

Direct Files

In direct files, the storage address of a record on the hardware device is related directly to the key value of the individual record that uniquely identifies it (i.e. the key allows identification of storage location) and records are not stored in any particular order. Hashing is a popular example of this approach, where the storage address is derived from the key using a calculation. Hashing functions map from a (larger) set of possible keys to a (smaller) set of available storage blocks. Hence, overflow techniques are required when blocks become full. Hashing direct files is usually (if there are not many overflow records) very efficient as the retrieval of a record is typically one disk access for the corresponding block and then an internal search of the block for the record.

Indexed Files

Indexing is a technique that speeds up access to data in a database by maintaining an auxiliary data structure, an index. An index is made up of basically two columns. One column contains the values of a given attribute or combination of attributes. For a single attribute, the second column associates with each value the records that take on that value for that particular attribute. For indexes based on a combination of attributes, the records associated with a particular set of values must take on jointly those values for the given attributes. An index becomes a list of <attribute

value(s), record pointer(s)> pairs. A primary index is based on the key field and so contains only one record pointer for each entry. A secondary index is based on a non-key field and so may contain more than one record pointer per entry. The key values can be hashed or sorted into some order (permitting faster table searching techniques e.g., binary search and key order processing). Accessing records is better than a sequential search of the file because the index will be smaller than the entire file. It is also a more flexible approach than hashing because it provides a level of indirection; keys in terms of choice and structure are independent of storage structure and keys do not necessarily share an algorithmic relationship with locations.

Indexed files may be either indexed non-sequential or indexed sequential. In an indexed, non-sequential file, records can be stored in contiguous or scattered storage space. Therefore, in a sorted index, there needs to be one (key, address) entry for each record in the file and is known as a dense index. With this file organisation, access is via the index and cannot be sequential. As there is no particular organisation of records, they will be blocked as they are received by the system, the corresponding index entries requiring sorting and structuring as necessary. Alternatively, it is possible to group records with similar frequencies so that from the central cylinder outwards/inwards records will be accessed with decreasing frequency. Hence on the whole, the disk head will remain around the central cylinder and will only occasionally have to move to inner or outer cylinders. Further, the index could also be stored on a central track or cylinder where the majority of records will be accessed.

In an indexed sequential file, data records are stored sequentially in order of their access key. This file organisation represents a compromise option in terms of the sequential and direct file design options. Sequential processing and sequential files are good for batch style applications whereas direct access processing and direct access files are good for on-line style applications. If both types of processing are required then index-sequential files are appropriate. This approach also means that only one entry for each data block is required in the index (a non-dense index). This entry provides the maximum or minimum key to be found in that block (block anchors). This results in a smaller index and as a consequence faster retrieval. The search for a record becomes a search of an index for the given key to find the block in which the record is maintained, accessing that block and sequentially searching the block for the record. When blocks become full, records are placed in overflow blocks which may add to the length of a search for a record.

In the same way as an index is built for a data file, an index can be built for a large index. An index to a data file is typically larger than the block or buffer size and so cannot be maintained in memory or searched in a few disk accesses. As most accesses require the use of the index it makes sense to build an index or a number of indexes to it until the topmost index is small enough to fit into memory and therefore more efficient to search. The set of indexes constitute an hierarchy and each index is called a level. The levels are numbered from the topmost index at the root of the hierarchy (level 0) to the index directly on top of the data file (level n, where n is the total number of levels minus one). Any number of levels can be built on top of a data file but Wiederhold (1987) concludes that the need for many levels of indexing is rare.

To reduce seek times during accesses to records, index blocks and the corresponding data blocks can be stored on the same cylinder. This means that the hierarchical index structure is now split along branches and not levels. As far as possible each branch is stored on contiguous single cylinder space. It is possible to reduce the seek time to only the initial access of the top level index if the complete file can fit onto one cylinder.

B-Trees

The B-tree is a particular type of tree-structured index. Since its introduction by Bayer and McCreight (1972), there have been many variations on the original theme and it is the index structure that has become the most popular for structuring the physical database. Most relational databases offer some variation of the B-tree with some not providing any other option (Date, 1994).

The popularity of the B-tree stems from the fact that its associated algorithms for inserting and deleting a key value ensure that the tree remains balanced. Balanced trees are those where the leaf nodes are the same distance (i.e. the same number of levels) away from the root node. Other tree structures become unbalanced over a period of time and as a consequence have unpredictable search path lengths (as accessing a multi-level index requires accessing one node on each level).

Each node in a B-tree contains pointers which act as delimiters of key values. The structure of a node is as in figure 2.1 where P_0 to P_n are pointers to sub-trees and K_0 to K_n are key values which appear in ascending order. Nodes of the original B-tree also contained the addresses of the records of the corresponding key values (Loomis, 1983). A B-tree of order m has m or less pointers in each node (i.e. n <= m-1). Following a middle pointer will lead to values between the value on its left

and the value on its right. Following the leftmost pointer P_0 will lead to values less than K_0 and following the rightmost pointer will lead to values greater than K_{n-1}.

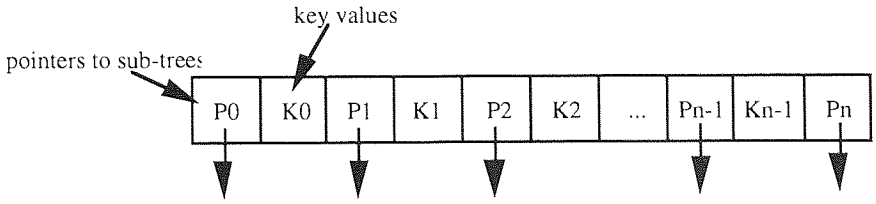


Figure 2.1 The structure of a B-tree node.

For example, consider the partial B-tree of figure 2.2. Following the left most pointer will lead to values less than 60, following the right most pointer will lead to values greater than 93 and following the middle pointer will lead to values between 60 and 93. Note that this principle is applied all the way down the tree. For example, following the pointer *b* takes us to values greater than 36 but also less than 60 because pointer *a* must have been followed to get to node *B*. In addition to the property that the tree remains balanced, other properties include that all nodes are at least half full (between $\frac{1}{2}$m and m pointers) and that the root, unless it is a leaf, has at least two sub-trees (Loomis, 1983).

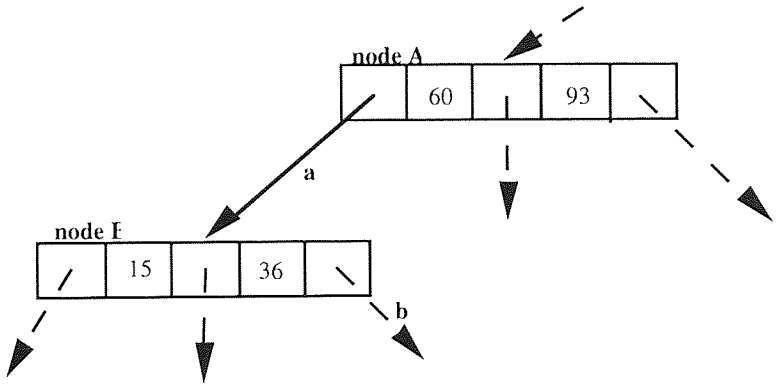


Figure 2.2 An example of part of a B-tree.

The insertion algorithm of the B-tree inserts the value to be added in its appropriate node unless it is full in which case splitting is used. The node is split into two separate nodes and the lower half of the values are placed in a left node and the upper half in a right node with the middle value being promoted to the parent of the original node. Of course, the splitting process may filter all the way up the tree in which case a new root node is created and the height of the tree increases by one.

The deletion algorithm is the inverse of the above using a merging process that has the potential of decreasing the height of the tree by one.

There are many variations on the above basic principles. One variation, highlighted by Date (1994) and introduced by Knuth (1973), is made up of two parts, the sequence set and the index set (see figure 2.3). The sequence set is a dense single level index providing (sequential) access to the data. The index set is a tree-structured index providing direct access to the sequence set and therefore also to the data. The index set is the B-tree proper with the combination of both the sequence set and the index set being known as the B+-tree.

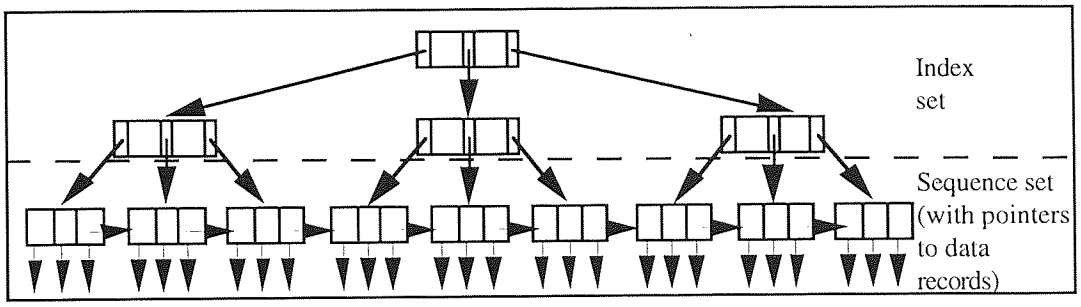


Figure 2.3 Knuth's variation of a B-tree *after* Date (1994).

2.8 PHYSICAL DATABASE DESIGN

This chapter has explored the fundamental file organisations that exist. In addition, there are numerous variations on the basic themes and new mechanisms continue to emerge each claiming to deliver better performance for certain types of access (Chen and Vitter, 1984; Litwin, 1980; Nievergelt et al., 1984). A choice must be made from the various options that exist for any particular database. The process of choosing appropriate file organisations and access mechanisms is known as physical database design.

The database designer must choose from those options supported by the particular database. Although this narrows the range of options, the process is still fraught with difficulties because different choices will have a different impact on performance. Some file organisations are good for one type of access whilst being poor for other types. The database designer must therefore analyse the type of applications the database will serve and attempt to choose the most appropriate file organisations available with a view to achieving good performance.

Performance can be improved by choosing appropriate auxiliary structures i.e. indexes and by arranging the actual data on disk in an intelligent manner. Indexes

provide routes to data and are good for selective retrieval of data records. A number of indexes can be added but their impact is constrained. Indexes only reduce the amount of accesses required to retrieve a single object to n+1 (the number of levels, n, plus one access for the actual block containing the desired record). Access to a single record using an index will only be zero if the record is already in the buffer and this will be due to the buffer management strategy not the file organisation. In addition, continually adding indexes adds to the cost of maintenance; when a record is updated references in indexes may also require updating.

Conversely, the intelligent placement of data improves performance over a series of requests. Clustering aims to improve performance in this way by storing objects that are likely to be accessed together physically close on disk. If the next record required in a series of requests is always physically close, then disk access time should be negligible or even zero (if objects are as close as being on the same page).

Irrespective of whether hashing or indexing has been used to determine the position of the next record to be fetched, the R/W head still has to move to that position on disk to be able to read that block into memory. It is at this point that clustering can improve performance. Hence, indexing and hashing techniques work at a different level (a single request) to clustering which works at the level of a series of requests.

Different clustering strategies exist but each tends to be good for one particular type of access whilst relatively poor for others. However, only one clustering strategy can be applied to each file. One approach is to choose the strategy most suited to the application which is expected to predominantly use the records of the file. However, this means that other applications (and therefore users) are penalised.

It is a difficult task for the database designer to ascertain whether there exists a compromise option that would suit all applications equally (not being optimum for one application) as an alternative to giving high priority to one application to the detriment of all others. It is also difficult given a specific situation to determine whether a few high priority users are more important than the majority. Often, given a time limit and too much conflicting information, a good strategy and not necessarily the optimum one is reached. Indeed, there remains the issue whether it is possible to arrange the data for all applications.

2.9 CONCLUSION

This chapter has reviewed the various aspects of the physical model which prove critical to the performance of databases. The concepts of physical database design

and specifically clustering have been introduced highlighting the fact that the database designer is faced with a lot of uncertainty and not enough guidelines. This thesis aims to provide more insight into the applicability of different clustering strategies and (in doing so) to arrive at some hints and guidelines for the database designer.

CHAPTER THREE

THE RELATIONAL MODEL

Relational databases are now well established and popular. Even with the advent of object databases, it is unlikely that their popularity will diminish, particularly in view of the investment placed in them. Much research has taken place in order for relational database products to reach such maturity and this cannot be dismissed. The aim of this chapter therefore is to review the relational model of data and its implementation before proceeding to the more complex object data model in the next chapter. By looking at the implementation of the relational model, this chapter will bring together physical concerns discussed in chapter two with logical concerns. Before the relational data model can be discussed, it is sensible to reflect on what is meant by the concept of a logical model.

3.1 WHAT IS A LOGICAL MODEL?

Models are employed heavily in developing and implementing databases. Modelling creates abstractions from the complex real world. Models aid understanding by creating a structured, easier to comprehend representation of the real world.

Logical models (also called internal models) form the basis for implementing and presenting databases. For example, relational databases are centred around the relational model of data and network databases are based on the network data model. A logical model provides the means of modelling and defining the data of the application domain. Therefore, associated with any logical model will be a number of logical modelling constructs to specify entities (things or objects of interest that we would like to retain data about), their characteristics, constraints on them and relationships amongst them. The users perceive their data to be arranged into the elements of the chosen data model and this provides a basis for retrieval and modification of the data in the database. Hence, in addition to the ability to specify data structure and relationships, a logical model should also have a number of associated operations that transform one instance of the data model to another.

The logical model of the database ideally should be hardware independent. It should not contain or be concerned with any physical details. It therefore acts as a 'bridge' between high-level complex interactions in the real world and low level details of the chosen physical medium on which the database is to be stored. However, it is software-dependent (Rob and Coronel, 1993) because the DBMS of the chosen database dictates how the model is specified.

The relational model is a very popular example of a logical data model. It is explored in this chapter because of the maturity of relational databases compared to object databases. It is thought advantageous to have regard for the knowledge that has been gained over previous generations of databases before object databases are discussed.

3.2 THE RELATIONAL MODEL

The relational model of data was introduced by E. F. Codd in 1970 and has become renowned and popular for its inherent simplicity. Its simplicity stems from the fact that the only construct of the relational model is the table. In fact, there are no physical constructs and it was the first logical model to exhibit both data and structural independence. Tables are a particularly good construct to work with because humans are used to dealing with tables in everyday life e.g., a train timetable. The relational model is based on a strong mathematical theory, set theory, and so in relational terminology the table is known as a relation. A relation is a matrix (see figure 3.1) of single valued data items. This means that in any row-column intersection there can only be one data value (i.e. repeating groups are not allowed) which cannot be decomposed further (i.e. data values are atomic). Each table represents one entity set or entity e.g., customer, supplier, invoice. An individual entity or entity instance is reflected by the occurrence of a row (known as a tuple in relational terminology) in the corresponding relation e.g., a customer with name Fred Bloggs in the customer relation or a supplier with supplier number S5 and name ABC Ltd in the supplier relation. Columns of a relation represent characteristics or attributes of the entity instances e.g., customer name and customer credit limit describe the entity customer. The number of tuples in a relation is known as its cardinality and the number of attributes its degree. Both the order of the attributes and the order of the tuples is irrelevant.

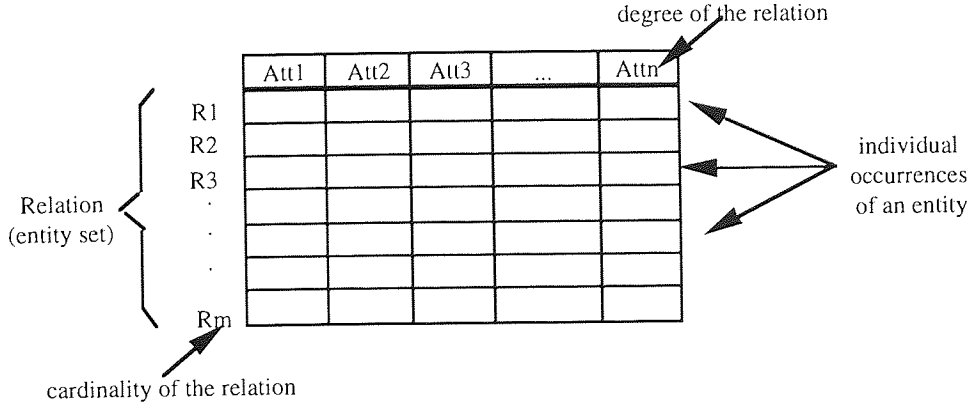


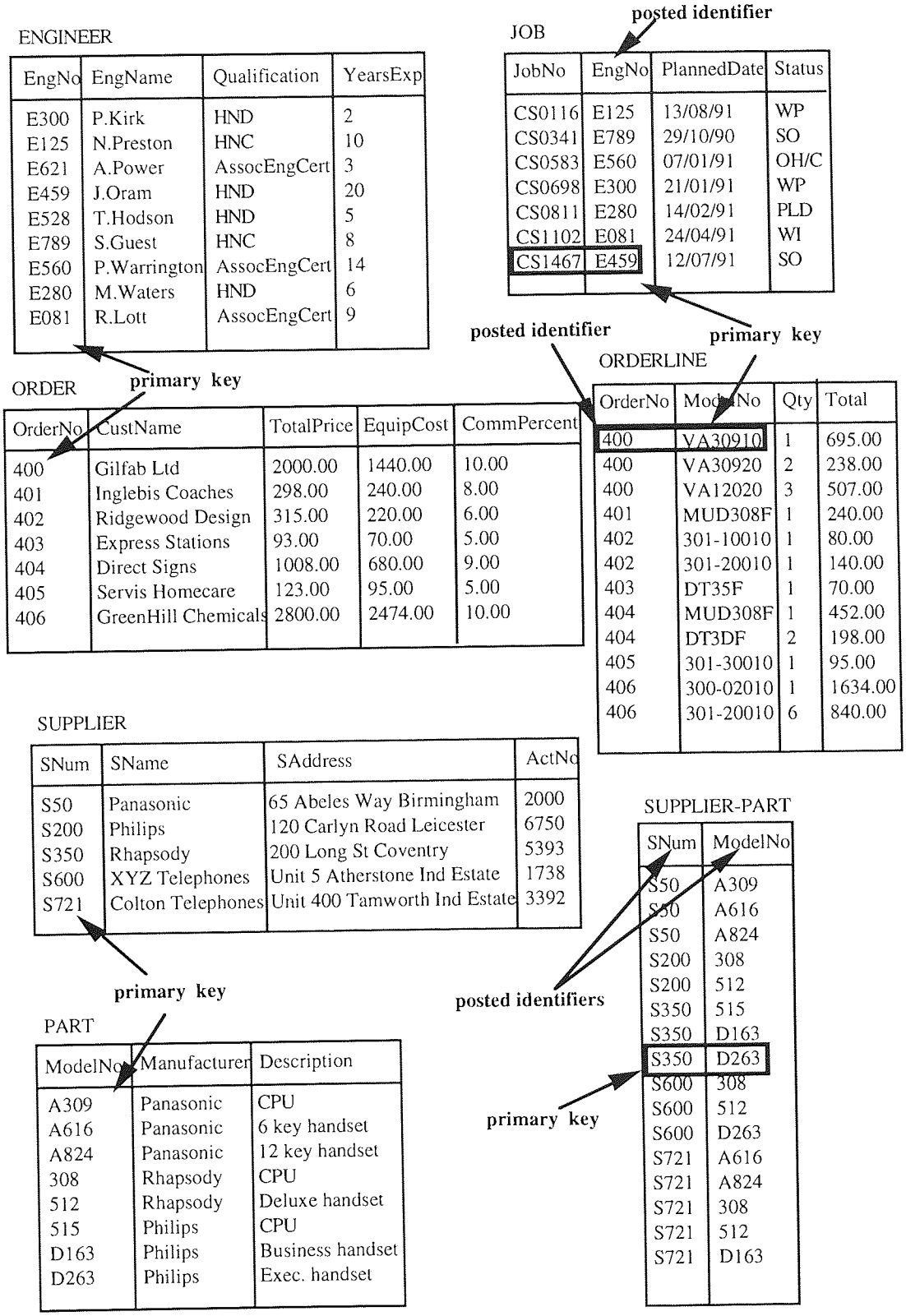
Figure 3.1 The structure of a relation.

Each tuple is identified by a key. A key is one attribute or a combination of attributes that uniquely identifies each entity instance so that there exists no duplicate tuples. These attributes may be naturally occurring characteristics or they may be artificially assigned so that each and every tuple can be identified. An associated constraint is the entity integrity rule that states that no component of the primary key of a base relation can be null. A null value indicates that information is missing or the value is unknown.

Each entity is specified in independent relations. Entities can be related by matching values in domain-compatible columns from different relations. This provides a link between independent relations and a way to represent relationships in the relational model. Three types of relationships can be modelled in this way, one-one, one-many and many-many relationships. One-to-one and one-to-many relationships are modelled by the appearance of the primary key from one table in the other table to which it relates. In the one-to-many case, the primary key from the 'one' end of the relationship appears in the relation of the 'many' end. When a primary key from another relation appears in a relation, it is known as a foreign key or a posted identifier. A many-to-many relationship is modelled as a relation in its own right and its primary key becomes a combination of the primary keys of the two relations taking part in the relationship. Relationships are constructed by matching values between two columns of different relations so we must always ensure that a foreign key has a matching primary key value. This is known as the referential integrity rule.

For example, consider the data analysis of ATelecomCo. The company purchases, installs and maintains telephone systems. A subset of a relational model for the company is shown in figure 3.2 below. The example database shows three different areas of the model, a one-to-one relationship, a one-to-many relationship and a many-to-many relationship. The company has a number of engineers and there are a number of installation or maintenance jobs to be completed at any current time. A job is performed by one engineer only and an engineer is responsible for only one job at any time. This one-to-one relationship is reflected by the relations ENGINEER and JOB and by the posted identifier EngNo in the relation JOB. A number of orders for new equipment will be made by the company's customers. The relationship between the relations ORDER and ORDERLINE is one-to-many and this is reflected by the posted identifier OrderNo in the ORDERLINE relation. The company has a number of suppliers of telephone equipment. Particular telephone systems can be purchased from both the manufacturer and a dealer and so the relationship between suppliers and parts is many-to-many. The SUPPLIER-

PART relation models the many-to-many relationship between the relations SUPPLIER and PART.



ENGINEER

EngNo	EngName	Qualification	YearsExp
E300	P.Kirk	HND	2
E125	N.Preston	HNC	10
E621	A.Power	AssocEngCert	3
E459	J.Oram	HND	20
E528	T.Hodson	HND	5
E789	S.Guest	HNC	8
E560	P.Warrington	AssocEngCert	14
E280	M.Waters	HND	6
E081	R.Lott	AssocEngCert	9

JOB

JobNo	EngNo	PlannedDate	Status
CS0116	E125	13/08/91	WP
CS0341	E789	29/10/90	SO
CS0583	E560	07/01/91	OH/C
CS0698	E300	21/01/91	WP
CS0811	E280	14/02/91	PLD
CS1102	E081	24/04/91	WI
CS1467	E459	12/07/91	SO

ORDER

OrderNo	CustName	TotalPrice	EquipCost	CommPercent
400	Gilfab Ltd	2000.00	1440.00	10.00
401	Inglebis Coaches	298.00	240.00	8.00
402	Ridgewood Design	315.00	220.00	6.00
403	Express Stations	93.00	70.00	5.00
404	Direct Signs	1008.00	680.00	9.00
405	Servis Homecare	123.00	95.00	5.00
406	GreenHill Chemicals	2800.00	2474.00	10.00

ORDERLINE

OrderNo	ModelNo	Qty	Total
400	VA30910	1	695.00
400	VA30920	2	238.00
400	VA12020	3	507.00
401	MUD308F	1	240.00
402	301-10010	1	80.00
402	301-20010	1	140.00
403	DT35F	1	70.00
404	MUD308F	1	452.00
404	DT3DF	2	198.00
405	301-30010	1	95.00
406	300-02010	1	1634.00
406	301-20010	6	840.00

SUPPLIER

SNum	SName	SAddress	ActNo
S50	Panasonic	65 Abeles Way Birmingham	2000
S200	Philips	120 Carlyn Road Leicester	6750
S350	Rhapsody	200 Long St Coventry	5393
S600	XYZ Telephones	Unit 5 Atherstone Ind Estate	1738
S721	Colton Telephones	Unit 400 Tamworth Ind Estate	3392

SUPPLIER-PART

SNum	ModelNo
S50	A309
S50	A616
S50	A824
S200	308
S200	512
S350	515
S350	D163
S350	D263
S600	308
S600	512
S600	D263
S721	A616
S721	A824
S721	308
S721	512
S721	D163

PART

ModelNo	Manufacturer	Description
A309	Panasonic	CPU
A616	Panasonic	6 key handset
A824	Panasonic	12 key handset
308	Rhapsody	CPU
512	Rhapsody	Deluxe handset
515	Philips	CPU
D163	Philips	Business handset
D263	Philips	Exec. handset

Figure 3.2 Part of the ATelecomCo database.

Maintaining data in independent relations reduces data redundancy. A process called normalisation is typically applied to a set of relational tables to ensure that data redundancy is controlled. Reducing data redundancy is desirable because it removes update anomalies. When a set of tables has been normalised, information can be inserted into smaller relations where the primary key is known, tuples can be deleted without losing information and an update of a single fact does not require a lengthy search through relations or the risk of producing inconsistent data. Normalisation is broken down into a number of stages. Each stage has a set of rules that produces a larger number of relations but which are in a more desirable form. First, second and third normal form were originally defined by Codd (1970). A set of relations in second normal form is more desirable than a set in first normal form and a set of relations in third normal form is more desirable than a set in second or first normal form. A set of relations in third normal form have no repeating groups and all non-key attributes are dependent on the whole key and nothing but the key. However, even when relations are in third normal form there still exists some undesirable properties and so a fourth and a fifth normal form have been defined.

Normalisation unravels relations into a larger set of relations, separating out the different relationships between the data (Date, 1994). Information is not lost; indeed it allows information to be maintained that previously could not because of the lack of a primary key. It therefore contains more real world information. However, a larger number of relations requires users to reconstitute the data more often. The DBMS must match values from different relations more often. This is a time-consuming process and is known as join processing. The join operator is the most significant operator of the relational model and will be discussed in the next section.

3.3 THE NATURE OF QUERIES

Every data model has a set of associated operators indicating how an instance of such a model can be queried and manipulated. The relational model has a firm mathematical foundation and so a set of rigourous operators has been defined. Codd (1979) proposed a relational algebra including mathematical set operators (union, intersection and Cartesian product) and special relational operators (restrict, project, join and divide). An alternative but equivalent mathematical abstraction proposed by Codd (1972) is relational calculus. Both languages have been used as a basis for implemented languages.

It is the join that is the most significant operator and as defined by Date (1994):

"Builds a relation from two specified relations consisting of all possible combinations of tuples, one from each of the two relations, such that the two tuples contributing to any given combination satisfy some specified condition."

This strictly describes the theta-join where the join condition is anything other than equality. When the condition is equality the join is known as an equi-join. A natural join is the equi-join with one of the join columns of the original relations removed so there exists no duplicate columns (figure 3.3). The word 'join' typically refers to the natural join because it is one of the most useful forms of join.

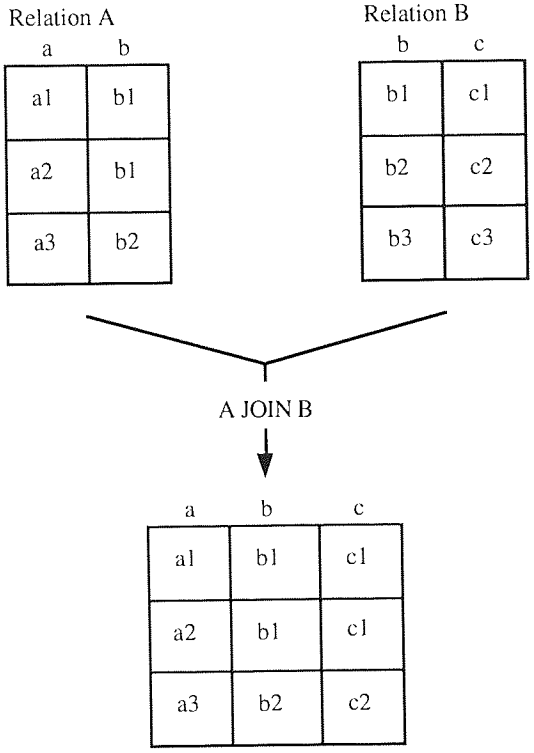


Figure 3.3 A natural join.

SQL (Structured Query Language) is the standard query language for relational databases and is based on tuple-oriented relational calculus. SQL is tuple-oriented as opposed to domain-oriented because variables range over complete tuples rather than complete domains in a relation.

Queries can be classified into:

- retrievals;
- updates.

A retrieval requests data that will be used for information purposes and leaves the database in the same state prior to the request. On the other hand, updates change the state of the database by altering the database content. SQL provides operators for retrievals and updates.

3.3.1 Retrievals

The basic form of an SQL retrieval command is:

```
SELECT   <target list>
FROM     <list of relations>
WHERE    <condition>
```

The target list specifies the attributes of interest which belong to the list of relations specified in the FROM clause. The WHERE clause is an optional component. A query without a WHERE clause represents a simple retrieval. The simplest retrieval is a request for all details of all records of a relation. On the other hand, a query with a WHERE clause with a condition is known as a qualified query. The condition determines the relevant tuples and is a rule consisting of one or more simple conditions. A simple condition may be a selection condition which restricts the tuples of a relation to a smaller set or a linkage condition which is used when there is more than one relation. To restrict a relation to a vertical subset (i.e. projection), the WHERE clause is omitted from the query. For example, figure 3.4 shows the effect of applying the query:

```
SELECT   Age, MaritalStatus
FROM     Person
```

Notice that duplicate rows have not been removed (there are two rows with values 28 and 'married'). SQL provides the keyword DISTINCT to explicitly specify that duplicate rows should be removed. The following query would produce a relation with one tuple containing the values 28 and 'married'.

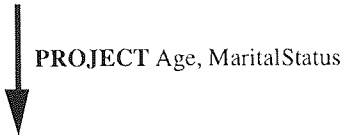
```
SELECT   DISTINCT Age, MaritalStatus
FROM     PERSON
```

The simple query,

```
SELECT   OrderNo, TotalPrice
FROM     ORDER
WHERE    CustName = 'Express Stations'
```

would produce a relation with a single tuple containing the values 403 and 93.00 for the attributes OrderNo and TotalPrice respectively.

Name	Address	Age	MaritalStatus	Occupation
J.Irons	12 Mancetter Road	23	single	VDU operator
L.Boal	40 Nursery Road	28	married	sales rep
J.Taylor	BrookDene Witherley Rd	18	single	student
L.Douglas	70 Friary Road	5	single	school
D.Beale	35 Margaret Road	30	divorced	teacher
T.Watts	13 Greendale Road	28	married	accountant
B.Draper	20 St. Georges Road	42	married	catering assistant
M.Preece	60 Stratford Avenue	51	divorced	solicitor
A.Williams	9 Covent Close	24	single	shop assistant



Age	MaritalStatus
23	single
28	married
18	single
5	single
30	divorced
28	married
42	married
51	divorced
24	single

Figure 3.4 The project operator.

The join operation is achieved in SQL by using a linkage component to relate attributes from two or more different relations. For example, the query

```
SELECT   ORDER.*, ORDERLINE.*
FROM     ORDER, ORDERLINE
WHERE    ORDER.OrderNo = ORDERLINE.OrderNo
```

joins the two relations ORDER and ORDERLINE on the attribute OrderNo (see figure 3.5). An asterisk (*) indicates that all attributes of the specified relations are required in the resultant relation.

The above query produces an equi-join. To produce the more popular natural join requires each attribute name except one of the joining attributes to be specified in the SELECT list. For example the query,

```
SELECT   CustName, TotalPrice, EquipCost, CommPercent,
               ORDER.OrderNo, ModelNo, Qty, TotalCost
FROM     ORDER, ORDERLINE
WHERE    ORDER.OrderNo = ORDERLINE.OrderNo
```

produces the relation in figure 3.6.

ORDER

OrderNo	CustName	TotalPrice	EquipCost	CommPercent
400	Gilfab Ltd	2000.00	1440.00	10.00
401	Inglebis Coaches	298.00	240.00	8.00
402	Ridgewood Design	315.00	220.00	6.00
403	Express Stations	93.00	70.00	5.00
404	Direct Signs	1008.00	680.00	9.00
405	Servis Homecare	123.00	95.00	5.00
406	GreenHill Chemicals	2800.00	2474.00	10.00

ORDERLINE

OrderNo	ModelNo	Qty	Total
400	VA30910	1	695.00
400	VA30920	2	238.00
400	VA12020	3	507.00
401	MUD308F	1	240.00
402	301-1001C	1	80.00
402	301-2001C	1	140.00
403	DT35F	1	70.00
404	MUD308F	1	452.00
404	DT3DF	2	198.00
405	301-3001C	1	95.00
406	300-0201C	1	1634.00
406	301-2001C	6	840.00

Equi-join ORDER, ORDERLINE on OrderNo

OrderNo	CustName	TotalPrice	EquipCost	CommPercent	OrderNo	ModelNo	Qty	Total
400	Gilfab Ltd	2000.00	1440.00	10.00	400	VA30910	1	695.00
400	Gilfab Ltd	2000.00	1440.00	10.00	400	VA30920	2	238.00
400	Gilfab Ltd	2000.00	1440.00	10.00	400	VA12020	3	507.00
401	Inglebis Coaches	298.00	240.00	8.00	401	MUD308F	1	240.00
402	Ridgewood Design	315.00	220.00	6.00	402	301-10010	1	80.00
402	Ridgewood Design	315.00	220.00	6.00	402	301-20010	1	140.00
403	Express Stations	93.00	70.00	5.00	403	DT35F	1	70.00
404	Direct Signs	1008.00	680.00	9.00	404	MUD308F	1	452.00
404	Direct Signs	1008.00	680.00	9.00	404	DT3DF	2	198.00
405	Servis Homecare	123.00	95.00	5.00	405	301-30010	1	95.00
406	GreenHill Chemicals	2800.00	2474.00	10.00	406	300-02010	1	1634.00
406	GreenHill Chemicals	2800.00	2744.00	10.00	406	301-20010	6	840.00

Figure 3.5 An equi-join on relations ORDER and ORDERLINE.

CustName	TotalPrice	EquipCost	CommPercent	OrderNo	ModelNo	Qty	Total
Gilfab Ltd	2000.00	1440.00	10.00	400	VA30910	1	695.00
Gilfab Ltd	2000.00	1440.00	10.00	400	VA30920	2	238.00
Gilfab Ltd	2000.00	1440.00	10.00	400	VA12020	3	507.00
Inglebis Coaches	298.00	240.00	8.00	401	MUD308F	1	240.00
Ridgewood Design	315.00	220.00	6.00	402	301-10010	1	80.00
Ridgewood Design	315.00	220.00	6.00	402	301-20010	1	140.00
Express Stations	93.00	70.00	5.00	403	DT35F	1	70.00
Direct Signs	1008.00	680.00	9.00	404	MUD308F	1	452.00
Direct Signs	1008.00	680.00	9.00	404	DT3DF	2	198.00
Servis Homecare	123.00	95.00	5.00	405	301-30010	1	95.00
GreenHill Chemicals	2800.00	2474.00	10.00	406	300-02010	1	1634.00
GreenHill Chemicals	2800.00	2744.00	10.00	406	301-20010	6	840.00

Figure 3.6 A natural join on relations ORDER and ORDERLINE.

3.3.2 Updates

A tuple or tuples can be inserted into and deleted from any table using the INSERT and DELETE operator respectively. An example of the INSERT operator is:

```
INSERT
INTO SUPPLIER (SNum, SName, SAddr, ActNo)
     VALUES  ('S899', 'PhoneBody', 'Unit 30, Lichfield Ind.
             Est. Tamworth', 6894)
```

An example of the DELETE operator is:

```
DELETE
FROM    ENGINEER
WHERE   EngNum = 'E560'
```

One or more field values of one or more tuples of a relation can be changed to new values using the UPDATE operator, as in the query:

```
UPDATE  ORDER
SET     CommPercent = 20.00
WHERE   TotalPrice > 1000.00
```

3.4 PERFORMANCE

Performance can be associated with the number of physical disk accesses a system performs over a period of time and this is dependent on the implemented physical model (discussed in chapter two). The logical model and physical model have been discussed independently and the logical model is data-oriented being structured (in theory) without consideration of the functions that will be applied to the database. The relational model, in particular, is produced without any consideration of physical details and is then the basis for implementing the physical model. After applying normalisation to the relational tables, each normalised table can become a physical file. Normalisation creates a large number of independent tables and so increases the amount of join processing required which seriously degrades performance. However, physical database design should consider the performance of the chosen physical model (an important user requirement) and this requires looking at the nature of the applications that the database will serve. This means that physical database design requires consideration of aspects totally overlooked at the logical database design stage and possibly partially 'backing-out' of the normalisation process in order to achieve good performance. These issues surrounding physical database design are explored below.

3.4.1 Physical Database Design

Physical database design is one of a number of stages involved in the development process of a database. The development process is a modelling approach which gradually progresses from the real world to the database as shown in figure 3.7. The logical model becomes the pivotal element in the process, forging the gap between high level concepts of the real world and the physical details of the computer. In order to understand the application domain and for the users and the programmers to communicate effectively, the data is initially analysed at a high level. This may include a business analysis stage. A popular model for an initial analysis is the E-R (Entity-Relationship) model by Chen (1976). The constructs of the E-R model are very user-friendly and are software and hardware independent (Rob and Coronel, 1993). The schema produced by applying the E-R model can then be translated into a logical schema that can be 'understood' by the chosen DBMS. The logical schema must finally be mapped to a physical schema (in terms of files and records) so that the database can be implemented on the chosen physical medium.

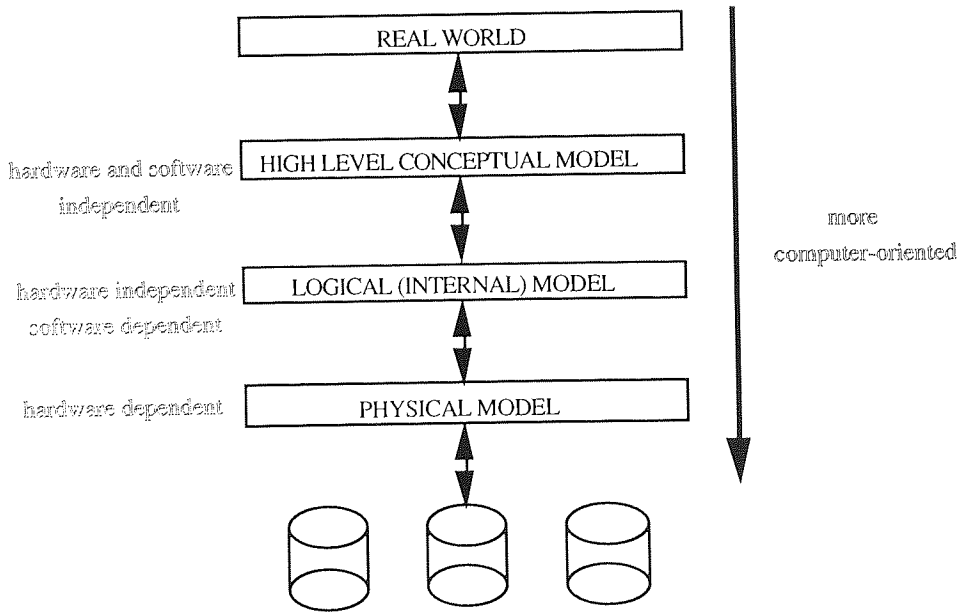


Figure 3.7 The different levels of abstraction in the development process.

The relational approach to logical database design produces a set of tables in third normal form. The normalised tables could be implemented directly with each relation becoming a file and each tuple a record within that file. However, simply applying the logical model and directly mapping logical constructs onto 'physical' objects means that no thought has been given to performance. Indeed, such an approach inevitably leads to poor performance. Therefore, mapping from logical to

physical objects should not necessarily be a direct process; physical database design should be, and is, concerned with choosing arrangements that achieve good performance. Although separating logical and physical concerns is convenient, it also means that some fine-tuning is required before the model can be implemented.

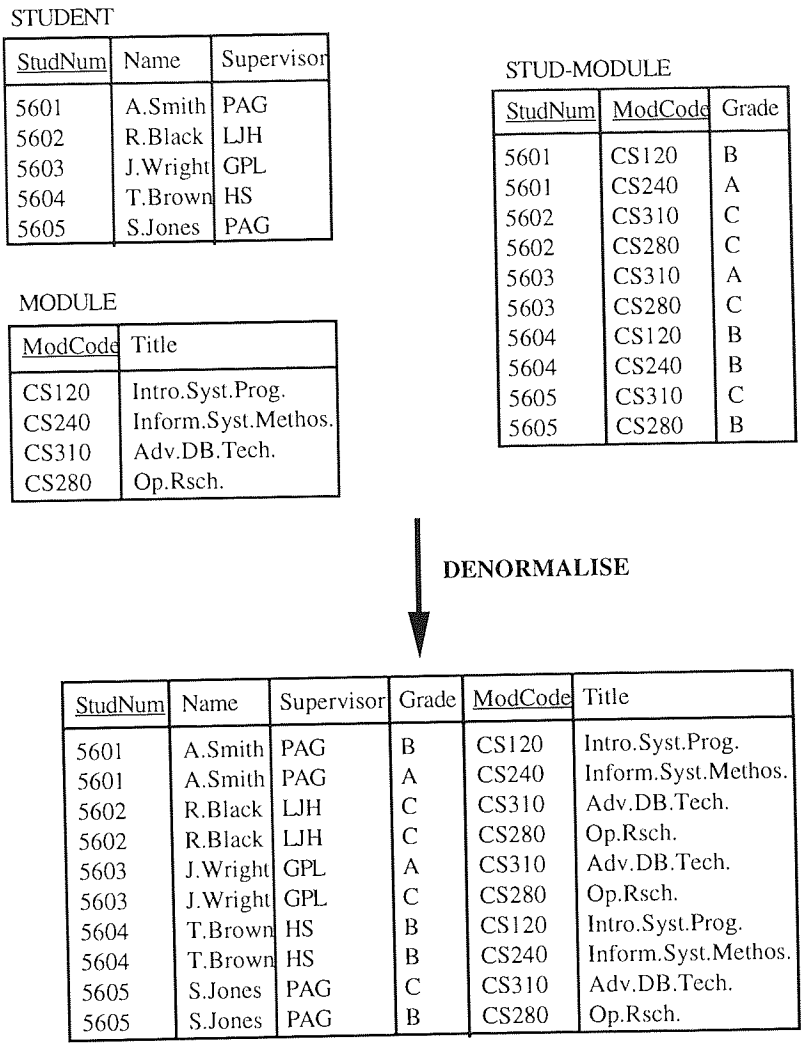


Figure 3.8 An example of de-normalisation.

Physical database design must consider conflicting requirements, those embodied in the very purpose of normalisation, to reduce data redundancy and therefore remove update anomalies and to provide adequate performance. An option is to de-normalise the set of relations that are in third normal form. A relation that has been decomposed into two or more entities could be merged into a single entity (see figure 3.8). This would remove the need to perform one or more joins to get all the information pertaining to all entities involved in a relationship. Choosing which entities should take part in de-normalisation depends on the frequency and type of requests that will be put to the database. It only makes sense to merge independent entities into a single entity if they are likely to frequently take part in a join

operation. Reversing the normalisation process reduces the amount of join processing but leads to update problems and some information cannot be represented.

In addition to mapping from logical to physical constructs, the database designer must decide on the type of file organisations to use for each file, particularly the number and type of indexes required for each file. This again requires an analysis of the type of requests that will be put to the database. For example, some file organisations perform better for retrievals rather than updates and frequent use of a particular search criterion suggests that an access path on the search field is required.

Prior to the existence of databases, where each transaction had its own set of files (the file approach) it was possible to 'tune' these files to best suit the nature of the transaction. However, the files of a database are shared and queried randomly posing conflicting requirements. Therefore, it is not possible in many cases to provide efficient access for every type of request. Typically, the option that performs the best for the operation most frequently applied to the database is chosen.

It is possible to argue from this that there exists a weakness in the current layered architecture proposed in the literature. In particular, the conceptual and logical design stage do not capture information in the structuring process (especially because it is data oriented) which must be recovered in an ad-hoc manner during the physical database design stage. An exception is the SSADM (Structured Systems Analysis and Design Method) approach which analyses and includes functions at the conceptual and logical level. This thesis will not resolve this issue, suffice it to say that it is important to be aware of the fact that a number of factors come to bear during physical database design which are not previously formally specified.

Despite the uncertainty involved in the physical design process, the designer may be able to choose from an array of storage management techniques including:

- file organisations (indexes, hashing, B-trees);
- pointer/chain links;
- clustering.

Due to the lack of formal guidance available for the designer in choosing from the above, obvious design choices have developed into rules of thumb. For example, if access is always based on one particular attribute of a record, then an index should be built on that attribute or if a report is to be generated in a specific order, it makes sense to place records in that order on disk preferably in adjacent blocks.

More rules of thumb exist for the use of indexes compared to clustering. Further, clustering strategies for the object data model have not been discussed to a great extent in the literature possibly due to the high level of complexity. Clustering is therefore the focus of this thesis in particular its effect on performance and is explored in the next section. Firstly, clustering with respect to the relational model will be examined in order to gain a level of understanding and insight before proceeding onto the more complex issues of the object model.

3.4.2 Clustering

Clustering places logically related data physically close on disk (or other secondary storage media) such as in the same segment of pages. As noted by Kim (1990a), this is useful to minimise the I/O cost of retrieving a set of related entities. The knowledge of which data are related can be gained from the nature of the applications that will be executed against the database and from the logical schema. Clustering can be considered to be a de-normalisation process whereby information ignored during the logical design phase is captured. Physical closeness of data that is likely to be accessed together or at relatively the same time improves overall performance due to:

- blocking;
- buffering;
- reduction in seeks.

