A Design Method for Object-Oriented Programming

Winnie W. Y. Pun

Thesis submitted for the degree of PhD

Department of Computer Science
University College London
Gower Street, London WC1E 6BT

Abstract

Object-oriented programming has come forth as an important programming paradigm in the 1980's. As more and more people practise object-oriented programming, the subject of how to design and develop a system towards an object-oriented implementation becomes important. There is, therefore, a general demand for a design method which is specially developed for object-oriented programming.

The theme of this thesis is to develop such a design method. The thesis starts off with a comprehensive background on object-oriented programming and how it differs from traditional programming. It also gives a detailed analysis of why existing design methods are inadequate for object-oriented programming. The thesis then presents the design method which has been developed in this research. The different stages of the design method and the various tasks that have to be performed at each stage are discussed thoroughly. The design method also embeds a design description language which allows system designers to communicate with each other during the design phase and this is also talked about in the thesis.

Inheritance is regarded as an important feature found in object-oriented programming. A design method without substantial support for inheritance is considered to be incomplete. Therefore, a mechansim which is called the inheritance factorisation process is developed to assist system designers to construct class hierarchies in object-oriented programming. The mechanism has a formal model which ensures its correctness. The details of the formal model and the issues concerning how to use the mechanism forms a crucial part of this thesis.

To examine the performance of the inheritance factorisation process, a factorisation engine is implemented and experiments have been carried out. To illustrate how the design method is used in system designs, two case studies have been carried out and are presented in this thesis.

The result of this thesis is a design method which guides system designers to organise the design activities towards an object-oriented implementation. It also forms the basis of future work which will lead to a computer-aided software engineering environment for object-oriented programming.

Acknowledgements

I would like to acknowledge my indebtness to my supervisor Russel Winder who assisted, advised and gave me tremendous support throughout this work. Without his guidance and continous encouragement, the thesis would not have been the success it is.

The list of people who I want to thank for their help and advices in the last few years are too numerous to list. I would, however, like to thank the following individuals in particular: John Campbell, Nigel Chapman, Helen Sharp and Graham Roberts for their invaluable comments on the first draft of this thesis; Simon Deal who patiently correct my usage of English in the thesis; Vernon who, probably didn't realised himself, first introduced the term 'object-oriented' into my life; Mildred, my old friend, who never fails giving me support in good and bad times.

Last but not least, I would like to thank Paul, my brother, who makes the past few years more tolerable (it helps to have a brother who is also doing a PhD at the same time). In addition, I have to thank my parents for giving me the opportunity to make this possible.

Of course, the thesis would not have seen the light of day without the financial assistance of the Croucher Foundation and the Oversea Research Scholarship; for this I would like to express my appreciation.

Contents

List of Figures

List of Tables

"There's always something wrong with a new idea. But you have to be careful of people who say there are no new ideas because they're likely to fool you into never getting any new ideas."

~ Marvin Minsky ~

Chapter 1

Introduction

1.1. The Evolution of Software Engineering

The term 'software crisis' refers to a number of problems that are encountered in the development of computer software [Pre87]. These problems range from whether the systems function properly to how to design and maintain the system. The consequence of the 'software crisis' is that a vast amount of money is being spent on developing software. In fact, Boehm has suggested that the cost of software is increasing at a rate of 12% per year and that the worldwide annual software costs will exceed $435 billion by 1995 [Boe75, Boe87].

Although there is not yet a satisfactory solution to the 'software crisis', it is believed that applying sound engineering principles is the best approach to obtain economical software that is reliable and works efficiently. This approach was introduced in the early 1970's and emphasises the use of:

i. comprehensive methods for all phases in software development,

ii. better tools to automate these methods,

iii. better techniques for software quality assurance and

iv. a more formal approaches to control and manage the development process.

As Sommerville [Som89] indicates, "The identification of the software crisis in the late 1960's and the notion that software development is an engineering discipline led to the view that the process of software development is akin to the process which has evolved in other

engineering disciplines. Thus, a model of the software development process which was derived from other engineering activities was suggested". This basic model of the software development process is divided into four stages (See Figure 1.1):

i. the requirement analysis stage,

ii. the system and software design stage,

iii. the implementation stage and

iv. the system testing and maintenance stage.

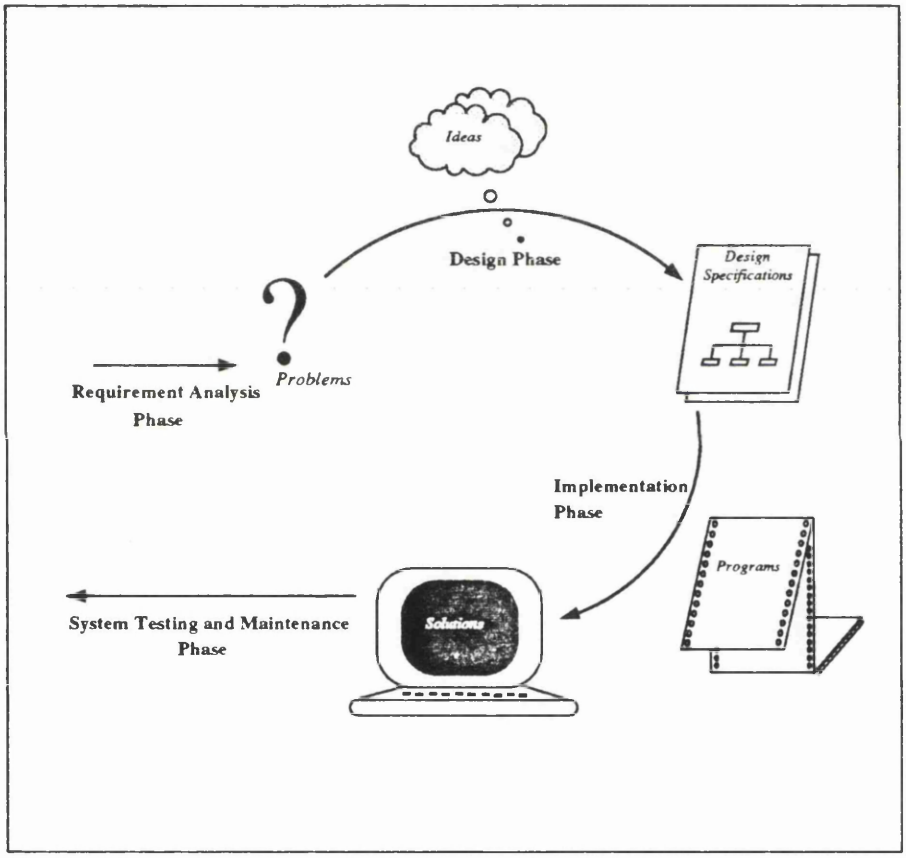


Figure 1.1: The Software Development Cycle

The process of software development can, in fact, be viewed as a series of transformations from one representation to another, each transformation introducing more detail and greater precision to the representation [ABF85]. Software engineers start off by developing an abstract model of the solution in their heads. As the development progresses, the abstract model is transformed into a conceptual model by introducing some primary concepts of the required computational model. Once the conceptual model is set up, more details about the implementation phase, for example, the characteristic of the style of the programming

language chosen for the computational model, are added to the conceptual model which transforms it into the implementation model. The implementation model is a representation which software engineers can take away to construct the system model. This kind of transformation is illustrated in Figure 1.2.

