

DIVISION OF COMPUTER SCIENCE

**A Technique for Clarifying the Implementation of
Relationships between Objects to Enhance Software Reuse**

(PhD Thesis - Research Programme carried out in collaboration with BNR)

Jeanne Audrey Mayes

Technical Report No.226

April 1995

A TECHNIQUE FOR CLARIFYING THE
IMPLEMENTATION OF RELATIONSHIPS
BETWEEN OBJECTS TO ENHANCE
SOFTWARE REUSE.

JEANNE AUDREY MAYES

A thesis submitted in partial fulfilment of the
requirements of the University of Hertfordshire
for the degree of Doctor of Philosophy

April 1995

This research programme was carried out
in collaboration with BNR



Abstract

The thesis describes a novel design technique, called the Sociable class design method, which can be used to improve the representation of one particular type of relationship between objects in object oriented systems. A study, using several object oriented programming languages, has been carried out to demonstrate the feasibility of implementing the design method. The proposed design method has been used in the development of a full case study which was implemented in Eiffel v2.3. The constructs used by the design method are intended to overcome some of the technical problems associated with the reuse of software components.

Traceability is identified, from a study of various engineering environments, as important for improving reusability of components in engineering. This thesis demonstrates that existing object oriented development methods do not provide traceability of all types of identifiable relationships between objects. The novel design method described improves the traceability of one of these types of relationship.

The proposed design technique is evaluated against other techniques which can be used to implement the same relationship. The evaluation indicates that the increased traceability provided by the new method simplifies the design of both the components and the overall system, thereby improving the reusability of the components and the extensibility of the system.



Acknowledgements

During the course of this research, I have received encouragement, support and help from many people. I would like to thank all those members of the department who have offered words of encouragement. I wish to thank some of them specifically.

Primarily, I must thank my principal supervisor, Carol Britton, and my two second supervisors, Bob Dickerson and Ruth Barrett. They have helped me to clarify my ideas and encouraged me to pursue the better ones. At a more practical level, I would like to thank Bob for his help with the coding of my systems, particularly for writing the C++ version, and for explaining the workings of latex and emacs.

The second group of people who deserve special thanks are my fellow research students, Mary Buchanan, Roger Collins, Caroline Lyon, Marie Rose Low and Paul Taylor. Mary deserves special thanks, firstly for the many stimulating discussions and secondly for the hours she spent reading and commenting on my thesis. I hope our discussions did not disturb Caroline and Paul too much when we were sharing a room.

I would also like to thank two members of my collaborating institution, BNR. These two people have formed my point of contact with the company. Ben Potter, who now works at the University of Hertfordshire, and Alastair Tocher have both been helpful in offering words of advice and encouragement.

Finally I would like to thank my family for their support. I am grateful to my husband, Phil, for taking an interest in my work and encouraging me when progress was slow. Thanks also to my children, Neil, Clare and Paul, for not moaning too much when my work disrupted their lives. Paul even acted as an audience during my preparations for seminar presentations although he hardly understood a word.



Contents

Abstract	ii
Acknowledgements	iii
1 Introduction	1
1.1 Aim of the project	1
1.2 The importance of reuse	1
1.3 Overview of the thesis	3
2 Reuse and object oriented techniques	4
2.1 Definition of reuse	4
2.2 Factors affecting reusability	5
2.2.1 Encapsulation and information hiding	5
2.2.2 Understandability	6
2.2.3 Compatibility	7
2.2.4 Standardisation	7
2.2.5 Reliability	8
2.2.6 Design complexity	9
2.2.7 Extensibility	10
2.2.8 Product efficiency	11
2.2.9 Traceability	11
2.3 Summary and Conclusion	12
3 Traceability in object oriented software development	14
3.1 The development process	15
3.2 Analysis methods	17
3.2.1 Approaches used	17
3.2.2 Classes and Objects	18
3.2.3 Relationships between classes and objects identified and modelled	20
3.2.4 Subsystems	25
3.3 Information modelled during analysis	26
3.4 Object oriented languages	30
3.4.1 Key features of the languages	30
3.4.2 Encapsulation based upon classes and objects	32
3.4.3 Inheritance	38
3.4.4 Polymorphism and dynamic binding	42
3.4.5 Data types provided	45



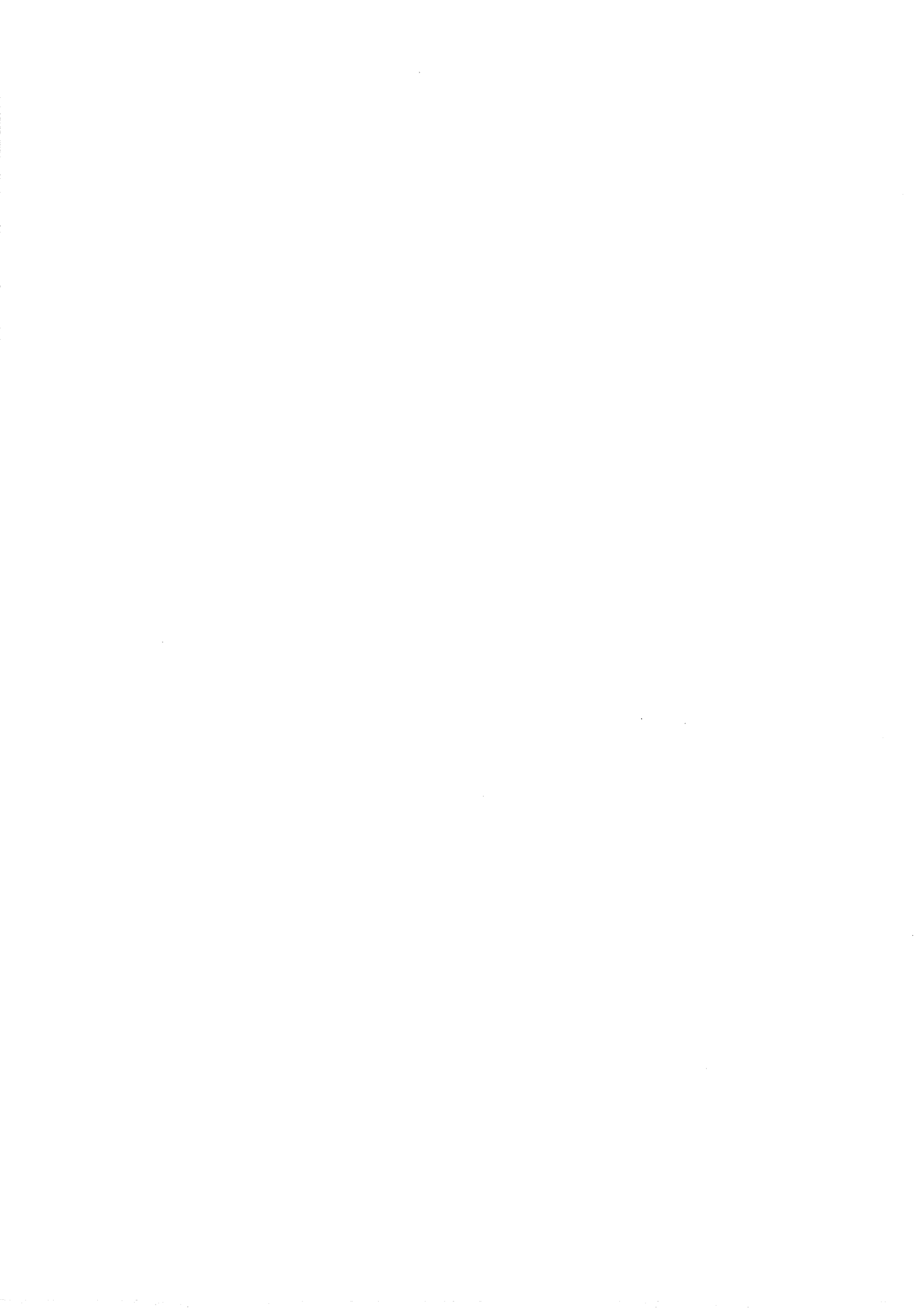
3.5	Information represented in object oriented programming languages	47
3.6	Design Issues—from analysis to implementation	49
3.6.1	Classes and Objects	49
3.6.2	Relationships	50
3.6.3	Subsystems	57
3.7	Design gains and losses	58
3.8	Discussion	63
3.9	Summary	64
4	Sociable classes—a novel design technique for improving traceability	66
4.1	Problems to be addressed	67
4.2	Design requirements for a solution	67
4.3	The definition of Sociable classes and related constructs	69
4.4	Class specification	71
4.4.1	Class Social	71
4.4.2	A Sociable class	72
4.4.3	Class Assoc	72
4.4.4	Class One_to_one2	73
4.4.5	Other classes of association	74
4.5	Using Sociable classes	76
4.5.1	Developing a new system	77
4.5.2	Extending a system	78
4.6	Features of the Sociable class method	79
4.7	Desirable language features	80
4.8	Feasibility study	81
4.8.1	Eiffel	81
4.8.2	Modula-3	83
4.8.3	Oberon-2	84
4.8.4	C++	86
4.9	Discussion	87
4.9.1	The technique	87
4.9.2	Current implementations	87
4.9.3	Alternative implementations	89
4.10	Summary	91
5	A comparison of the Sociable class technique with other techniques	94
5.1	Alternative design techniques	95
5.1.1	Object oriented databases	95
5.1.2	Combining object oriented and logic paradigms	96
5.1.3	Combination of inheritance hierarchies	97
5.1.4	DSM	98
5.1.5	Role-types	100
5.1.6	Design patterns	101
5.2	Traceability	101
5.2.1	Results	101
5.2.2	Conclusion	102
5.3	Other factors	102



5.3.1	Information hiding	103
5.3.2	Understandability	104
5.3.3	Standardisation	105
5.3.4	Reliability	105
5.3.5	Design complexity	106
5.3.6	Extensibility	108
5.3.7	Product efficiency	110
5.3.8	Language support	111
5.4	Discussion	111
5.5	Conclusions	114
6	Case Study	116
6.1	Requirements specification	116
6.2	Development	117
6.2.1	Analysis	117
6.2.2	Design	117
6.2.3	Implementation	120
6.2.4	Testing	123
6.3	Discussion	125
6.3.1	Use of the Sociable class technique	125
6.3.2	Traceability of Information	126
6.3.3	Reuse of classes	127
6.4	Conclusions about the use of associations	128
7	Conclusion	130
7.1	Reuse and traceability	130
7.2	The Sociable class design method	131
7.3	Future work	132
7.3.1	The Sociable class technique	132
7.3.2	Improving traceability of other relationships	133
7.4	Review of achievements	134
A	Eiffel classes used to implement the Sociable class method	135
A.1	Class Assoc	135
A.2	Class One-to-one2	135
A.3	Class Social	138
A.4	Class Person	140
B	Modula-3 classes used to implement the Sociable class method	141
B.1	Class Assoc	141
B.2	Class One-to-one2	141
B.3	Class Social	143
B.4	Class Person	145
C	Oberon-2 classes used to implement the Sociable class method	147
C.1	Class Assoc	147
C.2	Class One-to-one2	147
C.3	Class Social	148



C.4	Class Person	149
D	C++ classes used to implement the Sociable class method	150
D.1	Class Assoc	150
D.2	Class One-to-one2	151
D.3	Class Social	152
D.4	Class Person	152
E	Case study implementation	154
E.1	Root class	154
E.2	Class Display	156
E.3	Class Query	157
E.4	Class Plant_garden	160
E.5	Sociable classes	162
E.5.1	Class Crop	162
E.5.2	Class Garden	162
E.5.3	Class Section	163
E.5.4	Class Insect	164
E.5.5	Class Crop_rotation_rules	164
E.6	Other classes required	165
E.6.1	Class Gardenplot	165
E.6.2	Class Date	166
F	Association library classes	167
F.1	Class One-to-many	167
F.1.1	Class specification	167
F.1.2	Design issues	169
F.2	Class Many-to-many	169
F.2.1	Class specification	169
F.2.2	Design issues	170
F.3	Code	171
F.3.1	Class One-to-many	171
F.3.2	Class Many-to-many	175
G	An investigation into the type checking of generic types in ISE Eiffel v2.3	179
G.1	Eiffel terminology	179
G.2	Features used in the investigation	180
G.3	Tests carried out	180
G.4	Discussion and Conclusion	182
G.5	Code of root class for generic type conformance tests	182
G.6	Results of generic type conformance tests	187
H	Published paper	189



List of Tables

3.1	Inheritance terminology	31
3.2	Conversion of relationships	58
5.1	Design complexity	108
5.2	Design complexity of extended system	109
5.3	Comparative design complexity of original and extended system	110
5.4	Results summary	112



List of Figures

3.1	Life cycle models	16
3.2	Class and object notation	21
3.3	Inheritance notation	23
3.4	Is-part-of notation	24
3.5	Static association notation	25
3.6	Notation for processing dependencies	26
3.7	Pointers and expanded objects	36
3.8	Assignment of Records	37
3.9	Assignment of Pointers	37
3.10	Multiple inheritance (a)	51
3.11	Multiple inheritance (b)	52
3.12	Sample object models	53
3.13	Simple banking application object model	54
3.14	Implementing one way associations	54
3.15	Implementing two way associations	55
3.16	Object model of extended system	60
3.17	Extending a system	62
4.1	General object model	68
4.2	Sociable class hierarchy	69
4.3	Association class hierarchy	70
4.4	General associations	71
4.5	Ten forms of relationship from Shlaer and Mellor	75
4.6	A formalized association from Shlaer and Mellor	75
4.7	Sample object models	76
4.8	Simple banking application object model	77
4.9	Object model of extended system	78
5.1	Model showing role types involved in the simple banking system	100
6.1	Object model for kitchen garden system	118
6.2	Top level data flow diagram for kitchen garden system	118
6.3	Final model for kitchen garden system	119
6.4	Sociable class hierarchy used in the kitchen garden system	120
6.5	Association hierarchy used in the kitchen garden system	120
6.6	Model drawn from root class declarations	127
F.1	Many to many relationships	170



Chapter 1

Introduction

This chapter gives the background information to the research project. The aim of the project is described in section 1.1. The importance of research in the general area of reuse of software is explained in section 1.2. Section 1.3 outlines the structure of this report to give an overview of the research project.

1.1 Aim of the project

The broad aim of the project was to carry out research into the possibility of enhancing software reuse in object oriented development. This aim was achieved by research into the technical aspects of the systems development process. Non technical aspects such as managerial issues were not investigated. The technical area identified for detailed research was the development and use of software components. A more specific aim was identified. This specific aim was to improve the reusability of software components which can be developed in currently available object oriented programming languages. This precludes the development of a new language or the addition of features to an existing language.

The work was carried out through a number of distinct phases. Phase one was an investigation into the factors which affect reusability with the aim of selecting one factor for further detailed study. For phase two of the project, traceability of information from the analysis documentation to the final product was selected. The aim of this second phase was to identify aspects of the development process where this feature could be improved. The major finding was that information is lost during the design phase causing a reduction in traceability of information and leading to less reusable products. Phase three was to devise a design method that successfully transfers the information identified during analysis to the implemented system thereby enhancing the reusability of software components. This led to the development of the Sociable class technique. The feasibility of the method was tested in phase four by implementing a sample of the constructs required in four object oriented languages. These constructs were used in the development and extension of small systems. The next phase of the project was to evaluate the Sociable class technique against other design techniques which can be used to represent the information. Finally, the design method was tested by the development of a complete case study.

1.2 The importance of reuse

Reuse of software components is a long held aim of the software industry. It is seen as one way to mitigate the effects of the so-called *software crisis*. This term is used to describe the inability of

application developers to keep up with the demand for new software. This demand might be met by an increase in productivity which could be achieved by the efficient reuse of components.

There are many examples of studies highlighting the commonality of code between different systems. For example, Jones [1] reported that studies into the code used in different systems had shown that between forty and seventy percent of code could be found in more than one system. He suggested that up to eighty five percent of code in those systems could be replaced by standard, generic code. Another example was given by Meyer [2] when he identified the frequency with which table searching algorithms are used in programs. Reuse of code in these two examples would lead to an improvement in productivity. Of course, some reuse of code already occurs, for example, libraries of reusable algorithms have been provided for many years. Recent studies have shown that reusing code results in increased productivity. For example, Lim [3] reports studies which demonstrate that in some projects the productivity of programmers, as measured by lines of code produced per month, increases by up to 57 percent.

The advent of object oriented languages has increased the size of the reusable component. One ideal, quoted by Cox [4], is that systems should be developed

“from well-stocked catalogues of reusable software components”.

Software system assembly would then be similar to the assembly of engineering products. It appears that this ideal is not currently being met.

Hall [5] sees reuse as a means of improving productivity without compromising the quality of the product. Whereas Basili [6] sees reuse as a way to improve software quality. Meyer [2] concurs with this second view. He points out that reusing software results in less code being written. This should allow more time to ensure correctness of the software. In addition, the testing and correction of errors in a component is effectively continuing throughout the life-time of the system in which it is used. As Lim states [3], the effect of the correction of errors in reused software is cumulative and results in higher quality components. Thus, reusing components after they have been successfully included in a system should lead to improvements in quality.