As discussed in chapter two, a fixed size block of records is typically transferred from secondary storage to main memory as opposed to a single record. Therefore, there exists the potential for bringing records into memory that will be required by future requests. These requests can be satisfied without performing expensive disk accesses. Maintaining a number of buffers in memory instead of just one further increases the probability that a required record is in memory and hence does not require a disk access to be fetched. A suitable buffer replacement strategy will maintain the blocks in memory that are likely to be accessed in the near future and so should improve overall performance. Once a track is filled with related data, one might think the next stage would be to fill the next track on the same disk surface. However, related data should fill the disk on a cylinder basis to achieve a better improvement in performance. Physical closeness, as used in the definition of clustering, is therefore interpreted as the placement of records within the same cylinder or in the nearest cylinder and not as physically close as possible on disk. This reduces the amount and the length of seeks required between block accesses. This can have a marked effect on performance because seek time (as it is a mechanical movement) is a large component of disk access time whereas switching

from track to track is negligible. Rotational latency may also be reduced but this requires the detailed strategic placement of blocks. The heads are rotating constantly so we need to know how long it will take for the next block to be accessed in order to place it in its correct relative position on a track of the same cylinder. It is difficult to have an effect at this level.

There are essentially two types of clustering for traditional data models (Date, 1994):

- intra-file clustering;
- inter-file clustering.

Clustering all tuples or records belonging to a single file is known as intra-file clustering whereas clustering all records belonging to more than one file is known as inter-file clustering. For example, in a relational database all tuples that make up a single relation can be stored and clustered together in the same segment of pages on disk. A single column value (or a combination of column values) may be further used to determine the storage order of the tuples. This speeds up sequential scanning. Alternatively, especially if there exists a one-to-many relationship between two relations, each record on the one side of the relationship could be followed by its corresponding records of the many side of the relationship. This makes the joining of these two relations very efficient. The effect on performance is even more significant if the particular join is a frequently applied operation.

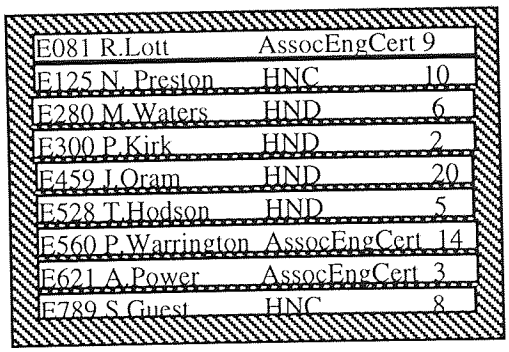


Figure 3.9 Clustering a single relation.

Consider the ATelecomCo example in figure 3.2. The relation ENGINEER could be clustered on the same pages and ordered by EngNo as in figure 3.9. Alternatively, ENGINEER could be clustered with its corresponding JOB record or vice-versa (see figure 3.10). Further, the one-to-many relationship, ORDER-ORDERLINE could be clustered as in figure 3.11.

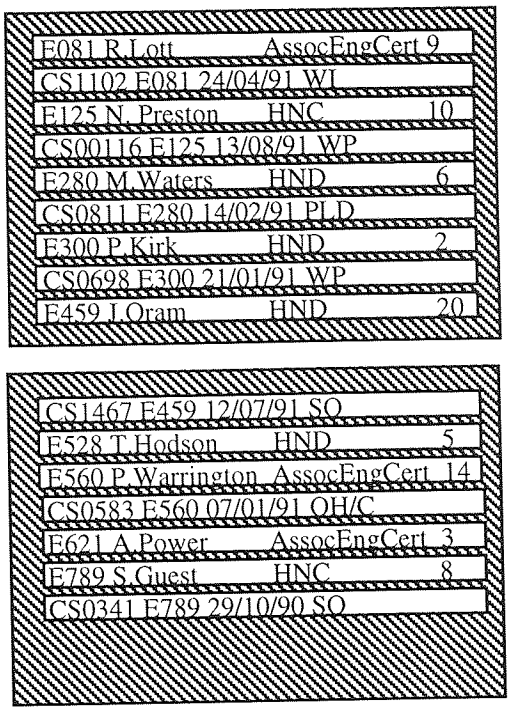


Figure 3.10 Clustering more than one relation.

In some DBMS's, clustering can be specified at the user level in the form of hints even though it should be a hidden implementation detail. Some employ the approach of taking hints from users as well as having a default clustering strategy. The data definition language of such DBMS's provides a syntax for specifying potential clustering strategies. The DBMS must then be able to specify lower level placement requirements at the file manager interface. Some DBMS file managers allow 'add new record near page X' to be specified. Others, such as the file managers of System R (Astrahan et al., 1976) and INGRES (Stonebraker et al., 1976), allow a tentative record identifier (a page number, slot number pair) to be specified when a new record is added. The file and disk manager can only attempt to store a new record on a specific page. Usually a certain amount of free space is retained on every page for this eventuality.

It is important to be aware that only one clustering strategy can be implemented for a given set of files. This presents a dilemma to the database designer because one strategy may be more desirable for one type of access whilst undesirable for most others. For example, clustering single relations is better for sequential scanning but not good for join processing whereas the converse is true for clustering multiple relations. This is because records from a single relation or set are dispersed over a smaller number of pages if they are clustered together than if records from different relations or sets are clustered.

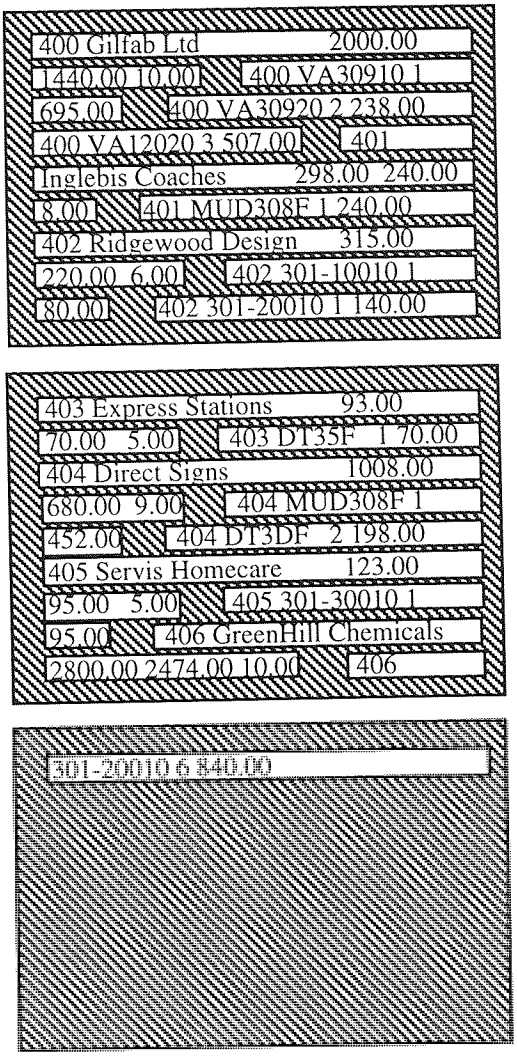


Figure 3.11 Clustering more than one relation with a one-to-many relationship.

Design decisions may therefore be more complex than simply choosing the clustering strategy that is best for the most frequently applied operation. At the very least, physical database design requires a good understanding of the type of applications the database will serve. Alternatively, one may perform a deeper analysis of the expected frequency and cost of different types of access to the database for the given user applications.

3.5 CONCLUSION

This chapter has brought together the logical model of data with the physical model discussed in chapter two by reviewing the popular relational model. In doing so, it has explored the physical design stage of the development process and in particular clustering with respect to traditional models. It has shown that even with simple logical models, the process is fraught with difficulties. Despite this, it is anticipated

that valuable lessons can be taken forward and used to understand object clustering later in the thesis.

CHAPTER FOUR

THE OBJECT DATA MODEL

This thesis is concerned with the performance of object databases. As highlighted previously, an object database is first and foremost a database irrespective of the data model it supports. Chapter two has reviewed the physical characteristics of databases which form a consistent thread through all databases. It is the nature of the data model which the database supports that makes a database differ from all others. The object data model is more powerful and more complex with respect to traditional data models. This chapter therefore provides a comprehensive review of the object model of data. The ideas taken on by the object data model stem from the general notion of object-orientation. Firstly therefore, the concept of object-orientation will be considered.

4.1 OBJECT-ORIENTATION

The concept of object-orientation began in the programming language area with the introduction of Simula in the 1960s followed shortly by Smalltalk. The ideas initiated by these two languages form the basis of the whole concept of object-orientation. However, as these ideas have been applied to many diverse areas, they have been manipulated and interpreted so as to be more applicable to the given context. Hence, there has been much variation on the basic themes. Further, this has been a critical contributing factor towards the general vagueness in the area. It is debatable which features are significant and contribute to the essence of object-orientation and which are seen as options that may be incorporated to enhance the core theme. However, Kim (1990b) believes there to be common features that can be found in object systems that enable core features to be specified. This unfortunately is a subjective process since some articles will neglect a characteristic that is described as fundamental in another. This chapter will take the lead offered by Kim (1990b) and consider the following key characteristics of object systems and consequently object databases:

- encapsulation;
- message passing paradigm;
- classes;
- object identity;
- complex objects;
- inheritance.

The following sections will explore each of these ideas in depth. Following these, some miscellaneous topics will also be covered which although not considered to belong to the core features, nevertheless they are felt important. Prior to leaping

into these topics it is perhaps beneficial to digress slightly and consider exactly what is an object?

4.1.1 What is an Object?

On a first approach to the subject, it is easy to find yourself asking exactly what is an object? However, there is not a single precise definition. Kim (1990b) identifies that there is little consensus as to the *precise* definition because of the development of the notion firstly in programming languages then in artificial intelligence and finally in databases. The fact that sometimes the answer is 'everything is an object' does not make the issue any clearer. It appears that each real world object in the application domain is uniformly represented as an object. Hence each person, each department, each manager etc. is considered an object. This is analogous to the entity instance in traditional database conceptual modelling. This may confuse some people whose knowledge lies in traditional databases. For some, it may be natural to anticipate that the entity instance concept align with the notion of an object and this is actually the case as discussed above. However, others may be confused because it was often the case that when an entity was discussed e.g., person, it was the entity set or group of entity instances that was being referred to and not a single entity instance. Therefore, the real difficulty arises when the uniform treatment of considering everything as an object applies to both classes (entity sets) and attributes. In the traditional database world, an attribute is quite clearly a separate concept and is not uniformly treated with entity instances. Indeed, the issue regarding attributes as objects is not clearly decided. This confusion in terminology can be attributed to the lack of uniformity apparent in the traditional view. In fact, uniformity in the object approach has been seen as an advantage in simplifying the users view of reality. What is agreed upon in such an approach is that each object be associated with a unique identifier known as the object identifier (see below) instead of being solely identified by value.

Objects are the units that object systems deal with. The representation of objects began with the development of object languages. Objects consist of a private data structure with a public interface where the private memory consists of a list of variables pertaining to each particular object. This description is essentially that of encapsulation, a concept supported by most object systems.

The vague definition of an object perpetuates confusion and it is useful to take a step back to some fundamental and general concepts of object-orientation. Indeed, the description of encapsulation clarifies the meaning of what constitutes an object.

Further, the concepts are intimately linked and provide a complete picture of object-orientation.

Blair et al. (1991) believe that object concepts all originate from the general notion of abstraction. This they consider to be the essence of object computing. Abstraction is the means by which we both analyse a situation and solve the problem. It is the manner in which we generalise and ignore details in order to dissolve complexities that we cannot comprehend as a whole. For example, decomposition is an abstraction mechanism whereby a problem is split into isolated subproblems that are more manageable to deal with. Abstraction manifests itself within the fundamental concepts of the object approach namely encapsulation, classes (behaviour sharing) and inheritance (behaviour sharing). Encapsulation is based on the notion of data abstraction and hence the object approach is less focussed on functional decomposition.

4.2 ENCAPSULATION

Data abstraction is the significant form of abstraction found in object computing and has adopted the term encapsulation. Encapsulation is considered the essence of the object approach and is closely associated with the definition of an object. Data is encapsulated within a single unit or module whereby the only access to that data is via a well-defined procedural interface. This means that every object is associated with a state (data) and a set of operations that operate and change the state of the object (Garvey and Jackson, 1989). A user cannot directly manipulate or inspect the data items and does not know the workings of internal algorithms or representation details. As both data (state) and operations (behaviour) are encapsulated in a single unit the object approach provides a behavioural view. The only concern to the user is the available operations and the associated semantics of those operations i.e. the what instead of the how.

A single object is therefore represented as a single module incorporating both the data representation (state) and its operations (behaviour), in addition to its interface.

The fact that data values are no longer directly accessible (contrast with conventional programming languages such as Pascal) has also inspired the term 'information hiding'. Implementation details are hidden from the user. Two different perspectives emerge; a specification (user) view concerning the interface and its semantics and an implementation view concerning attributes and their representation and algorithms of the associated methods. These views are also highlighted by distinguishing between the public and private aspects of an object. The interface of

an object is public to users. The implementation details (sometimes referred to as memory) are private to the object in that they are not visible to users and cannot be directly accessed or manipulated. The second viewpoint is emphasised when the definition of an object is considered to be an 'encapsulated private piece of memory'. Further, this separation of viewpoints provides data independence. If the internal details change the users of that object are not affected so long as it supports the same (compatible) interface. Any effects are confined within the module. For example, a change to the representation say from a linked list to an array would affect only the method code for the retrieval operations. No methods outside the module are affected because we know (due to the rule of encapsulation) that they cannot access the particular representation directly. All operations on an object must be via associated methods. Users are never allowed to write applications dependent on storage structure (Jackson, 1991). Further, the object is protected from external meddling (Khoshafian and Abnous, 1990).

The separation of implementation and representation details from the external view is not a completely new phenomena and is evident in a weaker form in traditional programming languages such as Pascal. The set of integers has an associated collection of operations such as "+", "-" and "*" to represent the behaviour of integers. As noted by Khoshafian and Abnous (1990), Pascal programs manipulating integers can be ported and compiled on systems that have a different internal representation. For example, one system might internally represent integers using 32 bit twos complement whilst another uses an 8 bit representation. The notion of encapsulation employs this principle on a higher level and extends to every object in a program so that for each object the external interface is not effected by changes in the separated internal representation and implementation.

The effect of modularity is also noted as providing systems with low coupling. Methods contained in one module (object) are not dependent on other methods or representation details of other modules and hence traditional "waterfall" repercussions are avoided. Traditional systems are notoriously difficult to change owing to their high coupling between elements and interdependences that arise from functional decomposition. The difference between object programming languages and conventional languages is attributed to the basis of modularisation with decomposition based on objects in the former and procedures in the latter. The avoidance of the "rippling" effect or autonomy of elements is just one of the benefits that arises from the nature of object modularisation as noted by Khoshafian and Abnous (1990). Modularisation provides the following benefits:

- improved modelling of the real world;
- correct applications are generated;
- stubbing of encapsulated data types;
- exception checking and error handling can be incorporated into the operations conveniently;
- reusability.

The approach of invoking and accessing from 'outside' leads to the notion of message passing. A message is sent to an object to invoke a given method. Object interaction is therefore visualised as message passing. This mode of computation is more delegatory (Khoshafian and Abnous, 1990) i.e. "Tell me what you want to accomplish and I'll do it my own way." From this it is clear that what underpins the object/message paradigm is the principle of encapsulation.

The above discussion implies that object systems must provide strict encapsulation (strictly no access to data values other than by methods). However, as Bertino and Martino (1991) describe, some systems do not provide strict encapsulation. For example in O2, attributes can be made visible from outside if a user demands that a given attribute be available for public reading or public writing. Dittrich (1990) also believes that encapsulation should not be an 'all or nothing' principle. Bertino and Martino (1991) further note that in some systems where both users define their own methods for accessing variables and the system provides methods such as Get and Set, there is no way to determine which method will be chosen. Hence, in some systems the user can re-define methods to ensure that the correct method with the correct semantics is always applied.

Encapsulation is used interchangeably with the term abstract data typing. In fact, some describe objects in terms of abstract data types. Abstract data typing was firstly introduced in programming languages to allow users to define new structures for their own data (Garvey and Jackson, 1989). Prior to this, programmers were limited to basic atomic types such as integers and string. Hence, object systems and programming languages allowing ADTs have the advantage that the application can be more easily and accurately represented. The definition as given by Khoshafian and Abnous (1990) considers an abstract data type (ADT) to be an extension of a data type where a data type is a 'set of objects with the same representation and associated set of operations'. An ADT extends this notion by 'hiding' the implementation of the user defined operations. Indeed, as presented in this definition, it is clear why the notions of encapsulation and abstract data types are considered as one, particularly as it tends to be said that operations are encapsulated within an ADT. However, Kim (1990b) pinpoints a difference with regard to the viewpoint on attributes as he states "this is contrasted to the term attribute in data

abstraction in which objects with the same abstract interface are grouped and the interface does not include attributes".

4.3 MESSAGE PASSING PARADIGM

The notion of strict encapsulation enforces that the state of an object can only be manipulated via the defined methods. Methods are invoked by sending a message to the appropriate object (similar to function calls in conventional programming). Interaction is therefore based on message passing and a message is merely a request to an object to carry out one of its methods. This approach supports data abstraction because message passing is not based on any assumptions regarding the implementation or the physical storage of data. Hence object databases support physical data independence. Smalltalk was the first advocate of this object/message passing paradigm.

A message usually consists of three parts:

- the object, also known as the receiver;
- the message (method name), also known as the selector;
- a number of arguments (optional).

The syntax of a message as given by Kim (1990b) is:

```
(Selector Receiver [Arg1, Arg2, .... ArgN])
```

where square brackets denote optional items.

The message name or the selector is the equivalent of the method name to be applied to the object. It is further noted that arguments are either objects or can be evaluated as objects. The Receiver is also an object. As a message may return an object, it follows that any argument and/or the receiver may constitute a message in its own right.

Arguments are optional; a message with no arguments is known as a unary message whereas a message with two or more arguments is known as a keyword message. In a keyword message, the selector is made up of two or more keywords preceding each argument separated by a colon. Garvey and Jackson (1989) give an example of a keyword message:

```
HouseFinances Spend : 30.50 On : 'food'
```

where the syntax of the message is:

```
<OBJECT> <SELECTOR> <ARG>
```

When a message is sent to an object, it is, in turn, represented by a method invocation. The selector of the message is the equivalent method name. The first important check the system must make is therefore that an equivalent method does exist for the class of the object specified as the receiver of the message (not neglecting inherited methods). The system must return 'message unknown' if a method cannot be found. If a method does exist, however, the system must be able to perform further type checks. The system must look to the method specification (as factored out in the class object) to ensure that all arguments of the type expected are compatible with those given in the message. Further, the system may need to use the specification to note the type of any returned arguments. This enables additional type checking similar to that found in conventional programming languages.

The specification, as discussed above, represents the external interface of an object. It is concerned with how users can communicate and interact with objects of a class. A method is therefore divided into two parts; the external interface and the implementation. The external interface specification is alternatively known as the method signature. It is similar to the declaration of formal argument types and names in traditional programming languages. This formal specification of a message includes:

- the method name;
- the names and classes (i.e. types in traditional programming) of arguments;
- the class of any returning arguments (if any).

However, Bertino and Martino (1991) note that systems do not necessarily have to formally declare the class of arguments and results e.g., Orion. In this case, type checking is carried out completely at run-time.

The method specification may contain additional components such as trigger methods and exceptions (Bertino and Martino, 1991). These methods are raised at the point of method execution and are used to modify inherited and system defined methods so as to make them more applicable.

Method invocation raises an issue in distributed and client-server architectures (Bertino and Martino, 1991). The issue poses a decision regarding the most suitable site for method invocation. There exists a choice between local and remote method execution. Either the object should be moved from the server to the workstation site (local execution) or the method should be executed on the server where the object resides (remote execution). Bertino and Martino (1991) establish

the various factors that should be taken into account and which contribute to a complex decision involving inevitable tradeoffs.

4.4 CLASSES

Behaviour sharing is another form of abstraction and one mechanism by which behaviour is shared is classification. Classification is a popular way of organising items or objects in particular. Indeed it is difficult to imagine a model without some form of organisation in some applications where there are a multitude of objects. Classification is the mechanism by which similar objects of the system are grouped together into classes that share the same set of attributes and methods. The class concept captures the generalisation is-a relationship between an object and the class to which it belongs. An object must belong to one class as an instance. The class can be viewed as a template with which to base an instantiation mechanism. Instantiation introduces one form of reusability (see inheritance for another). The same definition as specified by the template is reused to generate new objects that have the same behaviour and structure. Typically, this is achieved by sending the message *new* to a class object. The template therefore defines as given by Bertino and Martino (1991):

- the structure of objects (instance variables or attributes);
- messages defining the external interface of objects;
- methods invoked by the messages.

Note that given attributes and methods are shared explicitly, it is an implicit assumption that the external interface is shared. The declaration of the class therefore includes (at least) the naming of the above components plus the naming of the target object and (optional) number of arguments for the external operations and more obviously the naming of the class.

Simula introduced the class concept but with some deficiencies. Specifically, although it allows the declaration of a class, including its behaviour, it does not support encapsulation and so attributes can be manipulated directly. This was resolved by object languages which followed, notably Smalltalk and C++.

Bertino and Martino (1991) consider the template concept as essentially a specialisation of the instances (objects). Common definitions (methods, messages and attribute specifications) are said to be factored out of a set of objects. The factoring out process can also be viewed as an abstraction approach. Details that are the same between objects need only be held once for the class. All instances have common behaviour and so share the same code and only one code base is required.

Hence storage of redundant information is avoided. The information pertaining to individual objects i.e. the actual data values, is maintained separately with each object. These values are different for each object and only ever coincidently become identical. Therefore each instance of a class will be allocated memory storage to maintain the internal representation. The instance variable declaration may include:

```
<instance variable name> <instance variable type>
```

Explicit declaration of the type of these instance variables enables a type constraint to be applied. Only objects of the type declared can be assigned to the instance variable. Such type constraints are the essence of strongly typed languages (i.e. those with the ability to capture type errors). These also make use of algorithms to infer the types of expressions and variables. Class variables may also be maintained in the class object and are similar to global variables. They hold information pertaining to all shared instances of a class such as the average of a particular attribute e.g., weight.

The above implicitly describes a class as representing two different points of view. Firstly, a class can be viewed as the group of its objects that belong to it and hence a class denotes the collection of instances that constitute the class. Secondly, a class can be viewed as a definition template or specification of the objects that belong to it. The class in this case is therefore represented as an object that acts as a template. This viewpoint is associated with the perspective that *conceptually* a class "represents the set of all possible objects with prescribed structure and behaviour" (Khoshafian and Abnous, 1990). Data types in conventional languages are based on a similar idea. For example, the type integer relates to the set of all possible integers and hence is infinite. This contrasts with the first view which relates to the set of *actual* instances of a class that have been created and not destroyed i.e. those that exist within the given environment. The template view is known as the intension whereas the collection of actual instances is known as the extension. Khoshafian and Abnous (1990) consider that classes have the potential to monitor their own extension because a message *new* is sent to the class object each time an instance of that class is created. Under this scenario, the class object would also require informing of the deletion or removal of instances.

This separation of viewpoints is interpreted and addressed differently by various object database systems. Bertino and Martino (1991) review the alternative approaches amongst a number of different systems to conclude that:

"in general, we find the notion of decoupling specification from the notion of extension correct. The major drawback is that the data model becomes more

complex compared to a simpler model in which the class acts both as object template and object extent".

Smalltalk and C++ do not support class extensions but define a template that generates objects using the class construct with associated primitives to create and destroy instances of the class.

It is within the area of databases that class extensions become important with many object databases supporting this notion (Khoshafian and Abnous, 1990). Databases are renowned for bulk information processing of objects of the same type. Sets of objects aggregated together form the basis of a query. This is a critical distinguishing factor between databases and object languages; their view of classes and types is different. Consider the relational database which typically uses SQL as its data definition and manipulation language (Khoshafian and Abnous, 1990) as an example to show the different viewpoint concerning classes and types. In such a database, the emphasis is on the extent of a table i.e. the set of all existing instances that make up a tabular construct. The CREATE command of SQL therefore possesses two roles. In one action both the structure of an instance is declared and the declared table name (class name) acts as a mechanism to access the set of all existing instances. The database schemas type declarations actually identify the extensions of all tables in the schema so that the schema and the extensions are one. The user can explicitly insert records into and delete records from the extension with the 'handle' being the table or class name. Another view by Kim (1990b) is that the concept of class is one of the most important links (not a distinguishing factor) between object systems and databases, particularly as they are the basis of queries.

"Without the notion of a class to aggregate together related objects it is difficult to conceptualise (and evaluate) a query" (Kim, 1990b).

In a relational database, sets of objects are grouped in relations or tables. Similarly a query in an object database is issued against a collection of instance objects that make up one or more classes.

Khoshafian and Abnous (1990) discuss the role of collections to achieve the same goal as class extensions because of the lack of support of this notion in most object languages. Bertino and Martino (1991) also note the need for set constructors in order to group objects. Collections are built-in classes (types) such as sets, bags (sets that allow duplicates) and arrays that act as containers of other objects. A particular collection can therefore contain all instances pertaining to a particular class i.e. the extension and instances are created by sending an appropriate message to the

collection object. The instances that make up a collection must be of the same type for strongly typed languages whereas languages such as Smalltalk allow different types.

Kim (1990b) views a class as being similar to an ADT. An ADT in essence provides encapsulation and hence can be associated with the level of the object as described earlier. However, an ADT as developed in programming languages defines encapsulated *sets* of similar objects with an associated collection of operations. Hence, an ADT is more truly akin to the combination of the concepts of encapsulation and class and pertains to the class level rather than the instance level. Indeed, Khoshafian and Abnous (1990) believe that one of the three most fundamental concepts is that of abstract data types and that classes are a means of implementing these in object systems in particular programming languages. The benefits of encapsulation are still apparent at this higher level. Implementation and representation are still separated from the external view but for a group of objects rather than for an individual.

A class can be primitive (no associated attributes) or non-primitive.

"The value of an attribute of an object, since it is necessarily an object, also belongs to some class. This is called the domain of the attribute." (Kim, 1990b)

Domains of attributes can be any arbitrary class. This gives rise to the nested structure of classes forming a hierarchy known by Kim (1990b) as the class-composition hierarchy (nested hierarchy). The hierarchy takes on the form and characteristics of the nested relation if it is restricted to a strict hierarchy.

If a database consists of a number of classes there is an issue regarding whether an object can change class (Bertino and Martino, 1991). Allowing objects to migrate supports object evolution and indeed a philosophy behind the object approach is the support of evolution in general. To maintain the concept of object identifiers, a migrated object should change structure and behaviour but retain its unique OID. However, most systems do not support the migration of objects probably because a domain constraint problem may arise (Bertino and Martino, 1991). If an attribute A of a class C changes domain (i.e. migrates to another class) the possible change in attributes might be incompatible with the domain A as expected by the class C. Therefore, an incorrect object could be used for the value of attribute A, invalidating any methods referencing that attribute.

Kim (1990b) summarises the advantages which make the class concept critical:

- can act as a basis for the grouping of objects particularly useful for evaluating queries;
- good modelling concept capturing the instance-of relationship;
- introduces type checking and hence improves integrity by restricting values an attribute can take on with regard to the associated domain;
- removal of name and integrity related specifications; without the notion of the class more storage space would be required and dynamic changes would be rendered impractical.

Classes may be further organised and belong to a hierarchy of classes incorporating inheritance (see below).

4.5 OBJECT IDENTITY

The issue of object identity is a central theme in object systems. Khoshafian and Abnous (1990) consider it to be one of the three fundamental concepts of object-orientation. Individual objects are distinguished from each other according to their identity and not their value(s). A hidden object identifier (OID) must be maintained for each object (Garvey and Jackson, 1989). It must be unique, immutable and last the lifetime of the object. It should not be reused even after deletion of the object itself and should not change when any properties of the object change. Hence, if the values of any object change, then the object can still be identified as the same object. Systems supporting identity allow arbitrary changes state values whilst maintaining identity. However, those supporting a strong notion of identity will also allow structural modifications (changing of an object's class) without a change in identity (Khoshafian and Abnous, 1990). This is more in line with the real world. For example, a horse does not become a different horse simply because its name may change (Jackson, 1990). In summary, identity should be independent of how the object is accessed, what it contains and its location. It should be machine or implementation independent.

Bertino and Martino (1991) consider the importance of object identifiers as providing the ability to share sub-objects and to construct general object networks. Kim (1990b) agrees that this "natural representation of state" is one of two major reasons for the introduction of OIDs. Hence, the concept of object identifiers is strongly linked to that of complex objects. The second reason for the introduction of OIDs cited by Kim (1990b) is the fact that they were firstly introduced in object programming languages with little regard for large databases and the notion of queries. It was expected that objects would reside in virtual memory. Hence, the navigational model of computation was put in place and transported into the area of

object databases. The importance of OIDs is also as a mechanism to pinpoint an object for retrieval.

OIDs are used in place of values for attributes making the construction of complex objects feasible. The OIDs reference sub-objects that are components of the complex object. Khoshafian and Abnous (1990) believe that it would be difficult if not impossible to allocate self-contained objects to instance variables and let the same object be part of multiple objects without the notion of object identifiers. The construction of complex objects organises the object space being manipulated by a program. Further, object identifiers allow programs to dynamically construct arbitrary graphs of complex objects i.e. creation at run-time as opposed to compile-time in early conventional languages. This area is strongly linked to the idea of persistence; objects can even become persistent and subsequently accessed in a different program.

Without the use of identity or object referencing Khoshafian and Abnous (1990) cite two 'far from ideal' solutions. The first is to replicate the information pertaining to the same sub-object in each instance object that would share it if using identity. This approach clearly is a waste of space and causes consistency problems i.e. if one copy is updated all other copies must likewise be updated. The second approach is that used in relational databases. All sub-objects make up a collection, each with a unique identifier so that a number of instances in other relations can refer to the same copy of a sub-object by referencing the key. The use of keys is fraught with many problems and indeed provided the motivation for the RM/T model by Codd which makes use of surrogates similar to object identifiers. It is noted that the relation containing sub-objects is not storing instances of a class but strings of characters. Keys confuse identity and data values. This meant that it was necessary to enforce the integrity rule that key data values be unique. A problem with this is that certain updates that are valid in the real world are not valid in terms of the database. Taking the example from Garvey and Jackson (1989), a Person is identified by a name and address. In the case of a female, in the real world the event marriage updates (changes) the surname. However, this update would not be allowed by the database and so the database would become 'out of step' with the real world. Often the resolution of this type of problem was to introduce a surrogate identifier for example person_number. However, such identifiers are unnatural and often have no meaning in the real world application. The user is left to work with such obscure identifiers in order to put a query to the database. Even if the key is allowed to change (Bertino and Martino, 1991), there is still the problem that in database terms it is no longer recognised as the same object. A new

key implies a new and different object i.e. a discontinuity of identity. Khoshafian and Abnous (1990) cite a further problem with keys, non-uniformity. This has two aspects. Firstly, identifier keys have both different types and different combinations of attributes in different tables. This inconsistency makes keys more difficult to manipulate. Secondly, attributes for identifiers may need to change, for example, if two relations are merged with different identifiers. As noted above, changing the values of keys produces a discontinuity in identity. Normalisation and the 'flattening' out of relations is associated with this approach. This means that unexpected joins are required to re-associate data. In contrast using object identifiers allows storage and hence easier direct retrieval of complete objects. The decomposition of objects into a number of relations is a less intuitive approach to the representation of object spaces.

4.5.1 The Equality of Objects

The introduction of OIDs impinges on the meaning of equality. The simple view is that two objects are different if they have different OIDs even if all attributes have the same values. However, it is often viewed as a more complex issue and there are essentially two forms of equality.

The first form of equality, denoted by '=', is known as identity equality. The second form of equality, denoted by '==' is known as value equality. Two or more objects are identical if their OIDs are equal. Two or more objects are value equal if they have equal content, not necessarily equal OIDs. Equal content means that attributes that are values must be equal and attributes that are not values must recursively be equal. Hence two identical objects are also value equal but two value equal objects are not necessarily identical.

4.5.2 The Representation of Objects

There are different approaches to the representation of OIDs. Bertino and Martino (1991) provide an extensive discussion on the various approaches. In summary, the alternatives are,

- Logical identifier pair consisting of <class identifier, instance identifier>;
- Logical identifier only based on the instance identifier with class identifier kept as control information in the object itself;
- Physical identifier based on physical address on disk with location independence;
- Physical identifier incorporating the location by using the identifier of the site where the object is created for distributed systems.

The first two alternatives provide physical independence and require a table to be maintained to map from logical identifiers to physical addresses. This has the advantage that an object can change its physical address (i.e. move position on disk) without invalidating the logical identifier, as long as the table is updated. However, the performance of the system is inevitably worsened. This factor provides the motivation for the third alternative giving fast direct access. For example, the O2 system uses record identifiers as OIDs. A further aspect is the use of a forward marking technique so that records can be moved to a new page without changing the record identifier. However, Bertino and Martino (1991) cite a significant disadvantage of this approach in workstation architecture systems and distributed systems. If an object is created on a site that is different to the object store site then a temporary OID is required. This requires a permanent identifier at transaction commit time. This has provided the motivation for the last alternative, incorporating information regarding the site where the object was created. This site then retains forwarding information when the object is moved. The distributed version of Orion uses this approach.

The OID concept has links with the message passing paradigm. In an object, where a message is separated from the actual invocation of the method code, the system must determine on receipt of a message whether an equivalent method actually exists. If this process is successful, the system can then find the method code and apply it to the specified object. Due to the class concept, all methods are factored out and reside in the class object. Hence, the system must locate the class object to look up the method. In the first alternative, the system can extract the class identifier from the OID contained in the message and therefore go directly to the class object to determine the existence of an equivalent method name. Therefore, if a message attempts to invoke a method that does not exist, the system can return 'method unknown' immediately. However, in the second alternative where the OID only contains the instance identifier, the system must firstly fetch the object before it is able to locate the class object. Hence, objects are unnecessarily retrieved for invalid messages and type checking becomes expensive.

However, in terms of migration of objects from one class to another, the second alternative is better than the first. An object can reference any number of objects and it can be referenced by any number of objects. The objects that reference a given object contain the OID of the object in order to point to it. In the second alternative, when an object changes class and its class identifier changes, references to the object are not invalidated because they do not contain any reference to the class. The only update required is to the system information specifying the class of

the object within the migrated object itself. However, the first alternative's OID contains the class identifier. If the object changes to a new class then the OID must be updated with the new class identifier. This means all references to the object from any number of other objects are invalidated. It is difficult to locate and update all these references and proves to be an expensive process.

4.6 COMPLEX OBJECTS

The state of an object is represented by the set of values pertaining to the attributes of the object. The composition in terms of attributes reflects what is known as the aggregation relationship. In object systems, the values of an object's attributes may also be objects forming complex objects. This approach contrasts to the traditional relational approach where attribute types could only be from a limited set of basic atomic types such as integer or string.

The classes to which attributes belong are alternatively known as domains. If the domain of an attribute is of a basic atomic type such as integer or string then that class is known as a primitive class. Object spaces are built up on top of these base or 'rock bottom' types (Khoshafian and Abnous, 1990). These types are built-in and supported by the underlying system as in conventional programming languages. All other classes (i.e. those with associated attributes) are known as non-primitive classes. An attribute value may itself be an object as stated above and belong to either a primitive or non-primitive class. For example, a person may have the attribute address which belongs to the non-primitive class Address with its own attributes number, street, town and postcode. The fact that an attribute domain can be any arbitrary class gives rise to an arbitrary nesting of objects.

Instances of a primitive class such as the number 7 for example from the class (domain) integer have no associated attributes and hence are self-identifying. From this viewpoint, it is unnecessary for them to have associated object identifiers. Hence, most terminology refers to these instances as values rather than objects. In fact, Bertino and Martino (1991) state "that not all entities are objects" and that all primitive entities, e.g., integer or string, are values. Objects with no associated OID are not considered as objects whereas entity instances with associated attributes and OIDs are objects. Kim (1990b) takes an alternative viewpoint, uniformly treating everything as an object. The values of attributes are considered objects that may belong to either a primitive or non-primitive domain. Whatever the perspective, most object systems treat them differently by not associating an object identifier with instances of these basic types. For instance, it is not allowable to have two objects both with the value 7; there is just one object 7. Hence, the

statements '7 = 7' and '7 == 7' evaluate to true and the message new sent to the class Integer yields an error.

Either viewpoint leads to the same natural representation of the nested structure of complex objects. As objects are represented by OIDs it is a logical step to represent an object's state as a set of object identifiers of the objects that are the values of the attributes of the given object. Recursively, those objects are also represented by a set of identifiers. Objects with no OIDs or values are represented directly within the object. Hence, one can visualise an hierarchy from the nested structure of attributes. Kim (1990b) uses the term class-composition hierarchy to distinguish this from the class hierarchy of superclasses and subclasses. These hierarchies are noted as being distinct. Notice that it is the concept of object identity that allows such an elegant representation of complex objects. Hence, if conceptually there exists an infinite collection of identifiers I, Khoshafian and Abnous (1990) establish three properties of an object:

- as an instance of a class - the object's type;
- an associated identity I;
- the state of the object, the values of its instance variables.

If the instance variables or attributes are represented as:

```
A1, A2, A3 .... An
```

then the state of an object is often represented with a similar notation to:

```
A1 : i1,
A2 : i2,
A3 : i3,

An : in
```

where each ij is either an object identifier referring to a sub-object or a base object. Alternatively, these concepts are often represented graphically.

Bertino and Martino (1991) pinpoint the use of complex values in some systems. These systems allow the user to define complex values that use the same constructors available for objects. Complex values can be represented directly in an object instead of specifying an OID. However, they can not be shared and reside directly within a single object. Using OIDs as references to other objects within the state of an object allows more than one object to reference any other object as a sub-object. This gives rise to the sharing of sub-objects and provides certain advantages. For example, if the state of the sub-object changes then all higher level objects that reference it will 'see' such changes. Complex values may be useful in certain situations, for example, where aggregates or sets of objects are to be used as

part of objects but never on their own. They also provide better system performance because the system does not have to fetch sub-objects; complex values are fetched at the same time as the system fetches the object whereas with OIDs the system only fetches the higher level object first and extra fetches are required to get the whole of the object. However, if the complex value is large then it may have to reside elsewhere for instance on a separate page of secondary storage to the higher level object. Some systems allow the user freedom to build both complex values and to use OID references.

An attribute as noted by Kim (1990b) may be a single value or a set of values and generally complex value definitions require constructors such as set, list and tuple. These should be orthogonal in that any constructor can be applied to any object and any constructor can be used on an object that has been constructed out of any constructor. However, some systems may impose a restriction on the first level constructor to be applied.

One important relationship that can be superimposed on the complex object is that of the part-of relationship. Compilation of an object with these implied semantics forms what are known as composite objects. An example is a design object that consists of a multitude of design elements. Each element is part-of the higher level element which itself may be composed of a multitude of parts. The nested structure of a complex object representation is clearly applicable to the representation of a composite object. However, it should be noted that the notion of complex objects is more general and that the semantics of a composite object constitutes just one possible concept that can be superimposed on it. The composite object implies well defined semantics on allowable operations that can be applied to the object. For example, deletion of the higher level object requires propagation of deletion operations to all sub-objects. It is unfortunate that some confuse composite objects with complex objects to the extent that the term composite object is used when essentially it should be complex object. This may be the result of the large proportion of applications with such semantics of composite objects e.g., CAD. These applications can exploit the characteristics of object databases and have provided a strong impetus to their development. Some define an object database by solely describing these applications and associating the features of the object database with the characteristics that these new applications possess. Kim (1990b) argues that these applications provide only one potential exploitation of object database technology and that substantial modifications are required to the core object model for them to be truly suitable for such applications. Hence, the core object

database does not provide all the characteristics required for these applications and the potential of object databases is more far reaching and general than this.

4.7 INHERITANCE

A second behaviour sharing mechanism is inheritance. It is debatable whether this feature is considered essential or merely an option in an object system. Although inheritance is often included in a list of object features (Dittrich, 1990; Saxby, 1988; Kim, 1990b; Bertino and Martino, 1991; Garvey and Jackson, 1989), Zdonik and Maier (1990) do not include it within their 'threshold' model, believing inheritance to be useful but not definitional. The reasons for the exclusion of such a beneficial feature are because the terminology surrounding the concept varies and because other features can be combined in such a way as to achieve the same effect as inheritance.

Inheritance is a further stage in the organisation of data which is incorporated into object languages (typically) in the form of class inheritance. Classes are organised naturally into a hierarchy where classes inherit structure (attributes) and behaviour (methods) from their parent class. It 'taxonomises' objects in order to produce a well defined inheritance hierarchy (Khoshafian and Abnous, 1990). It is an important form of abstraction "since the detailed differences of several class descriptions are abstracted away and the commonalities factored out as a more general superclass" (Bertino and Martino, 1991). A given class known as the subclass, is derived from and defined in terms of an existing parent class known as the superclass using inheritance. Hence, new classes (or software modules) are easily added and extend the model by building them on top of an existing hierarchy of classes. In this sense, the object approach provides support for evolution. Inheritance has its roots in A.I. and knowledge representation which is explored in Khoshafian and Abnous (1990). This highlights the fact that inheritance is just one kind of relationship among elements although it is a significant one.

Inheritance is considered a form of specialisation (Garvey and Jackson, 1989). A new class is really a specialisation of an existing class, inheriting structure and behaviour and possibly extending and modifying the class with addition and/or substitution. An object of the subclass is-a specialised object of the superclass. Specialisation as used in this context is distinguished from the specialisation described in the creation of instances. The former type of specialisation is achieved by explicit declaration that a class is a subclass of another class whereas a message is sent to the class object in the latter case. A contrasting viewpoint is that the inheritance hierarchy also captures the generalisation relationship between direct and

indirect subclasses. Kim (1990b) further considers the class hierarchy to be the distinguishing feature between object languages and programming languages with ADTs.

Although a new subclass can be extended or modified, Khoshafian (1990) believes the essence of inheritance is due to the fact that object classes inherit most of their attributes from generic less specialised classes. This means that information specified once in a superclass is directly obtainable using inheritance. This removes the need to re-specify redundant information. The result of inheritance is the ease by which new classes can be created and so supports the object philosophy of evolution. Inheritance of behaviour enables code sharing and reuse and inheritance of representation enables sharing of structure among data objects. Bertino and Martino (1991) pinpoint it as the second reusability mechanism after the concept of classes. Inheritance also eases updating. A single update action changes all instances along the inheritance hierarchy. However, as noted by Khoshafian and Abnous (1990), there also appears a contradiction in the terminology with respect to inheritance by adding capabilities. A subclass is considered a specialisation of its superclasses. However, this means that we are assimilating specialisation with the addition/substitution of behaviour and structure. Extending the interface means that it becomes a superset of the interface belonging to its superclass. Therefore, Khoshafian and Abnous (1990) believe that the inclusion of additional capabilities should ideally be viewed as extension rather than specialisation in order to avoid confusion. It should be emphasised that specialisation is referring to the fact that the addition of new classes moves the model a step nearer the requirements of the application domain (Blair et al., 1991).

The is-a relationship between an instance of the subclass and an instance of the superclass implies that a subclass instance can always be used in place of its superclass. The reasoning behind this is that an instance of a subclass will have at least the behaviour (state and methods) of the superclass (Blair et al., 1991). The possible additional behaviour will not be called upon as the system expects an instance of the superclass.

The class composition hierarchy or aggregation hierarchy that represents complex objects is distinct from the inheritance hierarchy. A noted difference is that the class composition hierarchy may contain cycles whereas the inheritance hierarchy cannot. However, the two hierarchies may become relevant to each other because the domain of an attribute of a class may be any class and this class may be part of an inheritance hierarchy. It therefore follows from the is-a relationship between a

subclass and a superclass that an attribute A with a domain of class C can take on any instance of class C and any instance of any of the subclasses of C.

Bertino and Martino (1991) note that inheritance also includes the inheritance of messages (usually an implicit assumption). Further, Kim (1990b) considers that inheritance includes any integrity constraints specified on inherited attributes plus the objects that belong to the class. Therefore, in summary the possible components that may be inherited include:

- interface;
- code;
- representation;
- integrity constraints.

4.7.1 The Form of Specialisation

The addition of new attributes and/or methods (i.e. not inherited) introduces these properties into the subclass along with inherited properties but not in the superclass of the hierarchy (Garvey and Jackson, 1989). Substitution (overriding) occurs where the definition of attributes and/or methods (that could have been inherited) are overridden in the subclass. For instance, an attribute in the superclass is given a new specification in the subclass (possibly a different domain) or a method with the same name is re-coded. The definition of inheritance is associated with rules governing allowable operations. Garvey and Jackson (1989) state that

"all descriptions in a class (variables, properties, and methods) are inherited by a subclass unless overridden in the subclass".

Although in most cases subclasses specialise superclasses by extending representation and behaviour, there is also the issue regarding how exactly to specialise and whether to allow the restriction of behaviour. In this case, overriding is achieved by excluding methods. The approach taken is either to declare explicitly or implicitly that the method is not inherited or to override the method in the new class with a diagnostic message.

The possible forms of specialisation are summarised by Blair et al. (1991):

- Add new behaviour (state and methods);
- Change behaviour (re-define the implementations of a method or the domain of an attribute);
- Delete behaviour (remove state and/or methods from the existing class);
- Any combination of 1, 2 or 3.

Strict inheritance allows only the first form of specialisation to be carried out. Allowing behaviour to be changed or deleted does not guarantee that an instance of a subclass can always be used in place of an instance of its superclass.

The notion of overriding means that methods (similar to procedures) are completely re-coded and written again 'from scratch' as though nothing in the superclass's method is relevant. This defeats the central principle of reuse. Bertino and Martino (1991) believe this unit of change to be unsuitable and have noted that some systems are now allowing inheritance and overriding to be performed at a lower level. Different object systems take on different approaches but the aim is to allow parts of a method to be inherited while other parts are overridden.

4.7.2 Polymorphism

A limited form of polymorphism is introduced by the class hierarchy. Polymorphism is the ability of behaviour to have an interpretation over more than one class (Blair et al., 1991). Subclassing produces a number of classes that have the same method. Further, methods may be re-defined so that not only does a method name refer to more than one class but also to more than one implementation. Polymorphism will be explored in depth later.