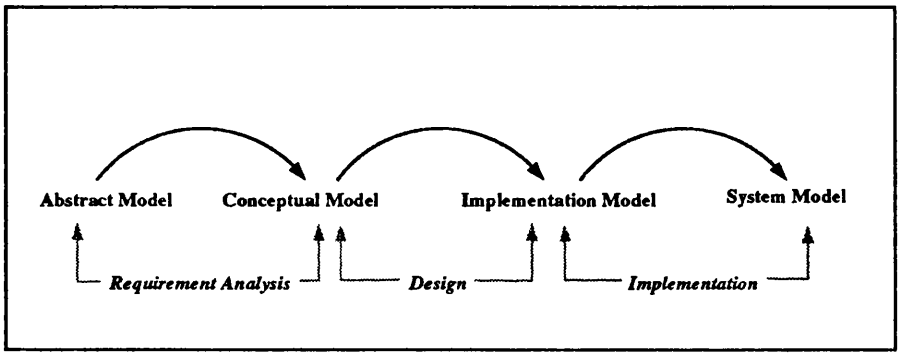


Figure 1.2: The Transformation of Different Represenation

The conceptual model is generally set up at the requirement analysis stage. The transformation process from the conceptual model to the system model is then carried out in the design and the implementation phases. With trivial problems, the transformation from the conceptual model to the implementation model, which takes place in the design phase, is not that significant. Human minds can cope with the thinking in terms of the programming language required to construct the system model. The system model can, therefore, be constructed directly from the conceptual model. However, with complex problems, the design phase becomes a significant stage in the development process and design methods become essential to guide software engineers in the transformation of the conceptual model into the implementation model.

Design methods are responsible for guiding software engineers in the organisation of the design activities towards a particular implementation. A design method should fulfill at least the following two goals [ABF85]:

i. defining the language or graphical notations so that software engineers can express their ideas and conduct a formal communication.

ii. giving rules or hints to guide the transformations smoothly from one stage to another.

The objective of a design method is to assist software engineers in constructing an implementation model. As programming language paradigms actually determine patterns of thought for problem solving [Weg88b], design methods are, therefore, inevitably found to be akin to the programming style chosen for the implementation phase. Thus, in order to obtain a full

picture of the development of design methods, it is necessary to examine the development of programming languages first.

1.1.1. Programming Languages and Design Methods

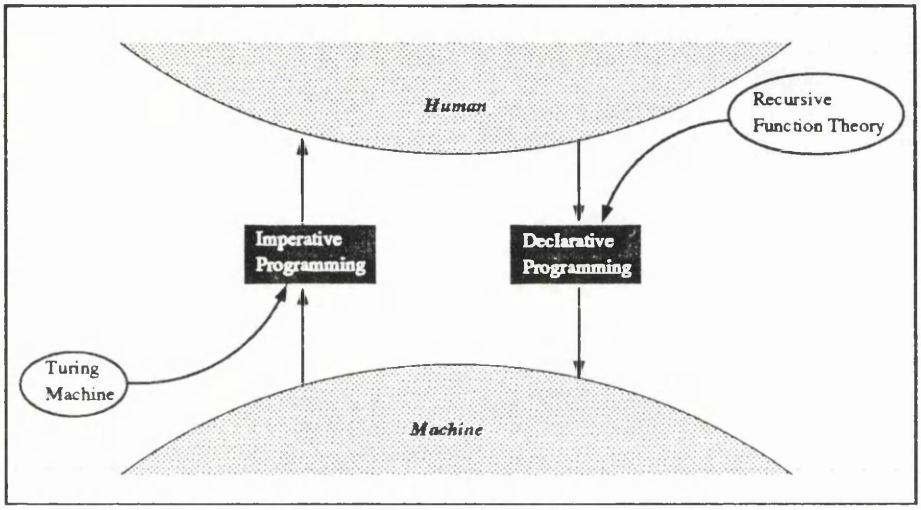


Figure 1.3: The Two Programming Paradigms

Generally speaking, programming languages can be classified into two major paradigms: the imperative and the declarative. The imperative programming paradigm has a machine architecture (e.g., von Neuman machine) as the underlying model of computation. It is concerned with specifying how a computation is performed as a sequence of state transitions. On the other hand, the declarative programming paradigm, such as functional programming and logic programming, has a completely different computational model. It is based on recursive function theory and is concerned with specifying what is to be computed.

As it stands, the declarative programming paradigm probably provides a more natural approach to solving problems. The languages developed are closer to how people solve problems because they are not based on the machine architecture but on the methods people apply to solve problems. Whereas, with the imperative programming paradigm, people are obliged to think in terms of how the machine executes. Although the declarative programming paradigm is a better approach when implementing solutions, to execute programs written in declarative languages efficiently has always been a problem. Since, the imperative programming paradigm is more widely used in industry and, as it is shown later, object-oriented programming fits more comfortably in the imperative programming paradigm, the following discussion about the evolution of programming languages mainly concerns the imperative programming paradigm.

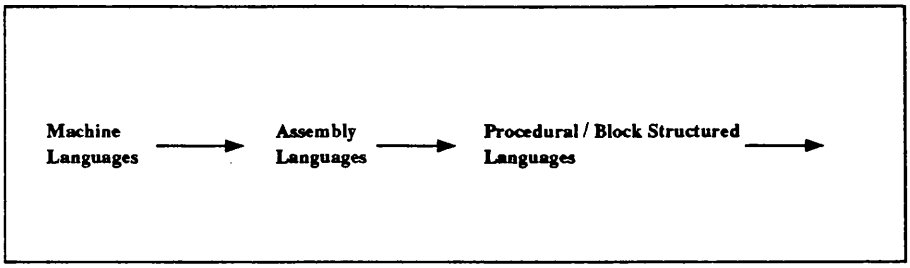


Figure 1.4: The Evolution of Programming Languages

The evolution of programming languages as illustrated in Figure 1.4, starts with machine languages†. A machine language is effectively a sequence of bits such as '01011100'. As one can imagine, to write or read a program in 1's and 0's is a painful task. So in the 1950's, assembly language was developed in which mnemonic names were used to stand for binary codes. The idea of the assembly language as described by Hofstadter [Hof79] is to 'chunk' the individual machine language instructions so that one can write a single symbol, such as 'add' instead of its binary code, say '01011100'. Although a program in assembly language is not very much different from its machine language equivalent as there is still a one-to-one correspondence between assembly language instructions and machine language instructions, assembly language makes the program more legible.