Reuse of components can occur at any stage of the development process. Identification of reusable components at an early stage of development will be more beneficial than identification during the later stages. There are several reasons for this. For example, the entities identified at early stages of development are less implementation dependent than the components of the actual product. Also, the early identification of a reusable component may save the development costs for that part of the implemented system.

The reuse potential of components varies. A small application independent component, such as a list or windowing system, will be more reusable than an application specific component. However, as Biggerstaff [7] points out, the cost savings gained by the reuse of larger units will be greater than those gained by the reuse of smaller, more general products, such as lists and windowing systems.

It should be appreciated that reuse may incur costs. For instance, a generalised component may cost more to develop than a specialised one. A component intended for reuse will also need to be tested more thoroughly than a single purpose component. The actual cost increase is not a constant value. The extra cost is reported by Lim [3] to vary between 111 percent and 480 percent of developing non reusable code. The increased cost of development should be offset against the reduced costs when reusing the code. In the same report, Lim suggests that the cost reduction of reusing code varies between 10 and 63 percent of the cost of developing new code. From these figures, code reuse appears to increase productivity in the best case after the first reuse of the component and in the worst case after reusing the code eleven times.

The ability to reuse components is important in improving productivity and the quality of the final product. The recent publication of papers extolling the virtues of reuse [3] and the benefits of systematic reuse [8] suggest that the potential for reusing software components is not fully exploited. Further research into this area is important in view of the fact that the demand for new systems exceeds the number of new systems that the suppliers are capable of providing.

A more detailed investigation into software reuse was carried out during the early stages of the research project. The results of this investigation are reported in [9].

1.3 Overview of the thesis

Chapter 2 begins with a definition of the term reuse. The chapter then discusses factors which are considered essential for the development of reusable components and identifies which of these factors are provided by object oriented development techniques. This discussion leads to the conclusion that the ability to trace information through the system development process is fundamental to improving the reusability of components.

The results of an investigation into the traceability of information in current object oriented development processes are presented in chapter 3. This investigation found that the analysis process identifies more types of information than can be represented in programming languages. The design process must convert the information in the analysis model into information which can be modelled by the languages. This conversion reduces the traceability of information with the effect that some of the information is lost. The reduction in traceability is shown to lead to several problems. In particular, it is difficult to look at the implementation of a software component and know which parts of it represent intrinsic properties of the component and which are required to represent the other types of information captured during analysis. This limits the potential for reusing the classes.

Chapter 4 proposes a design method, the Sociable class technique, which provides a mechanism for explicitly representing one type of information which is currently lost during the design process. The design method is assessed in a feasibility study. The traceability provided by the use of the Sociable class technique is shown to have the potential to reduce some of the problems encountered when reusing components.

Chapter 5 evaluates the relative merits of the Sociable class technique and other methods which can be used to represent the same information. Each method is initially assessed by considering the traceability of information. The Sociable class technique was developed to improve the traceability of information as a means of enhancing the reusability of components. The evaluation therefore begins by comparing the traceability of information provided by each design method. The broad aim of the project is to improve reusability. The reusability of components was shown, in chapter 2, to be affected by several other factors. The effects on the other factors of using the design methods is also considered. The evaluation showed that the Sociable class technique is potentially capable of providing a greater improvement in the reusability of components than the other methods selected for the comparison.

Chapter 6 describes the development of a complete case study using the Sociable class technique. The case study was chosen to demonstrate the use of a variety of different associations and was carried out to verify that the design technique can be applied to a complex system.

Finally, chapter 7 assesses the importance of the Sociable class design technique for improving the reuse of components and suggests further work which would be necessary to develop this design technique to a stage where it is viable for an industrial product. Other areas in which traceability could be improved are also noted.

Chapter 2

Reuse and object oriented techniques

It is claimed by Meyer [2] that object oriented development techniques increase the reusability of software components. The increase in reusability is brought about by the use of features such as encapsulation and information hiding. This chapter discusses these features and other factors which are considered important in enhancing the ability to reuse components. However, before discussing these factors, it is necessary to define what is meant by the term reuse. This chapter begins in section 2.1 with a discussion of possible interpretations of reuse and identifies the definition used in this thesis. Factors which affect the reusability of components are discussed in section 2.2. Section 2.3 selects one of the features for further investigation. The feature selected is traceability of information. This is shown to be a fundamental requirement when considering improvements in the production of reusable components.

2.1 Definition of reuse

The most obvious definition of reuse is the one used by Wirfs-Brock et al. [10]. They state:

"Software is reused when it is used as part of software other than that for which it was initially designed."

Reuse also occurs when an existing system is enhanced to add new functionality, that is the code is reused in successive releases of a product. This type of reuse occurs in many systems and, as Sommerville [11] and Lamb [12] point out, is often carried out under the guise of system maintenance. From this it appears that enhancements to a system are seen as comparable to the correction of errors. Naur [13] has shown that this type of reuse leads to loss of program structure and readability. There seem to be two possible interpretations of the term reuse, that is, a component is reused when it is included in a new system or in an extension to the system for which it was originally designed.

Further interpretations are possible. A component can be reused with or without modification. Wirfs-Brock et al. classify reuse with enhancement or modification as refinement rather than reuse.

These different definitions of reuse affect the reusability of the component. If the definition of reuse is restricted to mean reuse without change, the scope for reuse will be restricted but reliability should be enhanced. Increased reliability would be expected because the amount of testing is effectively increased as a result of the use of the component. If, on the other hand, the scope is extended to include refinement as defined by Wirfs-Brock, then the increase in reliability is likely to be compromised but reusability will be extended. Increased reusability should be expected because the changes to the component will make it available for use in more systems.

The above definitions are concerned with code reuse. Rubin [14] extends the potential for reuse to include analyses, designs, implementations, test cases, and documentation. A similar broad definition of reuse is used by the editors of Software Reusability [7]. The definition can be further extended to include the reuse of personnel and processes [6, 2].

This thesis is concerned with improving the reusability of components. The investigation of reuse is therefore restricted to the development and reuse of components. Thus, the areas examined are the technical aspects of development. The reuse of personnel and processes are not considered. Reuse, in this defined context, will provide most benefit if the term is considered in the widest sense. In view of this, the following definition is used.

Components are considered to be reused if they are used, with or without modification, in either a new system or an extension of an existing system.

2.2 Factors affecting reusability

This section discusses factors which affect reusability. The factors have been identified by examining reuse in traditional branches of engineering as well as software engineering. When reuse is defined to mean using software components in systems other than those for which they were originally intended, the situation equates to the use of 'off the shelf' parts in a traditional engineering environment. Factors which allow such parts to be used are considered in this section.

The factors which have been identified are:

- Encapsulation and information hiding
- Understandability
- Compatibility
- Standardisation
- Reliability
- Design complexity
- Extensibility
- Product efficiency
- Traceability

More details of this work can be found in [9, 15, 16].

2.2.1 Encapsulation and information hiding

Encapsulation and information hiding are identified by Lamb [12] and Lenz [17] as important factors in increasing the reusability of software products. These two features are used to control the amount of information which must be understood by developers. Encapsulation is the grouping of closely related functionality into a single component. The use of information hiding techniques means that the user of the component only needs to know what it does and not how it works.

The description of what a component does defines the interface of the component. The interface provides the only means of accessing components designed using encapsulation and information hiding techniques. The user of such a component cannot make use of any implementation details which

may be known but are not revealed in the interface. This has the result of appearing to reduce the complexity of a system which in turn makes the system easier to understand and develop. As Stovsky and Weide [18] point out, traditional engineers construct new systems primarily from reusable components. These components are chosen by their external behaviour, or interface, not by their implementation details. Lenz [17] further states that the provision of these features should allow software to be used as building blocks thus permitting the software industry to emulate the electronics industry in the method of construction.

The size of the encapsulated components is important. Rubin [14] suggests that reusability can be improved by factorization to encapsulate the information in smaller units. This involves breaking a large complex component into smaller, less complex components. Each of the smaller, less complex components can be reused individually. These smaller units have increased reuse potential when compared with the encapsulation of larger, more complex components. Thus, smaller components are more reusable but, as mentioned in section 1.2, provide smaller cost savings.

Information hiding and encapsulation are both provided by object oriented development methods. As Booch [19] explains, the object oriented approach to system development uses encapsulation to simplify complex situations by viewing the required system as a group of interacting objects which represent objects in the problem domain. A system built by following object oriented principles consists of components known as objects. These components communicate via interfaces to perform the required functions, in much the same way that a car consists of many parts all joined together to form a mode of transport. The encapsulation provided by objects is explained in more detail in chapter 3.

2.2.2 Understandability

It is necessary to understand a component before it can be used. Both Biggerstaff [7] and Rubin [14] agree that difficulty in understanding a component is a fundamental problem which limits its potential for reuse. Meyer [2] points out that it should be possible to understand each of the modules, or components, from which a system is built without needing to understand the rest of the system. This requires that the way in which components communicate should be explicit in the text describing the components.

It is important to be able to understand the complete system as well as the individual components. The time taken to understand a system increases the cost of both correcting errors and extending a system. The problem of understandability is identified by Standish [20] who quotes Lientz et al. [21] as showing that 50-90 percent of software maintenance time is spent in understanding the program. Software maintenance itself can account for anything up to 90 percent of the life cycle costs.

Lamb [12] sees understandability as a goal of preliminary design. He states that the purpose of preliminary design is to divide a system into modules which can be easily understood and which communicate in such a way as to make the whole system easy to understand. It is also stated that a system is easier to understand if the modules used communicate with few others and provide features which are closely related.

However, Naur [13] suggests that it is almost impossible to understand a program written by someone else even with all the relevant documentation. He further suggests that even a well structured program will not retain its structure if maintained by people not involved in its original design.

It seems generally agreed that the difficulties in understanding software increase the costs of maintenance and enhancement. Enhancement is included in the definition of reuse used in this thesis. Thus, it seems clear that improving the understandability of components is fundamental to improving reuse.

2.2.3 Compatibility

Biggerstaff [7] points out that reusing components involves composing them to make new computational structures. In order for a group of components to be used together they must be compatible.

Berlin [22] has reported problems caused by incompatibility of components. These problems resulted from the fact that code relating to decisions concerning error handling and validation of parameters was distributed throughout the hierarchy and not specified in one place. A decision to validate all the required parameters together rather than one at a time, was shown to mean that apparently useful classes required major re-working and re-coding before they could be used. This involved the breaking of encapsulation. These problems show that the design choice made concerning the way in which errors are handled can lead to incompatibility of components. In some systems, the provision of a default value may be the best way to deal with an error produced by a missing or invalid parameter. In another system the user may not wish to use the chosen default value and so is unable to use the class without modification.

Compatibility of components must be ensured before a system can be assembled. Compatibility should be considered when designing components for reuse.

2.2.4 Standardisation

Standardisation takes several forms. It ranges from the use of standard components to the use of standard methods and notations. This section examines two aspects of standardisation. These are components and design models. More details can be found in [16].

- Components

Stovsky [18] states that in traditional engineering, standard parts are used as much as possible both for reasons of cost and because their functionality and design characteristics have already been tested. The cost of developing a special design for a small industrial motor is estimated, by Leech [23], to be at least fifty percent more than the cost of the standard machine. Leech also suggests that if a non-standard design is needed it should be compatible with other standard parts, thus emphasizing the requirement for compatibility between components.

Most components used in both the electronics and manufacturing industries are readily available 'off the shelf'. A new product consists mainly of 'off the shelf' standard parts. These parts can be used because they:

1. have a well defined standardised interface, for example, size and/or connections to the outside world,
2. work almost as efficiently as specially designed parts,
3. are carefully tested and certified.

The proportion of standard parts varies in the different branches of engineering. Electrical goods contain more standard parts because of the relatively limited variety of those parts. Civil engineering is at the other end of the spectrum with standard processes rather than standard parts being used.

- Design models

In order to allow designs to be effectively communicated between developers, models of the products are developed. In traditional engineering these models can take the form of prototypes,

physical models or drawings. Different types of drawings are produced to serve different purposes [24]. For instance, separate diagrams are used to show the structure of the product and the control systems, that is the sequence of operations required.

Graphical standards for engineering drawings were introduced in the 1950's. All diagrams conform to these standards. The use of standards allows engineers from one branch of engineering to understand diagrams produced by an engineer working in another branch. These standards do not impose a certain style on the designer nor do they limit innovation. The first point is noted by D'Ippolito and Plinta [25]. They highlight the fact that architectural designs can be produced in many styles.

There are no such standards in software engineering. Biggerstaff [7] identifies this as a factor decreasing the reuse of design information. Horowitz in [7] has noted similar problems with recording the results of domain analysis. It is further suggested by Naur [13], that it is impossible using current design drawings to record all the information required to allow a newcomer to a system to understand it completely. Improvements in the area of representation of design information have been identified by the National Research Council [26], as necessary to prevent the loss of key pieces of information.

Standardisation may alleviate some of the problems associated with incompatibility of components. For example, a standard method for error handling would reduce the conflicts between components. Standard design models may help developers identify components for reuse.

2.2.5 Reliability

Reliability is a key feature when using components. It is clear that unreliable components produce unreliable systems. Unreliable systems in any environment will not be used, unless there is no alternative. Reliability of components and systems can be enhanced by thorough testing.

Traditional engineers base new products on carefully tested and certified components. Pugh [27] indicates that testing is included as part of the specification of each component and is not a separate activity. In fact, D'Ippolito [25] states that a chemical production plant was built without any tests of the design being necessary. This was possible because the models of all parts, that is the components, interfaces and controls, were validated prior to inclusion. The only testing performed was to ensure that the plant was an accurate implementation of the design. He also points out that engineers are now taught to design the unit operations and the objects that provide them, such as pumps, instead of being taught to design a particular kind of structure such as the chemical plant. He suggests that software engineers should be able to proceed in the same fashion. However, this view of testing in traditional engineering is not universal. Leech [23] states that it is almost certain that a product or its sub-assemblies will not work when first tested. Many redesigns and re-tests are said to be needed, leading to increased costs and causing difficulties in initial cost estimation. The redesigns and re-tests of parts after initial completion of the product suggest that the sub-assembly, interfaces and control models were not validated before assembly in the final product.

Reliability in a software environment is improved by using languages which are strongly typed and by basing the system on modules which are thoroughly tested before building and testing the complete system. Strongly typed languages are type checked statically by the compiler. The use of such languages helps to ensure that the system will not crash with a run time type error which could result in loss of information but does not, of course, ensure that the system provides the desired functionality.

The use of encapsulation and information hiding improves reliability. Encapsulation and information hiding ensure that the internal structure of a module cannot be accessed from outside the module.

The only access to the structure is via the features provided by the interface. These modules can then be used as pre-tested components with known behaviour. Components developed during object oriented development are produced following the principles of encapsulation and information hiding. It should therefore be possible to produce reliable reusable components from which to build a system.

2.2.6 Design complexity

A complex system is more difficult to understand than a simple system. It seems clear that a simple design will be easier to understand than a complex design of the same system. Understandability has already been identified as a factor which affects the extensibility of a system. In view of this, design complexity will also affect the extensibility of a system.

In the traditional engineering field, Leech advises the use of the simplest workable design. He suggests that the number of parts should be reduced and the product made as small as is compatible with other requirements.

Metrics have been developed by Pugh [27] to calculate the efficiency of an engineering design. One metric is called the complexity factor and takes into account the number of parts (N_p), the number of types of parts (N_t) and the number of interfaces (N_i).

$$\text{complexity factor} = \sqrt[3]{N_p * N_t * N_i}$$

A design with a low complexity factor is assumed to be better than a design with a high complexity factor.

An alternative metric is the design efficiency which uses the minimum number of parts (NM) and the time taken to assemble them (TM).

$$\text{design efficiency} = 3 * NM / TM$$

A higher value for design efficiency is assumed to be better than a low value. Thus a design with more parts which can be assembled quickly is considered more efficient than a design with fewer parts which take longer to assemble. The exact correlation between these two metrics is not known but a high design efficiency is assumed to correlate with a low complexity factor.

There is no agreed list of good design characteristics in object oriented software. Wirfs-Brock et al. [10] use simplicity as their empirical measure of design quality. They suggest several criteria on which to base a judgment of the design. A simple design will have:

1. fewer more intelligent classes or components,
2. more subsystems encapsulating the application specific functionality,
3. few contracts per class—this corresponds to a smaller interface between components,
4. deep inheritance hierarchies—the use of inheritance is explained in sections 2.2.7 and 3.2.3.

However, the use of deep inheritance hierarchies can also be viewed as adding complexity. As Rubin [14] points out, the use of inheritance leads to distributed definitions of classes. This makes each class more complex and also more difficult to understand. Rumbaugh [28] gives no guidelines for selecting a good design. He suggests that the design should be optimized to make the design more efficient. Optimizations include storing a derived value to save the processor time required for recalculation every time the value is needed. Thus, Rumbaugh bases the criteria for a good design on improved product efficiency rather than reduced design complexity. Booch [19] suggests evaluating designs by comparing facets such as computational efficiency, synchronization problems, independence from hardware

and simplicity of implementation. Again these features are not readily measurable and combine both design complexity and product efficiency. Lamb's view on preliminary designs are relevant here. As mentioned in section 2.2.2, his view is that systems should be composed of modules each of which provides a closely related set of features and that these modules should communicate with as few others as possible, that is they should have as small an interface as possible.