4.7.3 Method Binding

The requirement for method binding is a consequence of the polymorphism that the class hierarchy introduces. The method name must be bound to the correct implementation for the type specified. The mapping between method names and implementations is particularly important as overriding requires that a one-to-many mapping is resolved. The general rule is that when a message is sent to an object the most specialised method is applied. Hence, methods in the subclass with the same name as methods in the superclass override those in the superclass. Khoshafian and Abnous (1990) formalise this search process for the appropriate method into an algorithm:

Method Invocation Algorithm

```
Initially set cC (the 'current' class) to C.
If method called S is declared in cC
then it is M.  Stop searching and execute (invoke) it.
If cC is the root of the class hierarchy
then generate an error and stop; the method is undefined.
Else set cC to its parent and go to step 2.
```

This algorithm sets cC to the superclass only if specified explicitly. This is sometimes useful in an overridden method. To refer to a method in the superclass and distinguish it from the subclass either the pseudo-variable super is used (as opposed to self to refer to the subclass) or the name of the superclass acts as a qualifier using dot notation.

A critical issue is the timing of method binding with respect to program compilation and execution. The issue is whether mappings should be carried out statically when programs are created or dynamically as programs are executed (Blair et al., 1991). This will be explored in a later section on method binding.

Class inheritance was introduced by the language Simula. Although this allowed classes to inherit from one another, only attributes and not methods were inherited because instance variables could be accessed and updated directly (public manipulation). Other languages that followed Simula incorporated the notion of encapsulation (private manipulation) and hence inheritance was based on both attributes and methods. Some languages allowed the instance variables to be declared as either public or private. The introduction of inheritance yields yet a third option for the description of instance variables, subclass visible.

Hence, the simple rule by Garvey and Jackson (1989) above is more complex. Not all descriptions are necessarily inherited, particularly as there is a range of support for encapsulation. Further, with strongly typed languages 'overriding' properties in the subclass is a more complex issue than it appears at first sight. To avoid type errors we must tread with care when properties are overridden. These complexities are expounded by Khoshafian and Abnous (1990) and will be considered in the section on types and typing.

It is often implied that an object can be an instance of only one class. However, an instance of a class is-a specialisation of its superclass and can be substituted whenever an instance of the superclass is expected. This gives rise to the distinction between instances and members of classes (Bertino and Martino, 1991).

"An object is an instance of a class C if C is the most specialised class associated with the object in a given inheritance hierarchy. An object is a member of class C if it is an instance of C or some subclass of C."

Systems usually place the restriction that an object can be an instance of only one class although it can be a member of several classes (i.e. all the superclasses along the inheritance hierarchy). This derives from the transitive rule regarding the superclass-subclass relationship. This rule states that if class X is a subclass of

class Y and class Y is a subclass of class Z then class X is also a subclass of class Z. However, Bertino and Martino (1991) highlight a difficulty that arises in certain modelling cases. The example given is that of the hierarchy involving the classes Person, Student and Pilot where Person is the superclass of Student and Pilot (figure 1.1).

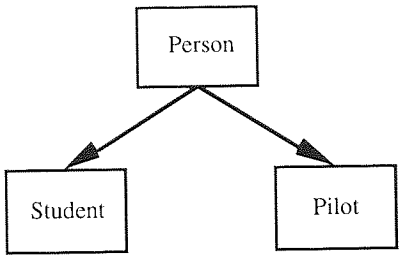


Figure 4.1 Inheritance hierarchy for classes Person, Student and Pilot.

The classes Student and Pilot are not so clear cut in the real world. Some people are both students and pilots. If an object could be an instance of more than one class then a student-pilot would be represented by an instance in the Student class and an instance in the Pilot class, as well as being a member of the superclass Person. However, name qualifications are necessary. This problem would be resolved if the system allowed multiple inheritance (see below). A class Student-Pilot would be formed that was a subclass of both Student and Pilot and hence would inherit attributes and methods from both these classes. The student-pilot would be an instance of this subclass.

4.7.4 Multiple Inheritance

The complication above has given rise to some systems allowing a class to inherit from one/more distinct classes (although up until now it has been assumed that there is a single superclass). In many situations, it is convenient to allow the modelling of the above situation. There are numerous examples in the real world of 'combination' classes. The restricted type of inheritance based on allowing only one superclass is known as single inheritance whilst the more flexible approach allowing one or more superclasses is known as multiple inheritance. In the single inheritance case, classes form a hierarchy known as a class hierarchy and can be visualised as an 'upside down' tree. When multiple inheritance is allowed, classes form a rooted directed graph sometimes known as a class lattice (Kim, 1990b). Multiple inheritance expands the union of superclass instance variables and methods to *all* immediate parents.

Whatever form is allowed, inheritance gives rise to conflicts. Resolution of conflicts requires rules to be established. In the single inheritance case, only the first type of conflict is possible. This is the conflict between a class and its superclass. However, this factor is not always distinguished as a conflict in the literature but merely a necessary process when supporting other concepts of object-orientation. The notion of overriding means that a single name represents many implementations. The system must know which implementation to use for a particular object of a class. As discussed previously, the system chooses the implementation most local to the subclass i.e. the first occurrence of the implementation in an upward search of the hierarchy. The superclasses are repeatedly searched because attributes and methods are inherited from a superclass which may itself have inherited properties from its superclass. If the root object is reached and the method cannot be found, it is necessary to return the statement 'method unknown' to the caller.

Multiple inheritance is considered more complex because it gives rise to another type of conflict in addition to the one above, namely the conflict between the class's superclasses. The union of all instance variables and methods is not as simple as it first appears. If a variable or operation name is defined in two or more superclasses, it is not clear how it applies to the subclass. Predecessors may use the same name for an instance variable or operation but they may represent totally unrelated semantics. For example, an attribute such as Height may be defined in two superclasses of a class but each may be defined with a different unit such as metres and feet. If a message call constituted the name of a clashing operation and an object of the subclass as the target object the system would have more than one valid search path and it would not be clear which one to take. In the multiple inheritance case, the search upwards presents a choice. If the item to be found is defined in only one of its superclasses there is no problem. However, which definition should the system choose if it is defined in one or more superclasses? How should the system assign precedence to the superclasses when both appear applicable? Khoshafian and Abnous (1990) consider the bulk of the problems with multiple inheritance to be dealing with conflict resolution strategies.

Another conflict that arises with multiple inheritance is conflicting instance variables and methods from a common ancestor (Khoshafian and Abnous, 1990). This is where a class is specialised into subclasses which in turn are specialised into a combination class. For example, Student and Employee are subclasses of Person and also superclasses of the combination class Student-Employee. Of course, there may be more levels than this involved. Instance variables and methods are inherited

along the subclasses from the common ancestor. In this situation therefore, these conflicting properties are related. This type of conflict is easily resolved by using only one copy of the instance variables from the common ancestor in the combination class as in Trellis/Owl and Eiffel.

Bertino and Martino (1991) establish a rule where two or more superclasses have an attribute of the same name but belong to different domains. The rule states that if the domains are related by a class hierarchy the most specific domain is chosen for the attribute in the subclass, otherwise the user must specify the superclass from which to inherit the attribute from. For example, in O2 the 'from' clause establishes which superclass the attribute is inherited from. Finally, if this is not specified the system uses an order of precedence as default.

A general solution for any name conflict is to assign a precedence to the ordering of superclasses (Kim, 1990b). Two alternatives are stated for establishing the order of precedence. Firstly, it may be specified in the class. This corresponds to a user defined order of precedence (the second option of the Bertino and Martino rule). Secondly, the order may be decided at run-time. This corresponds to the use of a default ordering by the system e.g., always left-to-right (the third alternative of the Bertino and Martino rule). Hence, user specification is considered as forcing a different order of precedence to that maintained as the default by the system. The order of precedence strategy also corresponds to the linearisation strategy described by Khoshafian and Abnous (1990). The inheritance graph is in effect being mapped by specifying the names of superclasses in a given order in the declaration onto a linear order. Yet another option is to rename or re-define the variables or operations from each superclass in the subclass to distinguish between them. In this way names could be used to indicate more clearly the semantics of each method or attribute e.g., EmployeeStatus and StudentStatus. The 'hint' to perform this renaming process may come from the system. This is a conflict resolution strategy discussed by Khoshafian and Abnous (1990). Here on declaration of superclasses that conflict, the system 'complains'. The system on complaining may then allow the renaming of variables or methods. Alternatively, the system may forbid conflicts completely. However, this is an extremely restrictive approach. Renaming is more flexible allowing the user to determine if a conflict actually exists and hence renaming appropriately. The system may allow conflicts as they arise but then on using a conflicting variable or method the user must qualify from which class it is being referred to. This is achieved by utilising a dot notation. For example, the user would have to specify Student.Status and Employee.Status.

Both the renaming and the qualifying approach place an extra burden on the user but they are considered the simplest forms and are easier to understand.

An example taken from Blair et al. (1991) serves to clarify this point. A Clock is considered to be both a specialised form of a Timer and a Gauge. Consequently, the class inherits from both the Gauge and Timer class as shown in figure 1.2 below.

The variable 'range' is defined in both superclasses. If a default approach to the resolution of clashes was taken e.g., left-to-right, the system would firstly traverse the left branch. Hence, the variable 'range', as specified in the superclass Gauge, is considered applicable. Alternatively, the user could specify that the variable 'range' as specified in the superclass Timer be inherited in the subclass. This may be because the clock in the application behaves more like a timer and its range refers to the semantics of a range implied by a timer. However, orders of precedence imply that one superclass becomes essentially the superclass of the other superclasses that were originally on the same level. Snyder (1986) believes this to cause significant semantic implications which would not have necessarily been the intent of the designer of the original classes. Alternatively, the class clock could inherit both variables but rename them to GaugeRange and TimerRange in its own class specification. Further, if conflicts are allowed on declaration then an alternative is to require qualification in the code i.e. Gauge.Range and Timer.Range.

The above presented the most commonly described strategies in the literature. However, Khoshafian and Abnous (1990) describe a further less common approach; the meet operation for subtypes. This is applied only to instance variables of records that are typed and evolves out of a subtype-ordering relationship of types. Hence, although the approach provides clean semantics, it is limited in its use. Khoshafian and Abnous (1990) describe the rules behind the meet operation;

The meet operation of T1 [a1:t1, a2:t2, a3:t3] and T2 [a3:t3', a4:t4] is the greatest lower bound of T1 and T2 using the subtype relationship:

```
T3 = T1 meet T2 = [a1:t1, a2:t2, a3:t3 meet t3', a4:t4]
```

For example,

```
if t3 is type Character[30] and t3' is type Character[40] then
t3 meet t3' is Character[30] i.e. the lowest bound.
```

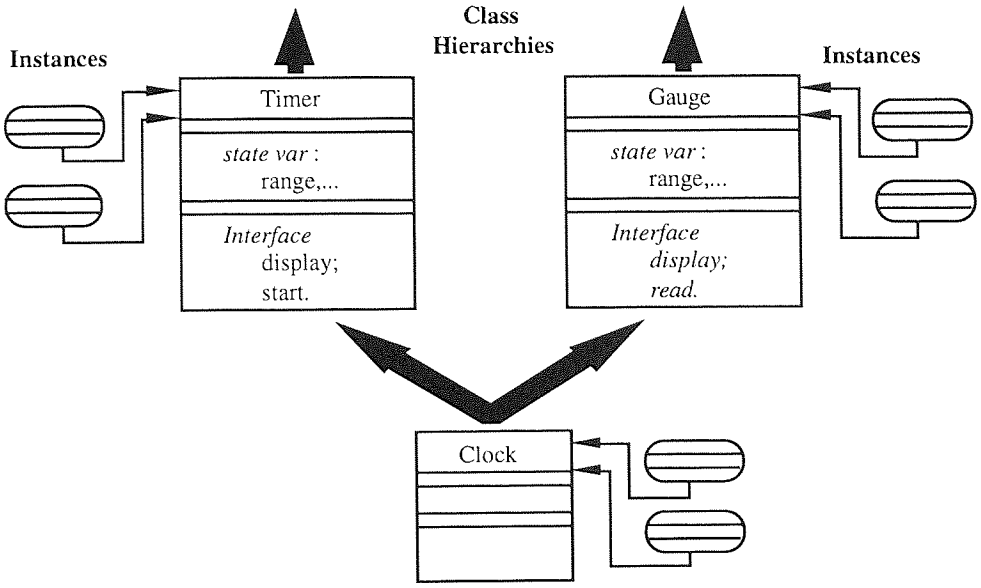


Figure 4.2 Inheritance from more than one class *in* Blair et al. (1991).

Khoshafian and Abnous (1990) conclude that "in terms of flexibility, generality, and ease of understanding, renaming attributes and qualifying attributes seem to be the most promising strategies".

Bertino and Martino (1991) cite a problem with inheritance. A method may be inherited from one class (say) class C based on attribute A with the domain as specified in class C. However, due to an order of precedence and a conflict regarding attribute A, an alternative domain is inherited from (say) class C'. Therefore a problem exists when the method is invoked because it expects the domain for attribute A to be that as specified in class C not C'.

Given that multiple inheritance gives rise to additional complications in terms of name conflicts it is questionable whether it is really necessary. Multiple inheritance certainly makes the model more complex and hence requires a more sophisticated system. However, there are advantages and disadvantages attached to each approach. Single inheritance is easier to deal with but cannot always model ideally the complexities of the real world. Kim (1990b) argues that multiple inheritance provides greater modelling power and flexibility. Single inheritance gives rise to duplication of information and "forces upon the user a less intuitive model of the database".

4.8 TYPES AND SUBTYPING

4.8.1 Types

A type is analogous to the idea of a class in that it restricts elements to a given set and has associated operations or methods. The commonalities of types can be abstracted to form subtype hierarchies in a similar manner to inheritance hierarchies.

4.8.2 Subtyping

According to Cardelli and Wegner (1985), the subtyping relationship among types is defined by the following,

"type T1 is a subtype of type T2 if every instance of T1 is also an instance of T2"

One conclusion drawn from this is the principle of substitutability. In fact, Bertino and Martino (1991) base their definition of a subtype on this principle where a type T is a subtype of type T' "if an instance of T can be used in place of an instance of T'."

In addition to the principle of substitutability, Khoshafian and Abnous (1990) describe other characteristics of the subtyping relationship. Firstly, subtyping is a reflexive in that every subtype is a subtype of itself. Relationships are also transitive:

```
if type T1 is a subtype of type T2
and type T2 is a subtype of type T3 then
type T1 is a subtype of type T3
```

Finally, if two types are subtypes of each other then they are considered the same i.e. the subtyping relationship is anti-symmetric.

4.8.3 Inheritance vs. Subtyping

The two concepts are often confused and the terms used interchangeably. The encapsulation paradigm can be strictly adhered to and subtyping can be based completely on behavioural subtyping. Behavioural subtyping as described by Bertino and Martino (1991) is based on the rule "type T is a behavioural subtype of type T' if T provides methods with the same name and the same (or compatible) arguments as type T'." The subtype may also provide additional methods. Behavioural subtyping used on its own makes sense because an ADT is defined solely through its interface with little regard for implementation.

On the other hand, inheritance may also be based on structural subtyping. This is based on the rule that "type T is a structural subtype of type T' if it provides the same attributes as type T' or attributes compatible with those of T'." It may also provide additional attributes.

Hence inheritance produces an implementation hierarchy whereas subtyping produces a behavioural hierarchy independent of implementation.

4.8.4 Subtyping and Dynamic Binding

In typeless languages such as Smalltalk, variables are not statically defined and can be arbitrarily bound to any type. Hence, it makes sense to employ dynamic binding. In strongly typed languages, variables are statically defined and so the assignment of a variable to a different type generates a run-time error. Nevertheless, dynamic binding is relevant to strongly typed languages because the inheritance hierarchy introduces substitutability. This simple idea poses a requirement for dynamic binding to allow an increase in flexibility whilst not forsaking correctness.

Khoshafian and Abnous (1990) use the following examples:

```
If T' is a subtype of T then
if X and Y are statically defined i.e.
Y : T;
X : T';
```

it is possible to assign object Y to variable X i.e.

```
X := Y;
```

and Y is dynamically being bound to X.

The dynamic binding of methods is useful in the following situation:

```
Mary : Employee;
Jill : SalesPerson;
SalesPerson is a subtype of Employee;
Mary.EvaluateBonus;
Mary := Jill;
Mary.EvaluateBonus;
```

where the method EvaluateBonus is overridden with a new implementation in the subtype SalesPerson. The identical messages (Mary.EvaluateBonus) are dynamically bound to the correct method implementation. Before the assignment the method implementation in the class Employee is invoked whereas after the assignment the method implementation in the subclass SalesPerson is invoked.

4.8.5 Subtyping Considerations for Inheritance

As shown above, inheritance allows substitutability which makes dynamic binding desirable even for strongly type languages. However, dynamic binding opens up the potential for run-time errors. The inheritance employed directly impinges on the likelihood of type errors. Subtyping rules that define when a type is a subtype of another type ensure that when substitutability is exploited no type violations are incurred. Hence, subtyping rules are useful to define the nature of inheritance to ensure a high level of correctness in an object system. Both what is inherited and in what manner it is inherited are influenced by the notion of subtyping. Bertino and Martino (1991) note that subtyping essentially effects two aspects of inheritance, overriding and multiple inheritance.

In essence, if a language is typeless e.g., Smalltalk, then it can override inherited properties arbitrarily. In Smalltalk, the types are not declared and so the greatest flexibility can be achieved. However, arbitrary re-definition does not guarantee a type safe program. Hence, if the language is strongly typed, avoidance of type errors at run-time can be achieved by introducing type and visibility constraints. There is evidently a trade-off between flexibility and type safe programming. Khoshafian and Abnous (1990) categorise the different alternatives for re-defining/overriding instance variables and methods.

At one extreme is the situation allowing no re-definition of instance variables whilst at the other is the allowance of arbitrary re-definition i.e. the type of the instance variable may be overridden by any type. The second extreme is relevant to such typeless languages as Smalltalk. An intermediate alternative is re-definition but with constraints on the allowable overridden types. This alternative uses the rule that re-defined types must be subtypes of the type in the superclass. This avoids run-time errors. For instance, inherited methods (i.e. not re-defined in any way) are applicable to the instance variables of the subclass. As the method is inherited and not overridden it will expect arguments with types as defined in the superclass. If these arguments relate to the overridden instance variables it is necessary (so as to avoid type errors) that they are subtypes of the types expected.

Yet a fourth alternative is hidden definitions as found in CommonObjects. Here the instance variables are hidden from inheriting subclasses so that independent definitions are required.

The re-definition of methods either takes the form of arbitrary re-definition or constrained re-definition. The former alternative places no constraints on the

argument types or behaviour. The latter alternative is governed by the rule that "methods in the subclass must have signatures that are subtypes of signatures they override". This rule is critical in strongly typed languages in order to avoid run-time errors. Further, any pre and post conditions associated with overridden methods must conform to the definitions as given by the superclasses.

In Smalltalk and other such typeless systems, the types of arguments are not specified and so this rule is not applicable to them. In strongly typed languages, however, the notion of subtyping must accommodate methods and the question "when is a signature a subtype of another signature?" arises.

The covariant rule although intuitive violates static type checking. It is used in some object languages such as Eiffel irrespective of the violation it may cause. The rule states that argument types and the result type of the method in the subclass must be subtypes of the arguments and the result type respectively of the corresponding method in the superclass (see subtyping rules for an explanation of the violation this causes). The contravariance rule is used to overcome this violation and avoid type errors. This rule changes the covariance rule regarding arguments by stating that argument types defined in the subclass must be supertypes and not subtypes of the corresponding arguments in the superclass.

The conclusion regarding pre and post conditions is that the pre-condition of the re-defined method should be weaker than that of the superclass whereas for a post-condition it should be stronger.

As with the overriding option, the restriction of behaviour (although not generally allowed) is also an area of concern with regard to type errors. The approach adopted to restrict behaviour in the inheritance hierarchy depends on whether the system utilises dynamic binding. Explicitly (e.g., by using the exclude construct) or implicitly (by declaring only those methods to be inherited) declaring that the method should not be inherited may lead to run-time errors. For example, if X is declared statically as type C and subsequently bound to a type C' (the subclass of C) invoking a method defined in the class C on X will cause a run-time error because it has been restricted in C'. Alternatively, the method could be overridden and merely cause the sending of a diagnostic message. Although this resolves the dynamic binding problem, it is a 'messy' solution. The former approach models more accurately what we are trying to achieve.

Bertino and Martino (1991) conclude that systems generally do not strictly adhere to the rules of subtyping. It is more convenient to provide a less restrictive type

system even though this does not guarantee that an instance of a class can always safely be used in place of an instance of a superclass. For example, O2 uses conditions that are different to those based on conformity and provides a more relaxed type system. Most object databases enforce only structural subtyping even if inheritance is based on both attributes and methods. One of the few systems that adheres strictly to the rules is VBase which uses both structural and behavioural subtyping based on conformity.

4.9 ENCAPSULATION AND INHERITANCE

Most languages only place the restriction that all access is via methods (i.e. encapsulation) to the users of the class. These are known as instantiating clients, creating instances of a class and manipulating the state of those instances only via methods. A second form of client is the inheriting client. These are subclasses of the class and inherit methods and structure from the class. Disallowing access to the first type of client whilst allowing access to the second violates encapsulation. It is the concept of inheritance that gives rise to this significant problem (Kim, 1990b). A major principle behind encapsulation is the ability to make changes to 'private' details of a class without causing any effects on the rest of the system. This is due to the rule that all access to and manipulation of data values is via the invocation of methods by message passing. However, inheritance allows direct access to instance variables of a class from a subclass. This means that methods in another class (the subclass) can be written that are dependent on the internal structure of inherited instance variables (attributes). Hence changes such as renaming or dropping of an attribute in the superclass invalidate any methods in other classes (the subclasses) that are allowed to reference that attribute. Kim (1990b) concludes the solution to be a restriction of access to only those methods defined for them. This is the reasoning behind hidden instance variables where both clients cannot directly access and manipulate the instance variables of the superclass. Methods of the subclass can no longer access instance variables directly but must invoke methods (these may be generated automatically for hidden variables). Further, Khoshafian and Abnous (1990) note that it is not necessary to incorporate both contradictory concepts, as encapsulation and overloading can be used to achieve a degree of inheritance. However, inheritance is still considered the more natural and direct approach to enable sharing of structure and code.

The visibility of instance variables can be categorised as:

- public - any client can directly access and update/invoke the instance variable;

- private - no client can directly access or manipulate the instance variable i.e. hidden;
- subclass visible - no access to instantiating clients but access for inheriting clients.

The third alternative is the most common approach. Further, the choice of whether an instance variable be public, private or subclass visible may be left to the responsibility of the designer which may be different at the different stages of development.

4.10 POLYMORPHISM

Polymorphism is defined as the ability for a value to have more than one type so that values can be used in a number of contexts demanding different types (Blair et al., 1991). Polymorphism represents both a shift towards more abstract behaviour and an increase in flexibility with the accent on what an object will do as opposed to how it will be done or implemented. Polymorphism is definitely not a new phenomena. Early forms were coercions (in-built mappings) and overloading. Today, object systems not only introduce polymorphism through subclassing but also more generally through overloading. Indeed, Garvey and Jackson (1989) establish overloading to be a main characteristic of object databases. Operator overloading was first introduced in conventional programming languages where operators such as multiply (*), add (+) and subtract (-) could be applied to a number of types such as integer and real. The above operators are said to be 'overloaded' in that they can be used by more than one type. The true meaning of the operator i.e. integer multiplication or real multiplication is resolved by inspecting the types of the operands. Overloading in terms of object databases "means that distinct methods can be given the same name for two different classes" (Garvey and Jackson, 1989). This is more general than subclassing because the meaning of a term can be overloaded in independent parts of the class hierarchy.

The user can therefore issue a more general message e.g., Display_Yourself for more than one type (class) of object. It is the responsibility of the system to inspect the message and object type and hence apply the correct implementation. The burden of choosing the correct implementation according to the type of object is removed from the user. Without polymorphism, each type (class) requires a more specific method name e.g., Display_X or Display_Y to be associated with a distinct implementation. The user must therefore ensure that the correct types are associated with the correct method name e.g., type Y used with Display_Y to avoid typing errors.

An important paper with respect to polymorphism is that by Cardelli and Wegner (1985) which includes a taxonomy of polymorphic techniques.

4.11 METHOD BINDING

Binding (in terms of object systems) has been described as the mapping from message name to the correct method implementation i.e. the resolution of levels of abstraction. It was concluded that two choices exist with respect to the timing of the binding, static binding (mappings resolved at compile time) and dynamic binding (mappings resolved at run-time). Both static and dynamic binding have advantages and disadvantages.

Static binding is the simplest of the two approaches whereby the actual calls to implementations are embedded in the code. This means that there exists no run-time overhead with this option and that binding is carried only once. Further, bindings that would cause an error can be captured from the outset. This removes the requirement to include error detection coding. However, a significant disadvantage with this approach, particularly as object systems profess to supporting the evolution of code, is that bindings cannot change without recompilation of code. This is found to be too restrictive.

With dynamic binding, mappings have to be carried out on every method invocation. Inevitably, repeated application of the search algorithm can prove expensive. This is particularly pointless if the bindings have not changed between method invocations. In addition, dynamic binding opens up the potential of 'method unknown' messages at run-time, which are essentially type errors. The code must therefore be able to handle the possibility of these errors, making it more complex. Despite these disadvantages, dynamic binding is considered the more sophisticated approach and is adopted in object systems more than static binding because of one crucial advantage, the ability to make changes without requiring recompilation from one invocation to the next. The price of dynamic binding is considered worthwhile given the increase in flexibility it provides. One of the principles that make up the object concept is the ability to allow change.

As brought out above, there is no guarantee that for every message call there will be a corresponding implementation (method code body). The search algorithm may proceed to the root of the inheritance hierarchy where an error is generated (the message 'method unknown' is returned). This is essentially a type error and has been the cause of the upsurge of interest in type checking. The result of this is a conflict in interest. On the one hand, there is the demand for correctness through

type checking, whilst on the other, there is the demand for flexibility through dynamic binding (Blair et al., 1991).

4.12 TYPE CHECKING

Type checking, when included in a language, ensures the elimination of type errors i.e. that no actions are invalid and that no type inconsistencies will result. Two areas which have the potential of yielding type errors are parameter passing and assignment. Actual parameter types must be compatible with formal parameter types. The resultant type of an expression must be compatible with the type on the left-hand-side of an assignment expression.

As with binding, a critical issue is the timing of type checking. Type checking may be carried out statically at compile time or dynamically at run-time. It must be stressed that type checking is concerned with the decision of whether an operation is valid and whether type inconsistencies would result. It is not concerned with how operations will be carried out or the code that will actually be invoked. This is the concern of binding.

Note that it is the existence of polymorphism in a system that makes binding and type checking more of an issue. If there exists a one-to-one mapping between a method name and its implementation, then the process reduces to a simple check of actual parameter types against formal counterparts and the location of the actual code body. With polymorphism, type checking ensures that an interpretation exists whilst binding determines the exact interpretation for the given type.

The separation of type checking and binding gives rise to four possible combinations although only three are valid:

- static type checking, static binding;
- static type checking, dynamic binding;
- dynamic type checking, dynamic binding;
- dynamic type checking, static binding.

4.12.1 Static Type Checking, Static Binding

At compile time, it is (both) known that a method exists and what that method is for a given object. Indeed, the fact that the method code can be linked, answers the query that it exists and hence renders type checking futile. This approach has the disadvantage of being severely inflexible. Re-compilation is necessary if the method code changes.

4.12.2 Static Type Checking, Dynamic Binding

At compile time, it is known that a method exists for a message call on a particular object but it is not linked with the actual method code until run-time. This approach is a good compromise and offers the advantage of correctness as well as flexibility.

4.12.3 Dynamic Type Checking, Dynamic Binding

It is not until run-time that both a check is made to determine whether a method code exists and a link is established to the actual method code. This approach is extremely flexible but has the penalty that run-time failures are possible The programmer needs to be aware of these and to make accommodations in the code.

4.12.4 Dynamic Type Checking, Static Binding

At compile time, a link is established to the appropriate method code. At run-time a check is made to clarify that there exists some method code. Clearly, the fact that code was found to link with the message call from the onset incorporates a type check and hence renders type checking at run-time unnecessary. This is an invalid approach and highlights the intimate relationship between the two processes.

4.13 QUERIES

There are a number of approaches to query languages in object database systems and a number of issues remain. Despite this, "most OODBs offer some form of querying capability to retrieve sub-objects from the persistent databases." (Khoshafian and Abnous, 1990). Brown (1991) identifies three categories of approaches. There are object query languages based on logic such as O_2 (Bancilhon et al., 1988), there are object query languages that are extensions of relational query languages typically SQL such as OSQL of Iris (Fishman et al., 1987) and there are object query languages which are extensions of object programming languages such as OPAL of GemStone (Ullman, 1988). A highly discussed issue with traditional databases is the fact that the application programming language and the data definition/manipulation language are based on two different models. This problem is otherwise known as "impedance mismatch". Object databases provide the opportunity to deliver a seamless database where each layer is based on the same model. This issue provides the impetus for the approach of extending object programming languages with persistent capabilities. On the other hand, the ease of use of associative languages of the relational model such as SQL provides a resistance to return to the navigational style languages of the network and hierarchical database era. In addition, it would be beneficial to a have

a standard object query language in the same way as SQL has become the standard relational query language. Indeed, standards have emerged. The standards committee have worked on the development of a standard object query language, SQL3 (Ullman and Widom, 1997) and the Object Database Management Group (ODMG) have defined an Object Query Language (OQL) to support their object model (Cattell, 1996). SQL3 is intended to keep all the features of ANSI SQL2 and to incorporate a subset of OQL.

An object query language must accommodate the differences the object data model brings compared with traditional data models. In particular, Kim (1991) pinpoints the following concepts, object identity, complex objects, the inheritance hierarchy and methods as impacting on the requirements of an object query language. An object query language should have the facility to distinguish between value and object equality. The class-composition hierarchy means that an object query language must accommodate predicates on nested attributes. The inheritance hierarchy means that the object query language must allow for queries to be constructed against "a single class or a class hierarchy rooted at the class" (Kim, 1991). A query which interprets the target class to be a single class is a weak query whereas a query which interprets the target class to be the class and all its classes on the class hierarchy rooted at that class is known as a strong query. An object query language must allow a query to be made up of derived attribute and predicate methods. A derived attribute method is similar to an attribute in that it returns objects but unlike an attribute which holds a value, a derived attribute method uses a program to return values based on other objects in the database. A predicate method returns the Boolean constants True or False and so can be used in a Boolean expression in a query.

This section has merely touched on some of the issues, approaches and requirements of object query languages. There is still much research in the area and so it is considered outside the scope of this thesis. For a more in depth look at object query languages, Brown (1991), Catell (1991) and Kim (1991) provide some excellent chapters on the issues, approaches and requirements respectively.

4.14 PROTOTYPE SYSTEMS

Although the class concept is considered an important feature, it is not necessary that object systems support it. This also impinges on the notion of inheritance. If classes no longer exist then class inheritance is not applicable. An alternative approach exists based on prototypical objects. Systems based on prototypical objects are known as prototype systems and use a process known as delegation. A

prototype is an individual object containing its own description. Generation of a new object is based on an existing object with possible modifications to its attributes and behaviour. There are two important differences between prototype systems and systems based on classes. Firstly, class inheritance implies inheritance of representation and behaviour whereas in prototype systems inheritance (delegation) includes operations and state i.e. the actual values. Inheritance of state is not completely ignored by class based systems, however, due to the inheritance of class variables. Secondly, the relationship 'delegates_to' can be formed dynamically whereas in class based systems inheritance relationships are established at the same time as the class is created.

The approach taken depends on the type of application as both have advantages to offer for certain situations. Prior to the arrival of a resolution at OOPSLA '87 in Orlando, known as the Orlando Treaty, this was an issue hotly contested. Groups from both sides believed their approach, because it could be employed in such a way as to model the alternative approach, to be more superior. The Orlando Treaty concluded that, in fact, the two approaches were valid and that each approach was more suited to certain types of programming situations than the alternative. However, most object databases have adopted the instantiation approach.

4.15 STANDARDS

The lack of a standard definition on the precise meaning of object-orientation is considered a fundamental contributor to the high level of confusion that exists in the area as cited by Kim (1990b). However, there are common core features to be found among the various systems that all tend to support. Furthermore, there is evidence of convergence of ideas and there exists a research effort dedicated to this goal. Specific groups of notable people in the area (M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik) are consciously considering the move to agreed terms. A result of this overt attempt to consolidate concepts into a solid agreed definition is the publication of the paper "The object-oriented database system manifesto" (Atkinson et al., 1989). More recently, there has been a move towards industry consensus with the formation of the Object Database Management Group (ODMG). This group "is a consortium of object-oriented database companies" whose technical representatives include Tom Atwood and Mary Loomis and "whose goal is to establish agreement for object-oriented database technology industry-wide." (Cattell, 1996).

4.16 CONCLUSION

This chapter has covered the complexities of the object data model. This thorough analysis was necessary to allow an understanding of its impact on the performance of object databases. It is anticipated that the differences with respect to previous models provides opportunities for an improved arrangement of data on disk. The next chapter will take on board the concepts of the object data model and review their implementation in terms of storage and the affect on performance.

CHAPTER FIVE

PHYSICAL DESIGN FOR OBJECT DATA MODELS

Chapter four has highlighted the complexities and richness of the object data model. Specifically, it has identified the following key characteristics:

- encapsulation;
- message passing paradigm;
- classes;
- object identity;
- complex objects;
- inheritance.

In order to understand how the performance of an object database is affected, this chapter needs to consider how these logical characteristics are actually implemented i.e. the object physical model. It is anticipated that the complexities of the object data model and the differences with respect to traditional data models will impact the physical model and storage management strategies of object databases. Figure 5.1 shows how these concepts are inter-related.

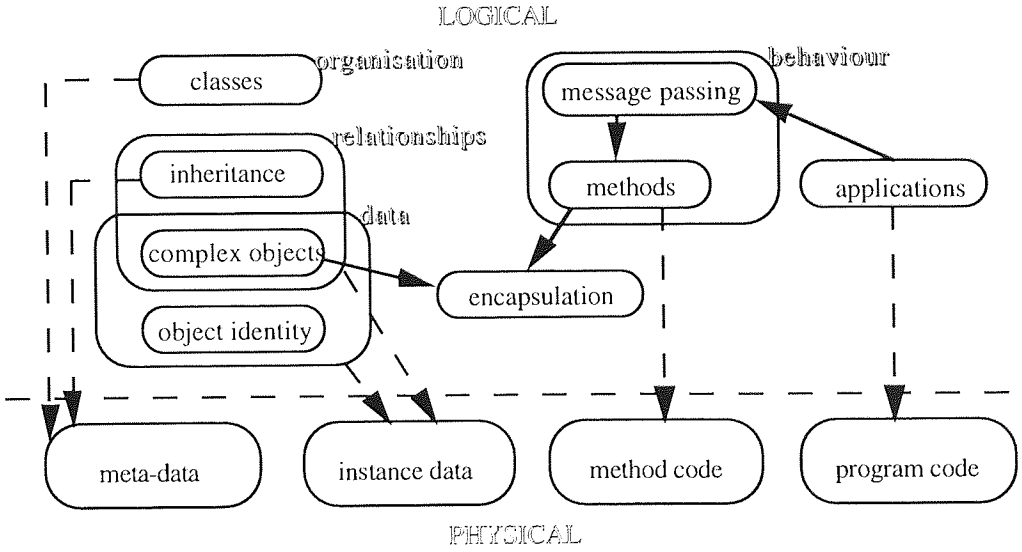


Figure 5.1 The relationship between logical and physical concepts of the object model.

In chapter one, the storage of the object data model, specifically the placement of data items on disk, has been identified as a potential source of improvement and consequently it is the focus of this thesis. Considering the object data modelling concepts, the message passing paradigm is a behavioural, dynamic aspect of the model and is not concerned with storage. Encapsulation implies the mode of access, specifically that data can only be accessed through prescribed methods. The concepts of encapsulation and classes are a means of organising the object space,

overriding the actual data content. The two concepts combined together provide an immediate improvement in performance. Similar objects belonging to a class may have common methods factored out and stored once. Storing method code once significantly reduces the volume of data so the issue of whether to replicate the code for each object can be disregarded. Classes provide an obvious performance clue. The fact that objects belong to the same class does not mean that they are necessarily stored adjacent to one another. However, storing objects belonging to the same class in the same place on disk is overtly sensible because queries against a database typically manifest themselves to be against a whole class of objects rather than just one object. The concepts of encapsulation and classes do not provide any more pointers to the arrangement of data on disk. On the other hand, the concepts of inheritance and complex objects, which are more directly concerned with the relationships between objects (instances and classes) may provide more intrinsic clues to the performance of databases. Moreover, by using the tools of inheritance and complex objects on some real world data yields a myriad of relationships that in turn require implementing as an actual physical model. Further, when the implementation of encapsulation and classes is considered against the implementation of inheritance and complex objects there is a difference in the volume of information to be represented. Information to represent encapsulation and classes may be considered to be meta-data and is of a smaller volume to the actual relationships and data. Inheritance and complex objects will therefore be reviewed below. Object identity impacts performance if queries are against specific objects as opposed to a class of objects. Object identity is also intimately linked to the concept of complex objects. The implementation of object identity varies from database to database and so it is included in the next section as a matter of interest.

The three fundamental characteristics of the object model that have significant impact on performance are therefore:

- object identity;
- inheritance, class hierarchy;
- complex objects.

The following sections will take these logical characteristics in turn and consider how they are represented at the physical level in object databases.

5.1 OBJECT IDENTITY

Khoshafian and Copeland (1986) advocate the use of surrogates to represent object identity as opposed to the use of physical addresses or values. Both physical addresses and values compromise identity. Physical addresses are not location

independent and values are not data independent. Surrogates on the other hand are globally unique identifiers generated by the system and are independent of how an object is accessed, what it contains and where it resides i.e. they are fully independent (Khoshafian and Abnous, 1990).

However, identity at the conceptual level and identity at the physical level need to be distinguished. The criticism by Khoshafian and Copeland (1986) that addresses and values make poor identifiers is largely levied at the application, end-user level. It is desirable for the higher levels of software to support surrogates. The object identifier is then isolated from any changes in an object's state, location or access route as far as the user is concerned. Despite the above advantages, there are benefits from having identifiers that are structurally dependent. For example, in ORION the system generated, unique identifier is made up of a <class identifier, instance identifier> pair. This enables the system to fetch the class object (so that it can check for the existence of the method being applied) prior to fetching the object. This eliminates unnecessary object fetches.

Surrogates may or may not be implemented at lower level storage modules. Identity may be implemented using physical addresses, structural names etc. For example, EXODUS uses an address of the form (volume#, page#, slot#, unique#) for its storage objects (Carey et al., 1988). However, Iris (Fishman et al., 1987), ObServer (Skarra et al., 1991) and GemStone (Maier and Stein, 1987) use unique surrogates. In particular, object identifiers in ObServer (UIDs) are 32-bit quantities allocated sequentially from a free list. Using surrogates requires an object table to map from the surrogate to the physical location of the corresponding object. However, the achieved level of indirection shields higher layers of software from underlying changes. On the other hand, using physical addresses as in EXODUS is more efficient. "Besides, a surrogate index can be implemented above the storage manager level for applications where surrogate support is required" (Carey et al., 1989). Indeed, in ObServer a server uses physical addresses to identify objects and an interpreter layer on top of the server uses logical object identifiers. A principle function of the server therefore becomes the maintenance of the correspondence between UIDs and chunks of memory (Hornick and Zdonik, 1987).

Tuple identifiers have been used internally in relational systems (Khoshafian and Copeland, 1986) e.g., System R (Stonebraker et al., 1976), INGRES (Astrahan et al., 1976) and WiSS (Chou et al., 1985). Although these are not fully structurally independent, they can be used to implement identity.

An additional level of indirection is incorporated into systems supporting replication. Here, an external OID maps to a number of internal OIDs, one for each copy of the same object. This is implemented in ENCORE (Hornick and Zdonik, 1987).

5.2 INHERITANCE HIERARCHY

The class hierarchy models is-a relationships between a set of classes. Semantically, this means that an instance of a particular class is also an instance of all its superclasses. For example, an object of type Manager is also of type Employee and type Person (see figure 5.1).

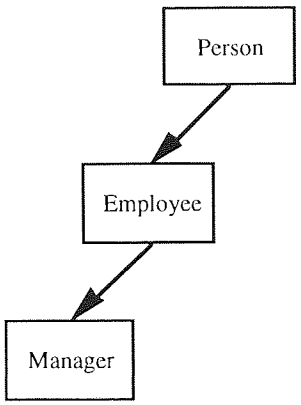


Figure 5.2 Part of an inheritance hierarchy.

However, there are two approaches to implementing an object's presence in an inheritance hierarchy. Yaseen et al. (1991) describe these approaches as the static storage model and the distributed storage model. Chan et al. (1982) also distinguish between complete grouping and semantic grouping.

In the first approach, objects are contained within only one class and within that class an instance contains all direct and indirect (i.e. inherited) attributes. For example, in ORION (Banerjee et al., 1987)) and in OZ (Weiser and Lochovsky, 1989) objects can belong to only one class. Henceforth, this strategy will be known as the Collected Storage Model (CSM). The second approach allows an object to be contained within more than one class. An object's structure can therefore vary and each instance takes on only those attributes defined (not inherited) in the class in which it resides. An object is therefore regarded as a number of instances distributed over a number of classes of a class hierarchy. Henceforth, this strategy will be known as the Distributed Storage Model (DSM) following Yaseen et al. (1991). For example, an object in ENCORE (Hornick and

Zdonik, 1987) is an aggregate of all instances that belong to it from a number of classes in a class hierarchy. If Toyota is a subclass of Car, Car is a subclass of Vehicle and Vehicle is a subclass of Object and there is a given Toyota object X, then in ENCORE there would be a chunk of storage for the part of X that is an instance of Toyota plus another chunk of storage for the part of X that is an instance of Car etc. A Toyota object is therefore an instance of four types, Toyota, Car, Vehicle and Object. GemStone also allows an object to be contained within more than one class because its implementors believe this to be a factor that differentiates it from traditional models (Stein and Maier, 1991). Indeed, even though Oggetto does not yet support multiple membership, it considers it to be a needed future extension (Mariani, 1992).

Most systems adopt the collected approach because it is easier to implement. The distributed approach must accommodate instances of variable length because each instance must specify a list of all classes containing its other instances (Banerjee et al., 1987).

In either case, the chosen implementation is a separate issue to how the semantics of a class hierarchy are incorporated into the operations on a class. The target class of the operation may be interpreted as the class itself or as the class hierarchy rooted at that class. For example, an operation to 'Get all Persons' may be taken as 'Get all objects of the class Person' or 'Get all objects of the class Person and the class Employee and the class Manager' because an Employee is-a Person and a Manager is-an Employee which in turn is-a Person. Therefore, if an operation is applied to a target class, then the latter (stronger) interpretation requires the system to apply the operation to each and every class on the class hierarchy rooted at that class. In the former (weaker) interpretation, the operation need only be applied to the target class. When operations are applied with a strong interpretation and the distributed storage model is employed then the system still only needs to look at the one class, the target class, if the operation is only interested in attributes belonging to the root of the hierarchy. This is indeed the case as the root of the hierarchy is the target class in the first instance.

5.3 COMPLEX OBJECTS

To implement the class-composition hierarchy, simple attributes and complex attributes are distinguished. Simple attributes are typically represented directly whereas an object identifier is used to reference an instance of a user-defined class for complex attributes. This means that the majority of systems do not assign identities to base objects i.e. those of integers or strings. In addition to the simple

and complex attributes, an object may also contain variable length, set or large attributes. To accommodate all these variations, a complex object is typically implemented by a separate fixed length and variable length portion, with the variable length portion kept at the end of the complex object so that it can grow. There may also be a header portion containing control information. The fixed length portion contains an entry for each attribute of the complex object. The entry for simple attributes is the scalar value itself whereas the entry for other attributes is a reference, pointer, offset or descriptor. For example, Kim et al. (1988) describe an implementation of a complex object where the fixed length portion contains either the scalar value itself, a variable length attribute descriptor consisting of a length, offset pair or a set attribute descriptor consisting of a count, pointer pair. Set instances are represented by a linked list. The implementation of a complex object by Khoshafian et al. (1990) includes a header portion containing the length, type and bitmap of the complex object. The bitmap shows the presence or otherwise of each attribute that makes up the complex object. Figure 5.3 shows the implementation of a complex object.

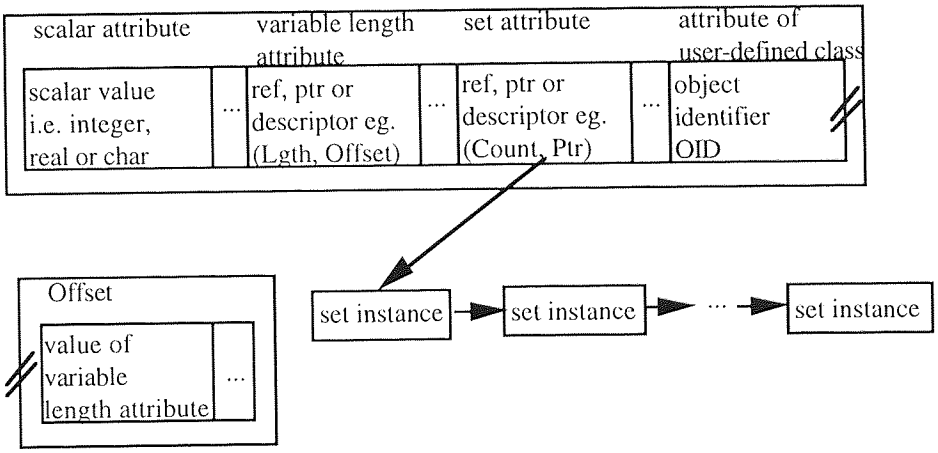


Figure 5.3 Implementation of a complex object *after* Kim et al. (1988) and Khoshafian et al. (1990).