After people had programmed in assembly language for some time, they found that there were some characteristic structures which kept reappearing in program after program. At the same time, assembly language was found to be inadequate for handling complicated programs. Hence, there was a need for higher-level languages which provided the ability to define new higher level entities. This led to the development of procedural and block-structured languages‡. With procedural languages, there is no longer a straight forward one-to-one correspondence between the statements in the procedure and the machine language instructions. A program segment is abstracted in terms of a procedure name and a parameter list. This procedure name and the parameter list will be expanded into the appropriate 'chunks' of instructions by the compiler or the interpreter which are then ready to be executed.

† Some people may consider that computer programming was originally done at an even lower level, i.e., connecting wires to each other, so that the proper operations were 'hard-wired' in.

‡ Block-structured languages are a specialisation of procedure languages in which procedures and data declarations are nested [Weg88b].

This advancement from the machine language to the assembly language to the procedural language can be viewed as the early development of programming languages.

The development of design methods has been closely related to that of the development of programming languages. When most programs were still written in machine language and assembly language, they tended to be small and therefore the need for a design method not that significant. However, when high-level languages were introduced, ways of solving problems and how to design the programs, emerged as a significant part of the software development process, for example, the step-wise refinement technique [Wir71], the top-down design approach etc. These programming and design techniques were later developed into a design philosophy called the structured design method [Pre87]. Structured design methods provide a systematic approach to software development and guide system designers to design solutions towards a procedural implementation.

The fact that the development of design methods is found to be dependent on the development of programming languages is rather disturbing. History shows that instead of designing programming languages and hence machines, that match how people think about solutions to problems, design methods were developed to guide people to think in a way that fits the programming language designed for the machine. This can probably be explained by the fact that people are more adaptable. It is easier for people to adjust their thinking to fit how machines operate than vice versa. Besides, the way in which people think is not as yet fully understood; each individual may solve the same problem differently. It is, therefore, difficult to come up with programming languages that suit every human and very expensive to build machines that execute such languages.

Although the development of programming languages and design methods has not been in the most natural direction, nevertheless the evolution of programming languages has indicated an effort to bridge the gap between how people think and how machines operate by advancing the *abstraction technique* in programming languages [PeW90].

1.1.2. The Road Towards Higher Levels of Abstraction

Abstraction is the key concept for controlling complexity [Gri79, Sha84, Yeh77]. A good abstraction is one that emphasises details that are significant and suppresses those that are immaterial. By having abstraction, one can concentrate on relevant information and ignore the irrelevant in problem solving.

When one examines the evolution of programming languages closely (see Figure 1.5), it is not difficult to notice that this evolution is actually heading towards the improvement of abstraction techniques. The movement from machine language to assembly language

provides a higher abstraction. The binary machine code is abstracted into meaningful names. Again, the development of procedural languages provides an even higher level of abstraction technique than that of the assembly language. With procedural languages, the implementation of the procedure is being abstracted under a name and a list of parameters and programmers can concentrate on the overall organisation of the program.

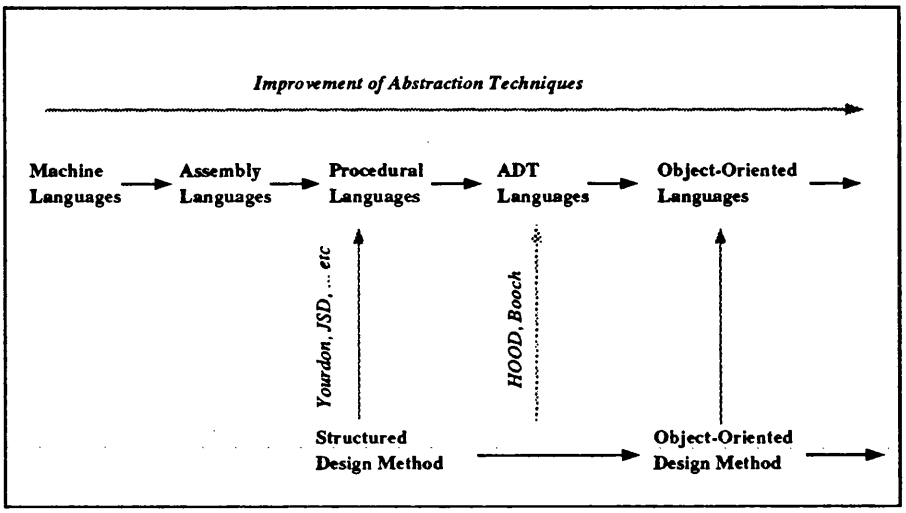


Figure 1.5: The Roads towards higher Abstractions

Although the abstraction techniques offered by these procedural languages, such as FORTRAN and Pascal, are quite considerable when compared to the earlier generation of machine code and assembly languages, when large programs are to be developed and maintained, these programming languages still fall short in handling the complexity of these programs [Pok89]. Therefore, new programming styles with higher levels of abstraction emerged, *abstract data type programming and object-oriented programming.*

1.2. Object-Oriented Programming

Abstract data type programming was developed independently of object-oriented programming. Languages such as CLU [LiG86] and Ada [Bar82] are examples of programming languages that support abstract data types. Abstract data type languages emphasise data abstraction whilst retaining the same control flow mechanisms developed for procedural languages. The essence is to package each data structure with its associative operations into a single construct. The resulting construct contains information necessary to treat the data structure and its operations as a type.

Object-oriented programming represents another step forward in programming development.

Its basic philosophy is to support programming in abstract data types [Boo86, Coo86, Mic88, PLT87, Ren82, Weg88a] and introduce message passing as the control flow. Here, each data structure and its associated operations are packaged in a construct known as a 'class'. Each object is created as an instance of a class and embeds a set of variables that describes the state of the object and a set of operations which can act upon that object. Unlike some abstract data types languages such as Ada, classes are first class citizens and they can be passed as parameters. Besides, in object-oriented programming classes are also used as a classification tool which capture the common attributes of a set of objects in a class definition while ignoring their differences. With this kind of abstraction technique, object-oriented programming brings with it an enormous increase in expressive power [Kay84]. In addition, the encapsulation of state and behaviour within an object allows one to model real world entities that possess state and behaviour [RoG89]. This explains why object-oriented programming is a natural step forward in the evolution of programming development and that it helps to bridge the gap between the problem and the solution spaces.