A law, called the Law of Demeter, has been developed [29] to help formalize the meaning of 'good style' for object oriented programs. It is a way of restricting and documenting the dependencies between classes in order to increase the ease of modification of the classes. Adherence to this law is said to help in the production of a 'good design' and could be used to assess designs by checking if they follow the law. Valid violations of the law are said to exist. A law such as this is not a simple design metric such as is available to traditional engineers.

It appears from the above discussion that the quality of a design is a difficult feature to quantify. One possible metric is design complexity. Several factors appear to be important when considering design complexity. These factors include the number of parts required, the size of the interfaces between the parts and the number of types of parts used in the system.

2.2.7 Extensibility

It is stated, in section 2.1, that reuse is considered to include the use of a component in an extension to, or an enhancement of, an existing system. The reason for this is that the expectations of software systems are constantly changing. As Lamb [12] points out, the term maintenance is used in a software development environment to include modifications and enhancements to the system as well correcting faults in the original system. Thus it appears that a requirement to improve a system by increasing its functionality should be anticipated and systems designed and documented accordingly. Rubin [14] states that the ability to produce systems which are adaptable to meet changing specifications is a fundamental requirement of an effective development methodology. This implies that the process of extending or adapting a system should maintain the original good structure of the program and not increase the design complexity unnecessarily. Naur [13] states that the structure of a program is often lost during repeated maintenance due to a lack of understanding of the system which is brought about by the inadequate mechanisms for explaining the development of the original system.

Some programming languages provide features which permit adaptations to be made. One such feature is genericity. This feature allows a general component, such as a list, to be declared. Specific lists, such as a list of customers, are then derived from this general list. This reduces the amount of code to be written and so reduces development time and cost.

In an object oriented development environment, inheritance is used to provide a means of producing incremental changes. Inheritance allows a new software component to be derived from an existing one. The new component shares all the properties of original component but defines its own additional properties. This sharing of properties reduces duplication of information with the result that the time required for testing is reduced. However, it is reported by Nino [30] that the method used to implement inheritance in many object oriented languages reduces the quality of encapsulation. Inheritance appears to improve reusability by providing a mechanism for incremental changes to be made but may also adversely affect reusability by weakening encapsulation.

Genericity and inheritance are important concepts for increasing the reusability of code. They are explained in more detail in chapter 3.

2.2.8 Product efficiency

Product efficiency is important because users of systems wish to have them work quickly and efficiently. In traditional engineering, component efficiency can be measured. This is documented as the performance characteristics of the component. The performance characteristics of each component will, at least in part, determine the product efficiency.

In software systems, efficiency often means speed of execution. This can be assessed by measuring the amount of processor time required to perform a function. Efficiency is affected by factors such as the algorithms used and the requirement for the processor to check at runtime that the operations are valid. This checking requires processor time and therefore reduces the speed of the system.

An alternative interpretation of software product efficiency is the amount of computer memory required to both store and run the system. Computer memory is becoming cheaper which reduces the importance of this measure of efficiency.

In systems where speed of execution and size of code is critical, such as operating systems, code reuse may not be seen as applicable. This is because a reusable component is likely to be more general than a specially designed component. As Rubin [14] points out, a general component is unlikely to be the most efficient solution to a specific problem. However, the analysis and design may still be reusable.

The importance of product efficiency as measured by speed of execution depends on the type of system being developed. Product efficiency is more important when developing operating systems and real-time applications than when developing systems which require a large amount of user interaction such as information retrieval systems.

2.2.9 Traceability

D'Ippolito [25] points out that it must be possible to trace the requirements from the initial documents through the models of the required system to the final product. This allows the user to have confidence that the system provides the desired functionality. The final product in a traditional engineering environment closely matches the models. Any changes made during production must be recorded to allow the same changes to be made in subsequent products. This maintains the traceability of information from the requirements to the implementation. The information required for construction and maintenance is contained in standard documents.

The software engineer does not have the same incentives to keep documentation up to date because software can be reproduced easily. Software reproduction is a simply matter of copying the code so there is no requirement to understand the structure of the system during manufacture. However, the structure must be understood if errors are to be corrected. In order to correct an error, the part of the code responsible for the affected functionality must be located. This process will be simplified if the structure of the implemented system directly reflects the analysed requirements. For example, understandability will be improved if constructs, such as data, functions or interactions, which are identified during analysis are clearly visible in the implementation. The improved understandability resulting from increased traceability should reduce the time required to trace an error and therefore reduce the costs incurred. The improved traceability should also help identify or even prevent any side effects caused by the corrected code. Traceability of information should be an important consideration during software development even if the system is designed to remain unchanged.

Traceability is even more important when considering systems which are designed to be upgraded or extended. The desired enhancements to a system can be shown by adding extra information to the analysis models. The same enhancements must be made to the implemented system. In order to do this, it is necessary to understand both the components which implement the current system and the way

they interact. As mentioned before, the process of understanding the current system will be simplified if all the information contained in the analysis models is traceable through the design and is visible in the implementation of the system. The design and implementation may, of course, contain additional information. Traceability of information requires that all the constructs visible in the analysis models have direct counterparts in the implemented system. The availability of such constructs should also facilitate the addition of the extra functionality. The process should involve the simple addition of the desired behaviour.

Improving the traceability of information by the provision of separate implementation constructs for each type of information in the analysis model will help to ensure that encapsulation and information hiding are fully provided. The types of information include both the entities required in the system and the relationships between those entities. Encapsulation and information hiding will be improved because one construct will encapsulate one piece of information. Identification of components for reuse in other systems should be easier because each construct will encapsulate closely related information. Improved traceability of constructs may also make it possible to maintain a program's structure even if maintenance is carried out by people other than the original developers. This would allow more incremental increases in a system functionality while maintaining the predictability of the behaviour. It should be remembered that Naur has identified problems due to the lack of adequate mechanisms to explain the development history of a system. The lack of such mechanisms causes the structure of a system to deteriorate during maintenance procedures. Providing traceability of information by representing each type of information by a separate construct may provide an adequate mechanism for explaining the development of a system.

The ability to trace a concept from the analysis model through to the design and implementation is likely to involve a standard mechanism for representing the concept at each stage. Thus, providing increased traceability might also improve standardisation which is another factor required to enhance reusability.

It seems clear that improving the traceability of requirements from the initial specification through to the implementation will improve the understandability of the components and enable their reuse.

2.3 Summary and Conclusion

This chapter has presented the background information to this research project. The term reuse is defined, section 2.1, to mean **use, with or without modifying the component, in an extension to the existing system or in a different system.**

All the factors identified in section 2.2 are important when trying to improve the reusability of components. Traceability of information is a factor which affects and is affected by many others. For example, as stated in section 2.2.9, providing traceability of the different types of information through the development cycle may provide improved encapsulation thereby simplifying the implemented system with the result that it is easier to understand. Such traceability would be provided by the use of distinct constructs for all types of information. Traceability may make the design appear less complex because a developer will be able to identify the relationship between the implemented system and the analysed requirements more easily.

Traceability is also related to standardisation. Improving traceability might lead to the development of standard mechanisms for implementing the constructs identified during analysis. Improved standardisation would have the added advantage of enabling users to understand the development of the implementation more easily. It was suggested that improving traceability of constructs might improve the extensibility of systems.

It appears that providing traceability of information by the provision of distinct implementation constructs to represent each type of information contained in the analysis models may lead to improvements in other factors required to enhance reusability. Traceability of information was therefore chosen for further investigation.

The next chapter, chapter 3, presents the results of an investigation into the traceability of information in several approaches to object oriented development. The aim of the investigation is to identify any shortfalls in traceability using current techniques. Such shortfalls can then be addressed with the prospect of improving reusability of components. The research presented in this thesis concentrates on developing an object oriented design method which promotes the possibilities of reusing the components identified during analysis by improving traceability of information.

Chapter 3

Traceability in object oriented software development

Traceability of information from requirements analysis through to the final product has been identified, in chapter 2, as a feature which should improve both the reusability of components and the extensibility of systems. This chapter discusses the traceability of information through the process of object oriented software development.

Traceability has been claimed to be one of the benefits of object oriented development. The use of such an approach is said by Coad and Yourdon [31, 32] to allow the analysis results to be systematically expanded into object oriented design and programming. Rumbaugh et al. [28] also claim traceability of information by stating that object oriented software development provides a seamless transition from analysis to implementation involving the clarification of features rather than requiring modification of work which has already been completed. This implies that it should be an easy task to trace information through the development process and to identify the entities from the analysis model in the final implementation.

Traceability of information during object oriented development was investigated by examining a development process which uses object oriented concepts throughout. Specifically, object oriented analysis and design are followed by implementation using an object oriented programming language. This process avoids any loss of traceability caused by changes in representation style such as might be encountered if following object oriented analysis with implementation in a relational database or structured programming language.

There are several different process models which can be used to describe the software development process. These are discussed in section 3.1 both to provide a basis for the selection of specific development methods and to define the approach adopted for the investigation into traceability of information.

The development methods chosen are the object oriented analysis and design strategy suggested by Coad and Yourdon [31, 32], the Responsibility Driven Approach (RDA) [10] and the Object Modelling Technique (OMT) [28]. These methods are described in section 3.2. The types of information identified and modelled by the analysis methods are discussed in section 3.3.

Section 3.1 shows that a major function of the design phase is to map the results of analysis on to implementation constructs. This means that before the design process can begin, it is necessary to understand the features of the implementation medium. The investigation into traceability during the development process therefore continues, in section 3.4, with a description of the main features of four object oriented programming languages. The languages described are Eiffel v2.3 [2], C++ [33, 34],

Oberon-2 [35, 36] and Modula-3 [37]. The different constructs and relationships provided by the languages are discussed in section 3.5.

Having ascertained both the types of information identified and modelled during analysis and the constructs and relationships available in a selection of programming languages, it is possible to consider the design phase of development. The description of the development process continues in section 3.6. That section compares the entities identified during analysis with those available in the programming languages to identify any differences which must be overcome during the design phase. The section also describes common design methods which are used to convert the information contained in the analysis models into implementation constructs. Section 3.7 discusses the consequences of the changes in the representation of information which are necessary during the design process. Section 3.8 discusses the traceability of constructs during object oriented development. Section 3.9 summarizes the findings.

3.1 The development process

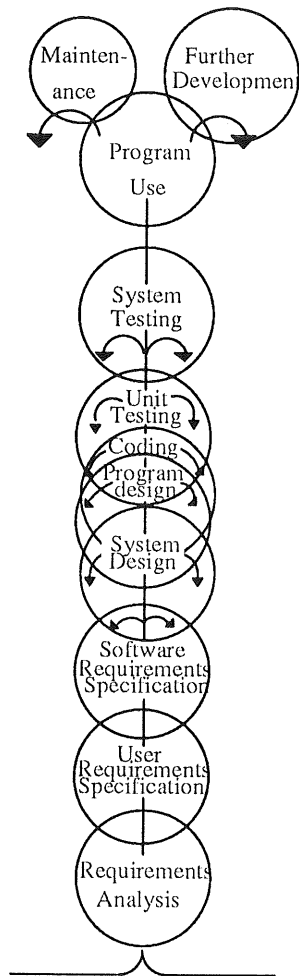
This section describes two models of the development process. The reusability of the artifacts produced by methods following each of the process models is assessed in order to identify the type of development process to consider for this research. Having identified the type of development process it is possible to define the method used for the investigation into traceability of information.

The process of system development can be described by several different models. Two of the models are the waterfall model [11] and the fountain model [38] as shown in figure 3.1. The waterfall model has been used to describe many different approaches to structured development. The fountain model was proposed by Hendersen-Sellers and Edwards as a process model for object oriented system development.

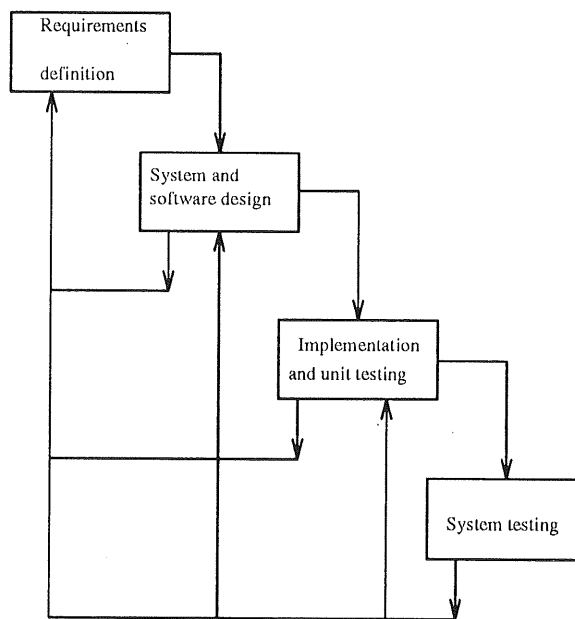
The phases of the process models are broadly similar. Both process models begin with the definition of the requirements of the system and the required software. The processes continue with the design of the system followed by coding, implementation and testing. Finally the system is installed for use. A minor difference between the models is that the fountain model shows more phases for the individual parts of the requirements analysis than the waterfall model.

There are, however, two major differences between the two models. The first difference is that the phases of the fountain model overlap whereas the phases in the waterfall model are shown as distinct stages with iterations between the phases. The authors of the fountain model stress the value of this overlap between the stages. For example, beginning the implementation before the design is complete is said to result in a more robust and more flexible design. The second major difference is that Hendersen-Sellers and Edwards suggest the use of the same programming environment throughout the fountain development process. This is said to result in a system that is easier to maintain and extend. It also allows classes available from previous systems to be considered for inclusion in the present system at an early stage.

The use of a method following the fountain process model, whilst offering the advantages mentioned above, has the disadvantage of merging the implementation with the rest of the development process. The use of specific features of the implementation medium, such as uncommon features of the language chosen, will be incorporated into the analysis and design models. These models may not therefore be language independent. This will probably mean that the analysis and/or design results cannot be reused either for implementing the system in a different language or for using a different style of implementation, thereby restricting the reusability of the results. It was stated in section 1.2 that the greatest benefit of reusing components is obtained by their identification at early stages of development



i) The Fountain model



ii) The waterfall model

Figure 3.1: Life cycle models

because these are more independent of the constraints of the final implementation medium. It appears that the use of development methods which follow the fountain process model restricts the reuse of analysis results. Therefore, methods which use the fountain approach, such as the method proposed by Booch [19], are not considered in this discussion.

The three different approaches discussed in this chapter follow the waterfall process model and complete the analysis and design stages prior to considering the actual implementation language. All the approaches assume that a natural language problem definition of the required system is available and thus begin part of the way through the life-cycle with analysis. The analysis phase is discussed in section 3.2.

The software design phase of development in the waterfall model involves deciding the style, such as object oriented or functional, in which the system will be implemented. The software design phase also determines the full specification of the required software in the chosen style without being committed to the use of a specific implementation language. Software design therefore involves mapping the results of analysis on to the implementation constructs which are available in the chosen style of implementation. An understanding of the general features of these languages is required before the process of object oriented design can be fully understood. The features of object oriented languages are described in section 3.4. Implementation constructs are discussed in section 3.5 prior to a discussion of design issues in section 3.6.

3.2 Analysis methods

This section presents the results of the first phase of the investigation into traceability of information during object oriented development. This phase requires the types of information identified and modelled during analysis to be ascertained. The analysis process produces models of the requirements to aid the understanding of the required system and provide a basis for the design and implementation of the required system.

The methods being discussed are the object oriented analysis and design strategy suggested by Coad and Yourdon [31, 32], the Responsibility Driven Approach (RDA) [10] and the Object Modelling Technique (OMT) [28]. All the methods begin with a natural language definition of the required system. Each method uses a different strategy and produces different models. This section identifies both the strategies used to produce the models and the way in which the information is modelled.

In order to produce a useful discussion, analysis has been defined as a process which identifies:

- classes and objects including the data associated with and behaviour required of objects,
- relationships between objects and between classes to provide the required functionality,
- subsystems.

This definition of analysis combines different phases in each of the methods examined. The definition has been chosen because it results in a specification of the required system being produced. The discussion begins in section 3.2.1 with a description of the different approaches adopted by the three methods. Sections 3.2.2 to 3.2.4 discuss the identification of each of the above constructs. The types of information identified during analysis are discussed in section 3.3.

3.2.1 Approaches used

Coad and Yourdon define object oriented development as including the use of objects and classes, classification, inheritance and communication with messages. They describe object oriented analysis as

combining features from semantic data modelling, object oriented programming and knowledge based systems [31]. Analysis is said to be the study of the problem and application domain. The initial phase of their method is heavily biased towards the stored data and the operational procedures which are an inherent part of that domain. The functionality of the application being developed is considered in their design stage which defines four separate views of the application producing four separate models. The models produced show:

- the problem domain,
- the user interface,
- the task management,
- the data management component.

Each model uses the same notation to show the classes involved, their structure and interactions. Class specifications are also produced.