Object references within complex objects may be internal references, for example, a relative page number within a logical address space. Therefore, there may be a one-to-one correspondence between identifiers and objects but a many-to-one correspondence between identifiers and references. Full object identifiers are only necessary for referring to objects outside a logical area such as a segment (Zdonik, 1990).

An instance of a complex object becomes a collection of records. To access the complex object, one record must be assigned as the root record (Kim et al., 1988).

Root records of the same complex object type are placed in a separate file to improve associative searches. Non-root records of complex objects are therefore clustered.

5.4 STORAGE MANAGEMENT

In addition to the mechanisms to physically implement the logical concepts of the object model, the object space can be further augmented by storage management techniques so as to improve performance. The real issue here is whether the concepts can offer more alternatives with respect to the techniques which have already been exploited by traditional models and as discussed in chapter two. In particular, the class hierarchy and the class-composition hierarchy give rise to more alternatives for both indexing and clustering. Clustering aims to store related objects or records (i.e. those that are likely to be accessed together) physically close on disk. Before clustering is considered in isolation, it is useful to explore how it is related to other issues in the wider problem of database design. Following this discussion, a review of the current state of the art with respect to clustering will be given and then an attempt made to provide some organisation to the alternatives.

5.4.1 The Database Design Problem

Clustering aims to collect together physically on disk, objects, sub-objects, or instances that are likely to be accessed at the same time in order to improve performance. Improvements in performance are achieved by bringing data into the buffer that will satisfy the current and future requests. Conversely, partitioning data from data that are unlikely to be accessed together also positively impacts performance. This ensures that fewer blocks of data are required to service a particular series of requests. Clustering (aggregating) and partitioning (segmenting) are therefore interrelated issues. March (1983) explores both aspects as belonging to the same issue.

Indeed, given a set of classes on a class or class-composition hierarchy, there are many variants or degrees of clustering and the process is really a mixture of clustering and partitioning. There exists a range of alternative clustering strategies with complete clustering (all classes are clustered in the same segment) at one extreme and no clustering or complete partitioning (all classes are partitioned into separate segments) at the other. The set of possible alternatives increases and becomes more complex as the number of classes belonging to a hierarchy increases. The database design problem is therefore concerned with finding the optimum

balance of clustering for any specific class or class-composition hierarchy in order to achieve an efficient system.

More alternatives exist if a 'child' class has more than one 'parent' class. This occurs both in the class hierarchy and the class-composition hierarchy. Multiple inheritance allows a subclass to have more than one 'parent' superclass and within a complex object, sub-objects may be shared by more than one 'parent' object. This presents more choices with respect to clustering strategies and becomes more complex as the number of parents per child increases. For example, if a 'child' class has two 'parent' classes, the choices are:

- cluster the child with parent one;
- cluster the child with parent two;
- cluster the child with either parent one or parent two;
- cluster the child with both parents i.e. replicate the child class;
- cluster the parents with the child object.

However, if replication is not allowed because there is no strategy in place to control it, sub-objects cannot be physically stored with each of its referencing objects (Zdonik, 1990). In fact, Kim (1991) states "at most one composition hierarchy can be chosen as the basis for clustering". ENCORE resolves this problem by supporting replication (Hornick and Zdonik, 1987).

The database designer must choose from a large set of alternative strategies which is made more complex by the object model. The choice is dependent on the nature of the queries put to the database. For instance, if queries only tend to refer to attributes two levels deep, then it is desirable to cluster only to this level. Further, a class may be the root of a number of class-composition hierarchies. However, they may not all be relevant to the queries that are put to the database and so it is desirable to cluster only those branches that are more likely to be useful. Banerjee et al. (1988) suggest that the three alternative clustering strategies for complex objects, depth-first, breadth-first and children-depth-first directly suit three types of traversal known also as depth-first, breadth-first and children-depth-first respectively. In general, the nature of the access patterns can be clarified by determining the type and frequency of each request put to the database. However, this often exposes conflicting requirements. One solution is to choose the strategy that is most suitable for the dominant user. This approach is far from ideal as the database often serves a community of users (March, 1983). There exists an overdue need for useful tools to aid the design process particularly as application domains become more complex. March (1983) developed a hierarchic aggregation algorithm intended for finding the optimum clustering/partitioning scheme for

hierarchical models. This could provide insight into the clustering of hierarchies found in the object model.

5.4.2 Clustering Strategies: A Review

A clustering strategy can be described essentially along two main dimensions, "how to cluster?" and "when to cluster?". The dimension of "how to cluster" is concerned with the relative placement of objects (instances, classes, relations) using constructs of the logical model as a guide. The dimension of "when to cluster" refers to the timing of the actual clustering of objects, before or after run-time. It must be noted that Bertino et al. (1994) also review clustering strategies with respect to 'how' and 'when' but in addition consider three other dimensions, namely clustering algorithm controls, clustering granularity and system performance evaluation. The dimension of clustering algorithm controls is concerned with the type and source of information utilised by clustering algorithms. There are two issues, whether it is a human or system function and whether the information is obtained statically by analysing the data model and method code or dynamically by monitoring query behaviour. It must be noted that this thesis is concerned with the link between the analysis of the conceptual model and the resulting clustering strategy. Clustering granularity is concerned with the mapping of logical to physical clusters taking into account storage constraints and the level of clustering offered by a particular system, page or segment level. System performance evaluation is concerned with the parameters which affect performance and the performance indices. However, the dimensions 'how' and 'when' to cluster represent the very essence of a clustering strategy whereas controls and granularity represent finer points. The evaluation of performance is not even considered a dimension at all because it is not felt to be a description of a clustering strategy but an overriding factor. Therefore, for the purpose of this thesis, the two pivotal elements of a clustering strategy, 'how' and 'when', will be reviewed separately in the following sections whereas the aspects of control, granularity and performance evaluation will not be considered further.

How to Cluster?

Kim (1991) cites five general alternative strategies for clustering in object databases, two of which can be found in the relational model.

In the relational model, tuples belonging to a single relation may be clustered in consecutive blocks ordered on some chosen attribute. In a similar manner, *objects from a single class may be clustered.*

To facilitate joins, tuples from different relations may also be clustered according to a common attribute. In this approach, 'children' tuples directly follow the corresponding 'parent' tuple where the values of the join attribute match. In a similar manner, *any two or more arbitrary classes which reference each other (via object identifiers in the object model as opposed to values in the relational model) can be clustered.* The object identifier in the referencing attribute is clustered with the corresponding object with that object identity. Any number of classes can be clustered because the referenced object may also reference other objects.

The above strategy may be applied to arbitrary classes. However, the mechanism of referencing objects belonging to other classes is used to form complex objects. Such referencing reflects the semantics within the application domain and so intuitively access to the database is likely to be along such class-composition hierarchies. It therefore makes sense to *cluster objects belonging to classes of a class-composition hierarchy.* Indeed, object databases should support access to complex objects as a whole and appropriate clustering is a key mechanism in the support of this facility (Deppisch et al., 1991). The requirement is even stronger for composite objects as dependent objects are often required at the same time as the root object (Banerjee et al., 1987).

When sub-objects of a complex object are iteratively clustered along a class-composition hierarchy, there are a number of alternative approaches. Khoshafian and Abnous (1990) describe two different approaches called depth-first and breadth-first that are available in GemStone. The difference between the two approaches is that sub-objects may or may not be clustered directly within its 'parent' object. In the depth-first approach, sub-objects directly follow the attributes that reference them. In the breadth-first approach, the sub-objects are placed after the complete 'parent' object but before the next 'parent' in the same 'parent' class (i.e. its 'uncle'). A Person instance could be clustered in GemStone in depth-first order:

```
clusterPersonDepthFirst
   self cluster.
   name cluster.
      name first cluster.
      name middle cluster.
      name last cluster.
   age cluster.
   address cluster.
      address streetNum cluster.
      address streetName cluster.
      address State cluster.
```

or, in breadth-first order:

```
clusterPersonBreadthFirst
   self cluster.
   name cluster.
   age cluster.
   address cluster.
      name first cluster.
      name middle cluster.
      name last cluster.
      address streetNum cluster.
      address streetName cluster.
      address State cluster.
      . . .
```

Banerjee et al. (1988) also describe three clustering approaches with respect to complex objects or class-composition hierarchies, depth-first, breadth-first and children-depth-first. However, the depth-first and breadth-first descriptions do not compare with those described by Khoshafian and Abnous (1990). Banerjee et al. (1988) represent a class-composition hierarchy as a graph. For example, in figure 5.4 letters depicting nodes of the graph represent the classes of the class-composition hierarchy and arrows represent the references between classes.

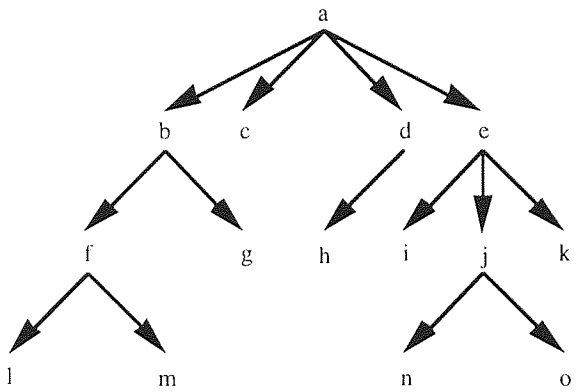


Figure 5.4 Representation of a class-composition hierarchy.

To map the graph to a physical arrangement requires the 'flattening' of the structure into a linear ordering of nodes. The three approaches differ with respect to the order of such nodes. The depth-first approach orders the nodes a branch at a time whereas the breadth-first approach orders the nodes a level at a time. For example, a depth-first clustering sequence of figure 5.4 is:

a b f l m g c d h e i j n o k

and a breadth-first clustering sequence is:

a b c d e f g h i j k l m n o

The children-depth-first approach is more complicated with all of a node's 'children' being placed as soon as possible after the node's 'brothers'. This is applied recursively so that given a node, the 'children' of its 'children' are placed before the 'children' of the next 'brother' of the node. For example, a children-depth first clustering sequence of figure 5.4 is:

a b c d e f g l m h i j k n o

Note that only when the tree structure is at least three levels deep and the leaf nodes are at different levels, is there a difference between the breadth-first and children-depth-first arrangements.

The class hierarchy leads to the fourth clustering strategy and is useful when a system employs a strong interpretation of the is-a relationship between classes. *Objects belonging to classes of a class hierarchy may be clustered.* However, this strategy only truly makes sense if objects are represented by multiple instances along a particular class hierarchy. This strategy is therefore a way of reconstituting the object. In fact, the DSM approach vertically partitions classes and class hierarchy clustering reverses that process. An example is in Chan et al. (1982) where "in general we allow multiple storage records representing the same underlying entity to be clustered together". The clustering is achieved by placing instances from different classes on a class hierarchy with matching object identifiers physically close on disk. This clustering is implicit in the CSM approach. In this case, it is only possible for all instances of one class to physically follow all instances of its superclass.

Class hierarchy clustering reverses the vertical partitioning that has taken place when an object has been decomposed into a number of instances along a class hierarchy. In addition to this, clustering can be combined with horizontal partitioning. In each class, the instances can be horizontally partitioned on a subclass basis. The instances of a subclass can then be clustered with their corresponding instances of its superclass. For example, Person records can be partitioned into instances where the Person is an Instructor and instances where the Person is not an Instructor. The Instructor instances of the Instructor subclass can then be clustered with the appropriate Person partition. This enables all information relating to Instructor entities to be readily accessible (Chan et al., 1982).

The fifth strategy is a hybrid approach combining the third and fourth strategies. A class may simultaneously be part of a class-composition hierarchy and a class hierarchy. *Objects belonging to classes of both a class hierarchy and class-*

composition hierarchy may be clustered. More precisely, it is possible to have sub-objects clustered with their root 'parent' object in addition to the clustered complex object being clustered with its corresponding clustered complex objects of its superclasses and/or subclasses.

The above strategies use knowledge of the object model to deduce the nature of clustering. An actual clustering arrangement can be produced by applying a clustering algorithm to a set of objects. These require input data indicating the priorities of object references which can be inferred from the above strategies. There are two types of clustering algorithms, sequence-based and partition-based (Kemper and Moerkotte, 1994). Sequence-based algorithms produce an object sequence from a set of objects according to a chosen traversal method. Link weights are utilised to reflect priorities between references when there exists a choice of paths to follow. Partition-based algorithms propose a mathematical approach, using link weights to form an optimisation problem.

When to Cluster?

Both static and dynamic clustering are supported in current object databases. Static clustering is specified once when the object is created and is fixed despite changes in access patterns. In systems with dynamic clustering, objects are continuously re-clustered to best suit current access patterns. Further, with dynamic clustering, re-clustering may take place at transaction commit time or at an arbitrary point in time (Kemper and Moerkotte, 1994).

The majority of systems support the more convenient approach of static clustering by taking hints from the DBA (Data Base Administrator) or user typically when an object is created. For example, in EXODUS it is possible to "place the new object near the object with id X" (Carey et al., 1988). EXODUS then places the new object in the same segment as the object with object ID X. Clustering in GemStone (Khoshafian and Abnous, 1990) and VBASE (Andrews and Harris, 1987) is also on a per object basis. For example in GemStone to cluster MyObject with previously declared objects requires,

```
MyObject cluster
```

Kim (1990b) notes that it is unfortunate that clustering relies on the specification of the sub-graph of the schema by users, due to the fact that it is expensive to cluster and de-cluster dynamically a set of classes in one segment for object and relational databases alike. Kim (1991) notes that to cluster a set of classes along a class

hierarchy or class-composition hierarchy requires the user to define the scope. In particular, for the user to define the scope, the following must be defined:

- root of desired class hierarchy;
- root of class-composition hierarchy;
- leaves of the class hierarchies.

Dynamic clustering is supported by Cactis (Brown, 1991). Statistics are collected by the system with regard to object retrieval patterns to enable re-organisation when necessary.

5.4.3 Clustering Strategies: A Topology

The various clustering strategies can be arranged into a classification of figure 5.5. At the top level is the strategy of no clustering. This means that the simple attributes are distributed across the database. At the next level, the simple attributes and references that make up a single object or record may be clustered i.e. aggregated. This can be extended to the clustering of all objects or records that make up a single class, table or relation. Clustering strategies labelled D, E, F, G and FG correspond to the clustering strategies described by Kim (1991). Clustering two or more classes belonging to either a class hierarchy or class-composition hierarchy can be thought of as a specialisation of clustering two or more arbitrary classes. The classification also reflects the fact that the clustering of a class and the clustering of a class-composition hierarchy can be combined (clustering strategy FG corresponds to the fifth strategy described by Kim (1991)).

Banerjee et al. (1988) and Khoshafian and Abnous (1990) both describe clustering of the class-composition hierarchy but they are essentially at different levels. Class-composition hierarchies can be viewed at the class/type level or at the object/instance level. Bertino et al. (1994) make use of graph schemas in order to clarify the difference. A class level graph (Type Graph in Bertino et al. (1994)) shows the relationships between classes i.e. types and an object level graph (Composition Object Graph in Bertino et al. (1994)) shows the relationships between individual objects. For example, the class level graph of Person may look something like figure 5.6 whereas the object level graph may look something like figure 5.7. Figure 5.6 and 5.7 show that a Person class is made up three attributes where one attribute (Age) is a simple value and two attributes are actually references to objects/instances belonging to other classes (Name and Address). It is necessary to be aware of these different levels when analysing clustering.

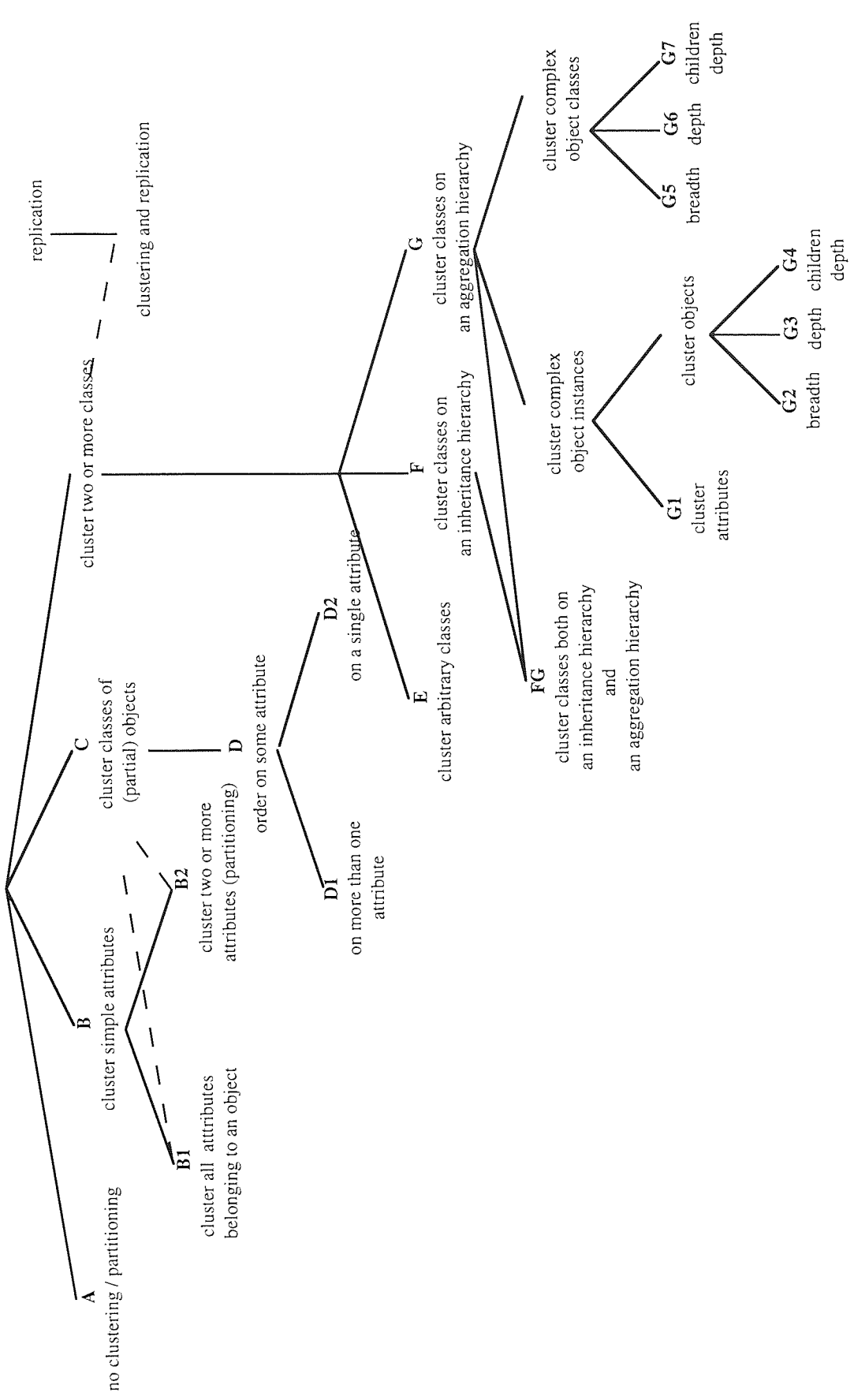


Figure 5.5 A topology of clustering strategies.

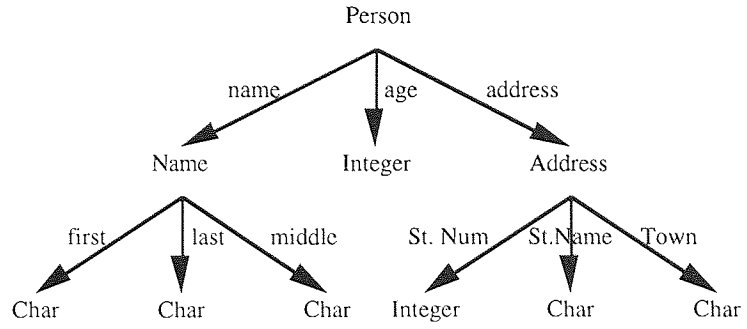


Figure 5.6 Class level graph for Person.

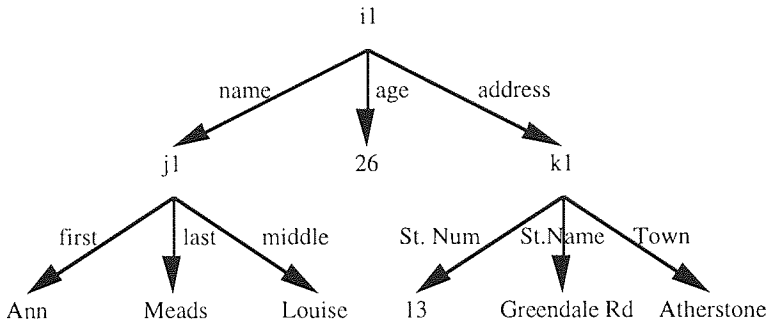


Figure 5.7 Object level graph for Person.

A single complete complex object may be clustered. At a low level, there is the clustering of sub-objects within its 'parent' object (attribute clustering). This corresponds to the depth-first approach described by Khoshafian and Abnous (1990). It is assumed that sub-objects two levels deep are also clustered within the top level root object directly following their reference within the first level object. For example, the linear arrangement of the instance level graph of Person would be as in figure 5.8. In effect, references to sub-objects become redundant in this arrangement as they are replaced iteratively with the actual objects.

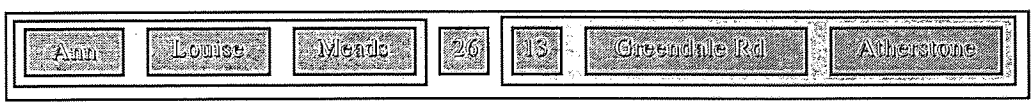


Figure 5.8 Attribute clustering for Person.

At the next level, a single instance of a complex object may be clustered but without the 'embedding' of sub-objects within their parents. This corresponds to the breadth-first clustering strategy described by Khoshafian and Abnous (1990). However, there is still a question of the ordering of sub-objects that follow a 'parent' object. Khoshafian and Abnous (1990) do not explore the alternatives past

a single level of objects. These can be arranged in depth-first, breadth-first or children-depth first order. For example, consider the extended class and object level graphs of Person as in figure 5.9 and 5.10 and the possible linear arrangements in figure 5.11. It is not clear whether Banerjee et al. (1988) are referring to these arrangements. Note that with this example, as there is no difference between the level of the leaf nodes, the breadth-first and children-depth-first arrangements are the same. Indeed, it is not typical for structures in the general business domain to go to many levels. The subtle difference between the two arrangements may therefore be more applicable to more complex, previously unsupported domains such as CAD and CAM.

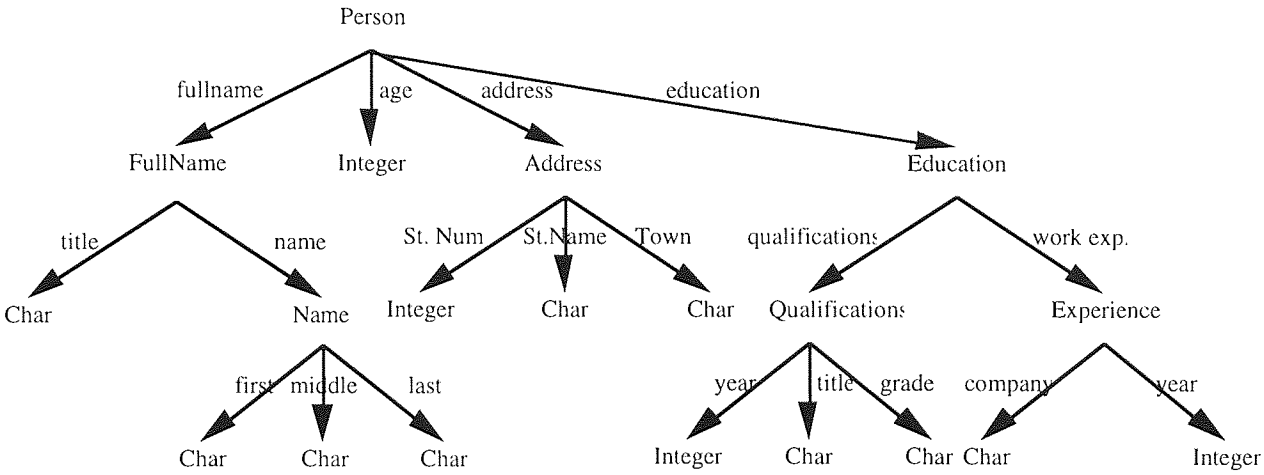


Figure 5.9 Extended class level graph for Person.

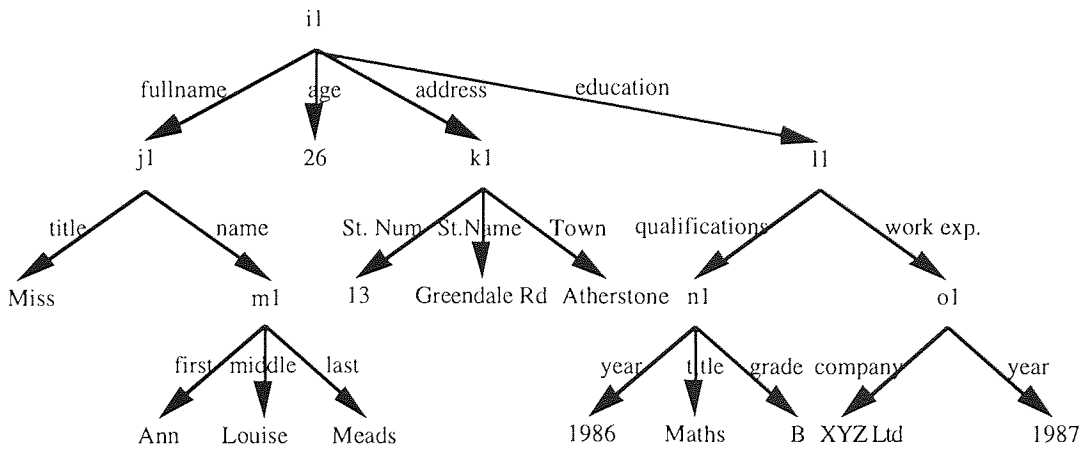


Figure 5.10 Extended object level graph for Person.

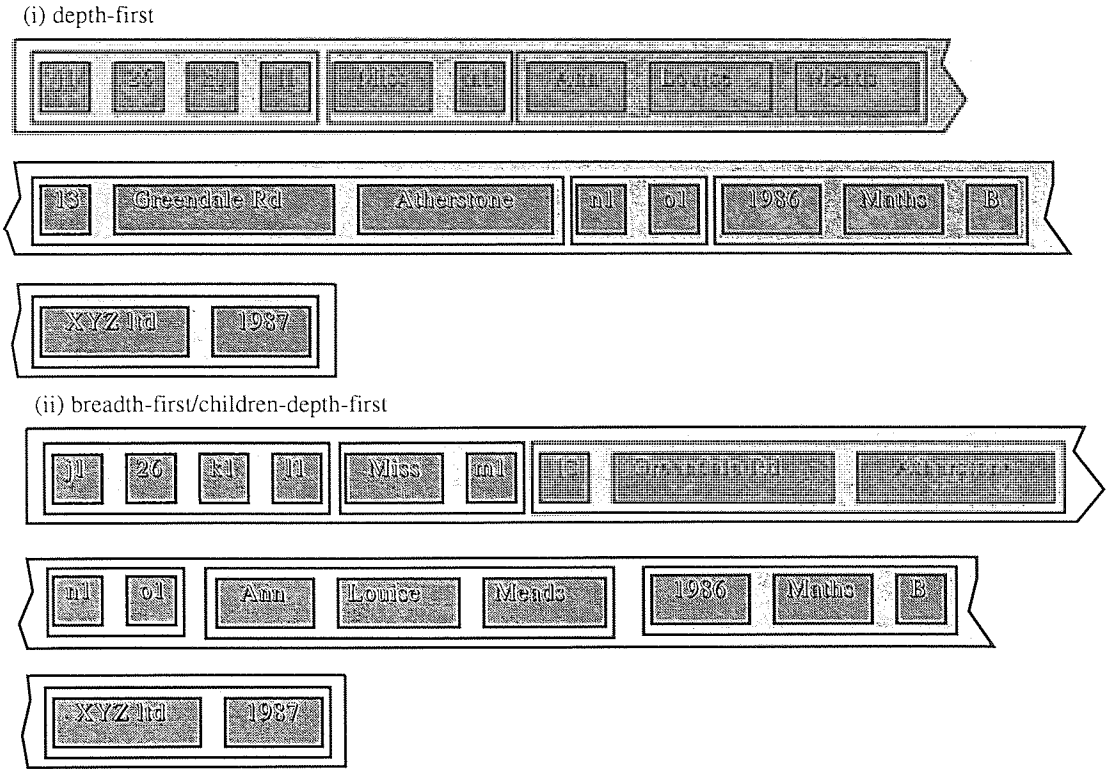


Figure 5.11 Different clustering strategies for Person at the object level.

At a higher level, the classes in their entirety may be ordered in some linear sequence. Each class may also be clustered on one or more attributes within this type of clustering. It is assumed that the three approaches described by Banerjee et al. (1988) refer to the different ways classes may be ordered. For example, consider the Person class-composition hierarchy again (see figure 5.6). Now imagine that there exists a number of objects/instances for each class on that hierarchy. For example, the system has a number of Person objects (i1, i2, ... in), a number of Name objects (j1, j2, ... jn) and a number of Address objects (j1, j2, ... jn). Class clustering is arranging all Person, Name and Address objects in their own segments on disk. It is assumed that object systems store each separate class (i.e. all objects belonging to a particular class) in a separate segment on disk unless stated otherwise. This implicit clustering is similar to relational systems whereby each relation is stored in a separate file on disk unless stated otherwise. However, the arrangement of the segments would be determined by the creation sequence and the availability of physical segments. Different class clustering strategies could be specified such that the arrangement of classes in their entirety could be depth-first, breadth-first or children-depth-first (see figure 5.12).

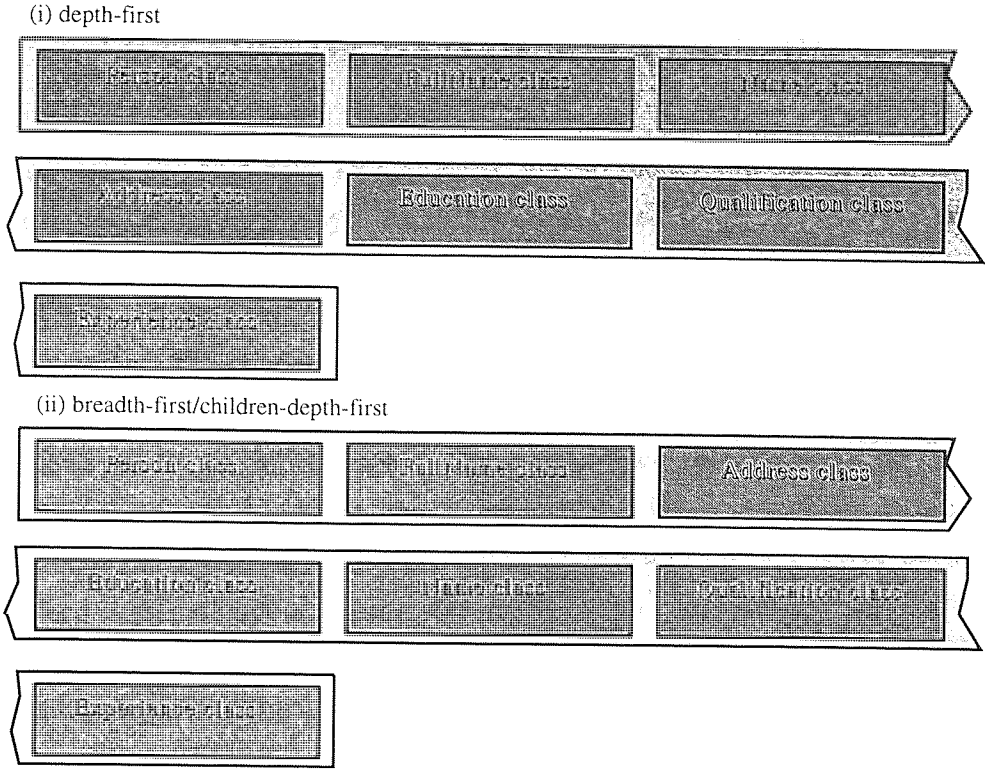


Figure 5.12 Different clustering strategies for Person at the class level.

5.5 CONCLUSION

This chapter has shown that object databases have addressed the physical storage issue to some extent. Although a large number of clustering strategies have been identified and discussed in the literature, there exists no proper framework and object databases have appeared to apply those strategies with a rather adhoc and opportunist approach. This chapter has helped to bring together the various ideas on clustering strategies. It is felt that the full potential of clustering based on a deeper analysis of the rich relationships of the object model has not yet been exploited. The next chapter will provide a detailed analysis of the relationships of the object model with a view to gaining more insight into clustering strategies for object databases.

CHAPTER SIX

ANALYSIS OF OBJECT RELATIONSHIPS

Chapters four and five have looked at the logical model and physical model for object databases respectively. This chapter and the next will bring these two concerns together. In order to bring together logical and physical concerns, it is necessary to consider the mapping of logical structures to possible physical implementations. The transitions required to map from a conceptual object model to a logical object model and finally to a physical object model are not adequately covered in the literature. In fact, the literature discusses conceptual structures, namely inheritance and complex objects extensively. It is accepted that in the early days of object databases, an understanding of the semantics behind the object model has indeed been paramount. Before the object model can be taken on board, it has been necessary to educate users, particularly as it follows a radically different procedural model of programming and a more simplistic relational database model. However, irrespective of the complexity or simplicity of the chosen model, all data must eventually be represented as 'bits and bytes'. It is time to move on and consider the implementation of such structures in more depth. This thesis will contribute to a better understanding of those relationships.

Before the implementation of structures can be achieved, it is necessary to explore all the possible types of conceptual structure. This chapter takes into account many factors in order to deliver a finite set of possible structures and an attempt is made to fit each structure to a real world example. The next chapter will then consider the possible implementations of such structures in terms of clustering.

The literature, quite correctly, advocates the need to keep conceptual structures independent of physical structures during the development process. However, in reality the database designer needs to be aware of the ideal physical representations for different types of conceptual/logical structures so as to provide an efficient system. Therefore, these chapters also intend to uncover the relationships between a set of conceptual and physical structures with a view to developing a set of heuristics for the database designer.

6.1 ANALYSIS OF CONCEPTUAL STRUCTURES

The aim of this section is to enumerate and explore all possible relationships between a set of classes. From chapter four, the two key concepts of the object model concerned with relationships and storage are inheritance and complex objects. It must be noted that chapter four covered object model concepts from the

standpoint of object databases, as this is the general concern of this thesis. However, the object concepts taken on board by the object database field have been influenced by many other areas. Notably, as this thesis moves onto the analysis of relationships, the area of object analysis and design (OA&D) is of interest. Indeed, object databases have been influenced by semantic modelling, and development methodologies for object systems may include richer concepts than those of the core object model. Initially, the OA&D area was confused by the various notations and concepts as found in methodologies proposed by Coad and Yourdon (1991) and Rumbaugh (1991). More recently, there has been a concerted effort towards standardisation with the emergence of schema definition and modelling languages intending to consolidate the various ideas. This is marked by the general move towards standards by the Object Database Management Group (ODMG) who have defined a schema definition language, Object Definition Language (ODL) which supports their Object Model (Cattell, 1996). In addition, the OA&D area has seen the proposal of a standard modelling language, the Unified Modeling Language (UML), developed by Grady Booch, James Rumbaugh and Ivar Jacobson (Eriksson and Penker, 1998). In the object model, from the standpoint of object analysis and design (UML), classes belonging to a given relationship may be part of:

- a generalisation hierarchy (inheritance);
- an aggregation hierarchy (nested attribute or class-composition hierarchy);
- an association.

The two concepts of the core object model, inheritance and complex objects are included in the model with the terms generalisation (hierarchy) and aggregation (hierarchy) being used in place of inheritance and complex objects respectively. However, a new concept arises, namely association. This covers a more general, weaker type of relationship than inheritance and aggregation which is useful in the earlier stages of development. It provides the ability to more conveniently represent the real world, as not all relationships between objects are strongly object-oriented. Table 6.1 provides a consolidation of the different terms used across the various fields. The OA&D field have completely adopted the term aggregation whereas the terms generalisation and inheritance are used interchangeably. From now on, therefore, the term aggregation will be used in place of complex objects but the term inheritance will continue to be used. Sections 6.2, 6.3 and 6.4 will take inheritance hierarchies, aggregation hierarchies and associations respectively and explore the relationships found in that area.

Sections 6.2 and 6.3 will use a graphical notation that is a mixture of that used by Coad and Yourdon (1991) and Rumbaugh (1991), from which UML was derived.

A class will be depicted by a rounded box where a single edged box represents the class and a double edged box represents both the class and all its objects. Relationships between classes are represented by lines broken by a semi-circle to represent an inheritance relationship or a triangle to represent an aggregation relationship. An arc between two classes of an inheritance hierarchy terminates at the inner box whereas an arc between two classes of a complex object terminates at the outer box. This represents the fact that, conceptually, is-a relationships are between classes whereas attribute-domain relationships are between individual objects.

	OA&D			
UML	Coad and Yourdon	Rumbaugh	Object Databases	This chapter
generalization	inheritance, Gen-Spec Structure	inheritance, generalisation	inheritance	inheritance
aggregation	Whole-Part Structure	aggregation	complex objects	aggregation
association	association, Instance Connection	association, links	---	association

Table 6.1 Terminology of object model concepts.

As used by Coad and Yourdon (1991), numbers indicate the cardinality of relationships between two classes where a single number represents fixed participation in the relationship and two numbers (separated by a comma) represent variable participation over a set range. This notation allows us to neatly express the optionality or dependency of a relationship. A lower bound of 0 indicates an optional relationship e.g., a Car may or may not have a Sunroof, whereas 1 indicates a mandatory relationship e.g., a Car must have an Engine. An upper bound of 1 indicates a singular relationship e.g., a Car has exactly one SteeringSystem and no more whereas a value greater than one indicates a multiple relationship e.g., a Car has many Tyres. Note that the cardinality adornments shown at the other end of a relationship for a given entity set are considered to be the entity's upper and lower bound and refer to how many instances of the other entity set are related to a single instance of this entity set. Therefore, if an entity's lower bound is 0, its instances can exist in their own right i.e. they are not dependent on the existence of the entity at the other end of the relationship and are independent. Conversely, if an entity's lower bound is 1 the entity instances cannot exist in isolation and are dependent on the entity at the other end of the relationship. For example in figure 6.1, a Doctor may have one or more Patients whereas a Patient can belong to only one Doctor.

Figure 6.1 An example of cardinality of a relationship.

Inheritance hierarchies and aggregation hierarchies are explored to one level deep only. It is assumed that the conclusions drawn from looking at relationships between one set of 'parent' classes and one set of 'children' classes can be applied recursively along a tree. Moreover, exploring trees more than one level deep would make the analysis unnecessarily complicated and unwieldy.

Section 6.4 will use E-R modelling notation (Chen, 1976) to explore associations between a pair of classes (i.e. binary relationships). Relationships involving more than two classes (ternary relationships) are not explored, both because the analysis would become disproportionately more complicated and because such relationships are not common place. A class or entity set is represented as a box and a relationship between two classes as a line broken by a diamond that may contain the name of the relationship. Again, cardinality (in addition to optionality) is represented by numbers as used for inheritance and aggregation relationships.

6.2 INHERITANCE HIERARCHY RELATIONSHIPS

The is-a relationship between a 'parent' superclass and a 'child' subclass is represented by a semi-circular shape (see figure 6.2). According to Rumbaugh (1991), inheritance relationships do not require cardinality adornments. The reason for this is that the concern in object analysis and indeed with this analysis is with the conceptual inheritance hierarchy. This means that an object cannot simultaneously be an instance of a superclass and a subclass. Objects belonging to the superclass in a conceptual inheritance hierarchy are different objects to those in its subclasses and so there is no cardinality relationship to represent. The conceptual inheritance hierarchy is distinguished from the implementation inheritance hierarchy, where an object is an instance of more than one class on the path of an inheritance hierarchy i.e. the distributed storage model. However, it must be known for a conceptual inheritance hierarchy whether objects can exist in the 'parent' class and whether objects can simultaneously belong to more than one subclass.

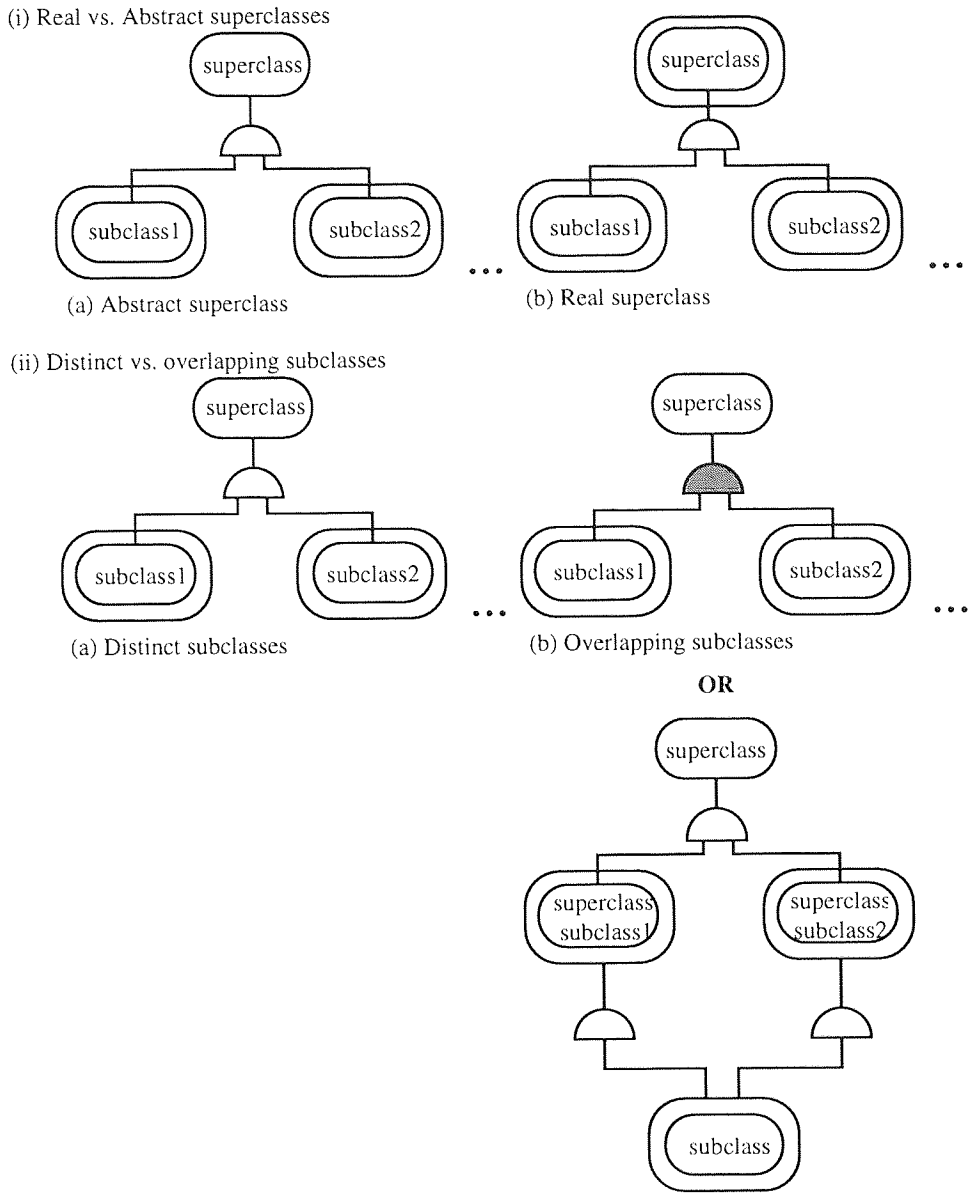


Figure 6.2 The different notations.

Objects tend to belong to leaf classes only but in some situations it may make sense to have objects belonging to non-leaf classes also. A single edged box representing the 'parent' class denotes the former case whereas a double edged box denotes the latter case (see figure 6.2(i)). Classes that do not directly contain objects are known as abstract classes whereas those that do are known as real classes. Superclasses tend to be abstract classes; their primary use is the specification of attributes and methods (excluding their implementation) which are then inherited by their subclasses. Nevertheless, superclasses may be real classes. The superclass may represent a basic type of object that may exist and the subclasses represent the optional "extras" which create new types of objects. This explanation is the only true conceptual reasoning for real superclasses. Other explanations involve

adjusting reality and lead to design and implementation decisions depending on the particular situation and requirements of the application. The next chapter will return to this issue when the concern shifts to implementation. This conceptual analysis will try to present examples that truly reflect reality but it must be stressed that the making of assumptions is necessary and the area between modelling and design is a fuzzy one.

Although the same object cannot belong to a superclass and its subclasses, an object may belong to more than one subclass (i.e. overlapping of classes). However, the ideal representation for this is as a separate class which has a number of parent classes (multiple inheritance), specifically the classes involved in the overlap. A shaded semi-circle is used to denote that an object can belong to more than one subclass (see figure 6.2 (ii)).