Although the basic principle of object-oriented programming is to encourage the usage of abstract data types, there are other features found in object-oriented programming that distinguish it from general abstract data type programming [BoS86, Pas86]. These features are:

i. Message Passing

Computation in object-oriented programming is achieved by passing messages to objects. Objects within the system respond to messages by evaluating the method matching the selector in the message and returning objects as a result. With the message passing mechanism, an object-centered approach to programming is further advocated. Instead of active procedures acting on the passive data passed to them, objects are asked to perform operations that have been assigned to them [Coo86, Pas86].

ii. Inheritance

Inheritance allows new classes to be created from existing classes via specialisation. The new class that is created as the specialisation is known as the subclass. The existing class, in this case, is known as the superclass. The subclass is said to inherit all the instance variables, class variables and methods of its superclass. The subclass may add more instance variables, class variables and methods on top of what it inherits. Inheritance enhances the resuability of software and is an important feature of object-oriented programming.

iii. Dynamic Binding

Dynamic Binding is an essential mechanism which supports polymorphism in

object-oriented programming. In conventional procedural programming languages, the names of operators and functions are bound to their operations at compile time. This leads to the result that an unique name is required for an unique operation. With dynamic binding, the name of the operators and functions are bound at run-time. This means that the same name can be used for different operations. For example, instead of using 'drawCircle' for the operation that draw a circle, 'drawSquare' for the operation that draw a square, a generic name 'draw' can be used for both operations. Dynamic binding provides a mechanism which enables the correct piece of code to be executed at run-time.

1.3. An Object-Oriented Design Method

As happened to traditional structured programming in the 1970's, object-oriented design methods have been greatly in demand as object-oriented programming became more popular in the 1980's. The need for an object-oriented design method can be viewed as two-fold.

i. As mentioned earlier, design methods are responsible for assisting system designers to organise design activities towards a particular implementation. Since object-oriented programming introduces new concepts in constructing software, a new design method is needed to help system designers to construct an object-oriented implementation model.

ii. Computer Aided Software Environments (CASE) have become more and more important in contemporary software development process. The environment comprises a set of tools which contributes to the automation of the software development processes. This has enabled dramatic productivity advances in software engineering [Gib89, McC89]. Although substantial tools have been developed to support software development in traditional design methods, very few have been developed for software development in object-oriented programming. In order to develop the suitable CASE tools to aid object-oriented software development, a proper design method for object-oriented programming has to be developed first. Such a design method defines the process to be automated and what sort of tools are required in the environment.

1.4. Thesis Goals

Object-oriented programming is not a revolution but a natural evolution in software development. It is a continuing evolutionary process that has spanned the past three decades.

The process started with writing programs in one-lined instruction statements. Then, structured programming came along which abstracted instruction statements into functions or modules with parameter lists. When abstract data types programming and object-oriented programming came, they advanced the abstraction into a higher level in which functions are abstracted into an object and behaviour is separated from implementations. Such an advent provides a better integration between human cognition and the programming process. It also helps to increase productivity, improve reliability and ease system modifications. Further, object-oriented programming encompasses new programming features such as inheritance to attack areas that conventional programming has been unable to address satisfactorily.

Although object-oriented programming has become more and more popular, so far a proper design method to support the construction of object-oriented software has not emerged. Therefore, the primary goal of this thesis is to develop a design method specially for object-oriented programming. The characteristic of this design method are:

i. It is dedicated for an implementation in object-oriented programming. It contains sufficient guidelines to assist system designers to organise the design activities towards an object-oriented implementation.

ii. It defines the design description language which is used throughout the design phase.

iii. It has substantial support to help system designers handle inheritance which is an important feature found in object-oriented programming.

The usage of this design method is then presented in two case studies.

1.5. Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2 provides a background to the research. It states the definition of 'object-oriented programming' as used in this thesis. It then analyses why existing design methods are inadequate to support systems design in object-oriented programming. The analysis includes both the popular structured design methods and the few newly emerged object-oriented design methods. The results of the investigation highlight the necessity of this research to look for a more appropriate design method for object-oriented programming. It also helps to specify the requirements of such a design method.

Chapter 3 presents the primary framework of the design method developed in this research. The design method is divided into three levels: the conceptual level, the system level and

the specification level. The details of each level and the tasks which are required to be performed at each level are discussed.

The design method embeds an inheritance factorisation process at the system level. This process helps system engineers to construct class hierarchies during the design stage. Chapter 4 discusses the details of this process. The formal model which lies behind the inheritance factorisation process is presented. The application of such a process in system designs is also discussed.

In order to examine the performance of the inheritance factorisation process, the first prototype of the factorisation engine is assembled. Chapter 5 describes the implementation of this prototype. The algorithms which are employed to implement the prototype and their efficiencies are discussed and compared.

Chapter 6 is the evaluation chapter. Both the inheritance factorisation process and the design method are evaluated. Plans for further evaluations are also presented in this chapter.

It is recognised that this research is only the first step towards a complete computer aided software development environment for object-oriented programming. Therefore, Chapter 7 talks about the further work involved in this area.

Chapter 8 is the conclusion of the thesis. It compares the work accomplished in this research with the original aim of this thesis.

"What is object-oriented programming? My guess is that object-oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is."

~ Tim Rentsch ~

Chapter 2

Background

Object-oriented programming increased in popularity during the 1980's. As mentioned in Chapter 1, this new programming paradigm encompasses distinguishing features such as data abstraction, dynamic binding, information hiding, message passing and inheritance which are not found in conventional programming. As Auld [Aul89] explains the reason why object-oriented programming is better than conventional programming style is because "it attacks the area that traditional methods have been unable to address satisfactory: reusability, extensibility and modularity". However, the excellence of object-oriented programming also causes a casual usage of the term 'object-oriented'. Stroustrup [Str88] mentions that the term 'object-oriented' has become a synonym for 'good' and people tend to use it anywhere they can. Thus, it is necessary to define clearly the definition of 'object-oriented' used in this research before any further discussion. This chapter begins with a rigorous definition of 'object-oriented programming'. It analyses the features found in object-oriented programming and highlights the difference between object-oriented programming style and the conventional procedural programming style.

Once the definition of the term 'object-oriented programming' has been stated, one can then embark on the development of a design method for object-oriented programming. Prior to this, an investigation of why existing design methods are inadequate for object-oriented programming is presented. Such an investigation is important because it lays down the foundation of this research. The investigation is divided into two parts. First of all, it examines the compatibility of conventional procedural design methods and object-oriented programming. Then it inspects a few existing 'object-oriented' design methods which claim

to be suitable for object-oriented programming. The results obtained from this investigation show the necessity of this research. It also helps to specify the requirements of the design method that has to be developed.

2.1. What is 'Object-Oriented Programming'?

The notion of 'object' was first introduced in the programming language Simula [Bir73, Nyg86] in the 1960's. The basic idea is to have the programming activity developed around the data structure, 'object'. An object encapsulates a set of variables that describes the state of the object and a set of operations which can act upon the object to change its states. Although the idea of object-oriented programming was around then, it did not take off as an important paradigm until the late 1970's when Smalltalk-80 [Gol83a] appeared.

As more and more people recognised the usefulness of object-oriented programming in software development, the fundamental concepts and definitions of the term 'object-oriented' become misused. In order to impress customers, system designers casually attached the label 'object-oriented' to any software they developed and programming languages they used.