The Responsibility Driven Approach views a system as a group of objects which represent the roles required to provide the desired functionality. Each object is required to provide some services and may require information or services from other objects. Classes providing the services are called servers and classes using the services are called clients. The list of services which can be requested by a client form a contract between the client and the server classes. The relationships between classes are called collaborations. The system is developed in terms of the Client-Server model. The method discussed here was developed by Wirfs-Brock et al. [10] and deals with all aspects of the required system during all stages of development. The models produced are :

- hierarchy graphs to show the inheritance hierarchy of the classes,
- collaboration graphs to show the way the objects interact to provide the functionality required,
- the specification of the classes, subsystems and collaborations.

The OMT method, in common with the RDA, analyses the required system and not the problem domain but also suggests using domain knowledge while identifying classes. Three models are produced to show different aspects of the required system. These are:

- the object model to show the static structures or classes involved in a system, the relationships between the structures and the operations performed by the objects.
- the dynamic model to describe the changes which occur over time in the system.
- the functional model to show how the data flows through the system.

A different notation is used for each model. These models must be combined before the class specifications can be produced.

It can be seen from the above descriptions that the three methods adopt different approaches to the modelling and production of the specification of the required system.

3.2.2 Classes and Objects

This section discusses the first stage of the analysis process as defined in section 3.2, that is the identification of classes and objects. In order to gain a good understanding of the classes and objects identified during analysis this discussion compares the meaning given to the terms class and object, the means of identification of classes and objects, the notation used and the behaviour required of objects.

1. Meaning of the terms

The Coad and Yourdon view of objects is that they encapsulate attribute values and exclusive services. Objects represent an abstraction of something in the problem space, reflecting the ability and requirement of the system to store information about it and/or interact with it. The Responsibility Driven Approach defines objects as specifying the roles and responsibilities of each object. The responsibilities can involve the requirement to store knowledge, that is data, or behaviour. In the Object Modelling Technique, objects are viewed as combining data and behaviour. Thus all the methods view objects as combining data and processing.

Classes in all the methods describe groups of objects with the same attributes and behaviour and also describe how to create new objects of the class. The classes do not involve any notion of defining a set of objects or operations on the group of objects produced. Collection classes, such as 'sets of customers', are not usually identified during analysis.

2. Identification

The Coad and Yourdon strategy identifies the initial classes and objects by studying the problem domain—the problem statement and any other information which can be found which is relevant to the problem domain. The guidelines for this process suggest looking for: things and events remembered, other systems, devices or terminators the system will react with, structures, roles played, operational procedures. The objects should: require remembrance, provide services required by the system, have more than one attribute, not be purely derived data. The attributes of classes are simple values. The classes and objects required in the models showing the user interface, task management and the data management component of the system are identified by treating each aspect of the system as the problem domain.

In the RDA approach, the classes are found by examining the requirements specification document and making a list of all the nouns and noun phrases, taking care that the phrasing of the document does not disguise nouns as verbs. The list is reduced by removing duplicates and alternative names for the same things and obvious nonsense. The remaining nouns and noun phrases are then examined to make sure that they fall into one of the following categories.

- physical objects such as a display screen,
- conceptual entities such as a PIN on a card,
- external interfaces such as the user interface,
- values of attributes such as float or real for the value of an attribute length.

The candidate classes are then examined for groups with common attributes. These are used to define preliminary inheritance hierarchies. The final list is then transferred on to cards or other storage medium. Each card contains the name of one class and a sentence describing the purpose of that class.

The OMT method identifies and selects classes from the problem statement in a similar fashion to the RDA but suggests the use of knowledge of the application domain to aid the process. The names and descriptions of the classes are entered in a data dictionary instead of on cards. The attributes of classes are simple values not objects.

All the methods identify classes which are functional in nature and provide the processing required by the system. The books describing both the OMT method and the RDA use an automatic teller machine as an example system. They identify transaction as a class. This class controls the accesses to account objects. The objects of this class, in the RDA, do not require storage. A new

instance of transaction is created when required. The objects are transient objects representing the invocation of a function. They can also be called functional objects. The Coad and Yourdon approach identifies this type of class during the analysis of the task management component. This suggests that objects must have data associated with them but do not necessarily require storage.

3. Behaviour

The Coad and Yourdon strategy and the OMT method both use state diagrams to help identify the required behaviour. The information gained is noted directly on the model of the relevant object in the Coad and Yourdon method but is noted on the dynamic model in OMT and has to be combined with the object model at a later stage.

In the RDA method, the behaviour is defined by the object's responsibilities which encompass both the knowledge maintained by an object and the functions it can perform. The requirements specification is the starting point for the identification of responsibilities. This time, verbs and verb phrases are extracted. The purpose recorded for each class is also useful. The relationships between classes are also used to help identify responsibilities. These responsibilities are then allocated to classes. The following guidelines are given.

- State the responsibilities as generally as possible.
- Distribute the intelligence evenly.
- Put all the information about one thing in one place.
- Keep the behaviour with the related information.
- Share responsibilities among related classes.

The attributes of a class are not modelled directly. Only the type of the attribute is considered to be important, for example whether the attribute is an integer or string. The attributes required by a class can be added as a responsibility for the class to know something, for example

```
Class: Account  
Responsibility : Know the account balance.
```

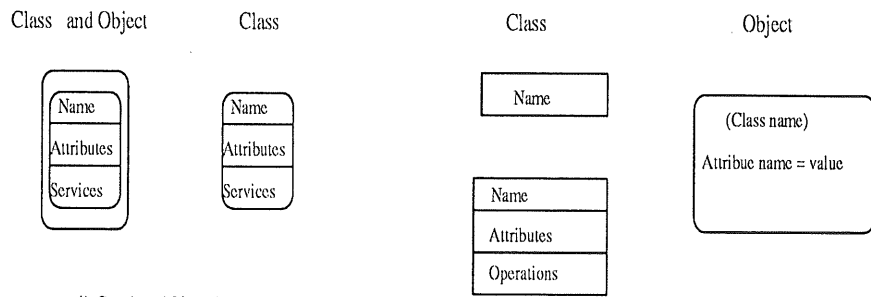
4. Notation

Figure 3.2 shows the notation used to model the classes and objects in the system. It can be seen that the notation used by the three methods is different which reduces the possibility of developers understanding the models produced by the different approaches.

This discussion shows some areas of commonality and difference between the methods concerning the identification of classes and objects. The classes and objects identified by the different methods all represent groups of objects with the same types of attributes and the same behaviour. The classes represent both structures present in the physical system and the functions to be performed by the system. The main differences are the techniques used to identify the classes and the notation used in the models.

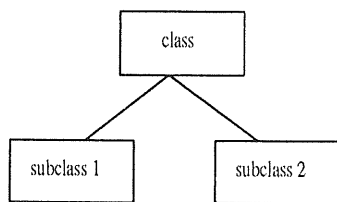
3.2.3 Relationships between classes and objects identified and modelled

This section discusses the second stage of the analysis process—that is the identification of relationships between objects and between classes. These relationships are used to define many aspects of the objects, such as the structure of objects, the static connections between objects and the processing



i) Coad and Yourdon

ii) OMT



class name	concrete or abstract
superclass name	
subclass name	
responsibility	collaborates with

iii) Responsibility Driven Approach

Figure 3.2: Class and object notation

dependencies which are needed to provide the system behaviour. Different types of relationship are identified and modelled by the different methods. Both the OMT method and the RDA method suggest identifying relationships by examining the verbs on the problem statement. Coad and Yourdon give little help with identifying relationships other than defining the different possibilities. The relationships between objects and between classes are discussed under the headings:

- structural relationships,
- non structural static connections,
- processing dependencies.

1. Structural relationships

There are two types of structural relationship. One type is where one class is a specialised type of one or more other classes. This is sometimes known as the *is-a* or the inheritance relationship. It is modelled in all three methods. The other type of relationship is where one object is composed of objects from other classes, an *is-part-of* relationship. These are static relationships and both are therefore noted on the object model in the OMT method. In the responsibility driven approach, the inheritance relationship is shown on the hierarchy graphs and the *is-part-of* relationship on the collaborations graph. Both types of structural relationship can be found on any of the Coad and Yourdon models.

- The inheritance relationship.

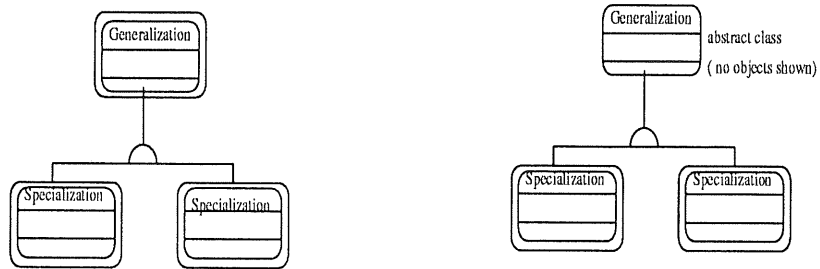
This relationship is identified in the same way by all three methods. However, different terminology is used. Coad and Yourdon use the term Generalization-Specialization (Gen-Spec). The RDA uses the term superclass-subclass relationship. The OMT method uses the term generalisation.

Inheritance is used where an *is-a* or an *is-a-kind-of* relationship exists between objects. For example, a car is a special type of vehicle. All methods recommend that the base class only contains data and operations required by all its subclasses. The structures produced can be either hierarchical, involving inheritance from a single class, or lattice like, involving inheritance from more than one class. All the methods suggest that where classes fulfill the same responsibility, such as display, a common abstract superclass is introduced from which the classes inherit the responsibility. This results in increased multiple inheritance.

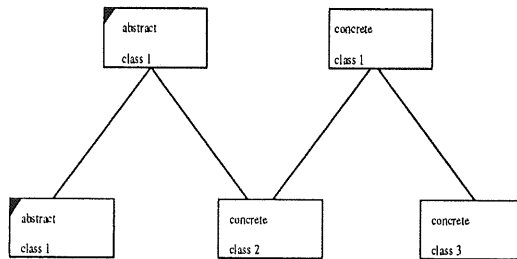
The three methods use different notation to model the relationships. This is shown in Figure 3.3. The OMT and Coad and Yourdon notations are similar in that the class hierarchies are shown as levels of specialisation. These levels could be taken to indicate more compatibility between classes on the same level than is intended. The RDA notation shows no connection between subclasses except that they have the same superclass. The OMT notation makes no distinction between abstract classes which have no direct instances, such as the display responsibility mentioned above, and concrete classes which do have direct instances. The distinction must be made by adding the word {abstract} after an operation name. Figure 3.3 shows that the other two methods make the distinction.

- The *is-part-of* relationship.

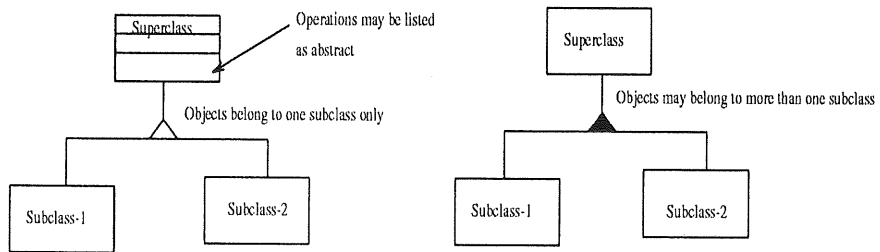
All three methods identify this type of relationship. Wirfs-Brock identifies two types of whole-part relationship. One of these is a composite class in which all the classes involved collaborate to provide the required functions. The other form of whole-part relationship is between a container and its contents. In this type of relationship there is usually very little,



i) Coad and Yourdon notation



ii) RDA notation



iii) OMT notation

Figure 3.3: Inheritance notation

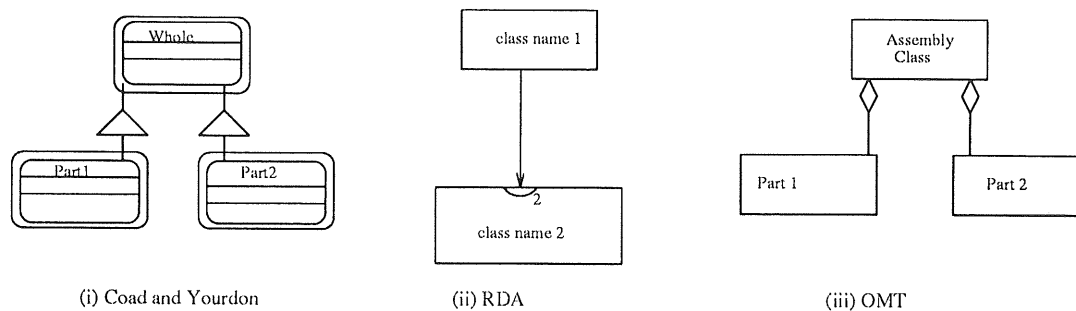


Figure 3.4: Is-part-of notation

if any, collaboration between the whole and its parts. Both types of whole-part relationship are modelled as a collaboration between the objects involved. The notation includes the number of the contract, group of services, which is used by the object.

Coad and Yourdon call objects involved in this relationship a Whole-Part structure. It is used to denote a container and its contents, a collection and its members or a class formed from many parts.

The OMT method identifies the is-part-of relationship as one type of association between objects (the other type of association is discussed in the next section). This type of association represents a physical connection between the object instances. Distinct notation is used to signify this type of association on the object model.

The notation used by each method is shown in figure 3.4. It can be seen from figures 3.3 and 3.4 that the OMT notation for the is-a relationship and the Coad and Yourdon notation for the is-part-of relationship are very similar.

2. Non-structural static relationships

Non-structural static relationships can involve objects from one or more classes. This type of relationship is modelled by Coad and Yourdon as instance connections. These are lines which show which classes participate in a relationship and the number of objects of each class involved.

The OMT method identifies non structural static connections as a second form of association. These associations represent a conceptual connection between objects. They are modelled as lines between the classes involved in the association. The lines are labelled to indicate the name of the association. The line representing the association describes a group of connections between specific objects. These associations are said to model a concept which is meaningful when read in either direction. Examination of the models in the OMT text suggests that this type of relationship is very common.

The multiplicity of the relationship is also noted. Link attributes are also identified. These represent a property of the association. For instance an association between a person and a company might have the properties job title and salary.

Qualifiers can also be added to associations. These are attributes, the value of which usually identifies uniquely a particular instance of a class. For example, the value of the bank code attribute might uniquely identify a bank computer.

The RDA method models all relationships of this type as collaborations.

An example of the notation used by each method is shown in figure 3.5. It can be seen that the OMT notation contains more information than the other notations.

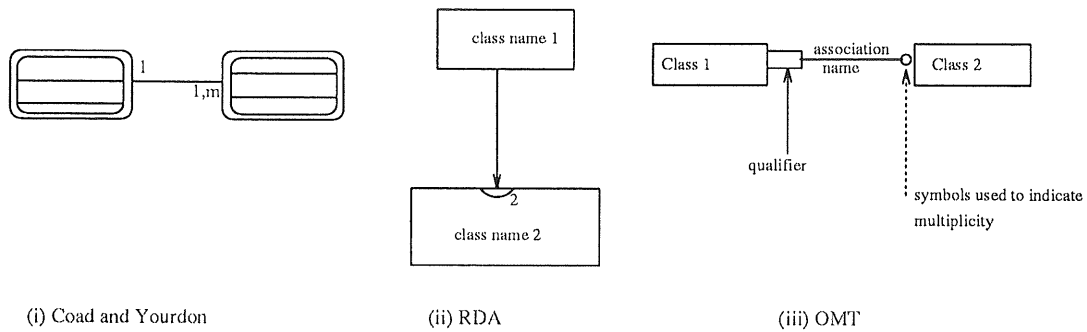


Figure 3.5: Static association notation

These non-structural relationships are clearly different from structural relationships because they do not model inherent properties. Non-structural relationships are formed between specific instances of the class. This is evident from the name instance connection used by the Coad and Yourdon method and the description of these connections as conceptual connections between object instances in the OMT method.

3. Processing dependencies

These relationships are used to show how the objects interact to provide the required system behaviour. They are dynamic links between objects. The RDA approach identifies these relationships as collaborations. Coad and Yourdon use message connections to indicate processing dependencies between objects. These are shown on each model of the system. The system requirements are defined by identifying user task scenarios. In OMT the required system behaviour is modelled on the dynamic and functional model. Scenarios are used to identify external events. The information in these models has to be combined with the object models to produce the process specifications.

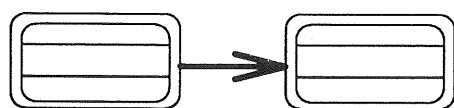
The notation used to model processing dependencies is shown in figure 3.6. It can be seen that there is little correlation between the models produced by the different methods.

This section has shown that the methods identify different relationships between classes. The RDA method identifies many different relationships but models only two. These are collaborations and the inheritance relationship. The other two methods identify and model four different relationships. These relationships are the inheritance relationship, is-part-of, non-structural static relationships and processing dependencies. The three methods use different notations to represent the relationships.

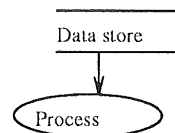
3.2.4 Subsystems

This section discusses the final stage of the analysis process defined earlier. The final stage is the identification of subsystems. Subsystems are considered here to be units into which the system can be divided. They are identified for two purposes. They can aid understanding of the system and allow parts of the system to be developed separately. They are conceptual entities introduced to make the system easier to understand.