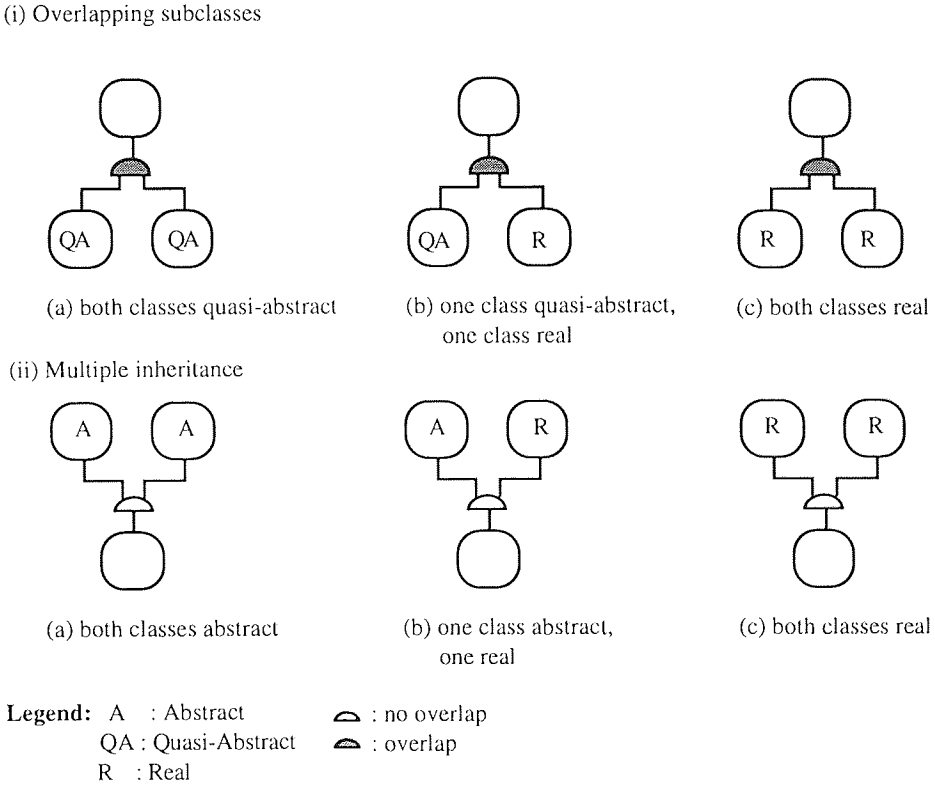


Figure 6.3 Alternative cases when two subclasses overlap.

With overlapping subclasses, there is an issue regarding abstract and real classes. This requires a deeper analysis to clarify the validity in the real world of different cases. A question arises whether it is a true reflection of the real world to have an inheritance hierarchy with overlapping subclasses where one or (possibly) both of the subclasses contain objects which belong to the 'overlap'? Such a subclass could be considered a quasi-abstract class. If this were feasible the question then is can

both classes be quasi-abstract? Following the above argument, it must be asked for multiple inheritance whether a single or both superclasses may be abstract? Notice the argument moves from quasi-abstract to abstract because the overlap is now represented by a subclass that inherits from two superclasses. Figure 6.3 is a sketch of the alternative cases. In addition to this issue, it must be asked if the answers to the above questions depend on whether the superclasses belong to the same hierarchy or distinct hierarchies, dismissing the root object (see figure 6.4).

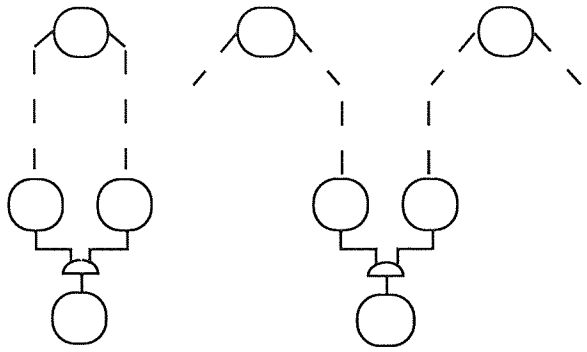


Figure 6.4 Same hierarchy versus distinct hierarchies.

6.2.1 Alternative Inheritance Structures

The above discussion illustrates that there are a number of variables that need to be decided in order to establish the variations of an inheritance hierarchy applicable to this analysis. The factors that need to be considered are:

- the number of levels;
- real vs. abstract subclasses;
- multiple vs. single inheritance;

and for single inheritance are:

- number of children classes;
- real vs. abstract superclass;
- overlap of subclasses;

and for multiple inheritance are:

- number of superclasses;
- parent classes belonging to the same or different hierarchies;
- real vs. abstract superclasses.

This analysis is concerned only with hierarchies of a single level (see figure 6.5).

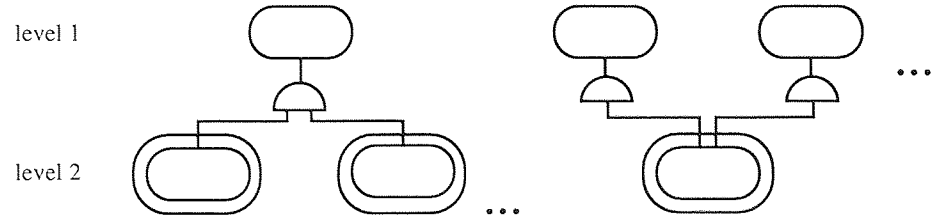


Figure 6.5 Inheritance hierarchies with two levels.

Further, multiple and single inheritance are considered in isolation. This means that the focus is on tree structures instead of the more complicated network structures. For example, in figure 6.6 structure (i) and (ii) but not (iii) will be studied. Structure (iii) can be considered a combination of two overlapping tree structures. It is anticipated that discussions for simple tree structures can be combined or extended for more complicated network structures.

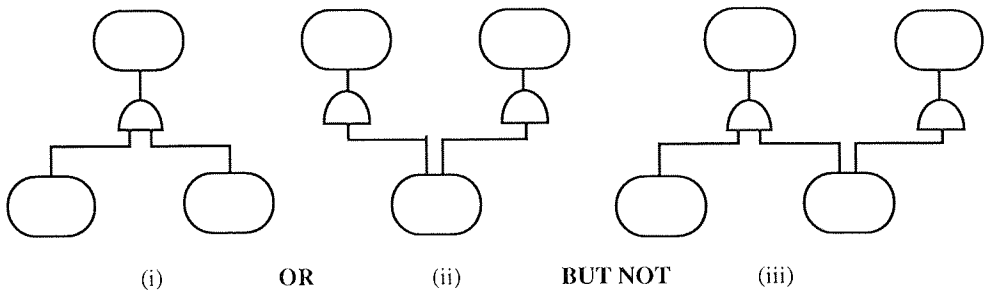


Figure 6.6 Different inheritance hierarchies.

For the purposes of this analysis, the subclasses involved in the two level structures are considered to be real classes only. Although any two level structure within an inheritance hierarchy may have abstract subclasses, they must have real descendant classes below it. The abstract descendant classes between a subclass and its first real descendant class are a means of modelling, understanding and logically specifying the problem domain. However, such a model (see figure 6.7) can be collapsed upwards from the first real class to the abstract subclass. The subclass then becomes a real subclass. The new structure still obeys inheritance/typing rules and so is semantically equivalent to the first. The decision to ignore abstract subclasses and to concentrate on real subclasses only, also removes the need to consider 'quasi-abstract' classes.

For single inheritance, structures with one subclass and structures with two subclasses are considered. By indirection, it is assumed that discussions for structures with two subclasses can be applied to structures with any number of

subclasses. At least two subclasses need to be explored so that structures with overlapping subclasses and structures with disjoint subclasses can be compared. For single inheritance structures, with one subclass or two subclasses, the 'parent' class may be real or abstract. Both cases must be considered because an abstract class does not necessarily have real ascendant classes and therefore the structure cannot be converted into a semantic equivalent consisting of direct real superclasses. Combining all three factors, the number of subclasses, the nature of the superclass (real vs. abstract) and for structures with two subclasses, the nature of object membership (overlapping vs. disjoint), gives rise to a total of six variations of single inheritance structures.

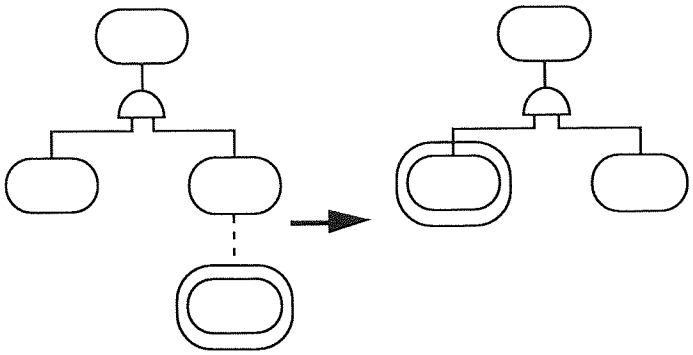


Figure 6.7 'Collapsing' an inheritance hierarchy.

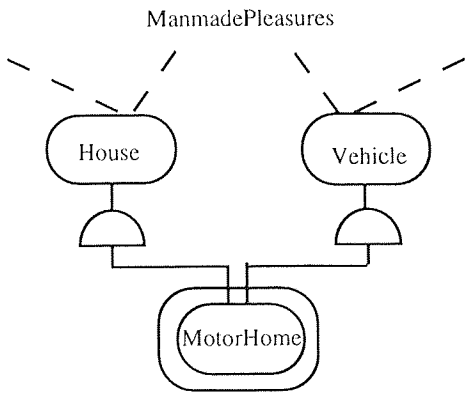


Figure 6.8 A multiple inheritance example.

Structures of multiple inheritance have two or more superclasses by definition. Only those structures with exactly two superclasses are considered for the same reason that only single inheritance structures with two subclasses are considered. As for single inheritance alternatives, a superclass of a multiple inheritance structure may be real or abstract. Therefore, all combinations of real and abstract superclasses i.e. both real, one real and one abstract or both abstract are considered.

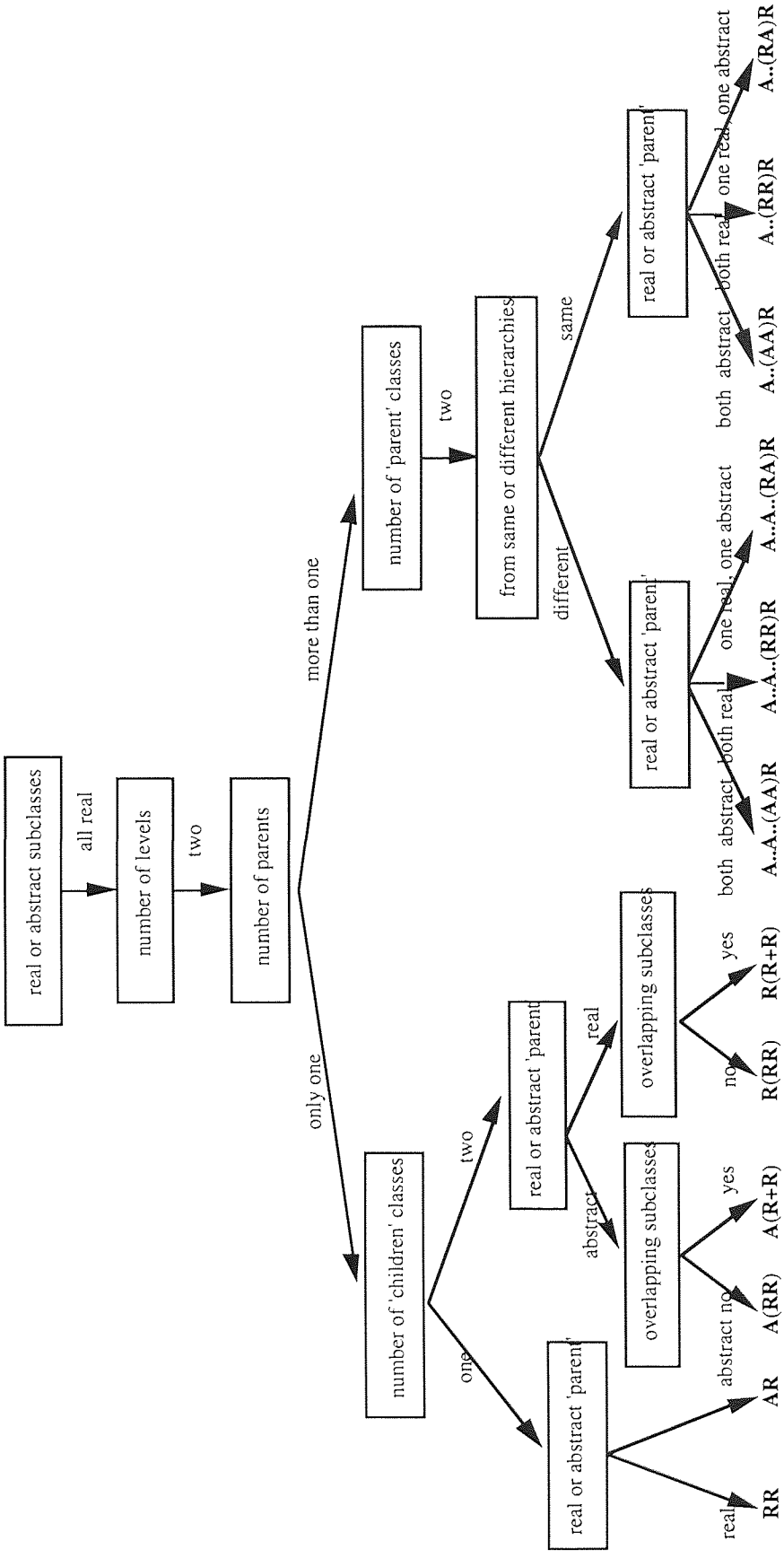


Fig.ure 6.9 Derivation of the alternative inheritance hierarchy structures.

Another source of variation is the nature of the arrival of the superclasses. The superclasses may descend from the same or different generalisation hierarchies.

The concern is whether the two superclasses under discussion have the same direct or indirect superclass. However, this decision depends heavily on the Universe of Discourse (UoD) represented by the particular application's requirements. For example, consider figure 6.8. One might consider the classes House and Vehicle to be two completely distinct entities from two separate hierarchies.

This is true if taken in a narrow context. However, one might argue that House and Vehicle both descend from the class ManmadePleasures if a much wider context is taken. The scope taken must depend on the requirements of the application and the resulting UoD. It must be stressed that, although some implementations view everything as deriving from the same generalisation hierarchy with the root being the very general class, Object, this analysis is concerned with more accurate reflections of the real world. Taking into account both factors, the nature of the superclasses (real vs. abstract) and how they have been derived (from the same or different hierarchies), leads to a total of six variations of multiple inheritance structures.

Combining single and multiple inheritance variations leads to a total of twelve alternative inheritance structures to be considered. The derivation of these alternative structures is shown in figure 6.9.

Each alternative is discussed below and an attempt made to match the structure with a real-world example. A number of sources (Bertino and Martino, 1993; Blair et al., 1991; Booch, 1994; Bowers, 1992; Brown, 1991; Coad and Yourdon, 1991; Khoshafian and Abnous, 1990; Kim, 1991; Kroha, 1993; Pratt and Adamski, 1994; Rumbaugh, 1991; Wirfs-Brock et al., 1990) were used to compile a list of inheritance examples. A total of 25 categories was established against which the twelve structures were compared.

1. R(R): Single Parent and Child, Real Superclass

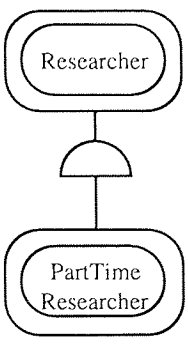


Figure 6.10 Inheritance relationship with single parent and child, real superclass.

A number of cases fall into this category because with a single 'child' class structure it is typical that the 'parent' class is real. There are two reasons for this.

- The subclass is used to represent a subset of the objects of the superclass that have specialised (extra/refined) characteristics. For example, there are a number of Researchers with general characteristics but a subset of these are PartTimeResearchers. The superclass represents the standard, general features. For example, sensors are generally StandardSensors but a subset may have additional features that make them CriticalSensors.

- The application in question may only be interested in one type of the 'parent' class although the complete set of objects could be specified by a number of types according to a particular characteristic. All objects of a type other than the type of the subclass belong to the superclass. For example, there may be a number of types of Queue but the application in question may only be interested in a GuardedQueue. All other Queue objects then belong to the Queue object.

2. A(R): Single Parent and Child, Abstract Superclass

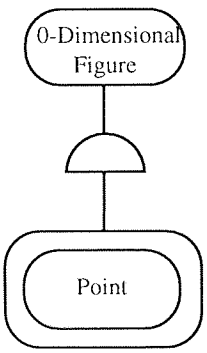


Figure 6.11 Inheritance relationship with single parent and child, abstract superclass.

Single 'child' structures, for the reasons discussed above, tend to fall into the category R(R). Therefore, the category A(R) is rare. It is difficult to see the need for an abstract class above a single 'child' class because all characteristics may be specified in one real class. The abstract superclass is redundant; it is not used to contain objects and it is not used to specify generalised information that would otherwise be spread over a number of classes.

The only 'true' example that falls into this category is shown in figure 6.11. This is extracted from a Figure hierarchy where a Point is a 0-DimensionalFigure. There are no other types of Figure objects that can be considered to be a 0-DimensionalFigure. Therefore the superclass 0-DimensionalFigure is an abstract class. The purpose of the 0-DimensionalFigure class is understood in the context of the complete Figure hierarchy as shown in figure 6.12 above. It makes sense to have classes 1-DimensionalFigure, 2-DimensionalFigure and 3-DimensionalFigure

because they specify general characteristics that pertain to a number of subclasses. It is then logical to have the class 0-DimensionalFigure as an abstract superclass of the class Point; the semantics of the application are more accurately reflected.

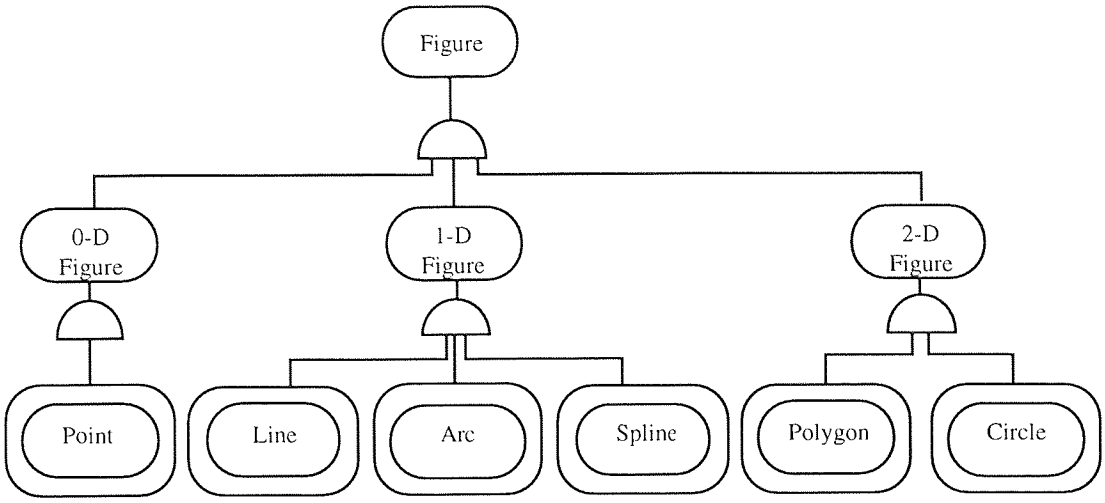


Figure 6.12 A Figure hierarchy.

3. A(RR): Single Parent, No Overlap, Abstract Superclass

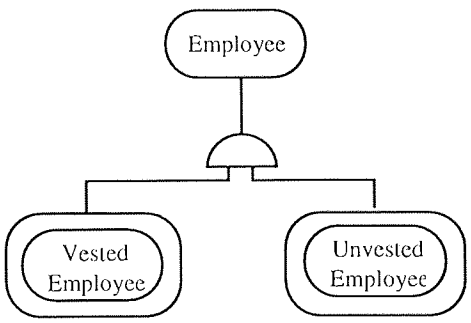


Figure 6.13 Inheritance relationship with single parent, no overlap, abstract superclass.

This is a popular structure where a set of objects falls clearly into a number of disjoint subsets according to a particular characteristic. The superclass is then simply a means of factoring out common information across the subclasses that represent the subsets. Employee or Person hierarchies are typical of this structure as are those hierarchies of everyday physical objects such as Vehicle, Sensor and Lamp. For example, an Employee can be characterised according to his/her pension status. An Employee must be either a VestedEmployee or an UnvestedEmployee i.e. no objects belong directly to the Employee class and it is therefore abstract. The characteristic has just two options, vested and unvested, and so the union of the two subsets is equivalent to the complete set of Employees. In addition, due to the

underlying semantics of the characteristic, an Employee cannot be vested and unvested at the same time. Therefore, the subclasses are disjoint and there is no overlap.

4. A(R+R): Single Parent, Overlap, Abstract Superclass

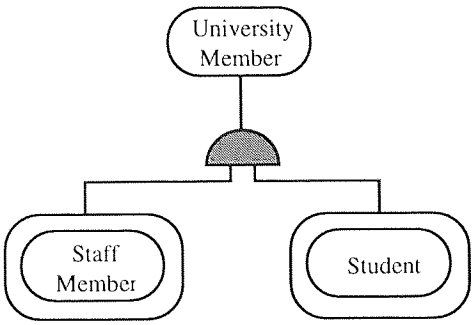


Figure 6.14 Inheritance relationship with single parent, overlap, abstract superclass.

This structure is typical within Person hierarchies where the subclasses reflect roles or jobs. There are many variations on the Person role theme and so it is a popular structure. A role does not preclude an object from taking on another role, and so in such hierarchies, there is often an overlap. For example, a UniversityMember may be a StaffMember or a Student. However, a Student may take tutorials (i.e. as an Instructor) and so also belong to the StaffMember class. An object, therefore, may belong to the StaffMember subclass, to the Student subclass or to both subclasses. The superclass, UniversityMember, is sufficiently generic to warrant it being abstract.

5. R(RR): Single Parent, No Overlap, Real Superclass

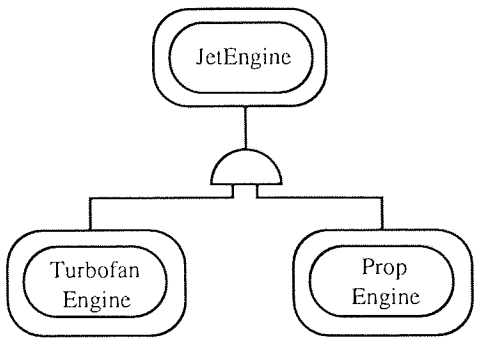


Figure 6.15 Inheritance relationship with single parent, no overlap, real superclass.

Examples pertaining to this structure are difficult to find. The usual purpose of a superclass is to factor out common information amongst a number of similar classes

and not to be a container of objects. A 'true' example of a R(RR) structure is shown in figure 6.15. This reflects the situation where there is a standard object that is utilised in its own right but which may be added to or modified to make a different object. For example, a JetEngine can exist in its own right and be used in JetAircraft. However, a JetEngine can be modified by a turbofan or a propeller and is known as a TurbofanEngine and PropEngine respectively. The same is true for a Window on a graphical user interface. There may be 'plain' Window objects. In addition, a Window object may have certain 'extras' such as a border or the ability to be edited. Therefore, an object may be a Window, a BorderedWindow or an EditableWindow.

It is possible to argue that some examples are of this structure, although they are essentially of the A(RR) structure. Such examples appear to be of this structure when all subclasses are not specified. The superclass becomes a real superclass because it contains objects that should ideally belong to unspecified subclasses. Consider the example hierarchies in figure 6.16. For some applications, it may only be necessary to know more detail for the subclasses specified. However, a Manager object, a Bus object and a Submarine object in the system must belong to the Employee, Vehicle and WaterVehicle classes respectively.

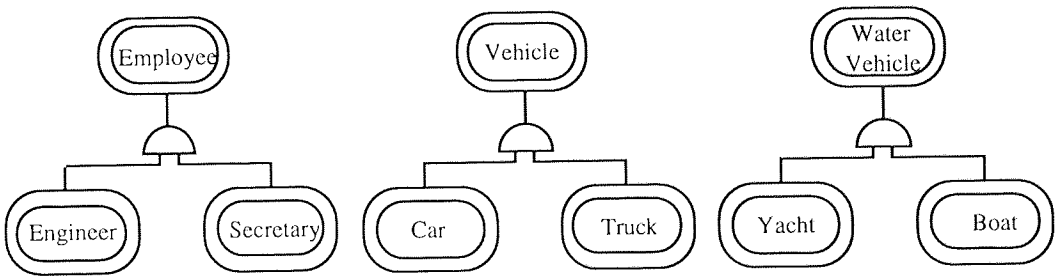


Figure 6.16 Example hierarchies with a limited number of subclasses.

6. R(R+R): Single Parent, Overlap, Real Superclass

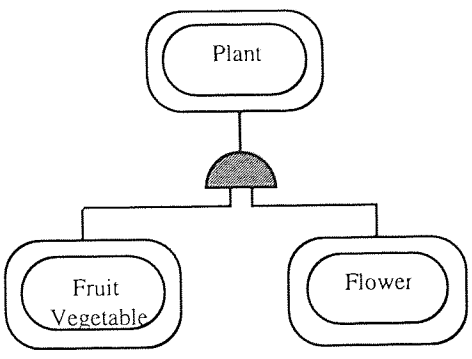


Figure 6.17 Inheritance relationship with single parent, overlap, real superclass.

The discussion above also applies to this structure. In the same way it possible to argue that some examples are R(RR) structures when essentially they are A(RR) structures, it is possible to argue for some examples that they are R(R+R) structures when essentially they are A(R+R) structures. A possible 'true' example is shown in figure 6.17. A Plant may produce a fruit or a vegetable, a flower, neither or both. Therefore, a Plant object may belong to the Plant class (real superclass), the Fruit/Vegetable class, the Flower class or both the Fruit/Vegetable class and the Flower class (overlap of subclasses).

7. A..A..(AA)R: Multiple Parents, Different Trees, Abstract Superclasses

It is difficult to find examples pertaining to this structure because multiple inheritance is often derived from similar classes that overlap. The above example is of this structure if it is assumed that Asset and InterestBearingItem are not immediately similar.

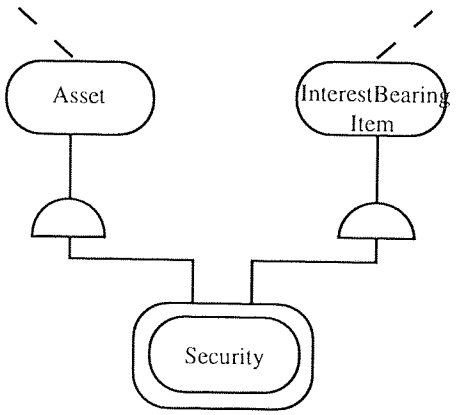


Figure 6.18 Inheritance relationship with multiple parents from different trees, abstract superclasses.

8. A..A..(RR)R: Multiple Parents, Different Trees, Real Superclasses

Again, this is a rare structure. If it is assumed that it is not required to know information below the JapaneseCompany or CarManufacturer level for other JapaneseCompanies or CarManufacturers respectively, then the superclasses can be considered to be real (see figure 6.19). However, this is really a condensed version of a larger hierarchy involving a series of multiple inheritance instances combining on the one side UKCompany, USACompany, FrenchCompany etc. and on the other ComputerManufacturer, WashingMachineManufacturer etc. Further, it may be considered that JapaneseCompany and CarManufacturer belong to the same

hierarchy and are both derived from the class Company. Hence, the argument that this example is of this structure is a weak one.

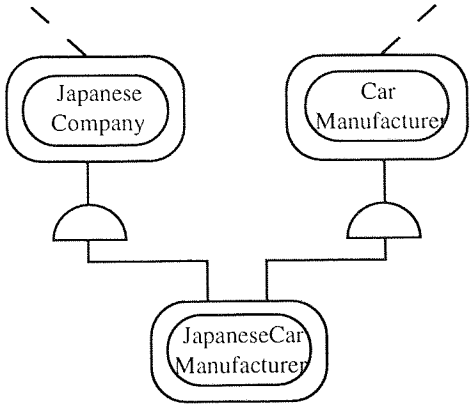


Figure 6.19 Inheritance relationship with multiple parents from different trees, real superclasses.

9. A..A..(RA)R: Multiple Parents, Different Trees, One Real, One Abstract Superclass

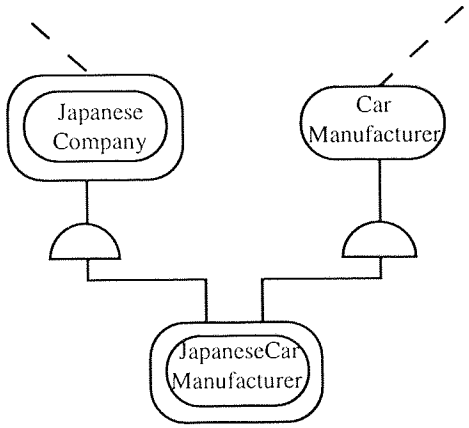


Figure 6.20 Inheritance relationship with multiple parents from different trees, one real and one abstract superclass.

A 'true' example of this structure could not be found. However, according to the application, it could be argued that the JapaneseCompany-CarManufacturer example can be of this structure. If the only specialisation of JapaneseCompany is JapaneseCarManufacturer (even though there exists other JapaneseCompanies), then the class JapaneseCompany can be considered to be a real superclass. On the other hand, all CarManufacturers may be further specialised into the various subclasses and so the class CarManufacturer can be considered to be an abstract superclass. It must be stressed that this argument is rather weak and that this structure may be invalid in view of the real semantics.

10. A..AA(R): Multiple Parents, Same Trees, Abstract Superclasses

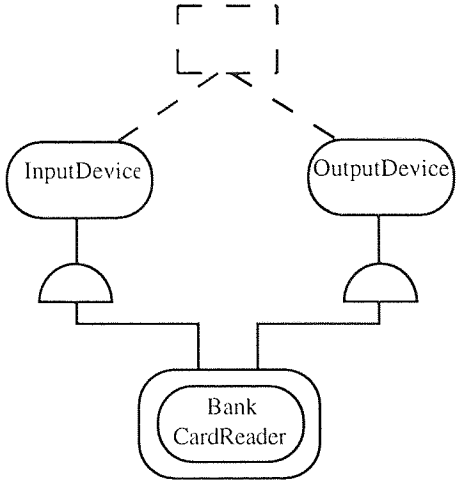


Figure 6.21 Inheritance relationship with multiple parents from the same trees, abstract superclasses.

This is a popular structure. It is typical of the situation where there are a number of objects that are definitely of one type and a number of objects that are definitely of another type but the odd case where an object can be considered to be a cross between the two types. For example, InputDevice has a number of subclasses such as Keyboard, PunchCardReader etc. and OutputDevice has a number of subclasses such as Printer and VDU etc. However, the subclass BankCardReader possesses both input and output facilities and so is a subclass of both InputDevice and OutputDevice. The classes InputDevice and OutputDevice are general classes for specifying common information across all types of InputDevice and OutputDevice respectively and do not directly contain objects, if it is assumed that all their subclasses are specified. Further, InputDevice and OutputDevice are both derived from the same 'parent' class Device. This structure also appears in hierarchies involving a series of combinations of characteristics. In this situation, an object inherits from at least two 'characteristic' superclasses out of a set of n superclasses and all combinations are exhaustively specified. For example consider an Aircraft hierarchy. An Aircraft may be a MilitaryJet, a MilitaryProp, a PropJet or a MilitaryProp. The characteristics Military, Jet and Prop need only be specified once in abstract superclasses.

11. A..(RR)R: Multiple Parents, Same Trees, Real Superclasses

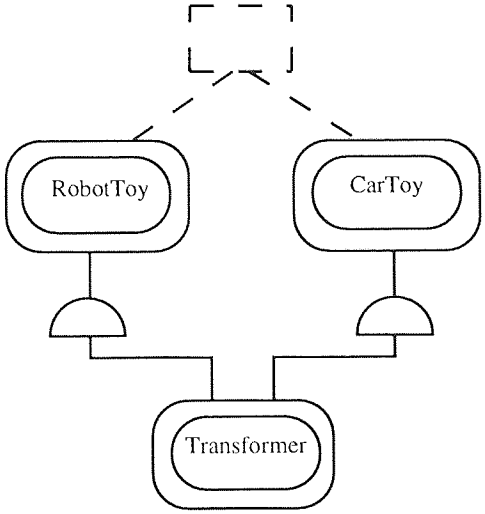


Figure 6.22 Inheritance relationship with multiple parents from the same trees, real superclasses.

Examples can easily be found representing this structure. The above is a 'true' representation of the A..(RR)R structure. A Transformer object can be considered to have both RobotToy characteristics and CarToy characteristics. The classes RobotToy and CarToy are derived from the single 'parent' superclass, Toy. This structure is also popular in Employee, Person and UniversityMember hierarchies (i.e. 'role' hierarchies), where a Person object may perform a single role or a number of roles e.g., the StaffMember-Student-Instructor hierarchy (see case 4).

12. A..(RA)R: Multiple Parents, Same Trees, One Real, One Abstract Superclass

This is not an obvious structure. The Employee hierarchy in figure 6.23 appears the only example that fits it accurately. Both the SalesPerson class and the Manager class derive from the same underlying class, Employee. A number of Employees are SalesPerson objects. A SalesManager can be considered to be both a SalesPerson and a Manager. The Manager class may be abstract if all Manager objects combine the characteristics of a Manager with the characteristics of another type of Employee e.g., TechnicalManager or AdminManager. However, this precludes the fact that some Employees may be general Managers over a number of different types of Employee and therefore do not have particular characteristics that belong to a single Employee type.

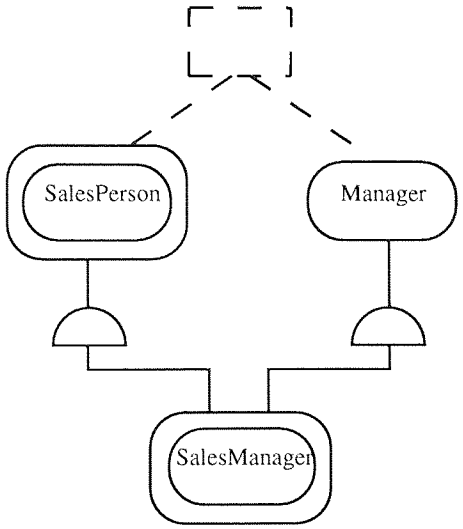


Figure 6.23 Inheritance relationship with multiple parents from the same trees, one real and one abstract superclass.

6.3 AGGREGATION HIERARCHY RELATIONSHIPS

An aggregation hierarchy is a set of relationships between 'whole' classes and their 'component' classes. Each association between a pair of classes can be described as a 'part-of' relationship and implies stronger semantics than a mere mapping between independent classes. In particular, such a relationship impacts on the semantics of creation and deletion operations.

An aggregation hierarchy may be used to represent a physical part hierarchy or a logical part hierarchy. If it is considered that one entity is part of another entity, then the relationship between them may be represented as an aggregation rather than an association. In fact, part hierarchies arise in three variations (Coad and Yourdon, 1991):

- Assembly-Parts e.g., an aircraft is made up of an engine, a body etc.;
- Container-Contents e.g., an aircraft contains a crew and cargo;
- Collection-Members e.g., an organisation has a number of clerks.

Kim et al. (1989) consider a relationship within an aggregation hierarchy to be a composite reference (as opposed to a weak reference) within a composite hierarchy. Such a reference may be dependent or independent and shared or exclusive. A dependent reference means that the existence of an object belonging to the 'component' class depends on the existence of a corresponding object in the 'whole' class whereas an object of a class constituting an independent reference may exist in its own right. A shared reference means that a 'component' object may be part of more than one 'whole' object whereas an exclusive reference means that a

'component' object may be part of only one 'whole' object. Kim et al. (1989) therefore categorise a composite reference as one of the following types:

- exclusive dependent;
- exclusive independent;
- shared dependent;
- shared independent.

Halper et al. (1992) further refine the characteristic of exclusiveness into inter-class and intra-class. Inter-class exclusiveness means that a 'part' object cannot be shared with any other 'whole' object of any type. Intra-class exclusiveness means that a 'part' object cannot be shared with any other 'whole' object of the same type but may be shared with an 'whole' object of another type. Hence, there are six alternative types of reference:

- inter-class exclusive dependent;
- inter-class exclusive independent;
- intra-class exclusive dependent;
- intra-class exclusive independent;
- shared dependent;
- shared independent.

The derivation of these alternatives is shown in figure 6.24. However, this views the nature of the relationship between a 'whole' class and a 'part' class in one direction only. The six alternatives are derived by focusing on the point of view of the objects of the 'part' class in terms of existence and participation in the 'part-of' relationship.

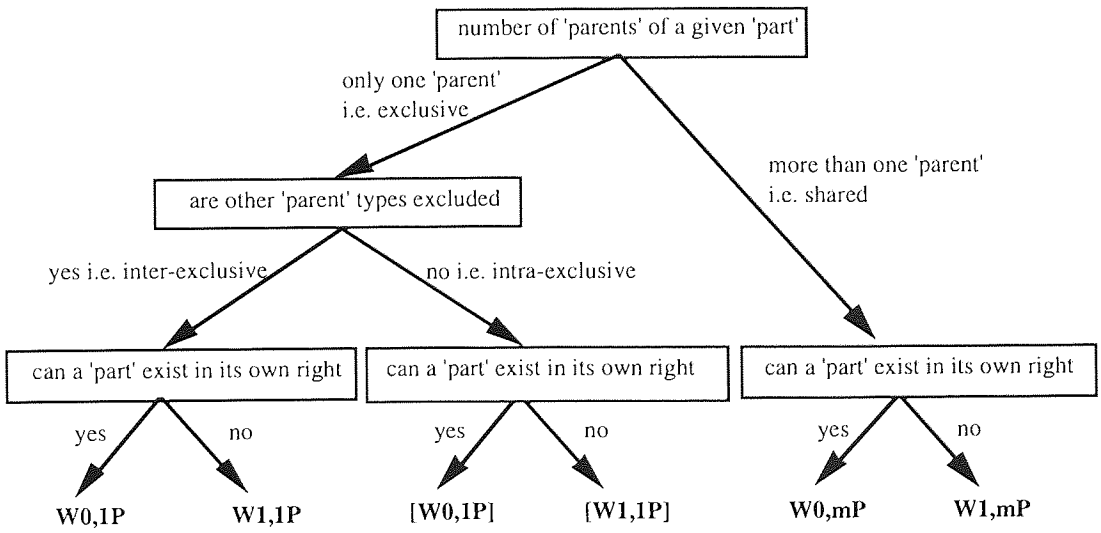


Figure 6.24 Derivation of alternative part-to-whole relationships.

These six alternatives can be specified with one set of cardinality adornments specifically those nearest to the 'whole' class as shown in figure 6.25. In particular, an upper bound of '1' indicates an exclusive relationship whereas an upper bound of 'm' indicates a shared relationship and a lower bound of '0' indicates an independent reference whereas a '1' indicates a dependent reference.

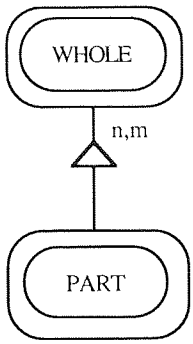


Figure 6.25 An aggregation relationship from the point of view of the 'part' class.

In the same way that the relationship can be viewed from the 'part' class end, it is felt that the relationship can be viewed from the 'whole' class end. Indeed, Halper et al. (1992) consider the characteristics of multi-valuedness and essentiality in addition to those explored by Kim et al. (1989) of exclusiveness/sharing and dependency. Dependency is also considered in both directions. These characteristics affect the cardinality adornments nearest to the 'part' class end (assuming the chosen notation) and so reflect the nature of the relationship from the point of view of the 'whole' class. When a 'whole' class references a 'part' class it contains an attribute that references objects of the 'part' class. This reference may be single-valued or multi-valued. A single-valued relationship references a single 'part' object whereas a multi-valued relationship is a set of references to a number of 'part' objects. A single-valued reference is indicated by the presence of an upper bound of '1' whereas a multi-valued reference is indicated by the presence of an upper bound of 'm'. An essential reference means that exactly one 'part' object must exist for each 'whole' object. This therefore represents a specialisation of the dependency characteristic from the 'whole-to-the-part', specifically when the relationship is single-valued. The cardinality adornment for an essential reference becomes a single value of '1'. In general, a dependent reference is indicated by a lower bound of '1'. Taking into account multi-valuedness and dependency gives rise to four alternatives from the point of view of the 'whole' class.

- single-valued, essential;
- single-valued, non-essential;

- multi-valued, dependent;
- multi-valued, independent.

For the purposes of this analysis, the terms essential and non-essential shall be used in place of dependent and independent respectively in order to distinguish whole-part from part-whole relationships. The derivation of these alternatives is shown in figure 6.26 below.

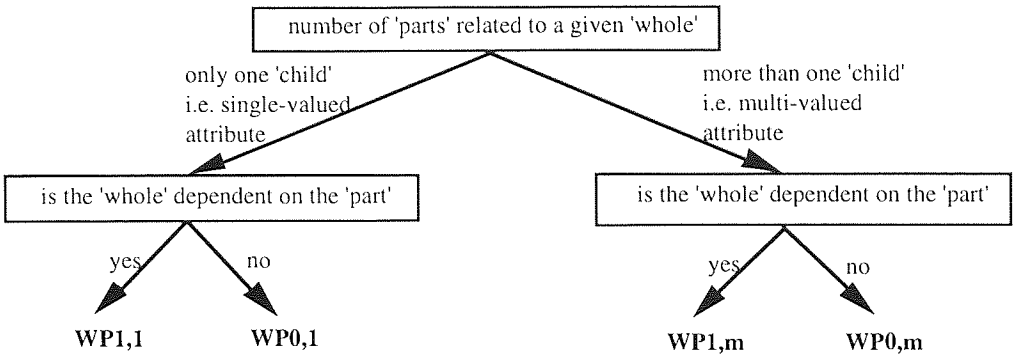


Figure 6.26 Derivation of alternative whole-to-part relationships.

Halper et al. (1992) also consider specialised cases of multi-valued references. A fixed-cardinality relationship means that both the lower and upper bounds have specific quantities that can be attached to them. A fixed-cardinality relationship means that a single known quantity of 'part' objects are related to a given 'whole' object (i.e. lower and upper bounds are equivalent). When a number of 'part' objects are related to a given 'whole' object, an ordering can sometimes be implied from the semantics of the application. Halper et al. (1992) use an ordered part relation to accommodate this situation. However, these are specialised cases and so they will not be considered further.

Combining the six alternatives derived from the point of view of the 'part' class with the four alternatives derived from the point of view of the 'whole' class gives rise to a total of 24 aggregation cases for consideration. See table 6.2 below. These categories will be the basis of the analysis that follows. Recursive references have been ignored for the purposes of this analysis. Note that a shorthand notation has been employed for convenience above, where W represents the 'whole' class and P represents the 'part' class. The subscripts of W represent "how many 'whole' objects are related to single 'part' object" and those of P represent the opposite viewpoint. Intra-class exclusiveness is represented by square brackets. A number of sources (Bertino and Martino, 1993; Blair et al., 1991; Booch, 1994; Coad and Yourdon, 1991; Khoshafian and Abnous, 1990; Kim, 1991; Kroha,

1993; Rumbaugh, 1991) were used to compile a list of aggregation examples. A total of 22 categories was established against which the 24 structures were compared.

		WHOLE VIEW			
		single-valued essential	single-valued non-essential	multi-valued essential	multi-valued non-essential
PART VIEW	inter-class exclusive independent	$W_{0,1}P_{1,1}$	$W_{0,1}P_{0,1}$	$W_{0,1}P_{1,m}$	$W_{0,1}P_{0,m}$
	inter-class exclusive dependent	$W_{1,1}P_{1,1}$	$W_{1,1}P_{0,1}$	$W_{1,1}P_{1,m}$	$W_{1,1}P_{0,m}$
	intra-class exclusive independent	$[W_{0,1}P_{1,1}]$	$[W_{0,1}P_{0,1}]$	$[W_{0,1}P_{1,m}]$	$[W_{0,1}P_{0,m}]$
	intra-class exclusive dependent	$[W_{1,1}P_{1,1}]$	$[W_{1,1}P_{0,1}]$	$[W_{1,1}P_{1,m}]$	$[W_{1,1}P_{0,m}]$
	shared independent	$W_{0,m}P_{1,1}$	$W_{0,m}P_{0,1}$	$W_{0,m}P_{1,m}$	$W_{0,m}P_{0,m}$
	shared dependent	$W_{1,m}P_{1,1}$	$W_{1,m}P_{0,1}$	$W_{1,m}P_{1,m}$	$W_{1,m}P_{0,m}$

Table 6.2 Derivation of aggregation cases.

1. $W_{0,1}P_{1,1}$: Inter-exclusive, Independent, Single, Essential

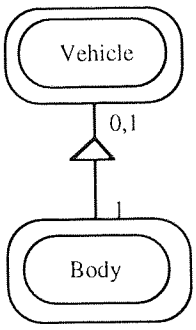


Figure 6.27 An inter-exclusive, independent, single, non-essential relationship *after* Khoshafian and Abnous (1990).

A number of independent exclusive references can be found in the aggregation example of a Vehicle. This is a typical example of a physical part hierarchy where an object is clearly made up of a number of parts where one of those parts is a Body. At any particular point in time component objects can only physically belong to one Vehicle i.e. exclusive relationships. However, component objects can exist

in their own right and be re-used in another Vehicle i.e. independent relationships. The relationship between a Body and a Vehicle is therefore both independent and exclusive. Taking the opposite viewpoint if it can be assumed that the very 'essence' of the Vehicle is the Body of that Vehicle then the relationship between Vehicle and Body is 'has-essential-part'. Further a Vehicle can only have one Body at any point in time and so the Vehicle-Body relationship is also single-valued.

2. $W_{0,1}P_{0,1}$: Inter-exclusive, Independent, Single, Non-essential

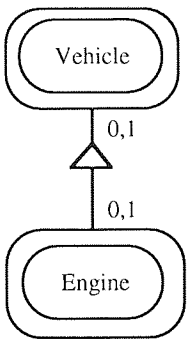


Figure 6.28 An inter-exclusive, independent, single, non-essential relationship *after* Khoshafian and Abnous (1990).

The physical part hierarchy of Vehicle is used again to show an independent exclusive relationship. Indeed, such a relationship tends to only be found in physical part hierarchies. Unlike the Vehicle-Body relationship however, the Vehicle-Engine relationship is not essential. It is assumed that the materialisation of a Vehicle is the Body and that it still exists with or without an Engine and that a Vehicle can have no more than one Engine. The acceptance of the former assumption depends on the nature of the application. If the ability of the whole to function correctly is emphasised then, it may be argued that an Engine is absolutely essential to the existence of a Vehicle.