In 1988, Peter Wegner published three papers [Weg87, Weg88a, Weg88b] about the definitions of 'object-oriented'. In these papers, he successfully differentiated between object-oriented programming and conventional programming. Furthermore, he distinguished object-oriented programming from object-based and class-based programming. This has helped to resolve some controversial issues regarding what object-oriented programming is.

According to Wegner, the term 'object-oriented' is defined as:

$$\textit{object-oriented} = \textit{object} + \textit{class} + \textit{class inheritance}$$

This means a programming language is 'object-oriented' if and only if it satisfies the following three criteria:

i. The language supports 'object' as a major language feature.

ii. An object is created as an instance of a class and a class is a template that specifies an interface of operations.

iii. Related classes are grouped into a class hierarchy via inheritance.

After Wegner, a number of other people have attempted to give definitions for the term 'object-oriented'. For instance, Saunders in his survey paper said that an object-oriented programming language should possess: an object creation facility, a message passing capability, a class capability and inheritance [Sau89]. Blair et al. generalised the basic requirements for an object-oriented programming language to: encapsulation, set-based abstraction and

polymorphism [BGM89]. Since, these alternative definitions are either modified or expanded from Wegner's, it is decided that Wegner's concise definition should be applied in this research. The following sections serve to explain these three criteria in more detail.

2.1.1. Object

In the new Collins dictionary [Han87], an object is defined as:

i. a tangible and visible thing,

ii. a thing seen as a focus for feelings, thought, etc, and

iii. an aim or objective.

In a way, such a definition has already captured the essence of an object in object-oriented programming. An object in object-oriented programming is a data entity, a target for messages, and each object implements a goal. An object is the basic unit of modularity in object-oriented programming. It is an abstraction of state and behaviour that consists of:

i. a set of variables which describe the state of that object and

ii. a set of operations which can act upon the object to change the state.

It is strongly believed that the major merit of object-oriented programming lies in its direct correspondence to the real world. The computational model of object-oriented programming is constructed around objects and their interactions amongst each other. Such a computational model is a natural metaphor of the real world [Boo86, Bor85, Gol83b]. This excellence of object-oriented programming has, in fact, been recognised in human factors research on programming languages. Curtis [Cur82] mentions that in a study, it was found that students spent more time with data manipulations and less time with transfer of control. He identifies that this as an important point. Conventional programming languages tend to provide massive control structures with embedded data manipulations. However, the natural human tendency seems to begin with data manipulations and add control structures as a qualification to the action. This implies that a programming language which concentrates on data manipulations, such as object-oriented programming, provides a better integration of programming process with the structure of human cognition. This is almost certainly one of the reasons for the success of object-oriented programming in software development.

The definition of an object being an encapsulation of state and behaviour is further extended in the concurrent programming domain. With concurrent programming [Kah89, ShT83], an object is viewed as a process that contains a mail address and its behaviour. The mail address indicates a buffer in which a sequence of messages is stored. The behaviour is denoted by its actions in response to a communication. Nevertheless, this research concerns the basic

definition of an object instead of its various extensions in different problem domains, hence the definition of an object as an abstraction of state and behaviour is sufficient.

2.1.2. Class

In object-oriented programming, an object is created as an instance of a class. A class is a template that specifies the state and behaviour of the object belonging to that class. Hence, a class consists of:

i. a set of variables, and

ii. a set of operations.

At a first glance, the definition of an object is very similar to that of a class. The main difference between the two actually lies in the fact that a class is a static template which defines the structure of a group of similar objects, a class exists in program text. Whereas an object is an active entity and its states are updated at run-time [Mey88].

Of course, this is not the only possible approach to look at classes and objects. In Smalltalk [Gol83a], classes are viewed as objects themselves. The main advantage of treating classes as objects is that it makes it possible to define class routines which can be applied to all classes, rather than to all the instances of a given class as standard features do. Further, there are discussions about 'classless' object-oriented programming languages such as Self [Hew77, UnS87] and Actor [Agh83]. In a 'classless' language, an object is created by copying an existing object, i.e., prototyping. The argument for a 'classless' object-oriented programming language is that it is simpler. It does not require object-oriented programmers to grasp two concepts namely the 'is an instance of' relationship and the 'is a kind of' relationship which are typical for a 'class-based' object-oriented language, in constructing software.

However, the view taken in this research is that the concept of 'class' is a nice structuring technique which helps system designers to understand the system better. It also reduces the search space when system designers try to locate a particular object in the system. Besides, if every object belongs to a class, it guarantees that all objects can be treated as first class items. With this, objects are allowed to be passed as parameters, a useful programming feature [Weg88b]. Moreover, it is important to distinguish the description of an object, i.e., the class, and the object itself. This is especially true when inheritance is concerned because with class inheritance, one inherits the structure or the description of an object rather than its value.

2.1.3. Class Inheritance

Inheritance is an important programming issue introduced by object-oriented programming. Inheritance is always associated with the specialisation/generalisation relationship between two classes. Its basic definition in object-oriented programming is analogous to the usual meaning which is familiar to everybody. For example, if classA and classB share an 'IS-A' relationship and classA is more general than classB, i.e. classB needs a larger set of attributes to describe its behaviour, then classA is defined as the superclass of classB and attributes defined in classA can be used by classB through inheritance.

Inheritance can be single or multiple. In the case of single inheritance, a subclass has only one superclass, hence the class hierarchy is a tree form. In the case of multiple inheritance, a subclass is allowed to have several superclasses which leads to a directed graph hierarchy structure. Although, it is said that multiple inheritance is extremely useful and provides more flexibility to object-oriented programmers [Car84, Mey88, Str88], it is also believed by some people that multiple inheritance can lead to very complicated systems which are difficult to comprehend [Bol82].

Danforth and Tomlinson say [DaT88], "a primary motivation for the use of inheritance in programming is that it provides both a specification structuring mechanism and a means of reusing specification that is based on common sense notions that are natural to our way of thinking". Indeed, the advantages of having inheritance are basically two-fold:

i. Reusability of Software
   Inheritance enhances the reusability of software which is an important aspect of software engineering. Inheritance in object-oriented programming brings along two kinds of reusability, 'reusing code' and 'reusing specification' [Joh88].

ii. Classification Technique
   Inheritance acts as a classification technique which groups related classes together in a hierarchical structure. This enhances the readability and understandability of a system.