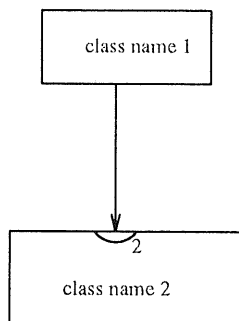
Coad and Yourdon divide the system into four components. These are the problem domain, human interaction, task management and data storage components. The task management and data storage components represent functional groupings. They also divide each component into subjects. The four



(i) Coad and Yourdon



(ii) OMT (Functional model)



(iii) RDA

Figure 3.6: Notation for processing dependencies

components represent different aspects of the functionality required by the system. The subjects represent classes grouped together by structural criteria. The classes at the top of Whole-Part structures become subjects containing the rest of the classes. Further simplification can be obtained by grouping subjects into subdomains of the problem domain.

The RDA approach defines a subsystem as a group of classes which work together to fulfill a set of closely related responsibilities, for example, a printing subsystem. These closely related responsibilities provide a small interface to the subsystem. The subsystems are identified and named after the classes and their collaborations have been identified. However, if the system is large, subsystems may be identified and named before classes are identified.

OMT suggests dividing the object model into modules. Modules are groups of classes and the relationships between them. The groups should represent a logical subset of classes, thereby producing a small interface to the module. OMT identifies subsystems after the three models have been produced. A subsystem in OMT is a group of classes which combine to perform a group of related services or functions. A subsystem should have a clearly defined interface with the other classes and subsystems involved in the application. Subsystems are used as a way of dividing the system to allow parts to be developed separately.

All the methods divide the required system into smaller units called subsystems. Subsystems in the RDA method are defined by functional criteria but in the Coad and Yourdon and OMT methods both structural and functional criteria are used.

3.3 Information modelled during analysis

This section discusses the types of information which are identified and modelled during analysis. The types of information define the starting point for the design process. The types of information are identified by discussing both the information given in the previous section and results from personal and

collaborative research into the use of the methods. The personal experience was gained by using the Coad and Yourdon and RDA methods to develop a kitchen garden planning system. This work is reported in [39] and [40]. The collaborative research involved the development of three systems using OMT [41, 42, 43]. Finally this section specifies the relationships as they are defined for this thesis.

All the methods begin with a problem definition of the required system. They all identify classes as groups of objects with the same attributes and behaviour. The methods used for identifying objects are different but all result in the identification of both structural and functional objects. All the methods grouped the objects into subsystems as a means of dividing the system into manageable components. This suggests that classes defining groups of objects which combine data and operations are not a sufficiently powerful means of managing the complexity of a system.

During the analysis process, some difficulties were encountered in distinguishing whether a particular object required a new subclass to be defined or whether it was just an instance of the class. Examples of this occurred whilst using of the OMT method. The developers of all three systems identified inheritance relationships in their initial analysis models. Subsequently some of the inheritance relationships were changed. In some cases an attribute called type was introduced to the base class and in other cases variant records were used. For example, the development of a kitchen garden planner [41] required the user to be able to add various types of crop to a data store. Some of these types of crop required sowing in a seed bed or greenhouse and then transplanting into the garden. These types were originally identified as subclasses of a class Crop. However, they were subsequently changed to be one class, Crop, with a variant field to contain the extra information required by some crops. The result was that a type parameter was required by the procedure which created the new crop objects and 'CASE' statements were needed to choose the correct code. This reduced the extensibility of the system and required all types of crops to be known when the system was written. The book describing the OMT method [28, page164] which the students were using as a guide also includes a type identifier in the Account class rather than using subclasses for each type. Possibly, this led to the problems.

Other difficulties were also encountered. The use of the Coad and Yourdon approach [39] led to the development of a model in which the required processing was distributed throughout the classes. The distribution of processing caused the problem domain model to look complex with all the classes being tightly coupled. This tight coupling might have been caused by including the reporting and processing requirements as part of the problem domain component rather than as the task management component. The classes developed after following this approach had limited reuse potential because they were tightly coupled.

The Coad and Yourdon method suggests examining the whole of the problem domain rather than restricting the scope of the development to the required system. It is suggested by Aksit and Bergmans [44] that such methods can result in the identification of an excessive number of classes making development difficult. This was not found to be the case when following the Coad and Yourdon method for the system chosen [39]. This was, however, a relatively small application domain. Development of the same system using the RDA method [40], which examines the whole of the required system including user interface and storage requirements, did result in an overwhelming number of classes. This suggests that the system should have been divided into subsystems before development began. Suitable subsystems might be stored data, user interaction, external devices and the required output. However, Aksit and Bergmans [44] suggest that dividing the required system into subsystems before development might not result in the optimal choice of subsystem boundaries. The control of the amount of information to be understood when analysing a system appears to be a difficult problem.

It is clear from the section 3.2.3 that there is a variation in the number and types of relationships identified and modelled by each of the analysis methods. The responsibility driven approach identifies

many different relationships but models them in terms of the inheritance and client-server relationships. This converts many different relationships into two relationships. The information portrayed by the different relationships has therefore been lost and cannot be traced from the requirements definition into the implementation.

The different methods identify different relationships between objects and adopt different names for the relationships when they do correspond. In order to allow discussion of concepts it is necessary to define the meaning of the relationships used in this thesis. The following list provides this definition. The list includes relationships not identified by the above methods but which appear to be important.

- **is_a**

An object is a special kind of something else. An extension of a class is defined to give specialisation. This is modelled by all methods as the inheritance relationship.

- **consists of**

The relationship defines the structure of a complex class, such as:

1. a house consists of one or many rooms.
2. a car consists of wheels, engine, seats etc..

The objects defined by these classes are composed of other objects. The addition of extra relationships of this type changes the structure of the objects produced. For example, the addition of a sun-roof to a car changes the structure of the car.

This relationship is modelled by Coad and Yourdon as a Whole-Part structure, by the RDA as a collaboration and by OMT as a special type of association.

- **contains**

An object contains other objects. Examples of such objects include lists and arrays which are usually identified during design and implementation. They might be used to hold the contents of structures such as a car boot, or the furniture in a room. There is little if any interaction between the container and the contents. An empty container is a realistic notion.

This relationship is modelled by Coad and Yourdon as a Whole-Part structure, by the RDA as a collaboration and by OMT as an association.

- **uses**

This represents a functional or processing dependency. An object requires access to another object to request information to be supplied or to request processing to be carried out. For example, a report object might require access to many objects to produce the required information. The state of the server object is not changed by the request.

This relationship is modelled by Coad and Yourdon as a message connection, by the RDA as a collaboration and by OMT as a flow of data from a data store on the functional model.

- **conceptual association**

This represents a logical relationship between objects which has significance over a period of time. For example, the relationship 'a person owns a car' is a conceptual association. The actual structure of the person and the car are not changed by the addition of the conceptual association. This type of relationship is relevant because of the application not by the nature of the objects

involved. This contrasts with a relationship, such as 'a car has wheels', which is a structural relationship and should be identified as a 'consists of' relationship.

The actual instances involved in this association are significant. For example, assuming the relationship 'a person owns a bank account', it is important that a person withdraws money from an account of their own. This type of relationship occurs between specific instances of objects rather than between classes of object.

These relationships are shown as instance connections by Coad and Yourdon, as collaborations by RDA and as associations by OMT. It appears from the example models in the texts describing the methods that many of the relationships come into this category.

- **temporary association**

These are also logical relationships between objects. They differ from the above in that the actual instances involved are not significant over time, for example, in a petrol filling station the actual instances of a car receiving petrol and the pump involved cease to be important after the petrol has been delivered to the car tank. These relationships are shown as message connections by Coad and Yourdon, collaborations by the RDA method and on the functional model by the OMT method.

- **instantiates**

This relationship is required when one class is responsible for creating an instance of another class. For example, a class will be responsible for starting a transaction in a system used to control an automatic teller machine [10]. The class responsible for the instantiation might be a functional class rather than a class requiring storage of data. This relationship is not identified by any of the methods but has a different meaning from any of the other relationships.

- **dependency**

This relationship is required when several objects cooperate to maintain a value or to perform a process. For example, a minimum balance in an account (A) could be maintained by transferring funds from another account (B). This method would be invoked on the basis of the balance in account A and cause a change in the value of the attribute balance in account B. This relationship is not identified by any of the methods but B. Helm et al. [45] recognise the importance of this type of relationship and have worked on the problems associated with its specification.

This relationship is not modelled by any of the methods.

- **property of**

Properties are single valued attributes, such as colour or length which do not have operations defined on them. The definitions of classes and objects, section 3.2.2, require both data and operations to be present. Properties do not have operations associated with them so this relationship is not considered to be a relationship between objects or between classes. The three methods model properties as attributes of classes.

This section has highlighted some similarities and differences between the three analysis methods reviewed in the first phase of the investigation into traceability of information. For example, they all model the required system as a group of interacting classes. The classes identified by all the methods represent groups of similar objects. The methods also identified a variety of relationships between the classes and between the objects defined by the classes. A major difference between the methods is the way interactions between classes of objects are recorded. It was shown that the RDA approach

models all relationships as either collaborations or inheritance relationships. The Coad and Yourdon approach models relationships as instance connection, message connections, is-part-of or inheritance structures. The OMT method models associations, is-part-of and inheritance structures on the object model and processing dependencies on the functional and dynamic models. It has also been shown that the three methods do not model all the possible relationships between classes and objects. The information modelled consists of the definition of classes of objects involved in the system, various types of relationships between these classes and the objects they represent, and the grouping of these classes into subsystems. Such types of information form the input to the design phase.

3.4 Object oriented languages

This section presents the information gathered during the second phase of the investigation into the traceability of information during system development. This second phase involved identifying the features available in the chosen implementation medium. Such information about the implementation medium is required before the design process can begin because, as stated in section 3.1, one of the functions of design is to map the analysis results onto implementation constructs. The development process under consideration involves implementation using an object oriented programming language. The characteristic features of four such languages are examined in order to identify areas of commonality and difference between them. Four procedural, class based object oriented languages have been selected for this investigation. They are Eiffel v2.3 [2], C++ [33, 34], Oberon-2 [35, 36] and Modula-3 [37]. It should be remembered that this chapter is investigating traceability of information in the context of improving the reusability of software so language features which affect other aspects of reusability of software are taken into account.

The essential features of object oriented languages are identified in section 3.4.1 together with other language features considered essential for producing reusable code. The following sections discuss the provision of the features identified. Section 3.4.2 discusses different aspects of the way in which encapsulation is provided. Section 3.4.3 describes the implementation of inheritance and some of the associated problems. Section 3.4.4 discusses polymorphism and dynamic binding and the implications of their use. The data types which form part of the languages are discussed in section 3.4.5. The information in this section includes the work reported in [46]. The types of constructs which are available for representing different types of information are discussed in section 3.5.

3.4.1 Key features of the languages

A commonly quoted definition of object oriented programming languages is given by Wegner [47]. The features required are:

- encapsulation based upon objects which belong to a class,
- inheritance—a means of deriving new classes, subclasses, by extending existing ones.

Objects, as defined by Atkinson [48], are entities which have a unique identifier and combine data and operations. The operations form part of the object.

Classes can be considered to be implementations of abstract data types [2, 48, 49]. The operations defined by the abstract data type form the interface of the class. A class may also require services from other classes so is more than a simple abstract data type. A class which provides services is called a server class. A class using the services provided by another class is called a client. The relationship between a client class and a server class is known as the client-server relationship. Classes are used

Language	existing class	new class
Eiffel	ancestor	descendant
C++	base	derived
Oberon-2	base type	extended type
Modula-3	supertype or ancestor	subtype or descendant

Table 3.1: Inheritance terminology

as templates from which a set of structurally equivalent objects are produced. Classes therefore define objects which are involved in client-server relationships.

The languages use different terminology when referring to classes and objects. In C++ and Eiffel, the term class is used to mean the language construct used to define abstract data types. The Oberon-2 term for a class is either a record type with procedure variables or type-bound procedures, or a pointer type bound to such a record type. The rationale for using the term 'record' is that programmers can readily understand the concept [50]. Modula-3 uses the term object type for a class. Opaque object types provide the required encapsulation. Future references to Modula-3 objects refer to those generated from opaque object types. All the languages use the term object to mean an instance of a 'class'. The terms class and object are used in the subsequent discussion.

The only form of encapsulation required by object oriented languages is based upon classes and objects. Eiffel is a pure object oriented language and provides encapsulation based upon classes and objects only. The file is the unit of encapsulation. This means that there must be one class per file and one file per class. The other languages, C++, Oberon-2 and Modula-3, are developed from non object oriented procedural languages by the addition of object oriented features. C++ developed as a superset of C. Both Oberon-2 and Modula-3 are developed from Modula-2. As a consequence of their development, these languages allow encapsulation to be based on the file structure of the underlying operating system. This means that in all these languages it is possible to declare more than one class in a file and also to declare functions or procedures which do not form part of a class.

Inheritance provides a means to build new classes as extensions of existing ones. Table 3.4.1 shows the terminology used by the different languages. The Eiffel terminology is used in the following discussion.

As well as providing the above features, the chosen languages provide static type checking. This feature helps to ensure that systems cannot fail due to a run time type error thus adding to the reliability of systems. Static type checking is considered essential in the context of reusable software because only reliable software can be safely reused. The prevention of type errors at run-time improves reliability. The system can also execute more efficiently because there is no overhead due to dynamic type checking. These two features were identified in chapter 2 as important in the production of reusable software components.

There are other similarities between the type conformance and type checking systems of Oberon-2, Eiffel, Modula-3 and C++. These include:

- type conformance based on the inheritance hierarchy.

In simple terms this means that a derived class conforms to its base class. For example, assuming that class **Customer** is derived from class **Person**, class **Customer** conforms to **Person**.

- type conformance governs assignment compatibility.

For example, objects of class **Customer** can be assigned to objects of class **Person**. An object of class **Customer** can be used wherever an object of class **Person** is expected. The extra features

defined by class **Customer** cannot be accessed via the object of class **Person**. Modula-3 also allows assignment to be made where the assignment *may* be legal dynamically. For example, the compiler allows a person object to be statically assigned to a customer object if the **Customer** class is derived from the **Person** class. Dynamic checks are performed to ensure the correct type is assigned.

- type checking as a purely syntactic mechanism.

There is no automatic semantic checking so it is possible to redefine a procedure to perform a totally different function using the same parameter types. However, Eiffel does allow the programmer to force some semantic checking. The keyword `ensure` can be used to implement post conditions. The use of post conditions prevents an `add` feature being redefined to multiply for instance. For example, an **Account** class might be declared with a feature `addFunds` as shown below.

```
Class ACCOUNT
feature
  addFunds (f : MONEY)
      ensure balance = oldbalance + f;
  end;
```

The `addFunds` feature cannot be refined by a descendant class to multiply the amounts instead of adding the value `f` to the balance.

The use of inheritance combined with the assignment rules above results in the possibility that each object can belong to more than one class. In general, the class of an object defines its type so the ability to belong to more than one class gives rise to the possibility that each instance variable can have more than one type. It will belong to its own type and the type of all of its base classes. Instance variables are said to be polymorphic. Each subclass can also provide its own implementation of an operation. For example, a **Person** class may define a `print` method which is overridden by a descendant class, **Customer**. It is possible that an object of type **Person** may have an object of class **Customer** assigned to it. The dynamic type of the variable is not the same as its static type. The result is that the required code may not be known until the `print` call is executed and the actual type of the calling object is known. This requires dynamic binding of code. Dynamic binding and polymorphism are both key concepts of object oriented programming.

There are three key features in object oriented languages. These are encapsulation based upon classes and objects, inheritance and polymorphism allied with dynamic binding. A further important feature of the languages is the types of data structures which are provided because the data structures available affect the information that can be represented. The rest of this section describes the provision of the key features of object oriented languages and the data types provided by the languages.

3.4.2 Encapsulation based upon classes and objects

Encapsulation is the grouping of related information into one place. As Rumbaugh states [28], in object oriented languages, encapsulation is used to mean both the grouping of data with the operations that can be applied to the data and the use of information hiding to control access to the features. Classes are the templates from which objects are produced and are used to provide encapsulation in object-oriented software production. The class is also the basic unit from which object oriented systems are

constructed. The features of a class which can be accessed by other objects form the interface to that class.

The languages differ in the way the parts of an object are named. Oberon-2 refers to the fields and procedures of an object. C++ uses the term member to cover all parts of the object. Modula-3 refers to the data and methods of an object. Eiffel refers to all parts of a class as features. The features can be either attributes or routines. The attributes of a class become the fields of objects. The Eiffel terminology is used for the following discussion which covers four aspects of encapsulation. These are the declaration of classes, the interfaces provided by classes, the mechanisms for providing routines and the mechanisms used to declare and access objects.

- Class declaration

In the Eiffel language, one class is declared in one file but the other languages allow more than one class to be declared in one file. In C++ and Oberon, one class must be implemented in one file but Modula-3 also allows the implementation of one class to be spread over several files.

Eiffel and Oberon-2 classes must be declared with the interface and implementation in one file. The use of one file is said to be quicker for the programmer and easier for the compiler which does not have to check consistency between the interface definition and the implementation modules. Another effect of using one file for both interface and implementation is that the interface can be altered simply by changing the export status of features. This does not encourage the completion of the design of a system before implementation begins. The interface of a module is bound to the implementation which makes it more difficult to change the data structures used to store the objects. Both languages provide tools to extract the interface from the implementation.