3. $W_{0,1}P_{1,M}$: Inter-exclusive, Independent, Multi-valued, Essential

A Machine is made up of at least one Assembly of Parts, typically many (see figure 6.29). A Machine would not exist without at least one Assembly of Parts i.e. it is difficult to envisage requiring information on a Machine without it having a physical materialisation. Before a Machine is constructed however, a number of Assemblies may exist in there own right i.e. the class Assembly is independent. Further, when an Assembly is part of a Machine, it can only physically be part of one Machine i.e. an exclusive Assembly-Machine relationship. This is similar to the Vehicle example

and indeed is the same for other physical part hierarchies such as Aircraft and PersonalComputer.

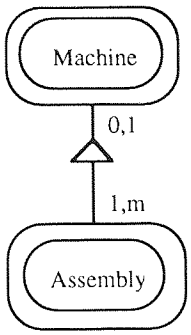


Figure 6.29 An inter-exclusive, independent, multi-valued, essential relationship *after* Rumbaugh (1991).

4. $W_{0,1}P_{0,M}$: Inter-exclusive, Independent, Multi-valued, Non-essential

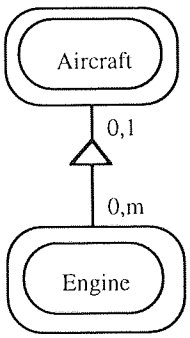


Figure 6.30 An inter-exclusive, independent, multi-valued, non-essential relationship *after* Coad and Yourdon (1991).

The relationship between an Engine and an Aircraft is as in the examples for Machine and Vehicle above. However, if it assumed that an Engine is not the 'essence' of an Aircraft (i.e. the Aircraft can be conceived even when all its Engines have been removed) and Aircraft may contain more than one Engine, the relationship between Aircraft and Engine is non-essential and is zero-to-many. To be more exact it is a range-restricted relation because Aircraft can have no more than four Engines.

5. $W_{1,1}P_{1,1}$: Inter-exclusive, Dependent, Single-valued, Essential

A Cooler can only be physically part of one EnvironmentalController i.e. an exclusive relationship (see figure 6.31). The Cooler class is also dependent, if it is assumed that it is not necessary to know about Coolers that are not part of some

Controller i.e. there is not a stock of Coolers. An EnvironmentalController requires one Cooler only (i.e. single-valued) and cannot function without it and so EnvironmentalController has-essential-part Cooler.

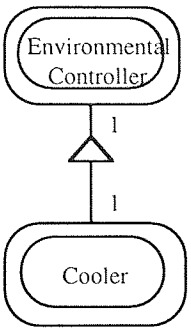


Figure 6.31 An inter-exclusive, dependent, single-valued, essential relationship *after* Booch (1994).

6. $W_{1,1}P_{0,1}$: Inter-exclusive, Dependent, Single-valued, Non-essential

This structure is found in physical part hierarchies where the part is an optional addition and is not necessary to the correct function of the whole e.g., a Mouse is an optional item for a PersonalComputer. This structure also appears in logical part hierarchies as shown in figure 6.32 below. A Block in a ComputerProgram is made up of either a CompoundStatement or a SimpleStatement. Therefore, a Block does not necessarily have to contain a CompoundStatement and therefore the relationship is non-essential. On the other hand, a CompoundStatement must be assigned to at exactly one Block of the ComputerProgram i.e. there exists dependent, exclusive relationship between CompoundStatement and Block.

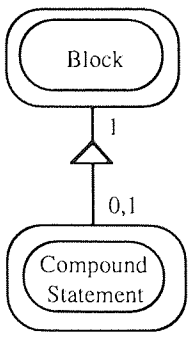


Figure 6.32 An inter-exclusive, dependent, single-valued, non-essential relationship *after* Rumbaugh (1991).

7. $W_{1,1}P_{1,M}$: Inter-exclusive, Dependent, Multi-valued, Essential

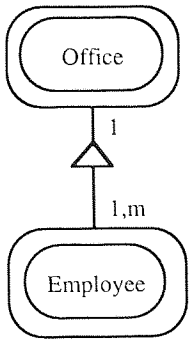


Figure 6.33 An inter-exclusive, dependent, multi-valued, essential relationship *after* Khoshafian and Abnous (1990).

This is a very popular structure that appears across a number of areas. For example, the structure exists in the physical part hierarchy of PersonalComputer between the classes SystemBox and RAM, the logical part hierarchy of ComputerProgram between the classes Program and Block, and the Collection-Members aggregation between the classes School and Club. Figure 6.33 shows a Container-Contents example and is extracted from an OfficeAutomation database. An Office has at least one Employee typically many (i.e. multi-valued essential) and an Employee belongs to exactly one Office at any particular point in time (i.e. exclusive dependent), if we ignore 'floating' Employees.

8. $W_{1,1}P_{0,M}$: Inter-exclusive, Dependent, Multi-valued, Non-essential

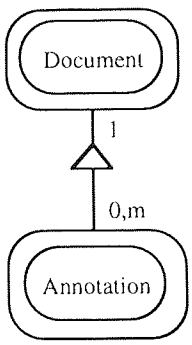


Figure 6.34 An inter-exclusive, dependent, multi-valued, non-essential relationship *after* Kim et al. (1989).

Again, this structure can be seen in a number of areas including Container-Contents aggregations and Collection-Members aggregations. Such a structure is found in the example of electronic Documents. A Document may have a number of Annotations associated with it, possibly none (i.e. multi-valued non-essential).

However, any specific Annotation can only be for a single Document i.e. an exclusive relationship. Further, an Annotation object is of no relevance without its corresponding Document object to which it belongs. The Annotation class is therefore dependent on the Document class. Therefore, the relationship between Document and Annotation is both dependent and exclusive as shown in figure 6.34.

9. $[W_{0,1}P_{1,1}]$: Intra-class exclusive, Independent, Single-valued, Essential

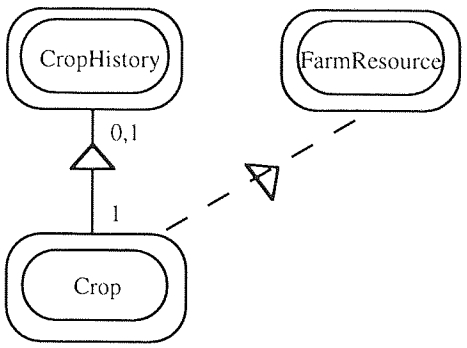


Figure 6.35 An intra-class exclusive, independent, single-valued, essential relationship *after* Booch (1994).

Intra-class exclusive references are difficult to find (i.e. structures 9 through to 16). The above example may be argued to fit this structure. A CropHistory is not valid without a Crop on which it is based. On the other hand, a Crop does not necessarily have a CropHistory recorded for it. It may then be reasoned that, although a Crop can only have one CropHistory at the most and is therefore exclusive with respect to the class CropHistory, it does not exclude references from other classes such as FarmResource. More precisely, the relationship between Crop and CropHistory is intra-class exclusive as opposed to inter-exclusive.

10. $[W_{0,1}P_{0,1}]$: Intra-class exclusive, Independent, Single-valued, Non-essential

The relationship between a FootballLeagueTeam and a Captain of such a team can be considered to be of this structure as shown in figure 6.36 below. A FootballLeagueTeam may have only one Captain but may not have one at a particular point in time such as when the position requires re-appointment i.e. single-valued, non-essential. On the other hand, a Captain can only perform that role for one team in the league i.e. an exclusive relationship. However, the relationship is only intra-class exclusive because a SportsPlayer may belong and be a Captain of other sports teams outside the FootballLeague. Further, if it is

considered that Captain represents a role taken on by a SportsPerson, then the Captain class is also independent.

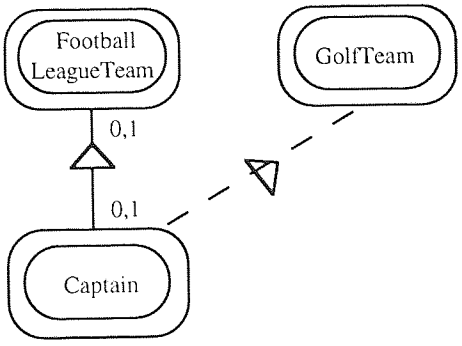


Figure 6.36 An intra-class exclusive, independent, single-valued, non-essential relationship.

11. $[W_{0,1}P_{1,M}]$: Intra-class exclusive, Independent, Multi-valued, Essential

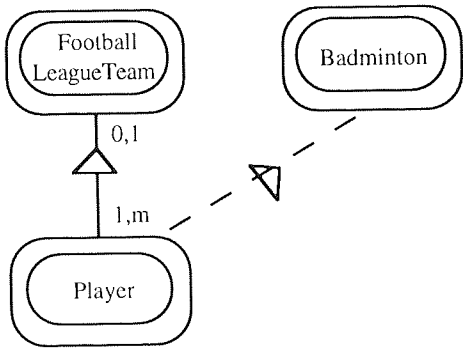


Figure 6.37 An intra-class exclusive, independent, multi-valued, essential relationship.

The FootballLeagueTeam example can be used again to show a representation of this structure. A FootballLeagueTeam must have some SportsPlayers i.e. multi-valued essential. Considering the relationship in the opposite direction, a SportsPlayer can only play in one FootballLeagueTeam at a time i.e. exclusive. However, a SportsPlayer does not necessarily have to play in a FootballLeagueTeam but may play in teams of other sports i.e. an intra-class exclusive, independent relationship.

12. $[W_{0,1}P_{0,M}]$: Intra-class exclusive, Independent, Multi-valued, Non-essential

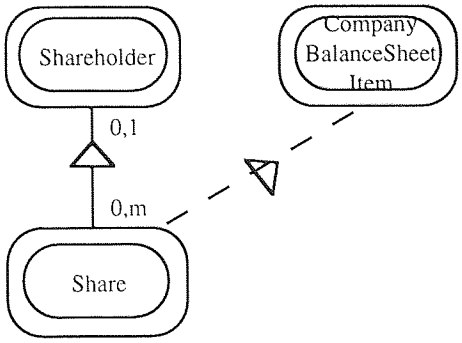


Figure 6.38 An intra-class exclusive, independent, multi-valued, non-essential relationship *after* Booch (1994).

It is possible to reason that the example in figure 6.38 fits this structure. A Share may belong to any Shareholder (i.e. if it has not been taken up on issue) or it may belong to exactly one Shareholder. On the other hand, a Shareholder may have from zero to many Shares. However, the exclusive relationship between the class Share and Shareholder can be considered to be intra-class exclusive. A Share may be referenced by other classes such as CompanyBalanceSheetItem for example.

13. $[W_{1,1}P_{1,1}]$: Intra-class exclusive, Dependent, Single-valued, Essential

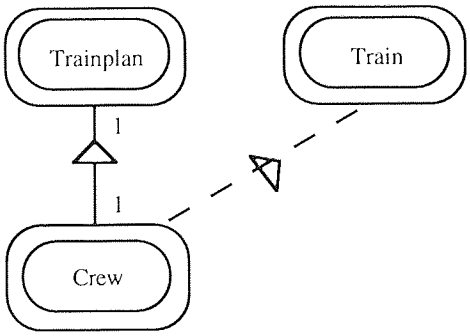


Figure 6.39 An intra-class exclusive, dependent, single-valued, essential relationship *after* Booch (1994).

Again, this is a difficult one to reason but the example in figure 6.39 represents a near fit to this structure. A TrainPlan must be based on a Crew and is therefore an 'has-essential-part' relationship. The Crew can only be part of exactly one TrainPlan at any particular point in time. However, the relationship can be considered intra-class exclusive if, for example, it belongs to the Container-Contents aggregation of Train.

14. [$W_{1,1}P_{0,1}$]: Intra-class exclusive, Dependent, Single-valued, Non-essential

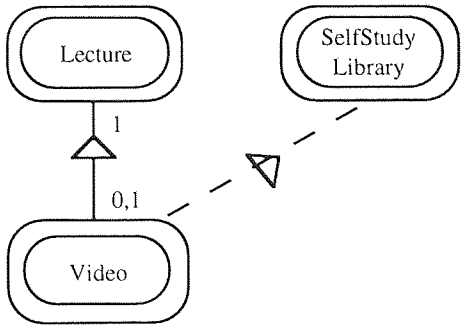


Figure 6.40 An intra-class exclusive, dependent, single-valued, non-essential relationship.

As shown in figure 6.40, a Lecture may have a Video made of it (say for long distance learning facilities), although it is not a necessary item i.e. a single-valued, non-essential relationship. From the opposite viewpoint, a LectureVideo by definition must be associated with a Lecture and it is assumed that only one Lecture is contained on a Video i.e. a dependent, exclusive relationship. Further, the relationship is only intra-class exclusive if Video can be referenced by other classes such as SelfStudyLibrary.

15. [$W_{1,1}P_{1,M}$]: Intra-class exclusive, Dependent, Multi-valued, Essential

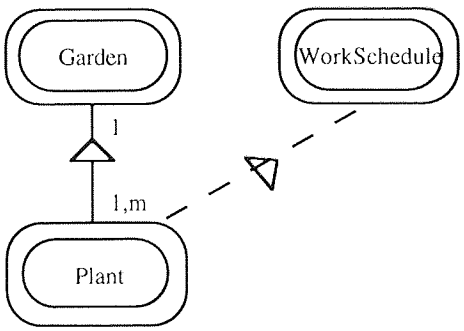


Figure 6.41 An intra-class exclusive, dependent, multi-valued, essential relationship *after* Booch (1994).

It is possible that the example in figure 6.41 fits this structure. A Gardener would expect to have at least one Plant in each of his Gardens that he tends, typically many, and a Plant can only be in one Garden at any particular point in time. If it is assumed that the Gardener does not keep unwanted Plants, then each Plant must belong to exactly one Garden at any point in time. The relationship may be

considered to be intra-class exclusive if the class Plant is referenced by other classes such as WorkSchedule.

16. $[W_{1,1}P_{0,M}]$: Intra-class exclusive, Dependent, Multi-valued, Non-essential

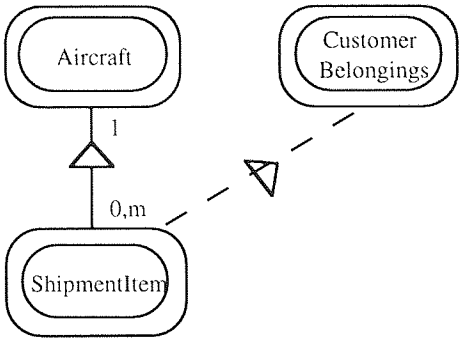


Figure 6.42 An intra-class exclusive, dependent, multi-valued, non-essential relationship *after* Coad and Yourdon (1991).

An Aircraft may contain many ShipmentItems, possibly none, and a ShipmentItem can only be physically part of one Aircraft at any point in time. However, the relationship can be considered intra-class exclusive if other classes such as CustomerBelongings can also reference the class ShipmentItem.

17. $W_{0,M}P_{1,1}$: Shared, Independent, Single-valued, Essential

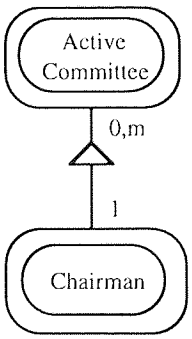


Figure 6.43 A shared, independent, single-valued, essential relationship.

Figure 6.43 shows the relationship between an ActiveCommittee and its Chairman. An ActiveCommittee must have exactly one Chairman at any particular point in time i.e. single-valued, essential and a Person may take on the role of Chairman for more than one ActiveCommittee i.e. shared. If it is assumed that the class Chairman represents a Member or more generally a Person set, then the relationship is also

independent i.e. not all Members will take part in the Chairman-of-ActiveCommittee relationship.

18. $W_{0,M}P_{0,1}$: Shared, Independent, Single-valued, Non-essential

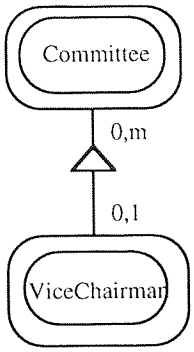


Figure 6.44 A shared, independent, single-valued, non-essential relationship.

Again, the Committee example is used to show a relationship of this structure. A Committee may or may not have a ViceChairman but can have no more than one i.e. a single-valued, non-essential relationship. The ViceChairman-Committee relationship can be considered to be independent and shared following the reasoning given for the Chairman-ActiveCommittee relationship.

19. $W_{0,M}P_{1,M}$: Shared, Independent, Multi-valued, Essential

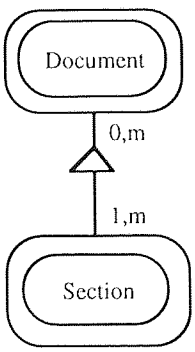


Figure 6.45 A shared, independent, multi-valued, essential relationship *after* Kim et al. (1989).

Returning to the (electronic) Document example, there is a independent shared reference as shown in figure 6.45 between the classes Section and SectionDocument. Each Section may be part of a number of Documents i.e. a shared relationship. A Section can also exist in its own right i.e. an independent relationship, if it is assumed that it is necessary to be aware of individual Sections which have not yet been fitted into a Document. This obviously depends on the

requirements of the users of the application and this is the opposite viewpoint of that taken by Kim et al. (1989). It is assumed that a Document cannot exist without it having at least one Section and will typically contain many (i.e. a multi-valued, essential relationship between the classes Document and Section).

20. $W_{0,M}P_{0,M}$: Shared, Independent, Multi-valued, Non-essential

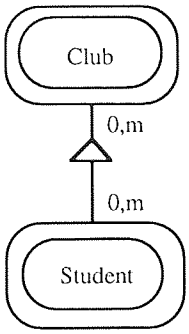


Figure 6.46 A shared, independent, multi-valued, non-essential relationship *after* Coad and Yourdon (1991).

The electronic Document example again contains a shared reference structure between the classes Document and Image. This structure is also found in the Collection-Members aggregation between the classes Club and Student as shown in figure 6.46. A Student may belong to zero, one, or many Clubs and a Club can have any number of Students at a particular point in time, possibly none. The Students still need to be aware of the existence of Clubs even if the membership is zero i.e. a non-essential, relationship.

21. $W_{1,M}P_{1,1}$: Shared, Dependent, Single-valued, Essential

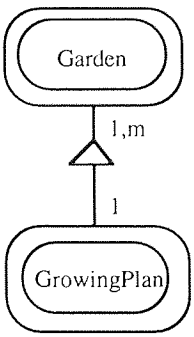


Figure 6.47 A shared, dependent, single-valued, essential relationship *after* Booch (1994).

A Gardener will have exactly one GrowingPlan for each Garden he tends (i.e. a single-valued, essential relationship) and the same GrowingPlan can be applied to

many Gardens but will exist for at least one Garden (i.e. a dependent, shared relationship) as shown in figure 6.47.

22. $W_{1,M}P_{0,1}$: Shared, Dependent, Single-valued, Non-essential

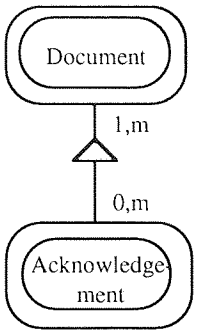


Figure 6.48 A shared, dependent, single-valued, non-essential relationship.

As shown in figure 6.48, a Document may or may not have an Acknowledgement i.e. a single-valued, non-essential relationship. On the other hand, the same Acknowledgement may be present in a number of Documents, particularly those of the same Author(s) i.e. a shared relationship. Further, it does not seem necessary to know about Acknowledgements which are not related to a Document and so the relationship can also be considered to be dependent.

23. $W_{1,M}P_{1,M}$: Shared, Dependent, Multi-valued, Essential

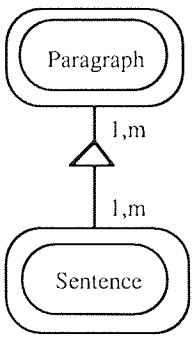


Figure 6.49 A shared, dependent, multi-valued, essential relationship *after* Rumbaugh (1991) and Blair et al. (1991).

Within the electronic Document example, the relationship between the classes Paragraph and Sentence fits this structure if certain assumptions are accepted. A Paragraph must contain at least one Sentence, typically many. From the other direction, if it is assumed that it is not necessary to know about individual Sentences, unless they are part of a particular Paragraph and that the same Sentence

can be used in many Paragraphs, then the Sentence-Paragraph relationship can be considered to be one-to-many i.e. shared, dependent. This is the stance adopted by Kim et al. (1989) throughout the electronic Document aggregation. However, although it is acceptable to disregard Sentences that do not belong to any Paragraph, it would be advantageous for a publishing company to retain isolated Chapters, Sections, and Paragraphs, and so these are considered to be independent classes in this analysis.

24. $W_{1,M}P_{0,M}$: Shared, Dependent, Multi-valued, Non-essential

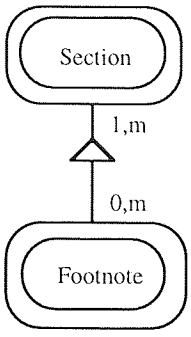


Figure 6.50 A shared, dependent, multi-valued, non-essential relationship.

The relationship between a Section of a Document and its Footnotes is of this structure as shown in figure 6.50. A Section may have a number of Footnotes but does not have to have any i.e. a multi-valued, non-essential relationship. Considering the relationship in the other direction, the same Footnote may exist in a number of different sections i.e. a shared relationship. Further, due to the fact that Footnotes act as a qualification and do not make sense on their own it is sensible to consider the relationship to be dependent.

6.4 ASSOCIATION RELATIONSHIPS

Robinson and Berrisford (1994) describe association as "a relationship between two independently identifiable objects" e.g., a Doctor is associated with a Patient. Although there is a mapping between the two classes, Doctor and Patient, they are otherwise unrelated. Associations (also known as instance connections (Coad and Yourdon, 1991) and as weak references (Kim et al., 1989)) are considered the most general type of relationship between classes. The semantic dependency that underlies an association relationship is weaker than its specialised form, aggregation. However, it is not always clear whether a relationship in the real world should be modelled as an aggregation or association. "The decision to use aggregation is a matter of judgment and is often arbitrary. Often it is not obvious if

an association should be modeled as an aggregation ... modeling requires seasoned judgment and there are a few hard and fast rules." (Rumbaugh, 1991).

Given an association relationship from one class to another class, there is typically a reverse relationship. For example, a Doctor sees many Patients and a Patient is seen by one Doctor. This is contrasted with an aggregation relationship which implies a direction (in addition to the ability to navigate) from a 'whole' class to its 'part' classes (Booch, 1994). An association also has a cardinality and can be described as optional or mandatory at either end. Combining these two aspects, the following combination of mappings exists:

- 1 : 1;
- 1 : 0, 1;
- 1 : 0, m;
- 1 : 1, m;
- 0, 1 : 0, 1;
- 0, 1 : 0, m;
- 0, 1 : 1, m;
- 0, m : 0, m;
- 0, m : 1, m;
- 1, m : 1, m.

These cases will form the basis of the analysis that follows. Although Robinson and Berrisford (1994) highlight additional mappings and semantic variations that can be taken into account with regard to associations, it is thought unnecessary to overburden the analysis with them, particularly as the specified mappings are considered rare. Further, they cannot be represented with the chosen notation and indeed with many other entity modelling notations. These cases include:

- 1 : 0 or m excluding 1;
- 1 : m excluding 0 or 1;
- 0, 1 : m excluding 0 or 1;
- fixed master;
- changeable master;
- monogamous detail;
- polygamous detail;
- variations on the meaning of optionality.

The set of 10 common alternatives is discussed below and an attempt made to match each structure with a real-world example. A number of sources (Bertino and Martino, 1993; Booch, 1994; Bowers, 1992; Brown, 1991; Coad and Yourdon, 1991; Khoshafian and Abnous, 1990; Kim, 1991; Kroha, 1993; Pratt and Adamski, 1994; Rumbaugh, 1991) were used to compile a list of association examples. A total of 79 relationships was established against which the ten structures were compared. Of these 79 examples, 20 constitute a Sales-Product database, 13 a Vehicle-Franchise database, 6 a Vehicle-Ownership database, 17 a

College/University database, 8 a ResearchGroup-Project database, 6 a Publisher database and 4 an Aircraft-Flight database.

1. $O_{1,1}O_{1,1}$: One-to-One Association

Figure 6.51 A one-to-one association relationship.

A few examples of this structure could be found. The example in figure 6.51 shows the relationship between a Chairman or Head of Department and a Department of a University. Typically, each Department will have exactly one Head of Department and a Head of Department can only be responsible for one Department at any point in time. The very semantics attached to the definition of a Chairman make the class independent i.e. if a Person is not currently a Head of Department then he/she is not a Head of Department and belongs only to say a Lecturer class. This relationship, therefore, depicts a role that the class Person or Lecturer takes on over a period of time. On the other hand however, the above structure does not however reflect the situation where the Head of Department has left the Department and the Department is seeking to fill the position.

2. $O_{1,1}O_{0,1}$: One-to-One Association with Optionality

Figure 6.52 A one-to-one association relationship with optionality.

Only three examples could be found to fit this structure where one class is dependent on another class but not vice-versa. Figure 6.52 shows that a Sale may or not involve a CreditCardTransaction, however a CreditCardTransaction must belong to exactly one Sale.

3. $O_{1,1}O_{0,M}$: One-to-Many Association with Optionality at the 'Many' End

Figure 6.53 A one-to-many association relationship with optionality at the 'many' end.

This structure proved to be popular amongst the examples analysed. The example in figure 6.53 shows that a ClerkPerson could be responsible for many LegalEvents related to the title and registration of a Vehicle. However, some ClerkPersons may not be responsible for such tasks. On the other hand, a LegalEvent must be carried out by exactly one ClerkPerson.

4. $O_{1,1}O_{1,M}$: One-to-Many Association

Figure 6.54 A one-to-many association relationship.

Again this proved a popular example amongst the examples analysed. The example in figure 6.54 shows the relationship between an Order and its OrderLines. An Order is for a number of Products (at least one, otherwise it would not exist) and so will have a number of OrderLines. On the other hand, each OrderLine must belong to exactly one Order i.e. it does not make sense to have OrderLines without its corresponding Order.

5. $O_{0,1}O_{0,1}$: One-to-One Association with Optionality at Both Ends

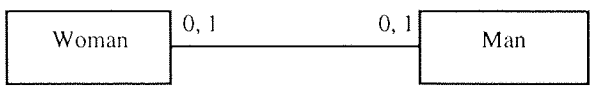


Figure 6.55 A one-to-one association relationship with optionality at both ends.

An example of this type could not be found in the relationships in the example database areas analysed. However, Bowers (1992) uses the example in figure 6.55 of the 'married' role between a Woman and a Man. In our traditional society, a Man *may* be married to *exactly* one Woman and vice-versa.

6. $O_{0,1}O_{0,M}$: One-to-Many Association with Optionality at Both Ends

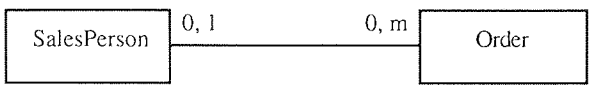


Figure 6.56 A one-to-many association relationship with optionality at both ends.

This proved to be one of the least popular structures. The example in figure 6.56 shows the relationship between a SalesPerson and the Orders he/she activates. A SalesPerson typically generates many Orders, and if it is assumed that the class Orders refers to only active Orders, then the SalesPerson may not have any active Orders at a particular point in time. If it is assumed that Orders can be generated by means other than the cold-calling of SalesPeople and that only one SalesPerson visits a Customer, then an Order may or may not be associated with one SalesPerson.

7. $O_{0,1}O_{1,M}$: One-to-Many Association with Optionality at the 'One' End

Figure 6.57 A one-to-many association relationship with optionality at the 'one' end.

Figure 6.57 shows the relationship between a Sale and Item where Item refers to the physical existence of a Product as opposed to the Product type i.e. stock. A Sale is for at least one Item (if it is assumed that there are no Sales of Services). An Item is either related to exactly one Sale or may not have been sold yet.

8. $O_{0,M}O_{0,M}$: Many-to-Many Association with Optionality at Both Ends

Figure 6.58 A many-to-many association relationship with optionality at both ends.

This did not prove a popular structure. The example in figure 6.58 shows the 'owns Company shares' relationship between a Company and Person. A Company may have a number of ShareHolders, possibly none, and a Person may hold zero or many shares in a Company.

9. $O_{0,M}O_{1,M}$: Many-to-Many Association with Optionality at the 'One' End

Figure 6.59 A many-to-many association relationship with optionality at the 'one' end.

This is a fairly popular structure and one such example is shown in figure 6.59. A Lecturer makes use of TeachingFacilities to be able to perform a Lecture. A Lecture requires a least one TeachingFacility e.g., a Room. TeachingFacilities will typically be used to perform a number of Lectures. However, a TeachingFacility may no longer be the state of the art and so may go unused.

10. $O_{1,M}O_{1,M}$: Many-to-Many Association

Figure 6.60 A many-to-many association relationship.

Figure 6.60 shows a many-to-many relationship which is fairly popular but which can sometimes be broken down into two relationships. For example, within the Vehicle-Ownership database, a Vehicle can be purchased by one or many Owners and an OwnerPerson can purchase one or many Vehicles. From the point of view of an office responsible for the legal events surrounding such purchasing, it is only necessary to be aware of Vehicles with Owners and vice-versa and hence there exists no optionality.

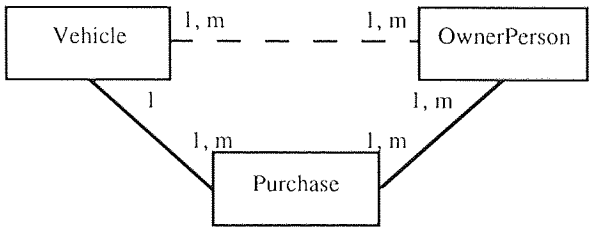


Figure 6.61 A many-to-many association relationship broken down into two relationships between a relationship class.

The relationship can be broken down to two relationships between the class Purchase as shown in figure 6.61. A Vehicle can be part of a number of Purchases but any particular Purchase is for one Vehicle only. An OwnerPerson can be involved in a number of Purchases and a Purchase can be for a number of Owners.

6.5 CONCLUSION

This chapter has identified and analysed a very large set of relationships found in the object data model. It has provided a theoretical framework for relationships between two objects through a synthesis of the ideas found in the literature. This has proved an education in itself and has given an insight into the likelihood of the

existence of such structures. Indeed, from a total of 46 cases, 15 were found to be less plausible. Specifically, it was difficult to find an example for 5 out of 12 of the inheritance structures, it was difficult to reason the plausibility of the intra-class exclusive aggregation structures, of which there were 8, and 2 out of 10 association structures did not have examples in the set of those studied. From this large set of possible conceptual structures, the next chapter will consider to what extent information can be gained to influence implementation decisions and storage strategies.

CHAPTER SEVEN

IMPLEMENTATION OF OBJECT RELATIONSHIPS

The performance of object databases, the subject of this thesis, has been cited as being a critical issue because of the impact on the take-up of the product. As indicated in chapter one, storage and specifically clustering strategies were found to be fruitful yet relatively unexplored territories. Prior to leaping into an examination of the object data model and more particularly object clustering strategies, it was felt wise to review previous experiences with traditional databases. Therefore, chapter two covered the physical model and chapter three reviewed the very popular relational model. Indeed, traditional concepts of the physical model are just as relevant today as they remain a stable and consistent element in any database. A review of the relational model provided an insight into how the concepts of a logical model can indicate clustering strategies at the physical level. Chapter four then covered the object data model. Alongside chapter three, it was apparent that the object data model is more complex and richer than previous models. It was expected that this aspect would provide greater opportunities with respect to clustering strategies in object databases. Indeed, a review of theoretical clustering strategies in chapter five shows this to be the case. However, these theoretical strategies have not been adequately thought through and applied. The designer still has a very difficult task. It became clear that there exists a need to consider more accurately the transition from logical modelling concepts to the implementation of clustering strategies. Chapter six started this process by providing a thorough examination of all possible types of conceptual object model structures. In itself, it has helped to clarify the validity of such structures in terms of clustering. This chapter will now consider the implementation of these structures in terms of clustering with a view to improving performance.

Throughout the following discussion, it is necessary to bear in mind the different viewpoints of chapter six and seven. Chapter six was concerned with object structures at the conceptual level whereas this chapter is concerned with the implementation level. An analyst or problem domain specialist represents this domain of discourse in terms of a conceptual model. In doing so, wider knowledge and assumptions come into play in order to make use of the tools of the model. With the object model, the analyst is particularly aware of inheritance and aggregation structures as aids to reduce complexity. Coad and Yourdon (1991) consider that "in using structures, analysts push the edges of the system's responsibilities within a domain uncovering additional Class&Objects (e.g., implicit in a "requesting document") that might otherwise be missed." The conceptual

model must then be transformed into an implementable design model bearing in mind the physical characteristics of the database.

The performance of any arrangement of data on disk depends upon the activity of the running of the database. This chapter therefore needs to consider both the theoretical clustering strategies of chapter five and the kind of queries which could be applied to the conceptual structures of chapter six. The process should bring together theoretical with actual concerns and so increase the understanding on how clustering can be applied. Taking the lead from chapter six, this chapter will look at inheritance, aggregation and association structures respectively. Each section will firstly consider the implementation of the various conceptual structures. Taking into account the various criteria applicable to each type of structure, a set of implementation structures will be produced. Feasible clustering strategies will then be considered against each implementation structure to produce actual clustering strategies. Finally, the types of queries that may be applied to the particular type of structure will be considered and then applied to each individual clustering strategy.

7.1 INHERITANCE

7.1.1 Implementation Structures

Within an inheritance hierarchy, relationships are not on an object to object level. Therefore, clustering is limited. Objects from two or more classes of an inheritance hierarchy may still be clustered in a single file. However, due to the lack of a relationship between individual objects, the clustering of objects, whereby similar objects (i.e. those with matching keys or OIDs) are placed physically adjacent to one another, is not possible. There are only two options with respect to the ordering of objects belonging to an inheritance hierarchy; order on a common attribute (which must therefore be an attribute of the root class) or order sequentially.

Now consider the significant criteria for inheritance hierarchies found in chapter six. The criteria for single and multiple inheritance will be considered separately apart from the criterion regarding real and abstract superclasses which applies to both situations.

Single inheritance
- the number of children classes;
- overlap of subclasses;

Multiple inheritance
- parent classes belonging to the same or different hierarchies;

General
- real versus abstract superclasses.

Real versus Abstract Classes

On a conceptual footing, abstract and real classes were distinguished in chapter six. The concern was to represent the real world as closely as possible. Abstract classes were used as a means of factoring out common elements among a group of classes. It was difficult at that point to accept the concept of real classes. However, moving onto the implementation of these structures presents a different situation. Here, the representation of real and abstract classes is concerned with instantiation i.e. whether a class can have objects created for it. The terms concrete and virtual will from now on be used in place of real and abstract respectively in order to distinguish between the conceptual and implementation viewpoints. The notation from chapter six for representing inheritance hierarchies will therefore change accordingly. Brackets and the '+' sign will continue to be used to represent subclasses and an overlap respectively. However, the letters 'V' and 'C' will be used in place of the letters 'A' and 'R' respectively, as for example A(R+R) will become V(C+C).

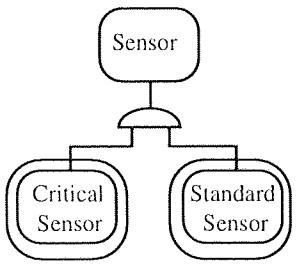


Figure 7.1 Virtual superclass and two subclasses *after* Coad and Yourdon (1991).

Although a class at the conceptual level may be abstract, in implementation terms, objects may still be instantiated for that class i.e. a concrete class. There are a number of reasons for this. One reason is that not all subclasses may be specified and so objects belonging to unspecified classes are placed in the superclass. A second reason is that it may not be necessary or desirable to know all information pertaining to all types of object. Robinson and Berrisford (1991) cite a third reason, the superclass is a parallel class to its subclasses whereby when an object is created in a subclass, an object is also created in the superclass. However, this reason is really confusing the virtual versus concrete issue with the type of storage model employed. The above is essentially describing the distributed storage model covered in chapter five. A final reason is the fact that the superclass represents a basic type of object and the subclass represents a type of object with extra services. This can be viewed in one of two ways, as highlighted by Coad and Yourdon

(1991) with the following example. The superclass Sensor may have two subclasses, CriticalSensor and StandardSensor (see figure 7.1).

Alternatively, as the StandardSensor class provides no extra services, another representation is shown in figure 7.2. The superclass Sensor is concrete, representing StandardSensor objects, and has a single subclass, CriticalSensor.

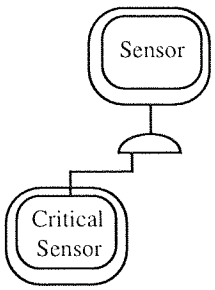


Figure 7.2 Concrete superclass and one subclass *after* Coad and Yourdon (1991).

This situation is really the only true conceptual reasoning for a real superclass. Concrete superclasses at the implementation level may therefore exist as a direct result of real superclasses at the conceptual level.

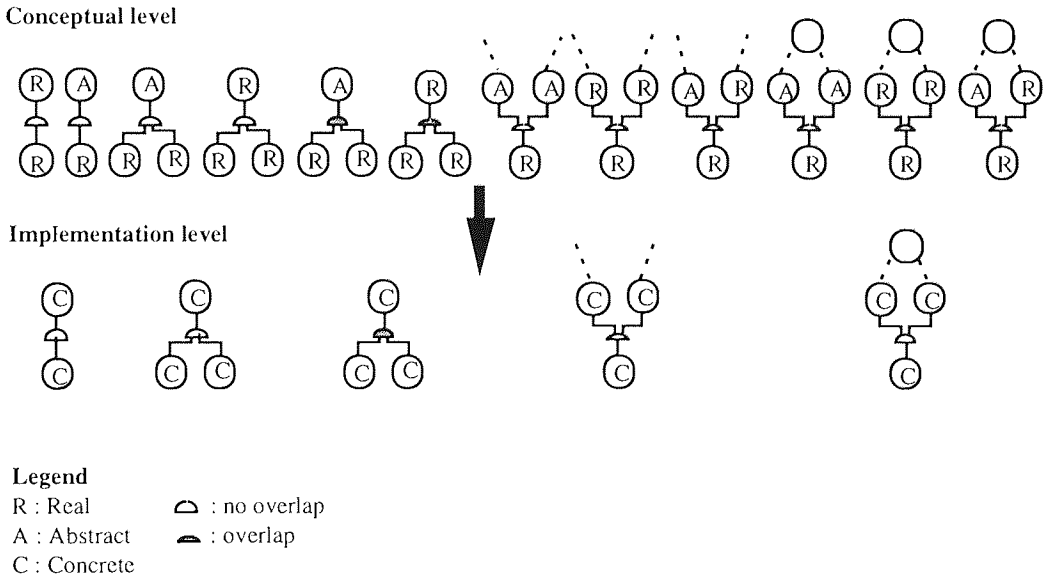


Figure 7.3 Inheritance cases after applying "all classes are concrete" assumption.

The above discussion shows that, in terms of implementation, the typical requirement is for concrete classes. Further, the implementation language may not allow specification of virtual and concrete classes and so all classes would have the potential to be instantiated. Therefore, this analysis will from now on assume all

classes to be concrete. Figure 7.3 shows the implementation structures to be further considered after the "all classes are concrete" assumption is applied.

Single Inheritance

Table 7.1 provides a reminder of the cases for consideration in this section.

SINGLE INHERITANCE
CC
C(CC)
C(C+C)

Table 7.1 Single inheritance hierarchy cases for further consideration.

Number of Children Classes

Figure 7.4 reflects the issue under consideration in this section. Conceptually for a given structure, there may exist (at the current point in time) a single subclass. However, this does not preclude the possibility of there existing another subclass. In fact, the very reason for modelling such a structure may be that the analyst envisages the class to be part of a more complex organisation of classes with potential siblings. Moving to an implementation perspective therefore, case C(CC) is more likely. The case C(C) is mapped to the more general structure C(CC) as shown in figure 7.5.

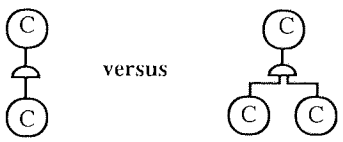


Figure 7.4 Single subclass versus two subclasses.

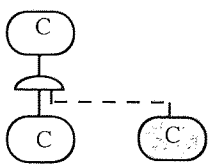


Figure 7.5 Mapping of case C(C) to C(CC).

Figure 7.6 shows the effect on implementation structures when the above decision is applied.

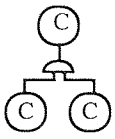


Figure 7.6 Inheritance cases after applying "two subclasses" decision.

Overlap of Subclasses

Figure 7.7 reflects the conceptual structure differences under consideration in this section.

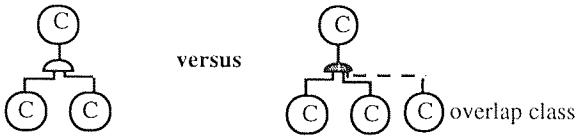


Figure 7.7 No overlap versus overlap structures.

When an overlap of subclasses exists, essentially this means that there are three subclasses. This is radically different to the case of two subclasses with no overlapping class from the point of view of implementation, particularly in terms of clustering, because it presents more options with respect to the placement of objects. Therefore, it is necessary to take into account both types of structure for the purposes of clustering and the two conceptual structures cannot be generalised into a single implementation structure.

Multiple Inheritance

Table 7.2 provides a reminder of the cases for consideration in this section.

MULTIPLE INHERITANCE
C..C..(CC)C
C..(CC)C

Table 7.2 Multiple inheritance hierarchy cases for further consideration.

Parent Classes Belonging to the Same or Different Hierarchies

Figure 7.8 shows the different multiple inheritance structures under consideration.

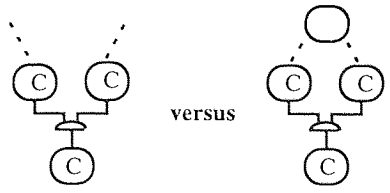


Figure 7.8 Multiple inheritance structures.

The difference in the above structures has an influence on storage. In terms of the placement of objects, the existence of a common parent presents different choices compared to the different parent case. Therefore, it is necessary to take into account both types of conceptual structure, producing an equivalent set of implementation structures.

7.1.2 Clustering of Inheritance Implementation Structures

The above has reduced the inheritance structures to essentially four types of implementation structure as shown in figure 7.9.

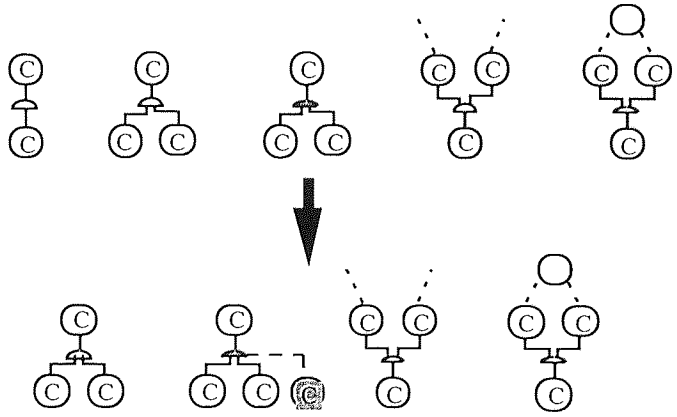


Figure 7.9 Inheritance implementation structures.

Now consider the clustering of these cases. As there does not exist object level relationships, the situation is very much simplified. It must be noted that with object level relationships, there exists a number of such relationships. However, with an inheritance hierarchy, the concern is at a higher level and the relationship between a set of classes is singular. Therefore, given a set of classes on an inheritance hierarchy, the choices are simply to cluster those classes in a single file or separate files. The clustering of a parent class with a child class is the same as clustering a child class with a parent class because the level of interest is not at the object level. A set of classes on an inheritance hierarchy may be thought of as a set

of relations or files in the traditional sense (see figure 7.10). These could simply be mapped to separate traditional files. This would provide the minimum level of clustering as in relational databases. It is assumed that object databases provide this level of clustering implicitly.

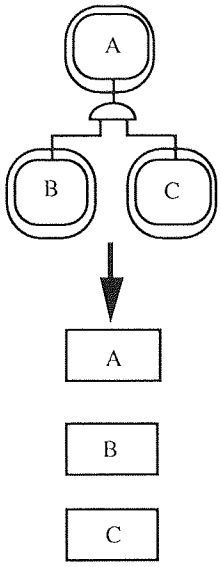


Figure 7.10 Mapping inheritance classes to files.

The process of implementation with clustering is concerned with which classes to cluster together into single files and which to keep separate. For instance, given three classes, A, B and C, A and B could be clustered in a single file with C in another file. Notice that clustering A with B is the same as clustering B with A because there is no concern for one-to-many or many-to-many object level relationships. The total clustering options therefore resolve to all possible combinations of the classes rather than permutations. Table 7.3 provides all the combinations for each of the above inheritance implementation cases.

On consideration of clustering, it is necessary to distinguish between the various classes of an implementation structure. Therefore, parent classes are labelled P and child classes are labelled C. P', P", C' and C" are used to distinguish more than one parent or child class. PP and PP' are used for root classes in multiple inheritance cases. A comma is used to show classes in separate files and a plus (+) sign is used to show classes clustered in a single file. The following sections discuss each case in turn.

	C(CC)	C(C+C)	C..C..(CC)C	C..(CC)C
1	P, C, C'	P, C, C', C''	P, P', C	P, P', C
2	P+C, C'	P+C, C', C''	PP+P, P', C	PP+P, P', C
3	P+C', C	P+C', C, C''	PP+P, P'+C	PP+P, P'+C
4	P, C+C'	P+C'', C, C''	PP+P', P, C	PP+P', P, C
5	P+C+C'	P+C, C'+C''	PP+P', P+C	PP+P', P+C
6		P+C', C+C''	PP+C, P, P'	PP+P+C, P'
7		P+C'', C+C'	PP+C, P+P'	PP+P'+C, P
8		P, C+C', C''	PP+P+P', C	PP+C, P', P
9		P, C+C'', C'	PP+P+C, P'	PP+C, P'+P
10		P, C'+C'', C	PP+P'+C, P	P+P', C
11		P+C+C', C''	PP'+P', P, C	P+C, P'
12		P+C'+C'', C	PP'+P', P+C	P'+C, P
13		P+C+C'', C'	PP'+P, P', C	P+P'+C
14		P, C+C'+C''	PP'+P, P'+C	PP+P+P'+C
15		P+C+C'+C''	PP'+C, P, P'	
16			PP'+C, P+P'	
17			PP'+P'+P, C	
18			PP'+P'+C, P	
19			PP'+P+C, P'	
20			P, P'+C	
21			P', P+C	
22			P+P', C	
23			P'+P+C	

Table 7.3 Inheritance hierarchy clustering combinations.