Although inheritance is such a useful feature, it is also a problematic issue. System engineers have encountered extensive difficulties in constructing class hierarchies [Joh88]. However, most of the difficulties evolved around the basic questions of how to use inheritance and what exactly should be inherited. The following discussion gives two different usages of inheritance. It also highlights two different school of thoughts concerning inheritance.

i. Non-Strict Inheritance
   Non-strict inheritance is also known as implementation inheritance or incidental inheritance [Sak89]. Generally speaking, non-strict inheritance implies code shar-

ing. Its essence is to reuse as much of the existing implementation as possible. Non-strict inheritance is usually found in weakly-typed object-oriented languages such as Smalltalk. In these languages, the notion of types is implicit and is always embeded in the notion of class. The term 'class', as mentioned before, denotes the template for a group of objects with common properties. These templates use symbolic labels to specify the state and the behaviour of a particular class of objects. Hence, a label either refers to a storage space or a piece of code. In the case of non-strict inheritance, system designers/programmers determine a superclass/subclass relationship by:

- applying their intuition to the two classes to decide whether an obvious 'IS-A' relationship exists. For example, the class 'car' is naturally a subclass of the class 'vehicle'.
- checking whether there exist some properties in the potential superclass that can be used by the subclass.

Since there is no type-checking mechanism in a weakly-typed object-oriented language to check the formal definition of a label, a more casual attitude to class hierarchies construction is allowed. For example, although it is required that there must be an 'IS-A' relationship between two classes for a superclass/subclass relationship to be established, there is no penalty for system designers/programmers that violate this rule. Therefore, a reasonable and meaningful class hierarchy construction depends very much on the self discipline of the system designers. In the extreme, one can put two conceptually unrelated classes in the same hierarchy provided they share some common attributes. For example, in the Smalltalk-80 system classes, the class 'semaphor' is created as a subclass of 'linked-list'. Conceptually, there is no obvious relation between 'semaphore' and 'linked-list' which may lead people to think that they should be put together. One suspects that the reason for doing this is because the class 'semaphore' can be implemented as a linked-list. This certainly breaks down the possibility of using inheritance as a classification technique. At the same time, it leads to the formation of a class hierarchy which is conceptually confusing. Furthermore, if multiple inheritance is applied, one can get a very unstructured network which is impossible to comprehend.

ii. Strict Inheritance

Strict inheritance is also known as specification/behaviour inheritance, essential inheritance and subtyping [Sak89]. Strict inheritance not only inherits the labels and the implementations associated with the labels but also the formal specification of these labels. The term 'formal specification' refers to the abstract data type

specification which involves the signature and the semantic meaning (axioms) of the operations in a class. With strict inheritance, a subclass can be derived from a superclass if and only if:

- All operations defined in the superclass mean something to the subclass, i.e. no redundant operations may be inherited.
- Operations which share the same name, types of input and output arguments must do the same thing, i.e. operations which share the same name must share the same semantics. For example, a 'stack' and a 'queue' may have the same operations such as 'add' to add an element and 'delete' to remove an element. However, a stack applies a 'last in first out' policy whereas a queue applies 'first in first out' policy. The operation 'delete' again employs different methods in the implementation. Thus, according to the definition of strict inheritance, 'stack' and 'queue' are not directly related.

As one can see, this kind of inheritance imposes a very strict discipline on building class hierarchies. A subclass must inherit both the syntactic and the formal semantic aspects of its superclass. As Meyer says [Mey88], this type of complete, non-ambiguous, precise manner of using inheritance can only be achieved by specifying the interface of a class in formal specification languages [GoM82].

However, strict inheritance can be partially fulfilled in strongly typed object-oriented languages such as Eiffel [Mey88] and Solve [RWW88]. Here, system engineers not only have to deal with a superclass/subclass relationship but also the supertype/subtype relationship in constructing class hierarchies. The term 'type' denotes the signature of an operation. The language provides type-checking facility at compile time. Therefore, a strongly-typed object-oriented language prohibits system designers/programmers from constructing meaningless class hierarchies, at least to a certain extent†.

Another mechanism which serves the same purpose as inheritance is delegation [Lie86, Weg87]. Delegation is always found in 'classless' languages. It is a mechanism that allows objects to delegate properties to one or more 'successors'. Since in this thesis, the term 'object-oriented' implies object-based, class-based and class inheritance, the fact that delegation is found in 'classless' languages makes it inappropriate to be discussed in this thesis.

† Besides type checking, Eiffel also allows programmers to use assertion statements to guard the semantic compatibility of the operations.

2.1.4. A Language that supports Object-Oriented Programming

The above discussion has stated the definition of the term 'object-oriented' as used in this research. Of course, object-oriented programming also contains other features such as dynamic binding and message passing as mentioned in Chapter 1. However, these two features are more to do with implementation and language issues rather than design. Thus, this research regards *object*, *class* and *inheritance* are the three basic criteria for object-oriented programming.

Though the term 'object-oriented' is now defined properly, it is still insufficient to give a clear idea as to what an object-oriented programming style is. To most system engineers, object-oriented software has to be implemented using one of the orthogonal object-oriented languages such as Smalltalk, C++ or Eiffel. On the other hand, there are programmers who claim that they can perform object-oriented programming in conventional structured programming languages such as C, Ada and Pascal [JaK87]. To these programmers, they have to exhibit extra effort or skill to write programs which are object-oriented. They have to write extra libraries to attain the object, class concepts and inheritance. The main question here is, 'how does one classify a language that supports object-oriented programming?'. The answer to this question is well-stated in Stroustrup's paper [Str88], "a language supports a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely enables programs to use the techniques".

Therefore, in this thesis, the term 'object-oriented programming' refers to a technique which allows programming in object/class concepts and reuse software via inheritance. Object-oriented programming has to be supported by an orthogonal object-oriented languages which comprises of facilities and constructs that help programmers to achieve this technique with ease†.

2.2. Designing Object-Oriented Software

Now that the definition of object-oriented programming is stated, one can investigate how software can be constructed in object-oriented programming.

The construction of reliable software depends a lot on using a proper design method. A design method gives guidelines for system designers to organise their design activities towards a

† There is on-going research concerns the usability of a programming language such as [Gre89].

particular implementation model. At present, most of the popular design methods are of procedural style. They target software developments which use procedural programming languages such as Pascal and COBOL. Although some people are currently adopting these methods in object-oriented programming, it is not a good practice. In order to justify this claim, the first part of this section analyses why conventional structured design methods are inadequate for object-oriented programming. As it stands, there are a wide range of conventional design methods around and they can be basically divided into two types: the data flow and the data structure design methods. Hence, the analysis mainly concerns these two types of structured design methods. In additional to this, a few people have suggested that entity-relationship modelling is useful in object-oriented programming [Pow87]. Therefore, there is a need to look into the possibility of using entity-relationship modelling in object-oriented programming.

Apart from conventional design methods, a few so called 'object-oriented' design methods have emerged recently. The main development includes Booch's object-oriented development [Boo86], Robinson's Hierarchical Object-Oriented Design (HOOD) [Rob89], Mittermeir's object-oriented software design [Mit86] and Jacobson's ObjectOry [Jac87]. These design methods are also examined in detail in this section to see whether they support object-oriented development to a satisfactory level.

Hence, the rest of this chapter serves as a detailed survey on whether these existing design methods are adequate for object-oriented programming.