The C++ language allows a programmer to declare the interface in a separate file from the implementation of the class. The interface can be declared in a header file which must contain details of the data structure. It is also possible to declare the interface and implementation in one file.

Modula-3 classes, opaque object types, require separate interface and implementation files to be used. The interface file contains the declaration of the public features of the class and declares the class from which it is derived, if applicable. The implementation module defines the features of the class which are not exported and also defines procedures which are assigned to the methods of the class. The separation of interface and implementation allows several implementations of the same interface to be available. The required implementation is linked to the system when the executable code is produced.

- Class interfaces

The parts of a class which are exported form the interface to that class. All the languages under consideration apply the same general rule to govern the export of features. They declare that all parts of the class will be hidden, or private, unless declared specifically as exported. This rule adds to the security of code. (Eiffel v3.0 has changed this rule. The default, in Eiffel v3.0, is that features are available to all classes unless access is explicitly restricted [51]. Future references to Eiffel refer to v2.3.) The details of the export policy in each language are different. It must be remembered that encapsulation is provided by declaring an interface which defines that information which is available to client classes and hides other information as well as implementation details. Access to features which do not form the interface of a class breaks this encapsulation. The class is no longer a black box and fails to provide information hiding which is one of the factors identified as important in permitting reuse of software, see section 2.2.1.

The Oberon-2 policy is the simplest. A class has only one interface. All users of a class are treated equally. There is no special arrangement for a class inheriting from another class. This means that it is impossible for a derived class to break the encapsulation of its parent class. Two grades of export can be specified by the programmer. These are read-only export and read/write export.

Modula-3 allows one class to have several public interfaces by permitting several public super-types of the same class to be declared. However, it is not possible to restrict the use of an interface to any group of classes. A subclass can access any of the features of its base class, thus there is no information hiding between base class and subclasses.

C++ and Eiffel allow classes to provide different interfaces to different classes. These interfaces are created by using a number of facilities.

1. General access by any class

Client classes are allowed access to any features labelled as public in C++, or listed as exported in Eiffel. Eiffel features which are listed as exported can be accessed by classes which are derived from class **ANY**. All Eiffel classes are descended from class **ANY** so gain access to those features exported. Eiffel automatically enforces that exported attributes are read only and routines are executable.

2. Access by some classes only

In C++, classes or functions can be declared as friends of the class being defined. These friend classes or functions can access any feature of the C++ class whether public, protected or private (hidden). Descendants of the class declaring the friendship do not inherit the friends. Friends must be declared at every level in the inheritance hierarchy.

Friends are usually used either to allow list manager type classes to access the parts of the list elements or to output user defined classes. An alternative way to output user defined classes is to declare global functions which overload the input and output operators. The friend relationship can be used to form a tight group of objects to simulate a complex type in the application domain. Another use for the friend mechanism is to increase the performance of the system [52]. This is not a good use because all the classes benefiting from access to the function or type will not be known in advance.

Eiffel allows a programmer to restrict access to exported features. Individual class features can be exported to another class or group of classes and their descendants. Any feature from the named class can access the individual feature. This is known as selective export. Inheritance can be used to change the export status of the features and thus allows more classes to be added to the specified list.

Both these mechanisms are restricted to use when the class is being developed. Unanticipated friends cannot be added except by either re-opening the class or using inheritance. Re-opening a class is possible only if the source code is supplied and may affect existing clients if the re-opened class has been used as a server in an existing system.

3. Descendant access

In Eiffel descendant classes gain access to all features of the parent class. Class invariants are inherited to help prevent the misuse of inheritance by breaking the encapsulation. C++ provides greater control over access to features descendant classes. Descendant classes can access the public or protected fields of a class but not the private fields.

- Implementation of routines

Routines are the means provided for accessing the state of an object or asking the object to perform a computation. Eiffel, Modula-3 and C++ provide one mechanism for implementing routines. These are the routines that are part of the class implementation.

Eiffel and C++ routines are declared in the class implementation which declares the signature of the routine, that is, the parameters, their types and return type, if any. C++ routines may be declared as `virtual`. This is required to allow the redefinition of features in descendant classes and will be explained in the next section.

Modula-3 routines are declared as methods in the class interface. The method declaration defines the parameters required by the method. These methods are implemented by procedures defined in the implementation module. A procedure is assigned to each method. The signature of the procedure implementing the method differs from the signature of the method. The procedure has an extra parameter which is an instance of the type itself or one of its ancestor types. This extra parameter must be the first one in the list of parameters. The other parameters must be the same types as those in the method definition.

Oberon-2 provides three different mechanisms. The mechanisms are type-bound procedures, procedure types and message records. Type-bound procedures are very similar to the routines in Eiffel and C++. Procedure types provide similar functionality to type-bound procedures but are more complex to implement [53]. Message records are not type checked as fully as the other means of implementing routines because the compiler cannot check that a particular message is understood by an object. The lack of static type checking means that message records cannot be shown to produce reliable, reusable software so are not considered to be a viable means of implementing the basic routines provided by an object. Future references to Oberon-2 routines refer to type-bound procedures.

All the languages therefore provide mechanisms for fully type checked routines to be implemented. A routine may require extra information such as the amount of money to be added to an account. This information is passed in the form of parameters to the routine.

- Declaration and access of objects

In order to be able to declare an instance of an object, the declaring class must become a client of the class, the server class, which defines the required object. In Eiffel, this involves the declaration of a variable of the required type and the inclusion of the directory containing the class in the system universe. In both Oberon-2 and Modula-3, the module which declares the class must be imported into the client module. An object of the required class can then be declared by qualifying the module name with the class name, for example, `p: Person.person` where `Person` is the module name and `person` is the class name. A C++ file can become a client of a class by including the file which declares the class in its header.

In Modula-3, objects can only be declared as pointers to the structure not the structure itself. It is possible in the other three languages to provide objects as either pointers or actual instances of the class. In all of the languages, objects declared as pointers have the value `NIL` or `VOID` until an actual object is assigned to it. This can be done either by assigning an existing object to the pointer or calling the function to create a new object. Objects declared as actual instances become part of the class in which they are declared. This means that they are not separate structures and so cannot be part of another object. Figure 3.7 shows the difference between the two forms of object. The mechanisms for providing the different forms of objects vary.

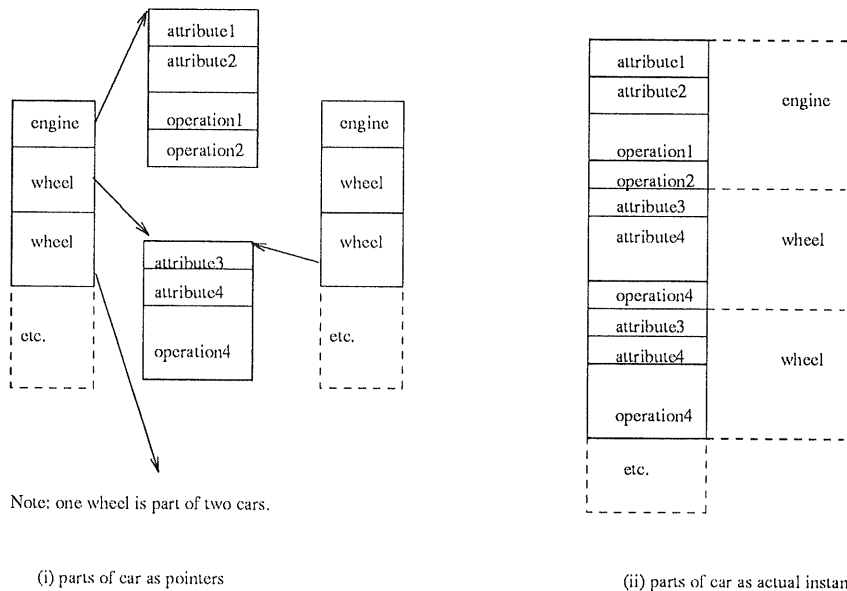


Figure 3.7: Pointers and expanded objects

In Eiffel, either the class definition or the client module controls which type of object is produced. The default behaviour is that the declaration of an instance results in the allocation of a pointer. In order to assign an actual instance to the pointer, a *Create* procedure must be invoked. If an actual instance of a class is required, the client module can declare the instance as *expanded*. For example, an actual instance of type *Person* could be declared as `p : expanded Person`; Alternatively the class declaration could begin with the keyword *expanded*, in which case all instances of the class will automatically be actual objects and not pointers.

In the C++ language, it is the client module which controls the production of objects. The programmer can declare either an instance or a pointer to an instance of a class.

In Oberon-2, a class can be declared either as a record with attached procedures or as a pointer to a record structure with attached procedures. The programmer declares instances as appropriate. The features of objects are accessed by dereferencing the object. In C++, a `→`, for example `object→feature`, is used to dereference a pointer variable and the dot operator, for example `object.feature`, is used for dereferencing an actual object. The other languages all use the dot operator to access features.

In all the languages, it is recommended that objects should be declared as pointers not actual instances of the class. There are two main reasons for this:

1. Not all the objects required in a system have to be declared statically. They can be created dynamically during system execution. The number of each type of object which will be created during the life of the system is not known at compile time making it impossible for the compiler to allocate the required amount of space in memory. Therefore objects, or instance variables, are usually declared as pointers or references, as these have a known size, rather than the structures themselves. Space can then be allocated dynamically as required.
2. The effect of carrying out an assignment depends on whether the variables being assigned are actual instances or pointers [53].

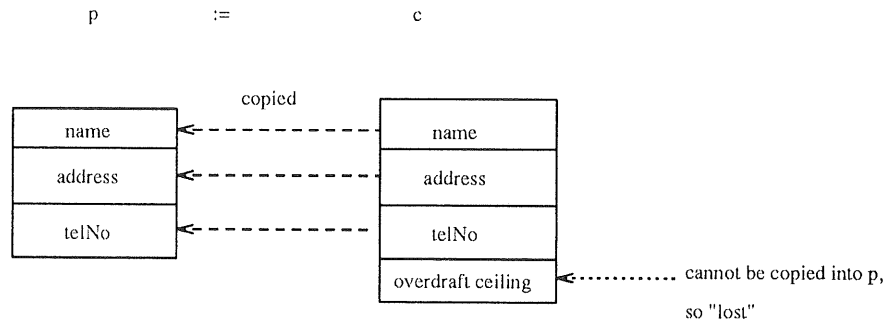


Figure 3.8: Assignment of Records

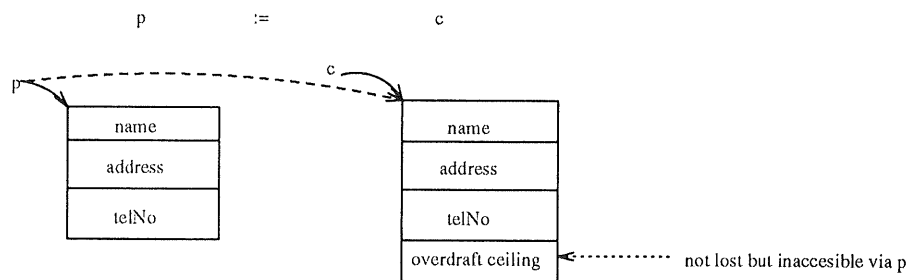


Figure 3.9: Assignment of Pointers

When the structures involved in the assignment, are actual instances, the values of the fields are copied as shown in Figure 3.8.

The variable *c* is an instance of class **Customer** so contains one more field than *p* which is an instance of class **Person**. The extra field cannot be copied and so is 'lost' in the assignment to *p*. The original variable, *c*, still contains the value.

Pointer variables contain the address of the actual structure. It is the address value that is changed by the assignment statement. If the assignment *p* := *c* refers to pointer types, any extra fields are not lost.

This is shown in figure 3.9. The extra fields are not accessible via the base variable because the language has static type checking. The variable *p* is declared to be of type **Person**. This means that its static type is **Person**. Consider the following statement sequence:

```

VAR
  p : Person;
  c : Customer;
  n : REAL;
BEGIN
  ...
  p := c    (*a valid assignment*)
  n := p.overdraftCeiling; (* invalid - type Person does not
                           contain this field*)
END.

```

The sequence is invalid because *p* has a static type **Person** so, during compilation, it does not have a field called `overdraftCeiling`. The assignment statement *p* := *c* results in *p* pointing to a variable of type **Customer** giving *p* the dynamic type **Customer**. Therefore, at run time, the variable *p* has the extra field of its extended type. However, all the

languages provide mechanisms to access these fields. These mechanisms are discussed in section 3.4.4.

Eiffel, Oberon-2 and C++ objects can only be produced as direct instances of the classes in the system. The result is all objects created from a class have the same structure and behaviour. Modula-3 allows different objects created from one class declaration to have different behaviour. This is possible because of the way classes are implemented. It was stated earlier that the implementation module assigns procedures to the method fields of a class. When an instance of a class is created, it is possible to assign different procedures to the method fields and change the behaviour of the object. The object is not considered to be an instance of the class of which it is declared. The object is not the same type as other objects created from the original class declaration. It is said to be an anonymous subtype of the original type.

This description of the encapsulation provided by the languages has shown that there are some areas of commonality and some where there are significant differences. The similarities are that the class constructs define both the structure and behaviour of the objects generated from them. All the languages provide information hiding and allow access to the object only via features declared in the interface. There are differences in the way the classes are declared. Oberon-2 and Eiffel classes are declared and implemented in one file. C++ classes can have separate definition and implementation files. Modula-3 must have separate definition and implementation files. Another difference is in the number of interfaces to a class. Oberon-2 classes have only one interface. Modula-3 classes can have more than one interface which are available to all classes. Eiffel and C++ allow a class to present different interfaces to different classes. A further significant difference is that the Eiffel, C++ and Oberon-2 languages allow objects to be declared as pointers to structures or as actual instances whereas in Modula-3 objects can only be declared as pointers. It is suggested that objects should be declared as pointers to allow dynamic allocation of space and to avoid problems with assignments. Modula-3 has one further feature which is not provided by the other languages. This is the ability to declare objects which have different behaviour from that defined by the class from which they are produced. These objects are not direct instances of any class.

3.4.3 Inheritance

This section describes the second of the key features of object oriented languages, that is inheritance. Inheritance can take various forms. The form of inheritance provided by the four selected languages is described first. The section continues with a description of the changes which can be made to the export status of the features which form the interface of the inherited class. The ability to redefine and redeclare inherited features is then discussed. The next part describes the use of abstract classes. The final part of this section identifies some of the type checking implications when using inheritance.

- Forms of inheritance

Inheritance provides a means to build new classes as extensions of existing ones. In simple terms, the new class has all the features of the existing class plus any new ones it defines. A descendant class can itself be used as a base class for new descendant classes. The inheritance relationship is transitive. Thus, a class is the ancestor of all classes derived from any of its descendants, or descendants of its descendants.

Languages, such as C++ and Eiffel, which allow classes to have more than one direct parent are said to support multiple inheritance. Eiffel permits the same class to be inherited repeatedly but C++ insists that a class can only be named once in its list of inherited classes. The languages

both have mechanisms for resolving any name clashes that occur due to multiple inheritance. Languages, such as Modula-3 and Oberon-2, which restrict classes to one direct parent are said to support single inheritance. Multiple inheritance gives the programmer greater flexibility than single inheritance when properties from existing classes are being combined. However, the resulting classes are more difficult to understand because the definition of the features is distributed between many classes.

The four languages have different policies governing access to the inherited features. This access was described in section 3.4.2.

- Interface changes

It is possible in both C++ and Eiffel to change the export status of inherited features. In C++, class features can be inherited using the keyword `public`. The export status of the inherited features remains the same as in the superclass. If the keyword, `public`, is not specified, another keyword, `private`, is assumed which results in all the features being private in the derived class. Eiffel allows the export status of inherited features to be changed from private to public or public to private. The ability to change the export status of inherited features is provided to increase reusability but means that a subtype relationship may not hold for derived classes in C++ and Eiffel. Oberon-2, and Modula-3 have a strict interpretation of inheritance. A derived class must export all the features exported by the base class. The restrictions on the redefinition of procedures and the export list enable the subtype relationship to hold for all derived classes.

- Redefinition and redeclaration

Before discussing redefinition and redeclaration, it is necessary to define clearly the meaning of the terms. Several definitions of the terms are possible. Meyer [51] defines redefinition as changing the implementation, signature (the formal parameters and result type) or specification of an inherited feature. He defines redeclaration as a more general concept including both redefinition and the implementation of a deferred feature inherited from an abstract class. Böszörményi [54] uses the following definitions. Redefinition is defined as changing the implementation of a function but keeping the signature and the specification the same. Redecclaration is defined as changing the implementation, signature and specification. Böszörményi's definitions are used because they allow the different aspects of the languages to be discussed more easily.

- Redefinition.

Redefinition of inherited features is permitted in all four languages. In Oberon-2, any type-bound procedure may be redefined by providing the new implementation in the module declaring the class. Modula-3 methods are redefined by assigning a different procedure to them in the implementation module.

Eiffel requires that the feature to be redefined is listed in the `Redefine` section of the class declaration. In C++, only functions declared as `virtual` in the base class can be redefined.