Case C(CC): Single Parent, No Overlap

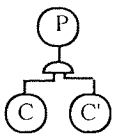


Figure 7.11 Labelled implementation structure for case C(CC).

Figure 7.11 provides a reminder of this structure and shows the labelling of the parent and child classes. The clustering options available with three classes are to cluster all classes in separate files, to cluster all classes together in a single file or to cluster two out of the three classes in a single file with the third class in another file. This leads to a total of 5 clustering strategies for this structure as shown in table 7.3 (column one).

Case C(C+C): Single Parent, Overlap

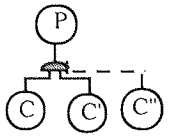


Figure 7.12 Labelled implementation structure for case C(C+C).

Figure 7.12 shows the labelling for this structure. Notice this structure effectively produces a fourth class containing those objects of the overlap, labelled C''. The clustering strategies for this structure therefore resolve to the total combinations for four items. This leads to a total of 15 strategies as shown in table 7.3 (column two).

Case C..C..(CC)C: Multiple Parents, Different Trees

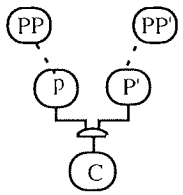


Figure 7.13 Labelled implementation structure for case C..C..(CC)C.

Figure 7.13 shows the labelling for this structure. This analysis is primarily concerned with a single level, parent-child relationship. However, to consider the affect of multiple inheritance, there is a requirement to appreciate the root parents in this structure. The clustering of the parent and child classes must be considered as well as the clustering of these classes with the root parents. The focus remains on the parent-child structure and so the clustering of the root parents is irrelevant on consideration of the parent, child clustering. Therefore, the clustering strategies for this structure are not simply the total combinations of five items. Instead, it is necessary to consider the clustering of PP, PP', P, P' and C as well as the clustering of P, P' and C. All possible combinations of clustering P, P' and C with PP and PP' are shown in table 7.3 (column three) but some of these strategies can be disregarded with more thought. For instance, as P has no direct inheritance relationship with PP', it would not make sense to cluster only these two classes together in single file. The same holds for similar strategies involving P' and PP. For example, consider the inheritance hierarchy of figure 7.14. Classes Car and House originate from different trees and so their objects are inherently dissimilar. A query directed at the class Vehicle would not be interested in House objects. The

clustering of a class with a class from a different tree is not a reasonable strategy. Therefore, strategies 4, 5, 8, 10, 13, 14, 17 and 19 of table 7.3 (column three) can be ruled out.

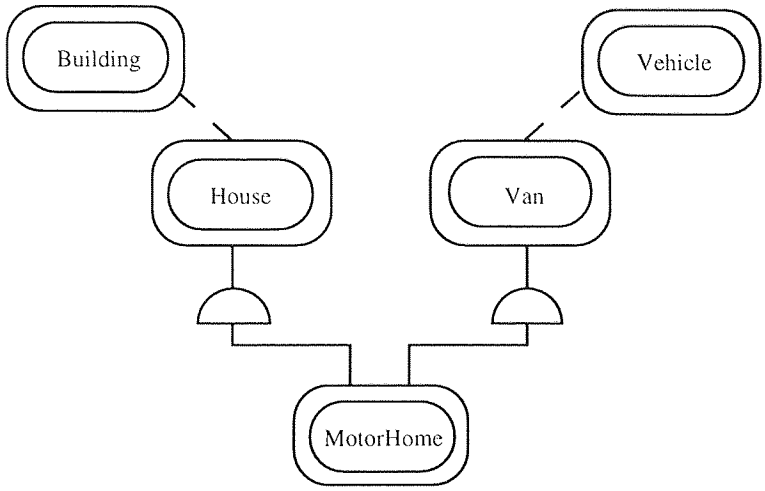


Figure 7.14 Example of multiple inheritance with root parents from different trees.

Intuitively, the clustering of two classes which belong to the same inheritance hierarchy but are more than two levels apart is not a sensible strategy. For instance, the clustering of PP and C only in a single file does not appear a fruitful strategy. It is difficult to envisage a query against a class that is concerned with that class and a class more than two levels apart. However, it makes sense for a query to be interested in all the classes of an inheritance hierarchy path. Therefore, strategies 6, 7, 15 and 16 of table 7.3 (column three) should be excluded but also, it is important to be aware that the clustering of PP with P (and possibly C) and of PP' with P' (and possibly C) means the clustering of all the classes from PP to P (or C) and PP' to P' (or C).

Case C..(CC)C: Multiple Parents, Same Trees

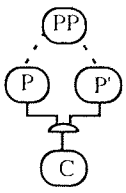


Figure 7.15 Labelled implementation structure for case C..(CC)C.

Figure 7.15 shows the labelling for this structure. Again, to consider the impact of multiple inheritance, clustering the parent and child classes needs to considered as

well as the clustering of these classes with the root parent. Here, however, there is a common root parent. It is anticipated that the clustering of the parent and child classes with the root parent makes more sense than the uncommon parent structure. Table 7.3 (column four) shows the clustering of three items P, P' and C and of four items PP, P, P' and C. For the same reasoning as above, strategies involving PP with just C are not viable. Therefore, strategies 8 and 9 of table 7.3 (column four) can be excluded. Given these arguments for excluding some strategies as being less viable, table 7.4 presents a new set of potential clustering strategies for inheritance hierarchies.

	C(CC)	C(C+C)	C..C..(CC)C	C..(CC)C
1	P, C, C'	P, C, C', C''	P, P', C	P, P', C
2	P+C, C'	P+C, C', C''	PP+P, P', C	PP+P, P', C
3	P+C', C	P+C', C, C''	PP+P, P'+C	PP+P, P'+C
4	P, C+C'	P+C'', C, C''	PP+P+C, P'	PP+P', P, C
5	P+C+C'	P+C, C'+C''	PP'+P', P, C	PP+P', P+C
6		P+C', C+C''	PP'+P', P+C	PP+P+C, P'
7		P+C'', C+C'	PP'+P'+C, P	PP+P'+C, P
8		P, C+C', C''	P, P'+C	P+P', C
9		P, C+C'', C'	P', P+C	P+C, P'
10		P, C'+C'', C	P+P', C	P'+C, P
11		P+C+C', C''	P'+P+C	P+P'+C
12		P+C'+C'', C		PP+P+P'+C
13		P+C+C'', C'		
14		P, C+C'+C''		
15		P+C+C'+C''		

Table 7.4 Inheritance hierarchy clustering strategies.

An added dimension with the clustering of inheritance hierarchies is the type of storage model employed. Recall from chapter five that there is the static storage model (SSM) and the collected storage model (CSM) for the storage of objects belonging to an inheritance hierarchy. This factor in itself provides degrees of clustering. In the SSM approach, objects contain both inherited and non-inherited attributes. In the CSM approach, objects contain only non-inherited attributes and therefore superclasses contain objects pertaining to that class and all its subclasses. One clustering strategy could therefore be to reverse the affect of the CSM approach and cluster all attributes pertaining to the same object together. However, it must be noted that this concept is separate (but relevant) to the issues presented above. The clustering strategies deduced above could be applied under the SSM or CSM approach. Under each approach, it is necessary to be aware of the nature of objects

contained in superclasses and subclasses. For example, for case C(CC) under the SSM approach, class P represents only P objects whereas under the CSM approach, class P represents objects of P, C and C'. This must be borne in mind in considering the impact of queries under different clustering arrangements.

7.1.3 Queries

Consider the type of queries that could be applied to an inheritance structure. Recall from chapter four that Kim (1991) distinguishes between weak and strong queries with respect to inheritance structures for the object data model. A query taken in the weak sense is applied to the target class only whereas a query taken in the strong sense takes on the is-a relationship between classes literally and the query is applied to the target class and all its subclasses. However, consider this strong versus weak issue more closely. Although there exists an is-a relationship between a set of classes, objects in a parent class *are* different to objects in a child class. Consider especially the structure where the parent class represents a basic type of object such as Sensor and the subclass represents objects with "extra" services such as CriticalSensor. Although it is acknowledged that a CriticalSensor is-a Sensor but with extra services, it is conceptually different to a StandardSensor and CriticalSensor objects are different to Sensor objects. Therefore, a query such as "Get all Sensors" should only really return standard Sensor objects. Alternatively, if the in-built assumption was that queries against a class on an inheritance hierarchy should be against the class and all its subclasses, then "Get all Sensors" would return not only standard Sensors but also CriticalSensors. The result is a set of varying objects which the user may need to distinguish. Therefore, this analysis will from now on take a weak interpretation of queries.

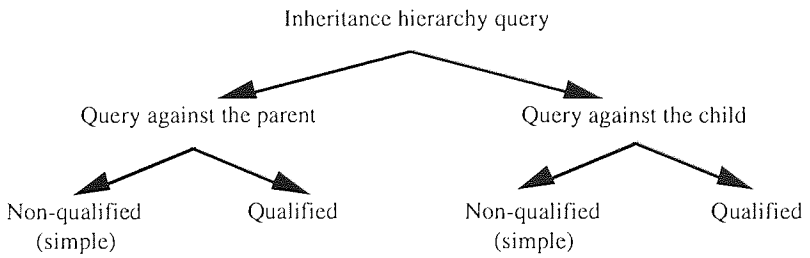


Figure 7.16 Inheritance hierarchy queries.

Recall that this analysis only needs to consider structures of two levels, as decided in chapter six. This allows a parent perspective and child perspective to be gained and these can be applied recursively. Queries against this type of structure therefore may be applied to the parent class or to the child class. Queries may also be

classified into simple and qualified queries (see chapter three). A simple query is a request for all details of all records of a relation, class or set whereas a qualified query is interested in a subset of those objects. Taking these two aspects leads to four basic types of query for further consideration, as shown in figure 7.16.

An inheritance structure adds another dimension to qualified queries. The attribute of interest may or may not be an inherited attribute. In this analysis, this factor has relevance to the child class. Therefore, for inheritance structures, the following queries are of interest:

- simple parent query;
- qualified parent query;
- simple child query;
- qualified child query based on an inherited attribute;
- qualified child query based on a non-inherited attribute.

7.1.4 Application of Queries to Clustering Strategies

Now consider the effect of performing the various queries deduced in the last section under different clustering scenarios. Take the first implementation structure as an example. Table 7.5 highlights the best clustering strategy for each type of query under the static and collected storage model.

			simple parent query	qualified parent query	simple child query	qualified child query inherited attribute	qualified child query non-inherited attribute
S S M	1s	P, C, C'	*	*	*	*	*
	2s	P+C, C'					
	3s	P+C', C			*	*	*
	4s	P, C+C'	*	*			
	5s	P+C+C'					
C S M	1c	P, C, C'	*	*		*	*
	2c	P+C, C'			*		
	3c	P+C', C					*
	4c	P, C+C'	*	*		*	
	5c	P+C+C'					

Table 7.5 The matching of queries to clustering strategies for case C(CC).

For the SSM approach, the first clustering option is best across the board. This clustering approach represents essentially a 'no clustering' strategy where each class

is placed in a separate file. For queries against the parent class, the separate file approach is the best option because P objects are not mixed with any other type of object. This makes a query more efficient because buffers are not filled with unwanted objects, a file is processed with fewer fetches of blocks into buffers and there is no sorting process required to decide the type of object. Likewise, for queries against the child class, strategies where class C stands alone are best.

For the CSM approach, the result is the same as the SSM approach for parent queries. Any strategies where P stands alone are best for the reasons presented above. Although the same strategies as for the SSM approach are best, a query against class P will not be as efficient under the CSM approach. When class P is processed under the CSM approach it is necessary to take into account the different types of objects, specifically P, C and C'.

The CSM approach makes a difference for queries against the child class. In this situation, a C object does not reside in one place. Therefore, for simple queries, the clustering strategy where C is clustered with P is the better strategy. The file per class clustering strategy would require the search of two files, namely P and C. Clustering C with P effectively reconstitutes C objects. For qualified queries concerning inherited attributes, any strategy where P stands alone are best i.e. strategies 1c and 4c of table 7.5. This is due to the fact that inherited attributes in the CSM approach reside in the superclass and not the class in question. This has the advantage that non-inherited attributes do not take up unnecessary space in the buffer but the disadvantage that P objects not only take up space in the buffer but must also be distinguished from C objects. The disadvantage unfortunately outweighs the advantage. For qualified queries concerning non-inherited attributes, as for the static storage model, any strategy where C stands alone is best. The CSM approach does not change the result because non-inherited attributes are contained in the class to which they pertain. The advantage over the SSM approach is that C objects do not contain inherited attributes making them smaller and quicker to process.

To decide the best overall strategy, it is necessary to decide whether the advantages the CSM approach brings outweigh the disadvantages. On the one hand, some queries, such as qualified queries on a non-inherited attribute, are more efficient than in the SSM approach. On the other hand, the CSM approach brings complications not found in the SSM approach, such as for qualified queries on inherited attributes. It is argued that the disadvantages found in the CSM approach make the SSM approach with the file per class clustering approach the best overall

strategy. The SSM approach means that the file per class clustering strategy is good for all queries.

This result can be found to be the same for the other inheritance hierarchy implementation structures. In conclusion, the file per class clustering strategy (essentially a 'no clustering' approach) is the best overall strategy for inheritance hierarchies. Indeed, although the above analysis has looked extensively at the various clustering options, on closer inspection of the application of queries to those strategies, it has been found that clustering per se is not a fruitful design strategy for inheritance hierarchies. The thesis that relationships of the object model can be exploited in the physical design stage still stands because inheritance hierarchies do not have relationships at the object level. This result therefore has been quite correctly predicted. The result is also important because many OA&D (object analysis and design) methods stress inheritance almost to the exclusion of aggregation yet this is unlikely to result in performance advantages.

7.2 AGGREGATION

7.2.1 Implementation Structures

In an aggregation hierarchy, unlike an inheritance hierarchy, relationships exist at the instance level. A part object will be directly related to a whole object. This provides more potential with respect to clustering. Clustering within an aggregation hierarchy can be more finer grained than in an inheritance hierarchy. Recall that for an inheritance hierarchy, objects of different types can be clustered in a single file but the ordering of those objects is limited to a sequential ordering or by a common attribute. An aggregation hierarchy, however, provides the opportunity to place objects physically adjacent to one another based on the whole-part relationship.

RELATIONSHIP OF PART-TO-WHOLE	RELATIONSHIP OF WHOLE-TO-PART
Exclusivity	Valuedness
Dependency	Essentiality

Table 7.6 Aggregation hierarchy criteria.

Now consider the significant criteria for aggregation hierarchies established in chapter six. Table 7.6 provides a reminder. Notice that aggregation hierarchies can be considered from the perspective of the part or the perspective of the whole object. Although, for convenience, this analysis has broken down the aggregation relationship into the two different viewpoints, it must be stressed that only one

clustering strategy can be employed and so only one direction will be able to take precedence. The issue then is which relationship is the most important, the whole-to-part or the part-to-whole relationship? This is an issue for consideration when the possible queries are analysed.

Relationship of Part-to-Whole

To recap on chapter six, this section is concerned with the relationship from the part class point of view and is signified by cardinality adornments as shown in figure 7.17. Therefore, the focus is on how many whole objects there are for an individual part object. Notice the different level of interest compared with inheritance. With aggregation, the concern is with individual objects and not just classes. Therefore, implementation issues must be focussed at the object level as well as the class level.

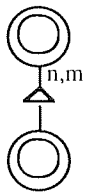


Figure 7.17 Aggregation relationship from the point of view of the part class.

Exclusivity

Figure 7.18 reflects the issue under consideration in this section. The concern is whether an individual part object is related to a single whole object or more than one whole object. This directly affects clustering. The placement of individual objects is straightforward if a part object is related to a single object. However, if a part object is related to more than one object, there exists a choice on the placement of that object.

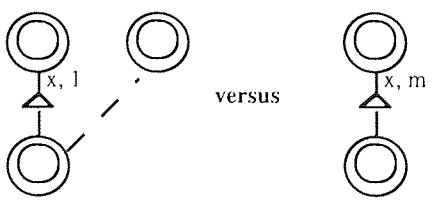


Figure 7.18 Exclusivity in an aggregation hierarchy.

Figure 7.19 reflects the relationship at the object level indicating the impact on clustering.

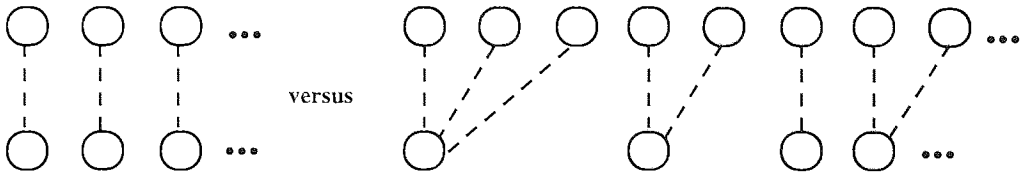


Figure 7.19 Part-to-whole relationship at the object level.

Inter-class exclusivity and intra-class exclusivity provide an added dimension. Recall from chapter six that an inter-class exclusive relationship means that a part object cannot be shared with any other whole object whereas an intra-class exclusive relationship means that a part object can be shared with another whole object of a different type. An inter-class exclusive relationship provides the simplest scenario in terms of implementation and clustering. The only option is simply to cluster the part object with the whole object it is related to. Intra-class exclusivity, however, like a shared relationship presents a choice with respect to where to place the part object. Unlike a shared relationship, the choice is across different types of objects. Therefore, at a class level, three (or more) classes come into the scenario with respect to implementation (see figure 7.20).

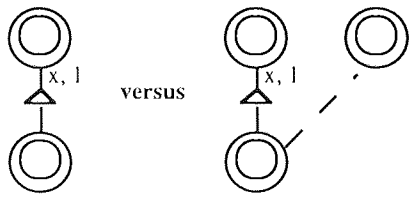


Figure 7.20 Intra-class exclusive, part-to-whole relationship.

In conclusion, exclusivity does impact clustering. Therefore, the conceptual structures cannot be reduced into a smaller set of implementation structures at this point.

Dependency

Figure 7.21 shows the difference in conceptual structure of interest here. To recap, a part object may exist in its own right and may not necessarily be related to a whole object. On the other hand, a part object may not be able to exist in its own right and so will always be related to at least one whole object.

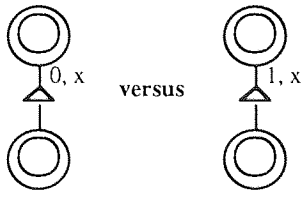


Figure 7.21 Dependency in the part-to-whole relationship.

In terms of implementation, specifically clustering, the dependency criterion does not represent a critical issue. Consider clustering a part object with its corresponding whole objects under the two different scenarios. If at least one corresponding whole object does not exist i.e. an independent relationship, then the part object can simply be stored elsewhere in the file. For instance, there could be a 'dummy' whole object where all part objects with no corresponding whole object could be stored. As far as implementation is concerned therefore, independent and dependent relationships are equivalent. This reduces the conceptual structures to a smaller set of implementation structures as shown in figure 7.22.

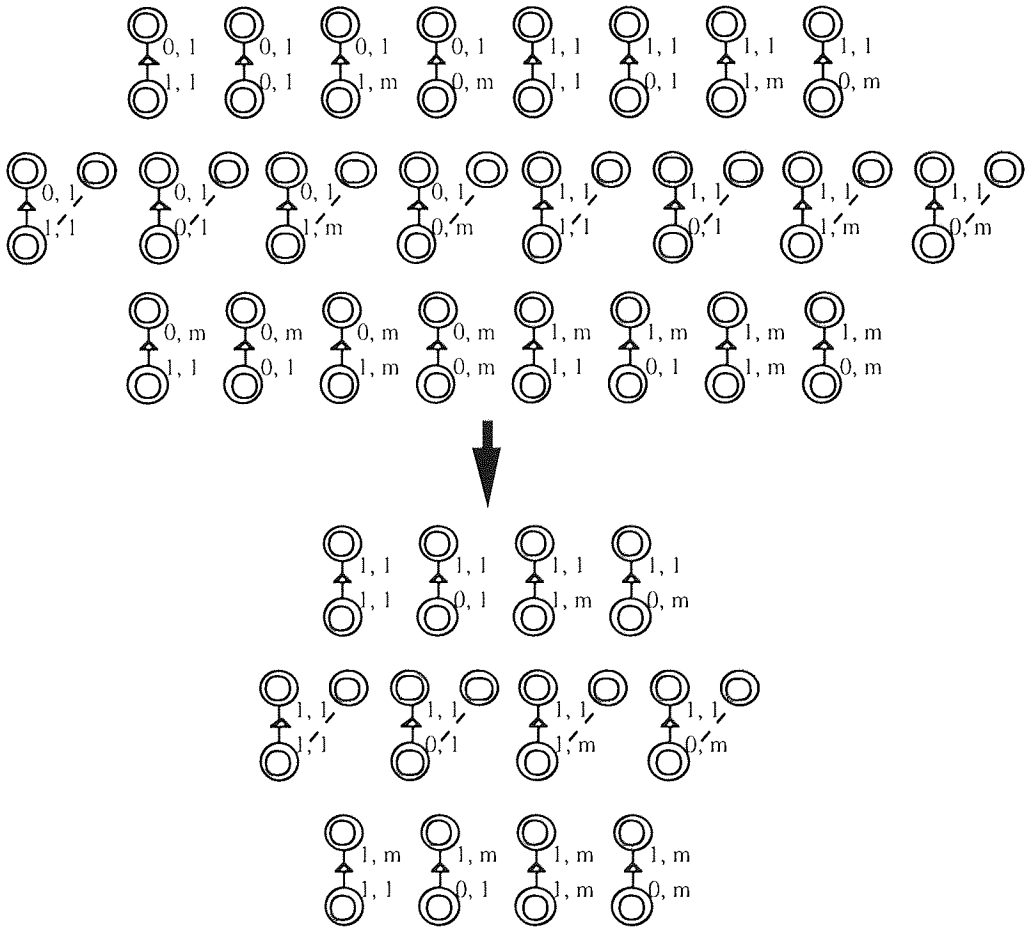


Figure 7.22 Aggregation cases after applying "all parts dependent" decision.

Relationship of Whole-to-Part

This section is focussed on the aggregation relationship from the point of view of the whole object. Recall from chapter six that this means the concern is how many part objects there are to a particular whole object (see figure 7.23). It must be noted that this relationship arises from the construction of complex objects and so the relationship is inherent to the representation of objects in the object model. Recall from chapter four that an object may have attributes whose domain can be a system-defined or user-defined domain. An attribute of a complex object therefore may be an object itself. A whole object therefore, with a number of attributes, will take part in a number of whole-to-part relationships. This must be borne in mind when clustering is analysed. Although, for convenience, a single whole-to-part relationship is analysed, there can be only one clustering strategy in place and all relationships should be viewed in terms of their precedence.

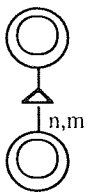


Figure 7.23 Aggregation relationship from the whole-to-part view.

Valuedness

To recap from chapter six, an attribute of a complex object (essentially the whole object) may be single or set (multi) valued. This issue is reflected in figure 7.24.

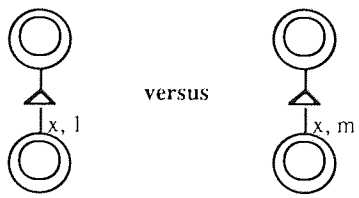


Figure 7.24 Single-valued versus multi-valued aggregation relationship.

As for exclusivity for the part-to-whole relationship, this criterion impacts on clustering as it presents more choices on the placement of individual objects. For single-valued attributes, the whole object may be clustered with its corresponding part object whereas for multi-valued attributes the whole object may be clustered with any of its relating part objects. Therefore, again, this criterion provides no reduction in the implementation structures.

Essentiality

This characteristic is the same as the dependency characteristic for the part-to-whole relationship. A part object may or may not be essential and so in essence the whole object may be dependent or independent of the part object. This is reflected in figure 7.25.

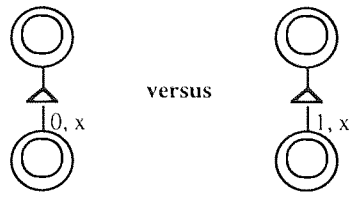


Figure 7.25 Essential versus non-essential aggregation relationship.

In terms of the representation of complex objects, this means that optional attributes are allowed where fixed attributes represent essential relationships between the whole and the part objects and optional attributes represent non-essential relationships. Further, this means that records are of varying lengths.

Again, to enable clustering of whole objects with their corresponding part objects for non-essential relationships, a 'dummy' part object could be employed. This effectively means that in terms of implementation, specifically clustering, optionality can be ignored. This again reduces the set of structures by half as shown in figure 7.26.

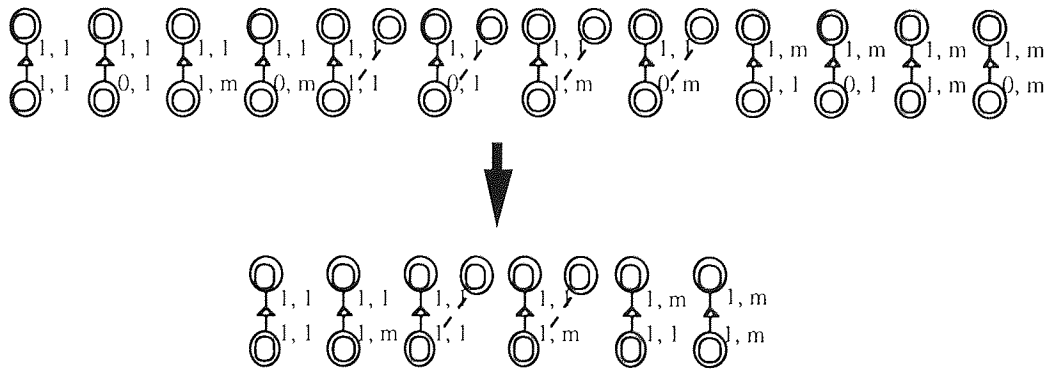


Figure 7.26 Aggregation cases after applying "all wholes dependent" decision.

7.2.2 Clustering of Aggregation Implementation Structures

The above has reduced the aggregation structures to essentially six types of implementation structure. Now consider the clustering of these cases. With

aggregation structures, unlike inheritance structures, there exists relationships at the object level. Notice that at this level there are multiple relationships. Further, the relationships between objects may be one-to-many in either direction. Unlike clustering for inheritance, it is not simply concerned with placing the objects of two or more classes into a single file, but with the actual placement of those objects and the clustering of sets of objects. Therefore, the clustering of (say) class A with class B is different to the clustering of class B with class A. Given a set of classes on an aggregation hierarchy, the clustering of these classes firstly resolves to the set of all possible permutations of those classes. Then, if there exists a one-to-many relationship, there a number of extra clustering options. Table 7.7 shows the various clustering strategies for each structure. The following sections will take each implementation structure in turn and consider the set of all possible clustering arrangements.

	$W_{11}P_{11}$	$W_{11}P_{1M}$	$[W_{11}P_{11}]$	$[W_{11}P_{1M}]$	$W_{1M}P_{11}$	$W_{1M}P_{1M}$
1	W+P	W+P	W+P	W+P	W+P (f)	W+P (f)
2	P+W	P+W (f)	W'+P	W'+P	W+P (r)	W+P (r)
3	W, P	P+W (r)	P+W	P+W (f)	W+P (c W)	W+P (c W)
4		P+W (c P)	P+W'	P+W (r)	P+W	P+W (f)
5		W, P	P+W+W'	P+W (c P)	W, P	P+W (r)
6			W, W', P	P+W'		P+W (c P)
7				P+W,W' (f)		W, P
8				P+W,W' (r)		
9				P+W,W' (c P)		
10				P, W, W'		

Table 7.7 Aggregation hierarchy clustering permutations.

Case $W_{1,1}P_{1,1}$: Inter-exclusive, Dependent, Single-valued, Essential

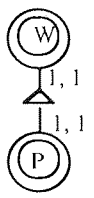


Figure 7.27 Labelled implementation structure for case $W_{1,1}P_{1,1}$.

Figure 7.27 provides a reminder of the structure and figure 7.28 shows more clearly the relationships at the object level. In this case, there are only two classes to consider and the relationships between objects are simply one-to-one. The set of

possible clustering strategies resolves to cluster W with P, cluster P with W or cluster W and P separately, as shown in table 7.7.

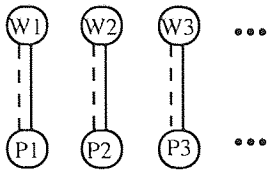


Figure 7.28 Object level relationships for case $W_{1,1}P_{1,1}$.

Case $W_{1,1}P_{1,M}$: Inter-exclusive, Dependent, Multi-valued, Essential

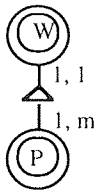


Figure 7.29 Labelled implementation structure for case $W_{1,1}P_{1,M}$.

Figure 7.29 provides a reminder of the structure for consideration here. It is necessary to be aware that the whole-to-part, one-to-many relationship is directly represented in the object model by set-valued attributes. The construction of complex objects therefore is focussed on the whole-to-part relationship. Figure 7.30 depicts the object level relationships for this case.

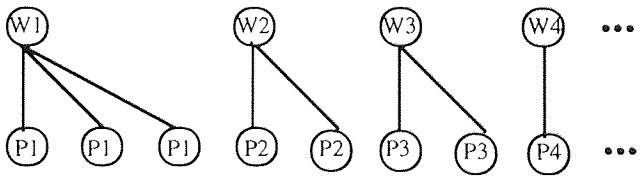


Figure 7.30 Object level relationships for case $W_{1,1}P_{1,M}$.

As there are only two classes involved, at the class level, the clustering options resolve to those above, to cluster W with P, to cluster P with W or to cluster W and P in separate files. However, due to the one-to-many relationship, clustering W with P presents more choices. If P objects are in an unordered, sequential file, then the corresponding W may be clustered with, for example, the first (related) P object, a randomly selected P object or a user-specified P object. On the other hand, if P objects are ordered by the relationship attribute, then each related W

object can follow each cluster of P objects. Figure 7.31 shows the different clustering options for this case using the object level relationships of figure 7.30.

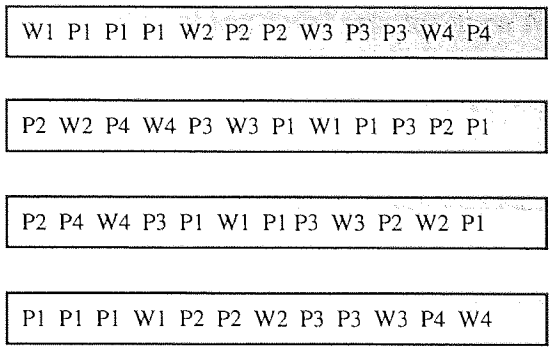


Figure 7.31 Different clustering strategy examples for case $W_{1,1}P_{1,M}$.

Case $[W_{1,1}P_{1,1}]$: Intra-class exclusive, Dependent, Single-valued, Essential

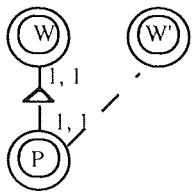


Figure 7.32 Labelled implementation structure for case $[W_{1,1}P_{1,1}]$.

Figure 7.32 provides a reminder of this structure. At the class level, with this case, there are three classes to consider. However, the clustering options available do not simply resolve to all possible permutations of three items because W and W' may share P but they are otherwise unrelated. Taking this into account, table 7.7 shows the possible clustering strategies for this structure.

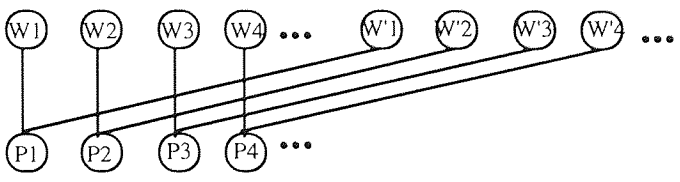


Figure 7.33 Object level relationships for case $[W_{1,1}P_{1,1}]$.

The use of object identifiers in the object model allows (different) whole objects to effectively share part objects. Ignoring replication, to cluster P with W therefore presents a choice, to cluster with W or W'. Notice that the discussion is at the class

level and that there may be any type of object level relationship between W' and P. On the other hand, all whole objects may be clustered with the relating P object. This means that all classes, in this case W, W' and P, are clustered in a single file. Figure 7.33 shows the possible object level relationships for this structure and figure 7.34 shows the corresponding clustering strategies.

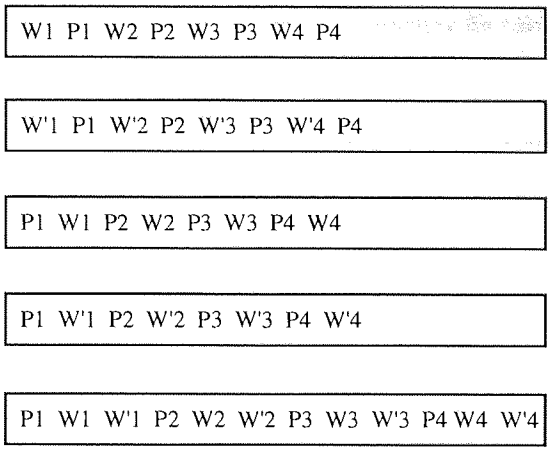


Figure 7.34 Different clustering strategy examples for case $[W_{1,1}P_{1,1}]$.

Case $[W_{1,1}P_{1,M}]$: Intra-class exclusive, Dependent, Multi-valued, Essential

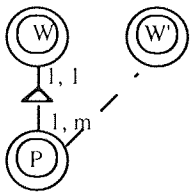


Figure 7.35 Labelled implementation structure for case $[W_{1,1}P_{1,M}]$.

Figure 7.35 provides a reminder of this structure. The discussions above for the case involving a whole-to-part, one-to-many relationship and the case involving an intra-class exclusive relationship are both relevant here. Taking the one-to-many relationship first, if W is clustered with P then there may be a choice of P objects. On the other hand, if P is clustered with W, then there is a choice between W and W'. The various strategies are shown in table 7.7.

Case $W_{1,M}P_{1,1}$: Shared, Dependent, Single-valued, Essential

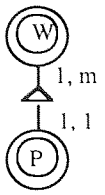


Figure 7.36 Labelled implementation structure for case $W_{1,M}P_{1,1}$.

Figure 7.36 provides a reminder of this structure. This structure arises in the object model because objects may be shared by the use of object identifiers. This is similar to the intra-class exclusive relationship above but sharing is between objects of the same type. In addition, this structure is the reverse of the $W_{1,1}P_{1,M}$ case with the one-to-many relationship being in the part-to-whole direction as opposed to the whole-to-part direction. Therefore, the discussion for the $W_{1,1}P_{1,M}$ case may be applied here.

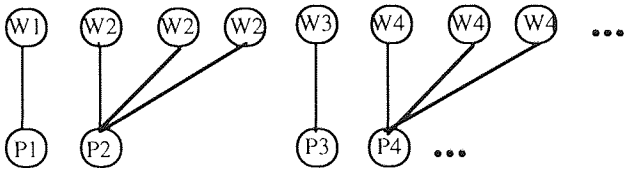


Figure 7.37 Object level relationships for case $W_{1,M}P_{1,1}$.

P1 W1 P2 W2 W2 W2 P3 W3 P4 W4 W4 W4
W3 P3 W1 P1 W4 P4 W4 W2 P2 W2 W4 W2
W3 P3 W1 P1 W4 W4 P4 W2 W2 W4 W2 P2
W1 P1 W2 W2 W2 P2 W3 P3 W4 W4 W4 P4

Figure 7.38 Different clustering strategy examples for case $W_{1,M}P_{1,1}$.

When P is clustered with W, there exists a choice and a P may be clustered with the first occurrence of W, a randomly selected W, a user specified W etc. Table 7.7 shows the various clustering strategies where 'f' indicates clustering with the first occurrence of an object, 'r' indicates clustering with a random object and 'c' indicates clustering with a set of clustered objects. Figure 7.37 shows example

object level relationships for this structure and figure 7.38 shows the corresponding clustering strategies.

Case $W_{1,M}P_{1,M}$: Shared, Dependent, Multi-valued, Essential

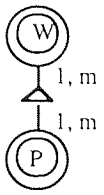


Figure 7.39 Labelled implementation structure for case $W_{1,M}P_{1,M}$.

Figure 7.39 provides a reminder of this structure. This is the most complicated case out of the set because there exists a one-to-many relationship in both directions.

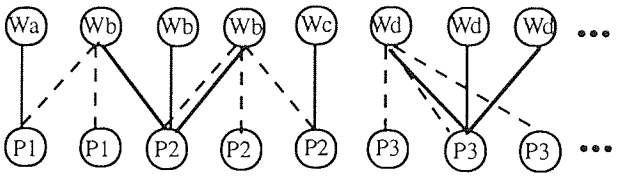


Figure 7.40 Object level relationships for case $W_{1,M}P_{1,M}$.

Wc P2 Wb P2 Wb Wd P3 Wa P1 Wd Wd Wb DummyW P1 P2 P3 P3
Wc P2 Wb Wb P2 Wd Wa P1 Wd Wd P3 Wb DummyW P1 P2 P3 P3
Wa P1 Wb Wb Wb P2 Wc P2 Wd Wd Wd P3 DummyW P1 P2 P3 P3
P3 Wd P2 Wb P2 P1 Wb P3 P1 P2 P3 DummyP Wa Wb Wc Wd Wd
P3 P2 P2 Wb P1 Wb P3 P1 P2 P3 Wd DummyP Wa Wb Wc Wd Wd
P1 P1 Wb P2 P2 P2 Wb P3 P3 P3 Wd DummyP Wa Wb Wc Wd Wd

Figure 7.41 Different clustering strategy examples for case $W_{1,M}P_{1,M}$.

The separate discussions for the one-to-many cases above apply here. However, it must be remembered that only one clustering strategy can be applied at any one time. Therefore, if P is clustered with W so that all parts are with the corresponding

whole, the need to cluster all wholes with its corresponding part cannot also be satisfied and vice-versa. Figure 7.40 shows example object level relationships for this structure and figure 7.41 shows the corresponding clustering strategies.

7.2.3 Queries

Consider the type of queries that could be applied to an aggregation hierarchy. Recall from chapter four that a query language for the object data model must accommodate predicates on nested attributes due to the nested definition of objects. A query therefore may be concerned with the top level object or a nested object. Given the inherent semantics of an aggregation structure, it is unlikely that a query will be posed directly against a nested object. It is more likely for a query to be essentially against the top level object but with reference to or involving nested objects to achieve the end result. It will be assumed from now on that queries are formulated against the topmost class of an aggregation hierarchy but may also involve nested classes. Indeed, if an aggregation hierarchy is representing an assembly of sub-objects of a complex object such as an aeroplane, the need to query the complex object in its entirety can be envisaged. In this case, the query is concerned with all levels of an aggregation hierarchy.

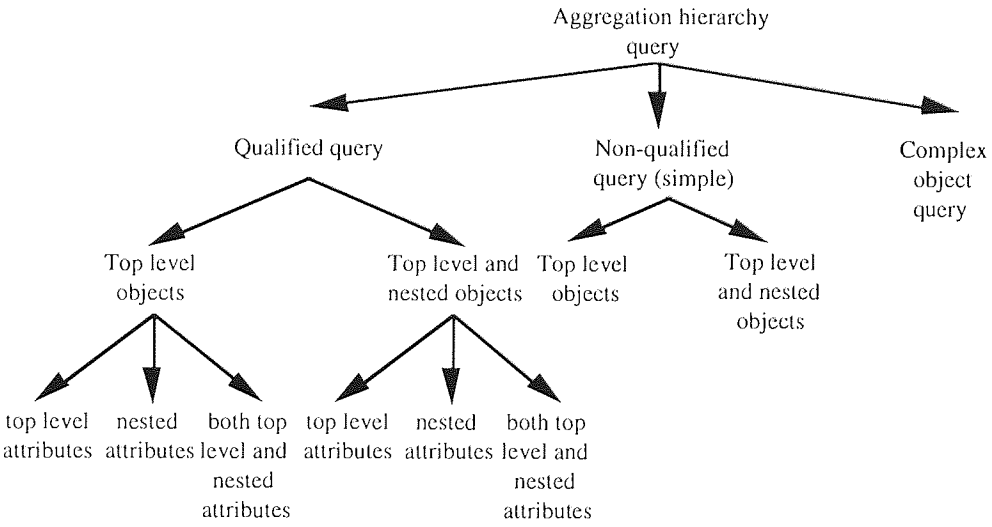


Figure 7.42 Aggregation hierarchy queries.

As for any query in an object database and as shown for inheritance hierarchies above, a query may be simple or qualified. To be more precise, a query may be concerned with all objects of a class on an aggregation hierarchy or with only a subset of those objects qualified by the value(s) of attribute(s). An aggregation hierarchy adds another dimension to both qualified and non-qualified queries. For

qualified queries, the attribute of interest may be at any level of the hierarchy. Although a query is formulated against the top level object, it may involve attributes at any level. Further, the result may involve just the top level objects or both top level and nested objects. For a non-qualified query, the user may be interested in all top level objects or all top level objects and some or all of their sub-objects i.e. complete or partially complete complex objects. Further, the user may be interested in a single complete complex object in its entirety. The various types of query for aggregation hierarchies are shown in figure 7.42.

This analysis is concerned with structures of a single level only. Therefore, queries may be formulated against top level objects or sub-objects of one level deep only. The top level aggregation class is considered the whole and the nested aggregation class the part. Therefore, for aggregation structures, the following queries are of interest:

- simple whole query;
- qualified whole query based on an attribute directly in the whole;
- qualified whole query based on an attribute in the part;
- qualified whole query based on attributes both in the whole and the part;
- simple whole and part query;
- qualified whole-part query based on an attribute directly in the whole;
- qualified whole-part query based on an attribute in the part;
- qualified whole-part query based on attributes both in the whole and the part.

7.2.4 Application of Queries to Clustering Strategies

Now consider the effect of performing the various queries deduced in the last section under different clustering scenarios. The following sections will take each implementation structure in turn.

Case $W_{1,1}P_{1,1}$: Inter-exclusive, Dependent, Single-valued, Essential

Table 7.8 shows the best clustering strategy for each type of query for this implementation structure, where W represents the whole and P represents the part. To clarify the various queries that could be applied to this structure, consider the following example. A Person lives at a Property. For the purposes of this analysis, assume that a Person can only live at one Property and that a Property does not have more than one Person living at it (say because relatives are not employed). Figure 7.43 shows one instance of this complex object where a Person instance is the top level object with an attribute Address whose domain is Property and which is an object in its own right.

	simple W query	qualified W query W attribute	qualified W query P attribute	qualified W query W and P attribute	simple W and P query	qualified W-P query W attribute	qualified W-P query P attribute	qualified W-P query W and P attribute
W+P			**	**	**	**	**	**
P+W			*	*	*	*	*	*
W, P	*	*	*	*				

Table 7.8 The matching of queries to clustering strategies for case $W_{1,1}P_{1,1}$.

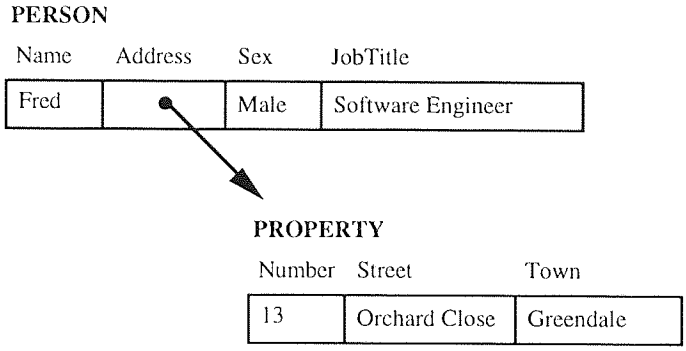


Figure 7.43 Example of a complex object.

The following queries can be envisaged as being feasible for the above structure:

- Get all Persons;
- Get all Persons with the name "Fred";
- Get all Persons who live in "Sheepy Road";
- Get all Persons who are "Male" and who live in "Atherstone";
- Get all Persons and their Addresses;
- Get all Persons and their Addresses who are "Software Engineers";
- Get all Persons and their Addresses who live in "Orchard Close";
- Get all Persons and their Addresses who are "Female" and live in "Long Road".

Consider firstly a simple query against the whole class such as "Get all Persons". As only whole records are of concern and no part records are required for this query, it is obviously better for W and P to be in separate files i.e. strategy W, P. This means that records are smaller, more records are brought into the buffers at any one time, there is no need to determine the type of record and hence the whole file is processed more quickly.

Now consider qualified queries against the whole class. Although the concern is essentially for whole records, the qualification may still be against either the whole or the part class or both. If the qualification is only concerned with whole records, the clustering requirements are the same as for a simple query against the whole

class i.e. strategy W, P. However, if the qualification is concerned with nested attributes such as "Get all Persons who live in 'Sheepy Road'", then it may be better for each P to be clustered with its corresponding W or vice-versa. Consider the processing required to achieve the result of this query. There are two approaches. Firstly, for each object in the Person class, the OID of the Address attribute could be followed, the corresponding Property object brought into the buffers and the Street attribute compared with "Sheepy Road". Alternatively, the Property class could be processed first to obtain a list of Address OIDs where the Street attribute is "Sheepy Road" and then this list processed against the Person class matching the Address attribute with each OID. The latter approach involves two separate activities and so suits the separate file approach i.e. no clustering. On the other hand, the former approach where the processing is integrated, would benefit from the P+W or W+P clustering strategy. If W is clustered with its corresponding P, the results can be achieved in a single run of the file as for each P, W is already in the buffer or is in the next block to be fetched. More naturally, if P is clustered with its corresponding W, then P replaces the OID in place and the complex objects are made into single large records (see figure 7.44). Again, the results can be achieved in a single run of the file and the clustering is akin to the semantics underlying the objects. The benefits this approach brings outweigh the fact that the records are larger. If the qualification involves both top level and nested attributes, then all clustering strategies are again valid, but with strategy W+P being more sensible if an integrated processing approach is taken.