2.2.1. Data Flow Design Methods

Stevens and Constantine were probably the earliest proposers of using data flow design in designing a system [Pre87] but it was Yourdon and Myers who widely propagated the method [YoC75]. Data flow design methods use 'functional decomposition' as the backbone but also focus on the movement of data in a system. It is widely used in data-processing.

Data flow design methods consist of a set of graphical notations which help to develop a structured functional specification from an user requirement, for example, data flow diagrams, structured English mini specifications and structure charts. The first step of the data flow design method is to use data flow diagrams to model the movement of data through a particular system. These data flow diagrams are later transformed to corresponding program structures in the form of structure charts. There are data dictionaries which work along with the data flow diagrams to give precise definitions of the terms used in the data flow diagrams.

The best way to find out whether a particular design method can be used in object-oriented programming is to actually design a system in the design method and implement it in

an object-oriented language. However, to carry out such investigations for each design method mentioned in this chapter is time consuming. At the beginning of this research, use was made of the fact that 35 second year Computer Science undergraduates at UCL have to design a system using Yourdon's design method and implemented in C++ as their Software Engineering coursework. Therefore, their courseworks were conveniently used in this research to analyse whether data flow design methods are adequate for object-oriented programming. When examining their courseworks, attention was paid to any significant effects cast from the design phase onto the implementation phase.

As a result, two observations were obtained from examining the students' courseworks:

i. The coding of most students showed that they were actually writing procedural programs in an object-oriented language, C++. They had not exploited the use of classes as data abstraction technique and inheritance as a means of achieving software reuse.

ii. A few students' implementations bore some kinds of object-orientation. However, their design specifications showed no direct correspondence to the decisions they made for the implementation phase.

Therefore, it seems that data flow design methods are not suitable for object-oriented programming†. The main reasons for this inadequacies are:

i. The basic concept of data flow design methods is to model a system in terms of data and transformation processes. It emphasises what input data and output data exist in a system and how the input data is transformed into the output data, i.e., identifying the required transformation processes in the system. This is very different from the basic model found in object-oriented programming. In object-oriented programming, one is concerned with the set of objects and the interactions amongst these objects in a system. Data flow design methods do not support the identification of objects and interactions of the system. There is no guidelines about how to construct an object, classifying an object and the concept of inheritance does not exist.

ii. Furthermore, data flow design methods are targeted at a procedural implementation in which modules are generated around operations; data structures are distributed between resulting routines. However, in object-oriented programming the reverse occurs and the emphasis is on data structures; modularisations and operations are generated around data structures.

† A discussion of this analysis on whether data flow design is adequate for object-oriented programing can be found in [PuW89a].

Such an analysis of the students' coursework not only has shown that data flow design methods are inadequate for object-oriented programming but also has highlighted the issues one has to pay attention to when developing a design method for object-oriented programming, for example, the identification and the construction of objects, the identification of the interactions between objects. These are very useful in setting up the specification for the design method for object-oriented programming.

2.2.2. Data Structure Design Methods

Data structure design methods are said to be useful for systems with well understood data structures [RPT84]. They produce a program structure by designing input/output data structures. A strong representation of this type of design method is Jackson System Design (JSD) [Jac83].

JSD is a design method that covers system development from definition through design into the production phase [BiO85]. Unlike data flow design methods, it does not start with functional requirements but concentrates on modelling the real world. The development steps of JSD can be classified into three stages: the modelling stage, the network stage and the implementation stage.

The modelling stage is to specify the model of the real world. It further subdivides into two steps, the entity-action and the entity-structure steps. In the entity-action step, all the entities and actions involved in a system are identified. In JSD terminology, an entity is characterised by the action which it performs or suffers. An action is always performed or suffered by one or more entities. An action must take place in the real world at a specific instant of time. The action not only occurs in the system itself but also in reality. The entity-structure step specifies how the actions are ordered within an entity. It defines the ordering of actions that describes the life cycle of the entity. This step also contains a set of graphical notations which supports the three classical algorithm/programming constructs: iteration, sequence and selection.

In JSD, the entity structures constructed in the first stage are regarded as model processes. The network stage concerns how to connect new processes to these model processes to form a network. New processes can be added to model processes either by state vector connection or data stream connection. This stage is further subdivided into three steps:

i. the initial model step which simulates the real world in terms of communicating model processes.

ii. the simulation of the real world resulting from the above steps provides the basis on which the system functions are to be specified. In this step, the developer inserts

the required system functions into the model in the form of function processes.

iii. system timing step which specifies the timing constraints in the system for use in the next step.

The implementation stage of JSD emphasises two important issues:

i. how to run the processes that comprise the specification?

ii. how to store the data that they contain?

An important point in the JSD design method is that it conceptually assumes that every entity runs on an individual processor. For example, each book in a library is a process that runs on a processor. Of course, this is not yet feasible. Hence, the main theme of the implementation stage is to transform such a design into a form that can be realised on the target programming language, operating system and computer.

The main reasons why JSD is said to be a potential design method for object-oriented programming are:

i. its entity-action step is about identifying entities and actions involved in a system and this is analogous to the step concerns identifying objects and interactions as found in object-oriented programming,

ii. its network stage concerns how to connect new processes to model processes to form a network and this can be viewed as identifying the interactions amongst objects in object-oriented programming.

However, when one examines the method more closely, JSD is not a good candidate for object-oriented programming as it seems to be. Firstly, although both JSD and object-oriented programming emphasises identifying entities/objects and actions within a system, there is a difference between the term 'entities' used in JSD and the term 'objects' used in object-oriented programming. According to JSD, an entity has to contain an ordered set of actions. In other words, an entity in JSD is just like a process which is defined as a sequence of actions. But in object-oriented programming, an object is defined as a set of variables which describes its state and a set of operations which describes its behaviour. The order of the operations is not required. Although, there are papers which mentioned that the object-oriented paradigm is similar to the process-paradigm and that one can be incorporated in the other [Str86a], it is strongly felt that to treat an object as a process imposes unnecessary constraints on the user. Moreover, object-oriented programming emphasises reuse. One way to achieve this is to design the object as general as possible so that its properties can be reused through inheritance. If one has to define the order of operations for an object, the object becomes specific to a particular application and difficult to reuse. Therefore, it seems

that JSD is much more of a 'modelling approach' than it is an 'object-oriented approach'. It emphasises creating time-based models of real world entities and models of their 'real world' interactions. Whereas, the objects found in object-oriented programming should not be designed based on their life cycles within a particular application.

2.2.3. Transformations of Conventional Design Methods

Since conventional structured design methods cannot be used directly in object-oriented programming, a few people have proposed some transformation techniques to convert specifications which are generated from conventional design methods to be used in object-oriented programming. For example, Alabiso [Ala88] has presented a transformation for data flow analysis models to object-oriented designs. Ward [War89] has introduced a method to integrate object-orientation with structured analysis and design. Poo et al. [PLK89] and Elizabeth et al. [EHZ89] have investigated how to modify JSD to be used in specifying object-oriented systems. The reasons for modifying conventional design methods to be used in object-oriented programming are two-fold:

i. Most system designers have been using these conventional design methods for a while. They are familiar with these design methods and may be unwilling to adopt a new design method.

ii. A popular and adequate design method for object-oriented programming has not yet emerged.