Oberon-2, Modula-3 and C++ allow access to the ancestor's version of the function. In Oberon-2, this is achieved by appending an uparrow, `^`, to the procedure call. Modula-3 provides two methods for accessing the ancestor's implementation of a function. Either the predefined function `NARROW` can be used or the method call can be prefixed by the type of the ancestor class. In C++, the ancestor's code is accessed by using the scope resolution operator, `::`, which tells the system to use the version in the named class. In Eiffel, repeated inheritance must be used to obtain two copies. One of the copies is renamed and the other redefined, the renamed copy is used to access the ancestor's version.

Eiffel permits functions to be redefined as attributes in descendant classes. Attributes cannot be redefined as functions.

– Redeclaration

Eiffel, Modula-3 and C++ permit the redeclaration of functions but Oberon-2 does not.

1. Eiffel

In Eiffel, any features being redeclared must be listed in the `Redefine` section of the class declaration. There are strict rules governing changes to the signatures and specification of functions. Any change of type in a signature must be to a type which conforms with the original type, that is to a subtype of the type declared in the original version. This is known as covariance.

Changes to the specification of a routine involve changing the pre- and post-conditions. Pre- and post-conditions are implemented by using the assertion mechanisms provided by the language. The rules governing changes to pre- and post-conditions state that preconditions can be weakened but not strengthened and post conditions must be strengthened not weakened.

Attributes can also be redeclared in Eiffel. The same rule applies that the new type must conform to the old type.

The Eiffel rules governing redeclaration are designed to enable the redeclared feature to be used wherever the original form is expected.

2. C++

C++ provides redeclaration by permitting a restricted form of function overloading. This means that a single name can be used for several different functions within the same scope. These functions must differ in both the parameter types required and return type, if any. The overloading of functions does not involve the virtual mechanism and so does not provide polymorphism. The new function 'hides' the implementation declared by the base class. It is possible for the derived class to access the original implementation by ensuring that the code is obtained from the superclass part of the object. This involves either the use of the scope resolution operator, `::`, or the assignment of the object to a superclass variable. An explanation of redeclaration in C++ is provided by Buchanan [55].

3. Modula-3

In Modula-3, a function can be redeclared by declaring a new method with the same name in the interface of the class. This new method can have a different signature from the ancestor's method. This declaration hides the ancestor's version and appears to be similar to the C++ implementation of redeclaration. The descendant class can access the ancestor's version in the same way as accessing a redefined function; that is, either by calling the predefined function `NARROW` or by prefixing the method call with the type of the ancestor class.

In all four languages it is possible to redefine procedural features to make the functionality applicable to the descendant class. It is also possible, via different mechanisms, to access the version of the feature defined by the base class. Eiffel, C++ and Modula-3 allow any procedural feature to be redeclared with a different signature. Oberon-2 does not permit redeclaration of features. Eiffel is the only one of the languages in which it is possible to maintain the semantics of an operation when it is redefined or redeclared. This is achieved by the use of pre- and post-conditions.

- Abstract classes

An abstract class specifies behaviour but does not provide a complete implementation. The complete implementation must be supplied by the subclasses. The implementation of an abstract class therefore involves writing a subclass which includes the code required to implement the feature. Each language has its own mechanism for providing this facility.

In Oberon-2, abstract behaviour can be specified by declaring type-bound procedures without implementing the function required.

In C++, pure virtual functions are used. These functions are declared with the return value of 0.

In Eiffel, the keyword `deferred` is used to replace the body of the feature. The derived class is responsible for supplying the code to implement the behaviour.

An abstract class in Modula-3 is defined by declaring the class with methods but not assigning a procedure to the method. The method is assigned the value NIL by the compiler.

The languages also vary in the way abstract classes can be used. In Eiffel and C++, the static type checking rules ensure that it is impossible to create instances of deferred classes. However, variables of deferred classes can be declared and have subclass objects assigned to them which allows polymorphic functions to be called. In Oberon-2, it is possible to create instances of abstract classes. The effect of calling a feature which has not been fully implemented is undefined. It is therefore suggested that the predefined procedure HALT is called in such features. Modula-3 also allows objects of abstract classes to be created but provides run-time checks to trap the error if a method with the value NIL, that is without an implementation, is called.

All the languages allow abstract classes to be declared but the amount of support given to prevent the improper use of such classes varies.

- Type checking implications

The assignment rules of the languages rely on there being a supertype-subtype relationship between ancestor classes and descendant classes. This suggests that the interfaces of descendant classes should include the interface of the ancestor class.

Oberon-2 allows only single inheritance and has strict redefinition rules which ensure that subclasses are subtypes. This makes type checking and type conformance a simple concept for the compiler to enforce. In situations where each subtype must be distinguished, for example when retrieving elements from a heterogeneous list, the programmer must use other features of the language to provide dynamic type checking. The features used are type guards and type tests both of which are explained later in this section.

Eiffel allows the developer to make a wider range of changes to inherited classes. In summary, Eiffel allows the types of attributes, parameters and return types to be redeclared providing the new parameters conform to, that is are instances of subclasses of, the original parameters. Eiffel also allows the derived class to hide features which were exported in the base class, thereby improving the possibility of code reuse by using inheritance to implement an is-like relationship such as a Queue is-like a list with limited add and remove functions. This hiding of features and redefinition of the signature of features destroys the subtype relationship between the classes. The assignment rules above may lead to an instance of a Queue being assigned to an instance of a list in which case all the list features will be available. Eiffel version 3 intends to introduce system validity checks in order to prevent such invalid access to features.

C++ has two modes of inheriting, `public` and `private`. The compiler enforces the rule that a class derived by public inheritance cannot hide any of the public fields of its ancestors. This should maintain the subtype relationship between the classes. However, redeclaration of a feature as defined above makes the base version of the feature inaccessible via the derived variable thus destroying the subtype relationship. The use of private inheritance allows features to be hidden. There is no longer a subtype relationship between the two classes. The C++ type checking system ensures that the two types no longer conform.

Modula-3 inheritance is similar to C++ public inheritance. This language also allows redeclaration of features and destroys the subtype relationship.

From the above it can be seen that Oberon-2 has adopted a simplistic approach to typing resulting in a greater burden of type checking being placed on the developer. Eiffel allows flexibility for the developer of new classes but has not yet developed a type system to deal with the problems that have arisen. C++ also provides flexibility but has mechanisms to allow the developer to control the type hierarchy. The result is a conceptually simpler static type checking mechanism in C++ than in Eiffel. Modula-3 provides dynamic as well as static type checking which may deal with any problems caused by the destruction of the subtype relationship.

The differences between superclass-subclass and supertype-subtype relationships are documented, by Cook [56], as causing problems for type checking systems but it is beyond the scope of this thesis to discuss the issue further.

The languages vary in the type of inheritance supported. Eiffel and C++ support multiple inheritance but Modula-3 and Oberon-2 support single inheritance. Eiffel, C++ and Modula-3 allow more changes to be made to the inherited features than Oberon-2. Consequently, they require more complex type checking systems.

3.4.4 Polymorphism and dynamic binding

This section discusses the remaining key features of object oriented languages. These two features are described together because the benefits obtained by the provision of both features increases the usefulness of the concepts. The two concepts are also more important when considering objects which are declared as pointers rather than actual instances so, in this section, an object is assumed to be implemented as a pointer to a variable.

Polymorphism was defined earlier, in section 3.4.1, as the ability of an object to belong to more than one class. This is not the only interpretation of polymorphism. The term can also be applied to the ability to call different code depending on the class of the receiving object. The need for this ability arises because of the binding of operations to classes. This allows any number of classes to define operations of the same name, a *print* procedure for example. Each procedure is uniquely identifiable because it refers to a feature of a class. Many objects can then respond to the same message. If all the objects are unrelated types the correct code can be chosen statically. However, redefinition of class procedures allows subclasses to declare their own version of the code which is suitable for their instances. The instances of a subclass may be assigned to a superclass variable. In this case, the type of the receiving object will not be known until run-time and dynamic binding is required to allow the correct code to be chosen.

Polymorphism provides the ability to assign an instance of a subclass to a superclass variable resulting in the possibility that an object may contain more fields than are accessible via the variable in which it is stored. It is desirable that these extra fields should be made accessible. This requires that a

variable is reassigned from a superclass variable to a subclass variable. This must be carried out in a type safe way if a system's behaviour is to be both predictable and reliable. This requires mechanisms to access the dynamic type of an object to ensure that the correct assignment is made.

This section first describes the way in which polymorphism and dynamic binding are provided and then describes ways to access the dynamic type of an object in order to provide assignment from a superclass instance to a subclass instance.

- The provision of polymorphism and dynamic binding

All the languages allow assignment of an instance of a subclass to a superclass variable. They also allow features to be redefined. The way in which code is bound to procedures varies. In Eiffel, all procedure calls are bound dynamically. The other three languages allow the programmer to choose dynamic or static binding to take advantage of the greater run-time efficiency provided by static binding. Static binding is the normal situation in C++. In order for C++ procedures to be bound dynamically, they must be declared as `virtual`. In Oberon-2, normal procedures are bound statically but type-bound procedures and procedure variables are bound dynamically. Modula-3 methods are all dynamically bound. If static binding is required this can be obtained by declaring a procedure in the interface of the class instead of declaring a method.

- Accessing the dynamic type of an object

This section explains in more detail the requirement to access the dynamic type of an object and then explains the mechanisms available in each of the four languages. The assignment rules allow a variable of a subclass to be assigned to a variable of its superclass. The superclass variable then contains more information than is available to the programmer. For instance, an object of class **Customer** may be assigned to an object of class **Person**. Any extra fields, such as a customer number, can be accessed via a polymorphic procedure such as *print* but not via an object of class **Person**.

A polymorphic procedure redefines a base class routine to provide class dependent behaviour. The redefined routines are then called as required because of the dynamic binding of code. Dynamic binding works well for a routine, such as *print*, which can be defined in a base class. However, dynamic binding cannot be used to access behaviour known to be present in the actual object but not provided when the base class was implemented. For example, a system may contain a list with elements of class **Person**. The list could also be used to store elements of subclasses of **Person** such as **Customer**. It might then be necessary to retrieve an instance of class **Customer** from the list of **Person** instances. This involves assigning a variable which was declared as a superclass variable to a subclass variable. Such an assignment is the opposite way round to normal assignment rules but is permitted in Modula-3 and checked dynamically by the system. Eiffel, C++ and Oberon-2 assignment rules do not normally permit the assignment. However, it is possible to make such assignments in all those languages.

The Eiffel language provides the `reverse assignment attempt`. This is most commonly used when accessing persistent data. The syntax of the call is

```
customer ?= people_list.get(i);
```

where `customer` is of class **Customer**, `people_list` is a list containing variables conforming to class **Person**, `get(i)` is the feature which retrieves the *i*-th element from the list.

If the *i*-th element was the required class it is assigned to the variable *customer*. If not, the value of the variable is void. This mechanism is possible because Eiffel objects contain information about their class.

Eiffel provides two other methods of accessing the dynamic type for use where a large number of types may be found.

1. The library class **INTERNAL** provides a feature *dynamic_type* which returns an **INTEGER**. The class **INTERNAL** is designed to be used as an ancestor class for classes which interface with other languages or database management systems. The use of these features is discouraged because they permit access to the internal representation of an object which breaks the encapsulation.
2. Eiffel classes are descendants of the class **ANY**. This base class provides generally useful facilities. One of the features is *conforms_to* which makes it possible to ascertain information about the dynamic type of an entity at run-time. However, a procedure call such as *p.conforms_to(c)* returning true does not permit access to any extra features present in *c* through variable *p*.

The C++ language does not provide a mechanism which corresponds to the Eiffel reverse assignment attempt because objects in C++ do not contain information about their class. Similar functionality can be obtained by using pointer casting. This uses the type cast mechanism available in C. Assuming the declarations:

```
person *p;  
customer *c;
```

the call *c = (customer *)p* converts a person variable to a customer variable.

This mechanism allows unconstrained changes between any types which can lead to many problems.

In some circumstances, such as retrieving elements from a heterogeneous list, it might be desirable to apply some constraint to the conversions available by determining the dynamic type of the elements. However, as C++ does not provide a procedure to allow the actual type of the object to be ascertained, it must be written by the programmer. One method involves the addition of a tag field to the object structure. This tag field is declared as a static feature which ensures that all objects of the same class contain the same data. The contents of this field can be tested before carrying out the pointer conversion.

Oberon-2 provides type tests and type guards to permit access to fields of a subclass variable through a supertype object. Type tests and type guards are used to check the dynamic type of a pointer variable or a formal VAR parameter of a record type. A type test takes the form

```
p IS Customer
```

and asserts that *p* has the dynamic type *Customer* or an extension of type *Customer*. After this type test, it is possible to access the fields added by type *Customer* or to assign a base type variable to an extended type variable. If, at run time, *p* does not have the dynamic type *Customer*, the result is undefined, so the type test should be part of an IF statement. For example:

```
IF p IS Customer THEN  
  n := p.customerNumber;  
END(*IF*);
```

Type tests used in conjunction with an IF ... THEN ... ELSE statement provide the similar functionality to the Eiffel `reverse assignment attempt`.

A type guard also performs a similar function. The required dynamic type is named in round brackets after the variable name. For example,

```
n := p(Customer).customerNumber;
```

The two dynamic type checking mechanisms described above apply to small regions of the program. A regional type guard is also provided to make the code clearer to read. For example,

```
WITH
    p : Person DO (* some lines of code specific to
                  Person variables*)
  | p : Customer DO (* some lines of code specific
                   to Customer variables*)
ELSE (* code to deal with unexpected type.*)
END
```

However, the form of the type test and type guards allow their use with known subtypes only. The programmer must know the name of all possible types in order to use them in the IF or WITH statements. This mechanism cannot be used to distinguish between unknown subtypes and so restricts the ability to write general code.

Modula-3 provides TYPECODE and TYPECASE functions as well as dynamic type checking of assignments. The TYPECODE is a built-in function which returns an integer which uniquely identifies the type of an object. This can be used to distinguish between different subtypes. The TYPECASE function can be used in the same way as the Oberon-2 regional type guard.

From the above, it can be seen that it is possible to access the dynamic type of objects in all these languages. All the mechanisms work with objects but not with fields of an object. For example, a class **Person** might declare an address field of class string. A Person object could have an instance of a subclass of class string assigned to its address field. This subclass might provide extra string handling facilities. These extra features cannot be accessed through the Person variable.

The four chosen languages provide similar functionality with regard to polymorphism and dynamic binding. They all provide the ability to bind code to procedure calls at run-time. In C++, Modula-3 and Oberon-2, the programmer must choose the correct method of implementation to permit dynamic binding. It is also possible in all the languages to access the dynamic type of an object and allow assignment against the normal assignment rules. Eiffel and Modula-3 provide built-in features to permit this type of access and assignment between a superclass and any of its subclasses. Oberon-2 provides features to access only the features of known subclasses. In C++, it is necessary to add tag fields to objects in order to provide the ability to assign a value from a superclass variable to a subclass variable.

3.4.5 Data types provided

This section is the final part of the investigation into the features provided by the selected object oriented programming languages. It identifies the data types provided by each language. Oberon-2, Eiffel, Modula-3 and C++ all provide simple types as types not objects. The support for generic types, enumerated types and subranges varies.

- Basic types

Basic types, such as character, integer and real are implemented as instances of the type not as pointers to instances of the type. C++ is the only one of the languages not to provide a type Boolean but uses the integers 1 and 0 to represent true and false in Boolean expressions.

- Structured types

Arrays and records are structured types. C++, Modula-3 and Oberon-2 provide arrays as basic language elements whereas Eiffel uses the generic type mechanism. Records are not available in Eiffel because they require access features and therefore become a class. They are available in the other three languages.

- Generic types

Generic types are commonly used to define data structures such as sets, lists and trees. The abstract structure of one of these data types is common to all instances of the type. Many operations on these types, for example *add* and *delete*, do not depend on the types of the actual elements contained. It is these operations that are specified in the generic type. Generic types are implemented as parameterized classes. The parameters for these classes represent types. Actual types, such as lists of integers, are produced by providing actual parameters for the formal parameters. This process is called instantiation.

Eiffel supports genericity. A generic class in this language is compiled using the formal generic parameters. The same code is used for all instantiations of the generic type. It is possible to implement both unconstrained genericity, where any type of class can be used as an actual parameter for the type, and constrained genericity where only specified types or their descendants may be used.

Some implementations of C++ provide generic types by using classes which are called templates. Other implementations provide a package, `generic.h`, which enables generic types to be declared [33].

Modula-3 allows programmers to rename imported modules. This facility has been extended to allow generic types to be implemented. A generic class defines the template for a class in a pair of generic modules. The interface module defines a list of interfaces called the formal imports. The formal imports must be replaced in an instantiation of a generic module by the required module names. When a new instantiation of a generic class is required, a new interface and implementation are produced by supplying actual interface names for the formal imports. New code is generated for each new instantiation. Constrained genericity is not supported but the instantiation will not compile if the actual generic parameters do not provide the required operations.