PERSON

Name	Address Number	Street	Town	Sex	JobTitle
Fred	13	Orchard Close	Greendale	Male	Software Engineer

versus

PROPERTY

Number	Street	Town
13	Orchard Close	Greendale

PERSON

Name	Address	Sex	JobTitle
Fred	OID	Male	Software Engineer

Figure 7.44 Part with whole versus whole with part clustering.

Now consider a simple query against the complex object involving both the top level class and nested classes such as "Get all Persons and their Addresses". In effect, the requirement is for complete objects. It naturally makes sense therefore to cluster these two classes in a single file. Clustering in separate files would mean

that two separate seeks would be required to obtain the "complete object". Clustering the part with the corresponding whole also makes more sense than clustering the whole with the corresponding part. As stated above, clustering the part with the whole is more akin to the underlying semantics and is more in line with the very way complex objects are represented by OIDs in the object model. The clustering of the whole with the corresponding part may reduce the number of seeks compared with the separate file approach but as shown in figure 7.44 it does not provide "complete objects".

Now consider qualified queries where both the top level class and nested classes are of interest in the final result. Contrary to qualified queries where only the top level class is the target class, there is only one sensible approach for processing this type of query, specifically the integrated approach. It would not make sense for a query processor to adopt the "separate activity" approach because objects and not just OIDs are required. Therefore, the clustering strategy based on separate files would not be appropriate in any of the cases for this type of query. For the same arguments as discussed above, both clustering strategies, P+W and W+P, are best for this type of query, but in particular, strategy W+P is the best overall.

Case $W_{1,1}P_{1,M}$: Inter-exclusive, Dependent, Multi-valued, Essential

Table 7.9 shows the best clustering strategy for each type of query for this implementation structure.

	simple W query	qualified W query W attribute	qualified W query P attribute	qualified W query W and P attribute	simple W and P query	qualified W-P query W attribute	qualified W-P query P attribute	qualified W-P query W and P attribute
W+P			**	**	**	**	**	**
P+W: f								
P+W: r								
P+W: c			*	*	*	*	*	*
W, P	*	*	*	*				

Table 7.9 The matching of queries to clustering strategies for case $W_{1,1}P_{1,M}$.

The situation for this structure is very similar to that of the previous implementation structure $W_{1,1}P_{1,1}$. All the arguments that applied in that case apply here. However, in this case, a set of OIDs must be managed and not just one. This factor affects qualification queries concerning the nested attribute. It means that the qualification becomes set-oriented with the requirement that at least one member of

the set or all members of the set take on a particular value. Further, the clustering of P with W means that a set of part objects are clustered in place within a whole object. As for the previous case, the clustering of W with P is a good but not the best strategy for qualification queries involving the nested attribute. However, for this case, there is a choice of how to cluster W with P. Recall, that if all P objects that relate to a single W object are not clustered, then W may be placed with the first P object in the file or any random P object. Consider the effect of applying a qualification query based on a nested attribute to this type of clustering. For each W object, to test if one or all of its P objects satisfies the condition requires the searching of the file a number of times. Compare this to the situation where P objects relating to the same W object are clustered together. In this case, such queries can be processed in a single run of the file. The clustering strategies P+W (first) and P+W (random) are therefore dismissed as valid strategies. Dismissing these strategies presents results similar to the previous case as shown in table 7.9.

Cases $[W_{1,1}P_{1,1}]$ and $[W_{1,1}P_{1,M}]$: Intra-class exclusive, Dependent, Single-valued or Multi-valued, Essential

	simple W query	qualified W query W attribute	qualified W query P attribute	qualified W query W and P attribute	simple W and P query	qualified W-P query W attribute	qualified W-P query P attribute	qualified W-P query W and P attribute
W+P								
W'+P								
P+W								
P+W'								
P+W+W'								
W, W', P	*	*	*	*	*	*	*	*

Table 7.10 The matching of queries to clustering strategies for cases $[W_{1,1}P_{1,1}]$ and $[W_{1,1}P_{1,M}]$.

An overall view can be taken in these cases. Here, the P objects may be referenced by another type of W object. Consider the other whole class to be W'. In previous cases, the clustering of P with W is of benefit for qualification queries based on the nested attribute. However, in these cases, W' cannot be ignored. Although, the clustering of P with W or vice-versa would benefit certain queries on W, it would be at the detriment of queries against W' involving the nested attribute. Clustering strategy W, W', P is therefore the only feasible strategy across the board for these cases as shown in table 7.10.

Case $W_{1,M}P_{1,1}$: Shared, Dependent, Single-valued, Essential

Table 7.11 shows the best clustering strategy for each type of query for this implementation structure. Note that clustering strategy W+P assumes that W objects are clustered according to the P they relate to. Clustering strategies W+P (random) and W+P (first) have been dismissed as valid strategies previously in the discussion for case $W_{1,1}P_{1,M}$.

	simple W query	qualified W query W attribute	qualified W query P attribute	qualified W query W and P attribute	simple W and P query	qualified W-P query W attribute	qualified W-P query P attribute	qualified W-P query W and P attribute
W+P			*	*	*	*	*	*
P+W			*	*	*	*	*	*
W, P	*	*	*	*				

Table 7.11 The matching of queries to clustering strategies for case $W_{1,M}P_{1,1}$.

As for previous cases, clustering strategy W, P is the most suitable for both a simple query against the whole class and for a qualified query where only the whole class is both the target and the basis of the qualification. On the other hand, for a simple query involving both the W and P classes, the separate file approach is not suitable as it would involve more seeks than necessary. Again, as for previous cases, clustering of P with W or vice-versa comes into play when either the qualification or the target class includes P. Where the target class does not include P, the query may justifiably be processed as two separate activities, specifically one activity to obtain a list of OIDs that satisfy any qualifications against the P class and one activity to match these OIDs with W objects. In this case, strategy W, P is valid. On the other hand, if an integrated approach to processing the query is taken, then the clustering of W with P or vice-versa is valid. Unlike previous cases, in this situation, the clustering of P with W cannot be used to create complete complex objects. Ignoring replication, a P object cannot be clustered in place of its OID in a W object because other W objects may reference that same object. Assuming the clustering of W objects, the clustering of P with W and the clustering of W with P now only differ in whether a P object leads or follows its corresponding set of W objects. Both strategies are equally valid. Where the target classes are W and P, the query is likely to be processed in an integrated fashion. Therefore, the clustering strategy W, P is not valid. Strategies where W and P are clustered in a single file would be beneficial to the integrated approach to processing such queries.

Where W is the only qualification it is perhaps better for the set of W objects to lead the corresponding P object and vice-versa.

Case $W_{1,M}P_{1,M}$: Shared, Dependent, Multi-valued, Essential

Table 7.12 shows the best clustering strategy for each type of query for this implementation structure.

	simple W query	qualified W query W attribute	qualified W query P attribute	qualified W query W and P attribute	simple W and P query	qualified W-P query W attribute	qualified W-P query P attribute	qualified W-P query W and P attribute
W+P								
P+W								
W, P	*	*	*	*	*	*	*	*

Table 7.12 The matching of queries to clustering strategies for case $W_{1,M}P_{1,M}$.

This is the most complicated situation because on the one hand, a whole object may have many part objects whilst on the other, a part object may be related to many whole objects. This means that there will be a set attribute within the whole object and a whole object may reference the same part as another whole object. Without replication, any clustering strategy will be good from one point of view but poor from the other. The problem is that only one relationship, the one-to-many or the many-to-one, can be the basis of the clustering. If all the P objects that relate to a single W object are clustered with that W object, then the fact that the W object may be part of a many-to-one relationship with a P object cannot be represented. In fact, without replication, P objects cannot be clustered with its corresponding W object for every W object precisely because the same P object may be related to more than one W object. Therefore, in this clustering scenario, some complex objects could be represented completely whereas others would still have to reference the part object that would now be embedded anywhere in the file. This would make the processing of these records very complex. For these reasons, it is sensible in this situation to adopt the separate file clustering strategy across the board. This conclusion is reflected in the table above.

In conclusion, with the exception of the more complicated implementation structures, the clustering requirements for aggregation hierarchies can be generalised. The more complicated structures can be considered to be the structures with more links and include the two intra-class exclusive structures, $[W_{1,1}P_{1,1}]$ and $[W_{1,1}P_{1,M}]$, and the many-to-many structure, $W_{1,M}P_{1,M}$. For the remaining

structures, which are inter-class exclusive and have no more than one 'many' relationship, the results are similar. For simple queries against W only and for qualified queries where both the target class and the qualification is W only, then the separate file clustering strategy is best. For all other queries, the best clustering strategy is W+P or W+P (clustered). This means that the creation of complete complex objects is sensible where part objects replace OIDs in place. The exception is the many-to-one relationship where complete complex objects cannot be created but P objects may follow clustered W objects to achieve a near approximation. The W+P clustering strategy was found to be valid for 6 out a set of 8 types queries. In general therefore, the clustering strategy W+P is best overall for aggregation hierarchies, unless simple or qualified queries involving only W have a higher priority. This can only be deduced from analysis of the actual queries for a particular application.

7.3 ASSOCIATION

7.3.1 Implementation Structures

Recall from chapter six that an association structure is very similar to an aggregation structure but with weaker semantics. Again, the relationships exist at the object level and so there exists the potential of clustering individually related objects into a single file. The previous discussion will be relevant to this section therefore but it must be borne in mind that the need for clustering to support the relationship will be less critical precisely because the underlying semantics are weaker. Indeed, for association structures, it is more necessary to be aware of the actual application to be able to ascertain the clustering requirements.

Figure 7.45 An association relationship.

Now consider the significant criteria for association structures established in chapter six. Unlike aggregation hierarchies, an association relationship does not imply a direction. Cardinality adornments signify the fact that there exists a relationship in both directions between objects of two classes, as shown in figure 7.45. The two perspectives of the relationship should be considered equally. The criteria relevant to an association relationship are valuedness and dependency.

Valuedness

Figure 7.46 shows the issue under consideration in this section. The concern is whether an individual object in one of the classes of the relationship is related a single object or more than one object of the class at the other end of the relationship. Notice from figure 7.46 that this criterion has been considered from the perspective of class O. The criterion can be applied in the same way from the perspective of class O'.

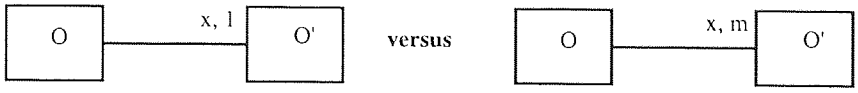


Figure 7.46 Valuedness in an association relationship.

As for aggregation hierarchies, this criterion impacts on clustering. A single-valued relationship is different to a multi-valued relationship with respect to clustering, as discussed in the previous section. Therefore, the conceptual structures cannot be reduced into a smaller set of implementation structures at this point.

Dependency

Figure 7.47 shows the differences in the conceptual structure with respect to the criterion of dependency. Essentially, dependency is concerned with whether an object in any of the classes in an association relationship can exist in its own right or depends upon the existence of a related object in the other class. Note again that the figure reflects the dependency criterion from the point of view of class O, but it could also be applied from the perspective of class O'.

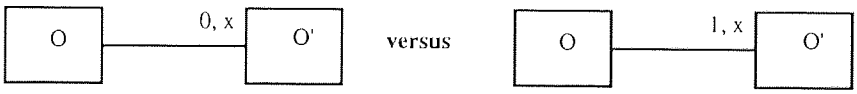


Figure 7.47 Dependency in an association relationship.

As discussed for aggregation hierarchies, dependency does not represent a critical issue in terms of implementation, specifically clustering. This allows us to reduce the set of 10 conceptual structures to a set of 3 implementation structures as shown in figure 7.48.

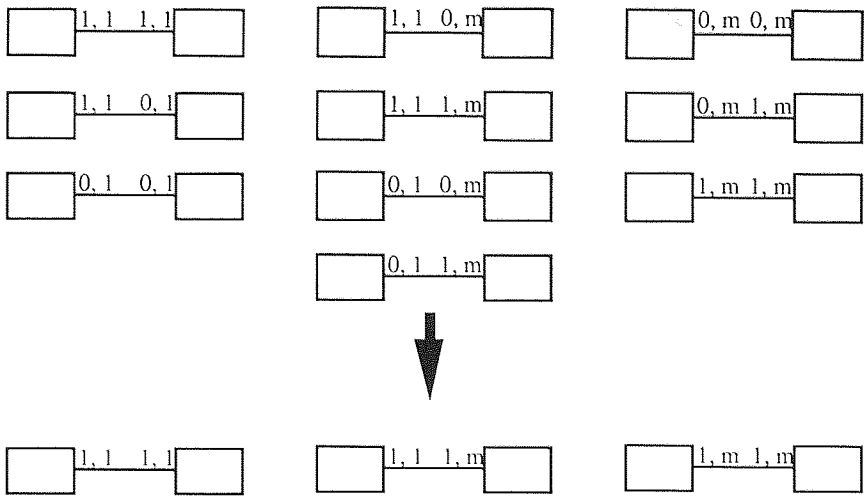


Figure 7.48 Association cases after applying "all objects dependent" decision.

7.3.2 Clustering of Association Implementation Structures

The above has reduced the association conceptual structures to a set of 3 implementation structures. Now consider the clustering of these cases. As for aggregation hierarchies, relationships exist at the object level and so clustering is concerned with the actual placement of individual objects. Apart from the underlying semantics, the four implementation structures for association are equivalent to implementation structures one, two and six in the aggregation section. Recall that it is not necessary to consider the relationship from both perspectives, as for aggregation hierarchies, because there is no implied direction for association relationships and so only one one-to-many relationship is included. The underlying semantics do not impact on the potential clustering options and so the discussion of the previous section applies here. Further, the clustering options for the association implementation cases can be directly produced from the previous section as shown in table 7.13.

	$O_{11}O'_{11}$	$O_{11}O'_{1M}$	$O_{1M}O'_{1M}$
1	O+O'	O+O'	O+O' (cluster W)
2	O'+O	O'+O (cluster)	O'+O (cluster P)
3	O, O'	O, O'	O, O'

Table 7.13 Association relationship clustering permutations.

Notice that the first and random clustering strategies have been dismissed as valid strategies after the discoveries made in the previous section upon consideration of queries. Given two independent classes O and O' therefore, with an association

relationship between them, the options available are to cluster O with O', to cluster O' with O or to cluster O and O' in separate files.

7.3.3 Queries

Consider the type of query that could be applied to an association structure. Association structures are not a new concept, unlike aggregation and inheritance hierarchies. Association structures represent a general relationship and can be found in previous models, notably the relational model. Therefore, consider the queries of the relational model. A query against a class or relation may be simple or qualified. Further, this may or may not involve the details of the class or relation at the other end of a relationship (known as a join in the relational model). For qualified queries, the related class may be the basis of the qualification, the target list or both. Figure 7.49 shows the types of queries applicable to association structures.

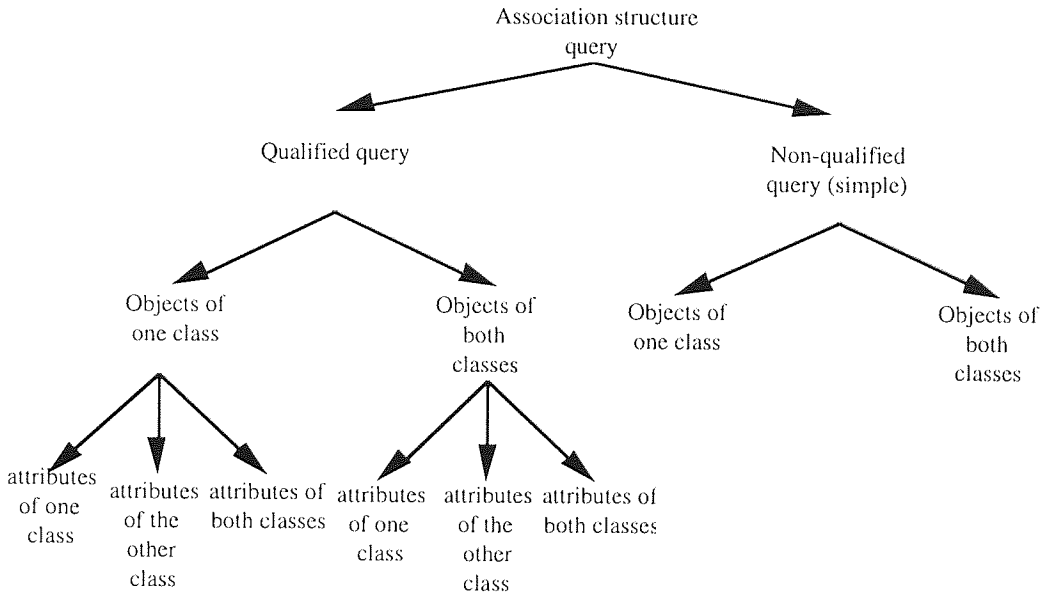


Figure 7.49 Association structure queries.

Unlike aggregation hierarchies, association structures do not imply a direction. Therefore, it is necessary to consider the application of queries to both classes. Therefore, for association structures, the following queries are of interest:

- simple query against a class O;
- simple query against a class O';
- simple query against two classes;
- qualified query where the target and the qualification are based on class O;
- qualified query where the target is class O but the qualification is based on related class O';

- qualified query where the target is class O but the qualification is based on that class and related class O';
- qualified query where the target is class O and related class O' and the qualification is based on class O;
- qualified query where the target is class O and a related class O' and the qualification is based on related class O';
- qualified query where the target is class O and related class O' and the qualification is based on both classes;
- qualified query where the target and the qualification are based on class O';
- qualified query where the target is class O' but the qualification is based on related class O;
- qualified query where the target is class O' but the qualification is based on that class and related class O;

7.3.4 Application of Queries to Clustering Strategies

Now consider the effect of performing the various queries deduced in the last section under different clustering scenarios. The following sections will take each implementation structure in turn.

Case $O_{1,1}O'_{1,1}$: One-to-One Association

Tables 7.14 and 7.15 show the best clustering strategy for each type of query for this implementation structure, where T represents the target and Q represents the qualification.

	T:O	T:O+O'	T:O'
O+O'		*	
O'+O		*	
O, O'	*		*

Table 7.14 The matching of simple queries to clustering strategies for case $O_{1,1}O'_{1,1}$.

	T:O Q:O	T:O Q:O'	T:O Q:O+O'	T:O+O' Q:O	T:O+O' Q:O'	T:O+O' Q:O+O'	T:O' Q:O'	T:O' Q:O	T:O' Q:O+O'
O+O'		*	*	*	*	*		*	*
O'+O		*	*	*	*	*		*	*
O, O'	*	*	*				*	*	*

Table 7.15 The matching of qualified queries to clustering strategies for case $O_{1,1}O'_{1,1}$.

The results are similar to those for aggregation hierarchies. Where the interest is focussed on a single class such as a simple query on only O or O' or a qualified query where both the target and the qualification are O or O', then the best

clustering strategy is the separate file approach, essentially a 'no clustering' strategy. On the other hand, for simple queries on both classes and qualified queries where the target is one class and the qualification includes the other, then all clustering strategies are valid. Again, as for aggregation hierarchies, the choice between separate file and single file clustering strategies depends on the type of processing likely to take place. For qualified queries, where the target includes both classes, the integrated approach to processing is more sensible than the separate activity approach because attribute values and not just OIDs are required. Hence, the separate file clustering strategy is dismissed as a good approach in these cases. Note with this structure, clustering strategies O+O' and O'+O are virtually the same. The only difference is the ordering of the objects, specifically whether an object instance of O precedes an object instance of O' or vice-versa. It may be debated that the instances of the class which is the basis of the qualification should come first. However, the advantages this factor brings are trivial and so for the purposes of this discussion, clustering strategies O+O' and O'+O are considered equally as good.

Case $O_{1,1}O'_{1,M}$: One-to-Many Association

Tables 7.16 and 7.17 show the best clustering strategy for each type of query for this implementation structure.

	T:O	T:O+O'	T:O'
O+O'		*	
O'+O: c		*	
O, O'	*		*

Table 7.16 The matching of simple queries to clustering strategies for case $O_{1,1}O'_{1,M}$.

	T:O Q:O	T:O Q:O'	T:O Q:O+O'	T:O+O' Q:O	T:O+O' Q:O'	T:O+O' Q:O+O'	T:O' Q:O'	T:O' Q:O	T:O' Q:O+O'
O+O'		*	*	**	*	*		**	*
O'+O: c		**	*	*	**	*		*	*
O, O'	*	*	*				*	*	*

Table 7.17 The matching of qualified queries to clustering strategies for case $O_{1,1}O'_{1,M}$.

The results for this structure are similar to those found for the previous structure. However, clustering strategies O+O' and O'+O (cluster O') may be different enough to make the ordering of object instances significant for qualified queries

where the qualification is one class. For instance, where the qualification is based on O' and there are a large number of O' instances to each O instance, then the strategy O'+O (cluster O') may be better than strategy O+O'. The query processor can then scan the list of O' instances to match against the qualification, and if successful, the current block or the next block into the buffers will contain the corresponding O' instance.

Case $O_{1,M}O'_{1,M}$: Many-to-Many Association

Tables 7.18 and 7.19 show the best clustering strategy for each type of query for this implementation structure.

	T:O	T:O+O'	T:O'
O+O' cluster O			
O'+O cluster O'			
O, O'	*	*	*

Table 7.18 The matching of simple queries to clustering strategies for case $O_{1,M}O'_{1,M}$.

	T:O Q:O	T:O Q:O'	T:O Q:O+O'	T:O+O' Q:O	T:O+O' Q:O'	T:O+O' Q:O+O'	T:O' Q:O'	T:O' Q:O	T:O' Q:O+O'
O+O' cluster O									
O'+O cluster O'									
O, O'	*	*	*	*	*	*	*	*	*

Table 7.19 The matching of qualified queries to clustering strategies for case $O_{1,M}O'_{1,M}$.

As for the many-to-many aggregation hierarchy case, the separate file clustering strategy is best across the board for this structure. For the reasons discussed in the aggregation section, any single file clustering strategy is not achievable with many-to-many relationships. Clustering each class in a separate file provides an uncomplicated approach. With association structures, it is feasible to represent many-to-many relationships as two separate one-to-many relationships. The previous discussion for one-to-many structures could therefore be applied separately to each relationship. This factor must be taken into account at the logical modelling stage.

In conclusion, the results for association structures are similar to those found for aggregation hierarchies. To re-iterate, for one-to-one or one-to-many association

structures, the following applies. For simple queries based on one class or qualified queries where both the target and the qualification is based on one class, the separate file clustering strategy is best. For simple queries based on two related classes or qualified queries where one class is in the target and the other class is in the qualification, all clustering strategies are equally valid. For qualified queries where both classes are in the target, a single file clustering strategy is best. However, it must be pointed out that an association relationship is not as strong as an aggregation relationship. A general assumption as to the popularity of such a relationship cannot be made. Therefore, the clustering requirements for association structures cannot realistically be ascertained without knowledge about the actual queries of an application. It is interesting to note however that for one-to-one and one-to-many structures the above tables show an equal overall preference to the various clustering strategies. In a particular query frequency analysis therefore, each strategy should have an equal weighting and simply the most frequently used query should dictate the type of clustering employed.

7.4 CONCLUSION

This chapter has analysed, in depth, the application of different clustering strategies to object model structures, specifically inheritance hierarchies, aggregation hierarchies and association structures. It has considered closely the effect of different clustering strategies for different types of queries. In doing so, it has revealed the true potential of clustering strategies for object model structures which has only been touched upon in the literature. It can be concluded from this analysis that clustering should be used for aggregation hierarchy and association structures but not for inheritance hierarchy structures. The results for aggregation hierarchy and association structures were found to very similar but it must be stressed that aggregation hierarchies, in particular, with stronger underlying semantics than association structures, provide the most fruitful opportunities for clustering.

CHAPTER EIGHT

CONCLUSION

This thesis set out to contribute to the existing knowledge on clustering strategies for object databases. More specifically, the intention has been to examine the connection between the logical constructs of the object model and clustering at the physical level, and in doing so to provide a set of rules of thumb for the designer or at least more guidance as to the when and how to use clustering. From a review of the literature, it was clear that there exists a lack of in-depth analyses on object clustering strategies. Further, there exists no overall framework on the subject. and there is no real guidance to designers. Although it is acknowledged in the literature that there are potential performance gains from object clustering strategies, there is not enough evidence or guidance to encourage designers to truly exploit clustering or, indeed, to dismiss clustering as a valuable physical design technique. This thesis has made significant contributions to these areas of research and these are explored in the following section.

8.1 REVIEW OF CONTRIBUTION

In order to understand the links between the various areas of concern effecting clustering, this thesis firstly looked at the more well established contributory factors. Chapter two covered the physical model of databases which has proved very stable over time. Chapter three looked at the relational model in order to gain insight into the development of a logical model. Moving onto the core elements of this thesis, chapter four covered the object logical model and chapter five covered the object physical model, in particular, object clustering strategies. Chapter five has made a significant contribution to the field by providing a consolidation of ideas in the literature and, for the first time, a framework for object clustering strategies has been established, as summarised in figure 5.5. The process, in general, has helped to increase the understanding on object clustering, but more importantly, it has filled in gaps in the literature, revealing new, previously unthought of strategies. Further, a framework encourages the path to more consensus on, and terminology surrounding, object clustering strategies.

In order to analyse the link between the logical and physical aspects of the object model, chapter six was devoted to an analysis of logical structures and chapter seven took the results of chapter six on board and considered the representation of logical structures at the physical level in terms of clustering. Chapter six gave an extensive coverage of object logical constructs found in the literature and matched

these to real life examples. Indeed, chapter six proved to be an education in itself. The mapping of the constructs to real life examples increases understanding of object model ideas, but also, it provides verification of those ideas, literally testing out the theories. The tendency in the current literature is to focus on a single area of concern to highlight the benefits of applying object concepts to that particular type of application. There exists no overall coverage of the mapping of object model ideas to a wide set of examples as provided by chapter six. Indeed, the literature is very weak on information on how object modelling constructs are mapped to logical model constructs and finally to physical model constructs. Taken together, chapters six and seven considerably increase understanding in this field.

Chapter seven took each logical concept in turn and considered its implementation in terms of clustering. In performing the process of mapping from logical to physical concerns, a better understanding of clustering strategies has been gained. Further, it has provided insight into the reality of applying clustering strategies. Indeed, some of the theoretical clustering strategies developed in chapter five proved to be futile. Specifically, clustering of one-to-many relationships in that direction are only truly valid if the objects of the 'many' end of the relationship are clustered. Clustering objects of the 'one' end with the first or a random (for example) object of the 'many' end is not a sensible approach and these approaches were eventually dismissed as invalid in chapter seven. The literature discusses theoretical clustering strategies separately to practical, working strategies and so it has been difficult to relate or even distinguish between the two aspects. The outcome of this research, which links logical to practical strategies, therefore makes a significant contribution to the information available to the physical database designer.

Chapter seven then proceeded with a cross-examination of clustering strategies and queries for each implementation structure. This has helped again to dismiss some clustering strategies as being invalid and so provide the physical database designer with more information. It has also revealed that the usefulness of clustering is different for the different object conceptual structures, namely inheritance, aggregation and association structures. This result supports the thesis that semantic knowledge at the conceptual level can be used at the physical level in the application of clustering. The resulting conclusions act as a set of rules of thumb to the physical database designer and a summary is provided below.

8.2 SUMMARY OF MAIN CONCLUSIONS IN THIS THESIS

The following conclusions were drawn from the research. Clustering has a potential use for aggregation hierarchies and association structures but not for inheritance hierarchies. It is interesting to conclude from this research that clustering has been totally dismissed for inheritance structures. However, this result contradicts the suggestions on object clustering found in the literature. In particular, Kim (1990a) believes that both inheritance (class) and aggregation (class-composition) hierarchies can be clustered. Despite the discrepancy between the outcome of this research and the existing literature, it is reasoned that the result still supports the thesis that semantic knowledge can be used to indicate clustering. From the analysis of relationships in chapter six, it was pointed out that inheritance is not concerned with relationships at the object level unlike aggregation or association. In hindsight therefore, the result that clustering is useful for aggregation and association but not for inheritance structures could have been predicted.

Clustering is useful for aggregation hierarchies and association structures and the results were found to be similar except for the direction of application. The following was concluded. Clustering should be utilised for simple aggregation or association structures. These are considered to be those with fewer links between classes, specifically structures with no more than one 'many' relationship and no intra-class exclusivity. For more complicated structures, specifically many-to-many relationships and those with intra-class exclusivity, the separate file approach, essentially a "no clustering" approach is best. For simple structures, the following results were revealed. For simple queries involving only one class or qualified queries where both the target and the qualification is one class, the separate file approach, again essentially "no clustering", is best. However, for all other queries, the best overall strategy is to cluster the classes in a single file, where objects of one class are clustered with the corresponding objects (matching OIDs) of the other class. In addition, if the target is one class and the qualification includes the other class, then the separate file approach is also a valid strategy. The direction of the clustering for association structures is not as relevant as it is for aggregation structures with their stronger underlying semantics. For aggregation structures, clustering should result in the creation of complete complex objects where part objects replace their references in the whole object in place. For many-to-one, whole-to-part structures, the clustering of whole objects which correspond to a single part object provides a near approximation.

The overall results are presented in table 8.1 below and provide the physical database designer with a set of rules of thumb. Note, however, that these results are on a per query basis. It is interesting to note that for simple aggregation and association structures, there is a fairly equal weighting to the clustering and no clustering approaches, but this does not provide a specific conclusion. A final conclusion can only be attained from the running of a particular database to assess the likelihood of individual queries for the particular application. It is anticipated that queries utilising the relationship will be paramount for aggregation hierarchies but association structures will be less predictable. In the absence of information about the types of queries expected, then the heuristics developed above will have to suffice.

Inheritance structures	Aggregation and Association Structures		
	complex structures	simple structures	
		queries involving one class	queries involving more than one class
no clustering	no clustering	no clustering	use clustering

Table 8.1 Summary of results on clustering.

8.3 FUTURE WORK

This research has focussed on utilising the structural elements of the logical model to aid clustering and in doing so has concentrated on static based optimisation rather than dynamic based optimisation. However, it is appreciated that an approach based purely on the logical model has limitations. Further, as this thesis has shown that different structures and different queries dictate clustering requirements, adaptive, learning databases may prove to be a fruitful area of research. Future work to extend this thesis therefore, could be based on the development of a system which measures the frequency of different queries and adapts the clustering strategies accordingly.

Dynamic based optimisation techniques would be good for situations where the query flow is relatively stable and unchanging over time to allow for the clustering strategy to have an affect. This is the case for typical transaction processing systems such as an airline booking system. For systems where the query flow is very unpredictable, then the updated clustering strategy will always be a step behind and this is precisely the typical domain which best utilises object databases.

Transaction processing systems are are adequately serviced by existing databases, notably the relational database. It is therefore questionable whether future research should concentrate on dynamic based optimisation techniques other than for stable transaction processing systems.

It is arguable that this research has been somewhat limited by assuming a two-layer model incorporating a single parent-child relationship. However, there is no logical reasoning why the analysis cannot be applied to a further depth with the established arguments. The arguments presented for a single parent-child relationship should hold for all relationships along the hierarchy such as a grandparent-parent relationship.

This research has presented its arguments in a mathematical logic fashion in preference to a more experimental approach such as simulation and real world examples. To continue the ideas developed in this thesis, the problem could be widened and other methods of research could be utilised. Specifically, an example database could be used to apply different clustering strategies to a few real world problem domains. The system should assess its performance to verify that the chosen clustering strategies, as concluded in this thesis, are actually valid.

8.4 CLOSING COMMENTS

This research has successfully shown that different logical structures produce different clustering requirements. Most notably, this thesis has shown that clustering is a useful tool for aggregation or association structures but not for inheritance structures. This makes a significant contribution to the field of physical database design which previously assumed that clustering could be usefully exploited for all types of logical structure.

REFERENCES

Andrews T. and Harris C. (1987), "Combining Language and Database Advances in an Object-Oriented Development Environment", *in* Zdonik and Maier (1990) pp. 186-196.

Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S. (1989), "The object-oriented database system manifesto (a political pamphlet)", *Proc. DOOD 89*, Kyoto, Japan, December 1989.

Astrahan M. M. et al. (1976), "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, **1**(2), Jun. 1976, pp. 97-137.

Avison D. E. (1997), *Information Systems Development: A Data Base Approach*, 3rd edition, McGraw-Hill, UK.

Bancilhon F. et al. (1988), "The Design and Implementation of O_2, an Object-Oriented Database System", *in* Dittrich, K. R. (ed.), *Advances in Object-Oriented Database Systems*, Lecture Notes in Computer Science, 334, Springer-Verlag, Berlin, pp. 1-22.

Banerjee J. et al. (1987), "Data Model Issues for Object-Oriented Applications" *in* Zdonik and Maier (1990), pp. 197-208.

Banerjee J. et al. (1988), "Clustering a DAG for CAD Databases", *IEEE Transactions on Software Engineering*, **14**(11), Nov. 1988, pp. 1684-1699.

Bayer R. and McCreight C. (1972), "Organisation and Maintenance of Large Ordered Indexes", *Acta Informatica*, **1**(3).

Bertino E. and Martino L. (1991) "Object-Oriented Database Management Systems: Concepts and Issues", *COMPUTER*, Apr. 1991.

Bertino E. and Martino L. (1993), *Object-Oriented Database Systems Concepts and Architectures*, Addison-Wesley, England.

Bertino E. et al. (1994), "Clustering techniques in object bases: A survey", *Data & Knowledge Engineering*, **12**, pp. 255-275.

Blair G. et al. (eds) (1991), *Object-Oriented Languages, Systems and Applications*, Pitman Publishing, London.

Booch G. (1994), *OBJECT-ORIENTED ANALYSIS AND DESIGN with Applications*, Second Edition, Benjamin Cummings, Redwood City, CA.

Bowers D. S. (1992), *FROM DATA TO DATABASE,* 2nd edition, Computer Press, UK.

Brown A. (1991), *Object-Oriented Databases : Applications to Software Engineering,* McGraw-Hill Book Company, London.

Cardelli L. and Wegner P. (1985), "On understanding types, data abstraction and polymorphism", *Computing Surveys*, **17**(4).

Carey M. J. et al. (1988), "The EXODUS Extensible DBMS Project: An Overview", *in* Zdonik and Maier (1990), pp. 474-499.

Carey M. J. et al. (1989), "Object and File Management in the EXODUS Extensible Database System" *in* Kim and Lochovsky (1989), pp. 341-369.

Cattell R. G. G. (1994), *Object Data Management : Object-Oriented and Extended Relational Database Systems,* revised edition, Addison-Wesley Publishing Company, Reading, MA.

Cattell R. G. G. (ed) (1996), *The Object Database Standard: ODMG - 93*, Morgan-Kaufman Publishers, San Francisco, CA.

Chan A. et al. (1982), "Storage and Access Structures to Support a Semantic Data Model" *in* Zdonik and Maier (1990), pp. 308-316.

Chen P. (1976), "The Entity-Relationship Model - Towards a Unified View of Data", *ACM Transactions on Database Systems*, **1**(1), pp. 9-36.

Chen W. C. and Vitter J. S. (1984), "Analysis of New Variants of Coalesced Hashing", *ACM Transactions on Database Systems*, **9**(4), Dec. 1984, pp. 616-645.

Chou H-T., DeWitt D J., Katz R. H. and Klug A. C. (1985), "Design and Implementation of the Wisconsin Storage System", *Software-Practice and Experience*, **15**(10), Oct. 1985, pp 943-962.

Clifton H. D. (1978), *Business Data Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Coad P. and Yourdon E. (1991), *Object-Oriented Analysis*, Second Edition, Prentice Hall, Englewood Cliffs, NJ.

Codd E. F. (1970), "A Relational Model of Data for Large Shared Data Banks", *Communications of the ACM*, **13**(6), pp. 377-387.

Codd E. F. (1972), "Relational Completeness of Data Base Sublanguages", *in Data Base Systems: Courant Computer Science Symposia Series 6*, Prentice-Hall, Englewood Cliffs, NJ.

Codd E. F. (1979), "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, **4**(4), Dec. 1979, pp. 397-434.

Date C. J. (1994), *An Introduction to Database Systems*, Volume I (6th edition), Addison-Wesley, Reading, MA.

Deppisch U. et al. (1991), "Managing Complex Objects in the Darmstadt Database Kernel System", *in* Dittrich (1991), pp. 358-375.

Dingley S. (1996), *A Composite Framework for the Strategic Alignment of Information Systems Development*, PhD thesis, Aston University, UK.

Dittrich K. R. (1990), "Object-Oriented Database Systems: The Next Miles of the Marathon", *Information Systems* , **15**(1).

Dittrich K. R. et al. (eds) (1991), *Object-Oriented Database Systems*, Springer-Verlag.

Effelsberg W. and Haerder T. (1984), "Principles of Database Buffer Management", *ACM Transactions on Database Systems*, **9**(4), Dec. 1984, pp. 560-595.

Elmasri R. and Navathe S. B. (1989), *Fundamentals of Database Systems*, The Benjamin/Cummings, Redwood City, CA.

Eriksson H-E and Penker M. (1998), *UML Toolkit*, John Wiley & Sons, NY.

Fishman D. H. et al. (1987), "Iris: An Object-Oriented Database Management System" *in* Zdonik and Maier (1990), pp. 216-226.

Garvey M. A. and Jackson M. S. (1989), "Introduction to object-oriented databases", *Information and software technology*, **31**(10), Dec. 1989.

Haberhauer F. (1990), "Physical Database Design Aspects of Relational DBMS Implementations", *Information Systems*, **15**(3), pp. 375-389.

Halper et al. (1992), "Part relations for Object-Oriented Databases", *in* Pernul G. and Tjoa A. M. (eds), *11th International Conference on the Entity-Relationship Approach*, 1992, Springer Verlag, Berlin, pp. 406-421.

Hornick M. F. and Zdonik S. B. (1987), "A Shared, Segmented Memory System for an Object-Oriented Database", *in* Zdonik and Maier (1990), pp. 273-285.

Jackson M. S. (1990), "Beyond relational databases", *Information and Software Technology*, **32**(4), May 1990.

Jackson M. S. (1991), "A Tutorial on Object-Oriented Databases", *Information and Software Technology*, **33**(1), Jan/Feb 1991.

Kemper A. and Moerkotte G. (1994), "Physical Object Management", *in* Kim W. (1994), *Modern database systems*, Addison-Wesley, NY.

Khoshafian S. (1990), "Insight into object-oriented databases", *Information and Software Technology*, **32**(4), May 1990.

Khoshafian S. and Abnous R. (1990), *OBJECT ORIENTATION Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, NY.

Khoshafian S. N. and Copeland G. P. (1986), "Object Identity", *in* Zdonik and Maier (1990), pp. 37-46.

Khoshafian S., Franklin M. J. and Carey M. J. (1990), "Storage Management for Persistent Complex Objects", *Information Systems* , **15**(3), pp. 303-320.

Kim W. (1990a), "Architectural Issues in Object-Oriented Databases", *JOOP*, Mar/Apr 1990, pp. 29-38.

Kim W. (1990b), "Object-Oriented Databases: Definition and Research Directions", *IEEE Transactions on Knowledge and Data Engineering*, **2**(3), Sep. 1990, pp. 327-341.

Kim W. (1991), "Introduction to Object-Oriented Databases", The MIT Press, Cambridge, MA.

Kim W., Chou H-T and Banerjee J. (1988), "Operations and Implementation of Complex Objects", *IEEE Transactions on Software Engineering*, **14**(7), Jul. 1988, pp. 985-996.

Kim W. and Lochovsky F. H. (eds) (1989), *Object-Oriented Concepts, Databases, and Applications,* Addison-Wesley, NY.

Kim W. et al. (1989), "Composite Objects Revisited" *in Proc. ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, Jun. 1989, pp. 337-347.

Knuth D. E. (1997), *The Art of Computer Programming, Vol. III : Sorting and Searching*, 3rd edition, Addison-Wesley, Reading, MA.

Kroha P. (1993), *OBJECTS AND DATABASES*, McGraw-Hill, London.

Litwin W. (1980), "Linear hashing: A New Tool for File and Table Addressing" *in* Stonebraker M. (ed), *Readings in Database Systems*, 2nd edition, Morgan Kaufmann Publishers, San Mateo, CA, 1994, pp. 570-581.

Loomis M. E. S. (1983), *Data Management and File Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Loomis M. E. S. (1990a), "The Basics", *JOOP*, May/Jun 1990, pp. 77-81.

Maier D. and Stein J. (1987), "Development and Implementation of an Object-Oriented DBMS", *in* Zdonik and Maier (1990), pp. 167-185.

March S. T. (1983), "Techniques for Structuring Database Records", *Computing Surveys*, **15**(1), Mar. 1983, pp. 45-79.

Mariani J. A. (1991) "Object-Oriented Database Systems" *in* Blair et al. (eds) (1991), *Object-Oriented Languages, Systems and Applications*, Pitman Publishing, London, pp. 166-202.

Mariani J. A. (1992), "Oggetto: An Object-Oriented Database Layered on a Triple Store", *THE COMPUTER JOURNAL*, **35**(2), pp. 108-118.

Nievergelt J., Hinterberger H. and Sevcik K. C. (1984), "The Grid File: An Adaptable, Symmetric Multikey File Structure" *in* Stonebraker M. (ed), *Readings in Database Systems*, Morgan Kaufmann Publishers, San Mateo, CA, 1988, pp. 582-598.

Oxborrow E. (1989), *Databases and Database Systems: Concepts and Issues*, Second Edition, Chartwell-Bratt.

Phillips E. M. and Pugh D. S. (1987), *How to get a Ph.D.*, Open University Press, Milton Keynes, Great Britain.

Pratt P. J. and Adamski J. J. (1994), *DATABASE SYSTEMS Management and Design*, 3rd edition, boyd & fraser, Boston, MA.

Rob P. and Coronel C. (1993), *Database Systems: Design, Implementation and Management*, Wadsworth, Belmont, CA.

Robinson K. and Berrisford K. (1994), *Object-Oriented SSADM*, Prentice Hall, Englewood Cliffs, NJ.

Rumbaugh J. (1991), *Object-Oriented Modelling*, Prentice Hall, Englewood Cliffs, NJ.

Sacco G. M. and Schkolnick M. (1982), "Mechanism for Managing the Buffer Pin a Relational Database System using the Hot Set Model" *in Proceedings of the 1982 Very large Database Conference*, Sep. 1982, Mexico City, Mexico, pp. 257-262.

Saxby B. J. L. (1988), "Conventional and Object-Oriented Databases", School of Information Systems, Kingston Polytechnic, Oct. 1988.

Skarra A. H. et al. (1991), "ObServer: An Object Server for an Object-Oriented Database System", *in* Dittrich (1991), pp. 275-290 .

Snyder A. (1986), "Encapsulation and inheritance in object-oriented programming languages", *Proceedings of OOPSLA '86*, Portland, Oregon.

Stein J. and Maier D. (1991), "Associative Access Support in GemStone", *in* Dittrich (1991), pp. 323-339.

Stonebraker M. (1980), "Retrospection on a Database System", *ACM Transactions on Database Systems*, **5**(2), Jun. 1980, pp. 225-240.

Stonebraker M. (1981), "Operating System Support for Database Management", *Communications of the ACM*, **24**(7), Jul. 1981, pp. 412-418.

Stonebraker M. (1987), "The Design of the POSTGRES Storage System" *in Proceedings of the 13th VLDB Conference*, Brighton, pp. 289-300.

Stonebraker M., Wong E., Kreps P. and Held G. (1976), "The Design and Implementation of INGRES", *ACM Transactions on Database Systems*, **1**(3), Sep. 1976, pp. 189-222.

Teorey T. J. and Fry J. P. (1982), *Design of Database Structures*, Prentice-Hall, Englewood Cliffs, NJ.

Traiger I. L. (1982), "Virtual Memory Management for Database Systems", *Operating Systems Review*, **16**(4), Oct. 1982, pp. 26-48.

Ullman J. D. (1988), *Principles of Database and Knowledge-base Systems - Volume I*, Computer Science Press, Rockville, Maryland.

Ullman J. D. and Widom J. (1997), *A First Course in Database Systems*, Prentice-Hall, NJ.

Weiser S. P. and Lochovsky F. H. (1989), "OZ+: An Object-Oriented Database System", *in* Kim and Lochovsky (1989), pp. 309-337.

Wiederhold G. (1987), *File Organisation for Database Design*, McGraw-Hill, New York, NY.

Wirfs-Brock R. et al. (1990), "Designing Object-Oriented Software", Prentice Hall, Englewood Cliffs, NJ.

Yaseen R. et al. (1991), "An Extensible Kernel Object Management System" *in Proceedings of OOPSLA 1991*, pp. 247-263.

Zdonik S. B. and Maier D. (eds) (1990), *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, San Mateo, CA.

BIBLIOGRAPHY

Bancilhon F. and Kim W. (1990), "Object-Oriented Database Systems : In Transition", *SIGMOD RECORD* , **19**(4), Dec. 1990.

Bertino E. and Kim W. (1989), "Indexing Techniques for Queries on Nested Objects", *IEEE Transactions on Knowledge and Data Engineering*, **1**(2), Jun. 1989.

Hammer M. and McLeod D. (1981), "Database description with SDM: a semantic database model", *ACM Trans. Database Syst.*, **6**.

Hull R. and King R. (1987), "Semantic database modeling: Survey, application, and research issues", *ACM Computing Surveys*, **19**(3).

Loomis M. E. S. (1990b), "OODBMS vs. Relational", *JOOP*, Jul/Aug 1990.

Loomis M. E. S. (1990c), "Database Transactions", *JOOP*, Sep/Oct 1990.

McLeod D. (1991), "Perspective on object databases", *Information and Software Technology*, **33**(1), Jan/Feb 1991.

Shekita E. J. and Carey M. J. (1989), "Performance Enhancement Through Replication in an Object-Oriented DBMS" *in Proc. ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, Jun. 1989.

Shipman D. (1981), "The functional data model and the data language DAPLEX", *ACM Trans. Database Syst.*, **6**(1), Mar 1981.