Oberon provides the type SET but this type can only be used with integers. It is claimed that Oberon-2 can be used to implement other generic types [35]. The suggested implementation of a FIFO queue relies on the “generic” elements being subtypes of the declared type. In the example [35, page 210], the declared type of element is an empty record, of type Node. All possible elements to be included in the list must be descended from this type, Node. The type conformance rules ensure that all the nodes will be compatible with the FIFO queue. Any instance of this type could be completely heterogeneous. In systems where it is desirable to declare more than one data structure, each of which must contain specific types only, the programmer must use type guards. The type guards could be used in one of several ways. Two methods are:

- use a type guard every time an element is added to the queue or removed from it.
- implement the FIFO using type-bound procedures and redefine each procedure to include a type guard.

- Enumerated types

Neither Oberon-2 nor Eiffel provide the facilities required to declare enumerated types. They were omitted from Oberon-2 because “*they defy extensibility over module boundaries*” and had been observed to have “*led to a type explosion that contributed . . . to program verbosity*” [57]. It is possible to implement enumerated types in Oberon-2 by using numeric constants or arrays. Meyer [2] declares that enumerated types are rarely needed and uses integer constants to implement them in Eiffel. C++ provides the *enum* construction tool to implement enumerated types. However, the underlying implementation uses integers which makes it possible to perform meaningless operations such as adding two members of the enumeration. Modula-3 provides enumerations as ordinal types. Binary relational operators are also provided to work on enumerated types. The use of < and > means that it is possible to compare values from enumerations of colours. For example, it is possible to test if red > blue. This kind of comparison will not always be meaningful. Enumerated types are a useful abstraction but are not satisfactorily provided by any of the languages under consideration.

- Subranges

In Eiffel, Oberon-2 and C++, subranges of ordinal types can be implemented by declaring classes to represent the required range of values. The programmer must supply the code to check that values fall within the desired range.

Modula-3 allows the declaration of subranges of any ordinal type. Dynamic type checks detect errors in assignments.

The languages supply a range of data types. It is possible to provide generic types, enumerated types and subranges in all the languages either by using built in constructs or by combining other language features to mimic the required type.

3.5 Information represented in object oriented programming languages

The previous section described the similarities and differences between the constructs and features common to the selected object oriented languages. This section discusses the types of information which are modelled by the chosen object oriented programming languages. The types of relationship which are provided are also discussed. This information is necessary to permit the definition of the end point of the design process.

Object oriented languages provide encapsulation based on the class construct. Classes define all the features of the objects generated from them. Thus, the classes define both the structure of objects and the relationships between objects. It can be seen from the description in section 3.4 that object oriented languages provide two basic relationships between classes. These are the inheritance and the client-server relationships.

- Inheritance

This is a transitive relationship between classes. All instances of a class share the same structure and operations. Modula-3 allows anonymous subclasses to be declared in a program. These subclasses have the same structure but different behaviour from the declared class.

The features of an inherited class can be changed by derived classes. The result is that inheritance can be used to implement an is-a relationship, an is-like relationship or to gain access to features to improve code reuse. Inheritance in object oriented programming is not a mechanism for subtyping.

- Client-server

Classes gain access to the features provided by other classes by declaring instances of the class. The relationship between the two classes is called the client-server relationship. This is a transitive relationship. A class is considered to be client of another class if it, itself, or one of its ancestors is a client of that class. The server class features are visible to the client class but not vice-versa. There are four variations of the client-server relationship. The relationship is defined by the client class but the first three of the variations define relationships between individual objects. The client-server relationship is used to implement relationships between classes and relationships between objects.

1. Simple client

There are two ways in which a class may become a simple client of another class:

- A variable, in the form of a pointer to an instance, of the server class is declared in the client class. This ability is provided by all the languages discussed above. The client class gains access to all the exported features of the server class.
- An instance of a class is required as a formal parameter to a routine.

2. Expanded client

An actual instance, not a pointer to an instance, of the server class is declared in the client class. This relationship is provided by Eiffel, C++ and Oberon-2 but not by Modula-3. Instances of the client class have an instance of the server as part of their structure.

3. Privileged client

This relationship allows a class to provide different interfaces to different classes of clients. It is only provided by C++ and Eiffel. Each has a different mechanism.

Selective export is used in Eiffel. A feature can be made available to specific classes and their descendants that is selectively exported to those classes.

The friend mechanism is used in C++. The declaration of a friend allows the named class or function access to any of the features of the class whether public or private. This contrasts with Eiffel where the selective exports give the privileged class access to specific features only.

4. Generic client

This relationship is provided by C++, Eiffel and Modula-3 but not by Oberon-2. A generic class is declared as requiring a class as a formal parameter. Inside the generic class, an instance of the formal parameter is declared. The generic class is therefore a client of the class of the formally declared parameter. When the class is instantiated with an actual class, the instance of the formal parameter is replaced by an instance of the actual class. The instantiation is then a client of the class of the actual parameter.

In order to use a class, all the classes on which it depends must be included in the system. This means that all the ancestor classes of a class must be present. They are required to provide all the inherited structure and functionality. As well as being dependent on its ancestor classes, a class is dependent on its server classes. This means that a class cannot be compiled unless all its server classes and its ancestors are included in the system. Thus, classes related by either the inheritance or client-server relationships are bound together. Classes which depend on many other classes are more difficult to reuse than classes which depend on fewer classes. This is because all the classes must be identified and included in the new system. The new system may then be larger than necessary and may have reduced performance characteristics.

From the above discussion, it is clear that object oriented systems consist of objects which are completely defined by their class definitions. The classes define both the structure of the objects and the relationships in which each object can be involved. There are two main relationships between classes allowing two types of information to be represented. This contrasts with the eight relationships between classes identified by the analysis process, as shown in section 3.3. Classes and the two relationships between them represent the form the output from the design phase must take.

Fewer types of information can be represented by distinct constructs in languages than are identified by the analysis process. The design process must convert the many types of information identified during analysis into the two types which can be represented in programming languages. The changes which are necessary to translate analysis information into language constructs are described in the next section.

3.6 Design Issues—from analysis to implementation

This section begins the final phase in the investigation into traceability of information from analysis to implementation. It examines one function of design which is to map analysis results on to programming language constructs. Analysis was defined in section 3.2 as the identification of:

- classes and objects including the data associated with and behaviour required of objects,
- relationships between objects and between classes to provide the required functionality,
- subsystems.

This section examines how each of these constructs can be implemented using the constructs and relationships identified in section 3.5. This process identifies the traceability of each of the types of information identified. The effects of the changes required to translate the information identified during analysis into programmable constructs is evaluated in section 3.7.

3.6.1 Classes and Objects

The main concept used during object oriented development is that of the class. The analysis methods and the programming languages view classes as defining groups of objects which combine structure and behaviour. Classes are used as templates for objects. It is not necessary in either part of the development cycle for an object to require storage. Classes can therefore define functional objects. The similarity in the meaning of classes and objects allows designers to carry the constructs through from analysis to implementation. The class and object constructs are therefore traceable from analysis through to the implementation.

3.6.2 Relationships

It is clear from sections 3.3 and 3.5 that more relationships are identified during analysis than are available in programming languages. In addition, the meanings of the relationships provided by programming languages are not necessarily the same as those identified by the same name during analysis. Mechanisms must therefore be devised which map object oriented analysis relationships to object oriented programming language relationships.

This section describes some mechanisms suggested by different authors for implementing the relationships defined in section 3.3. The traceability of each type of information is noted in this section. The effects of the changes required are discussed in section 3.7.

- **is-a**

The is-a relationship represents a specialisation relationship. This relationship is modelled by the inheritance relationship. This relationship is implemented by the inheritance mechanism in programming languages. The analysis methods stress that inheritance should be used where a subtype relationship exists. Inheritance as provided by the languages investigated does not always result in subtyping.

All the analysis methods identified and modelled multiple inheritance. Multiple inheritance can arise for several reasons. For instance, a class may be the base class for more than one hierarchy. These hierarchies may then combine to form one class as shown in figure 3.10 (i) which is taken from the OMT text [28]. Alternatively a class may inherit from two unrelated classes as shown in figure 3.11(i) adapted from the book describing the Fusion technique [58].

As was reported in section 3.4.3, multiple inheritance is not necessarily available in a programming language. It might therefore be necessary to reduce multiple inheritance to single inheritance during the design the system. Several mechanisms are suggested for bringing about this conversion by the authors of the RDA and OMT methods [10, 28].

1. The most relevant ancestor is chosen to become the sole ancestor [28]. In this case, the features from the other branch of the hierarchy are added to each descendant 3.10(ii).
2. The most relevant ancestor class is chosen to become the sole ancestor [10]. The code which would have been inherited from the other classes is copied into the new class.
3. the client-server relationship can be used to represent both branches in the hierarchy. [28]. This is shown in figure 3.11(ii).
4. The most relevant ancestor class is chosen to become the sole ancestor [10, 28]. An instance of each of the other ancestors is declared as an attribute of the new class. This is shown in figure 3.11(iii).

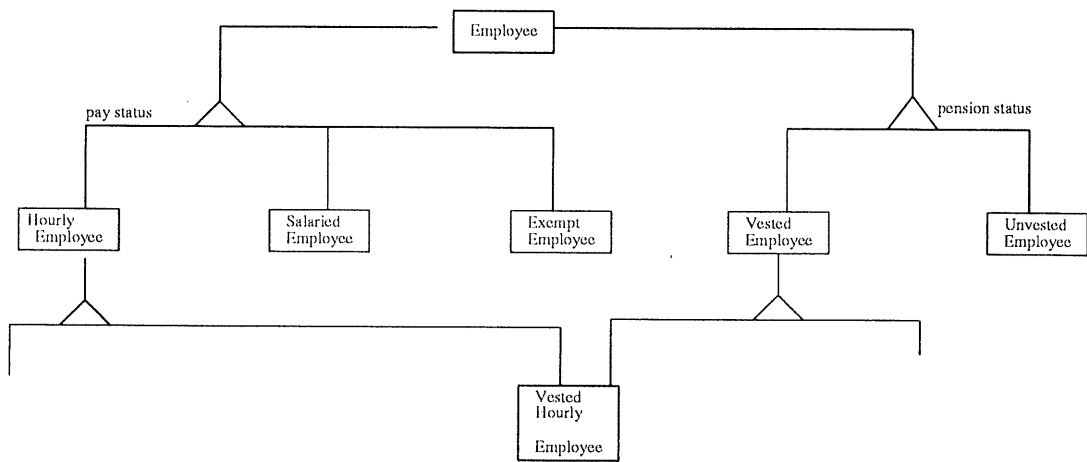
The first two mechanisms involve the duplication of code which could lead to confusion. The third and fourth mechanisms involve converting the is-a relationship to a client-server relationship.

The is-a relationship is common to both analysis and implementation but does not necessarily have the same form or meaning. The traceability of this relationship is questionable.

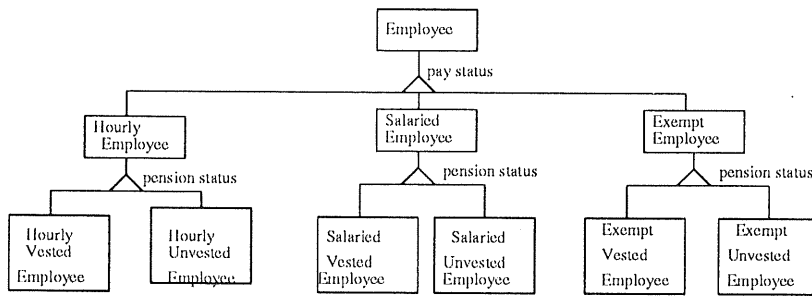
- **consists of**

This relationship is used to define the structure of a complex object.

This relationship is usually implemented by the client-server relationship. The composite class becomes either a simple client or an expanded client of the classes representing its parts. If the



(i)



(ii)

Figure 3.10: Multiple inheritance (a)

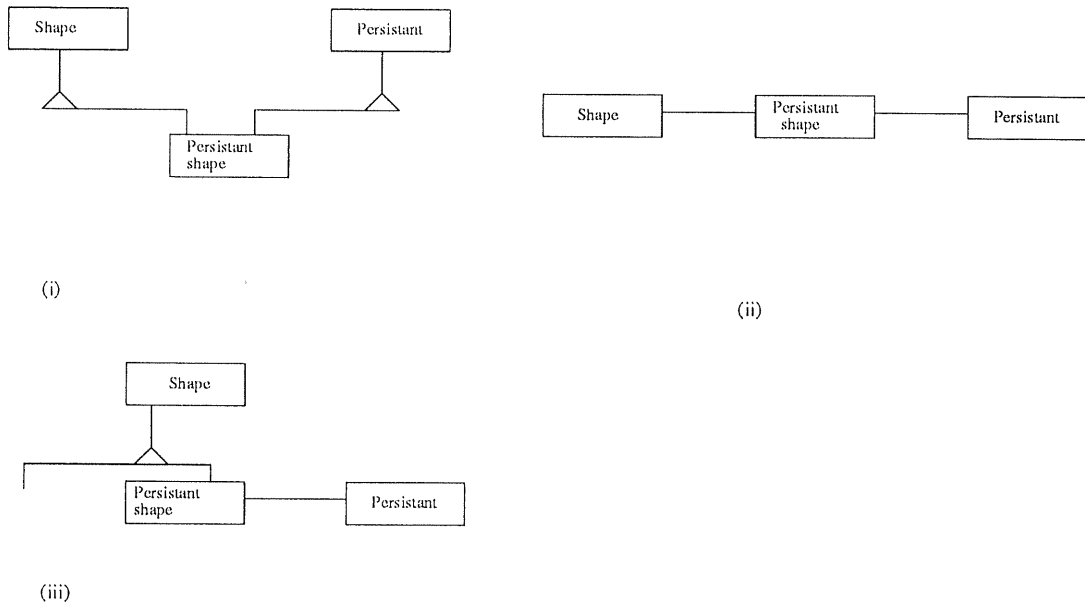


Figure 3.11: Multiple inheritance (b)

composite class consists of more than one instance of another class, the composite class may declare a list (or equivalent) structure of the part. The composite class becomes a generic client as well as a simple or expanded client.

Another way to implement this relationship is to use inheritance. For instance a car is an assembly of engine, body, seats etc.. A car class could be produced by inheriting from the classes representing its component parts. This use of inheritance is discouraged by many development methods. OMT [28] for example, points out that this use of inheritance can lead to incorrect behaviour and recommends the use of inheritance for the is-a relationship only. However, the Eiffel libraries use inheritance to increase code reuse without regard for the is-a relationship. This use of inheritance is possible because Eiffel allows developers to change the export status of features.

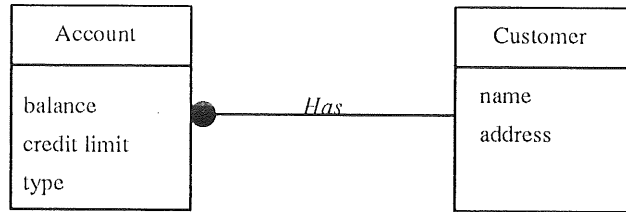
Eiffel and C++ provide a third mechanism for the implementation of the consists of relationship. The selective export feature in Eiffel and the friend concept in C++ can be used to build complex structures.

This relationship is not implemented by a unique construct so is not identifiable in the implementation.

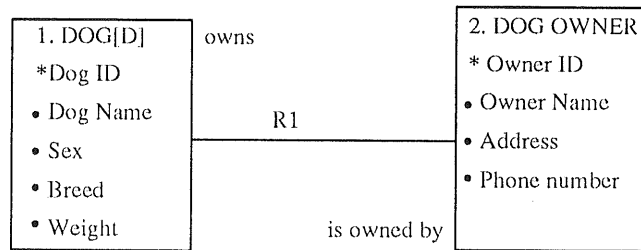
- **contains**

This relationship models the relationship between a container and its contents and is implemented by the client-server relationship. The containing class declares an instance of a generic class, or its equivalent, and so becomes a client of the generic class. The generic class is instantiated with a class representing the required contents and is a client of the contents class. The containing class is therefore a client of both the generic class and the contents class.

Generic classes are also used to contain stored data so this representation of the contains relationship is not unique. The traceability of the contains relationship cannot be guaranteed.



i) part of banking application (ATM) object model from OMT method



ii) Graphical representation of *Dog owner owns dog* and *Dog is owned by dog owner* from Shlaer and Mellor.

Figure 3.12: Sample object models

- **uses**

This relationship models a processing dependency between objects. It is implemented by the client-server relationship. The client class is usually declared as a simple client of the server class which supplies the required functionality. For instance, if class A requires class B to carry out some processing, class A would declare an instance of class B. Alternatively the client class can be named as a privileged client by the server. The uses relationship has been converted into a client-server relationship so the information cannot be traced.

- **conceptual association**

This relationship is defined as a logical relationship between objects. The identity of the objects involved is significant. The associations modelled by the OMT and Shlaer and Mellor [59] methods are two directional as shown by figure 3.12. This relationship is implemented by the client-server relationship. Various methods of conversion [28, 59] can be used depending on factors such as the visibility and the cardinality of the relationship required. A simple banking system, shown in figure 3.13, is used to demonstrate some possible representations of a one-to-one relationship between two objects.

The models shown in figures 3.12 and 3.13 imply that the relationship is important in both directions. This is not necessarily the case. It is possible that an association will be more important in one direction than the other. For example, it could be decided that for the simplified banking application, it is more important to access the person via the account than the other way round.