These people believe that changing the notations of the specifications in existing conventional design methods or re-interpreting a design specification in an object-oriented manner is sufficient to obtain an object-oriented design. This belief is analogous to the one which said that object-oriented programs can be written in procedural languages such as C and Pascal [JaK87]. As mentioned earlier, to write object-oriented programs in procedural languages may be possible but the extra efforts and skill required from the programmers are tremendous. Likewise, in order to perform a sensible transformation between the two different paradigms, the system designer has to be an experienced user of the structured design method, to be completely familiar with object-oriented programming, and to fully understand how the transformation works. Besides, just carrying out a transformation on a structured design specification to an object-oriented design specification is definitely not enough to construct object-oriented software successfully. Design has a lot to do with thinking and organising. To acquire a precise implementation model, the designer has to be aware of object-orientation in the early stages of the software development.

2.2.4. Entity Relationship Modelling

Entity-relationship modelling [Che76, Eas86] is a data modelling technique. It is used in the analysis phase of the software development cycle. It is complementary to methods such as Yourdon, JSD or SADT. Methods like data-flow design provide a global analysis for the system with emphasis on the data flows in to and out of a process. Entity-relationship modelling, however, concentrates on analysing what is in the data store. The technique, in principle, models a system using three building blocks: entities, relationships and attributes.

Entity-relationship modelling concentrates on the fundamental data items of a system. It analyses how these items are related to each other and examines the properties they possess. This corresponds to certain characteristics of object-oriented programming. Both entity-relationship modelling and object-oriented programming emphasise the identification and interaction of objects within a system. Furthermore, in object-oriented programming, the term 'object' is defined as a distinct element and resembles the 'entity' in entity-relationship modelling.

With respect to the relationship issue, there are three different kinds of 'relationships' found in object-oriented programming:

i. Inheritance

   This kind of relationship is found between two distinct classes. For example, if 'bus' is defined to be the subclass of 'vehicle' then the object of the class 'bus' will inherit all the properties of class 'vehicle'.

ii. Instantiation

   This kind of relationship is found between objects and classes. For example, Bus No.27 can be defined as an instance of the 'bus' class.

iii. Interaction

   This kind of relationship is found between objects and objects. It is normally denoted when one object sends messages to another object.

In entity-relationship modelling, there is only one kind of relationship and that is the association between entities. For example, 'attend' might be a relationship set between the entity sets 'Student' and 'Course'. This relationship is analogous to the 'interaction' relationship found between two objects in object-oriented programming. As mentioned earlier, the 'interaction' relationship found in object-oriented programming concerns message passing. Thus, the next step is to see whether the relationship in entity-relationship modelling gives an insight to the kind of messages that should be included in a certain object.

Consider an object-oriented programmer using the entity-relationship modelling technique

to analyse a system that dealt with school records. One may end up with diagrams such as that shown in Figure 2.1. To interpret this diagram in an object-oriented way, one may feel that the 'School' and the 'Teacher' are two distinct classes. The relationship 'employ' tells us that one or more messages† concerning this relationship should be implemented. For example, one could include a message 'employed_by' in the 'Teacher' class. By sending this message to a particular teacher e.g., 'Teacher_A employed_by:', one will receive the school which Teacher_A teaches at. It is also valid to include a message 'employed_by' which takes an argument, e.g., 'Teacher_A employed_by: School_A'. This message returns a boolean value which indicates whether Teacher_A is employed by School_A or not. At the same time, it appears that it is perfectly correct for a programmer to choose to implement the related message in the class 'School'. For example, one could implement a message 'employ' in the 'School' class so that 'School_A employ:' returns a list of teachers who are employed at that school.

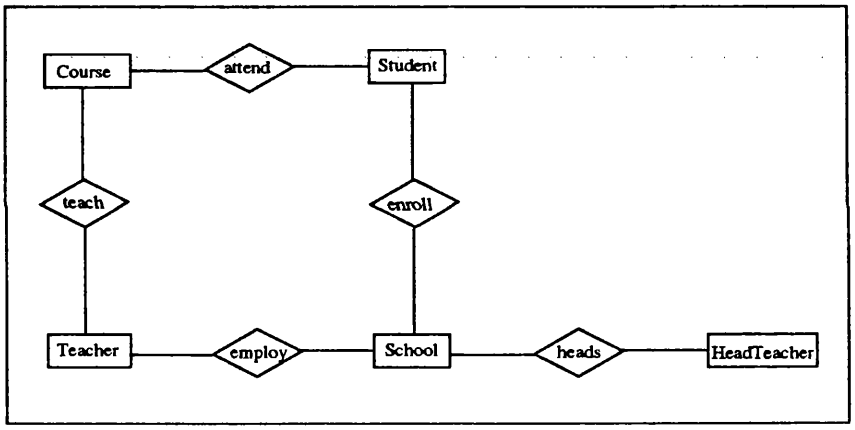


Figure 2.1: An Entity-Relationship Diagram

From the above example, it seems that messages have to be implemented whenever a relationship is detected. However, there are still problems:

i. A relationship is an association between two entities. If we are going to implement messages concerning a relationship, which entity set (class) should these messages belonged to?

ii. Is there any difference which class such messages is assigned to?

iii. What about including such messages in both classes?

† The term 'message' is originated in Smalltalk and it means a request for an object to carry out one of its operation.

Besides, there are three kinds of relationship found in object-oriented programming. However, entity-relationship modelling only addresses the 'interaction' relationship explicitly. Consider the previous example, the experienced object-oriented programmer has probably already noticed that the 'HeadTeacher' class is a subclass of the 'Teacher' class and the 'Teacher' class is a subclass of the 'Person' class. This kind of relationship, 'inheritance', is not included in entity-relationship modelling. The 'inheritance' relationship, however, is regarded as an important issue in object-oriented programming. It contributes a great deal to achieving reusability. Thus, it seems important to provide a notation to express this relationship in the analysis/design phase. Indeed, some people are trying to achieve this by extending the E-R notation to capture the inheritance relationship as well [War89].

Generally speaking, Entity-Relationship modelling is effectively used to analyse static aspects of a system. It concerns the data stores and the relationships between different data stores. Whilst it may help to identify some of the objects in an object-oriented system, especially the static ones, it does not help to analyse the interaction between objects, the dynamic part of a system.

Just as in conventional programming, entity-relationship modelling cannot be used on its own in constructing an information system. The modelling technique only models the data relations in the system. To be useful, the design method must provide a framework in which the architecture of such an information system can be constructed. Hence, entity-relationship modelling alone is insufficient to construct an object-oriented system.

2.2.5. Booch and HOOD Design Method

Booch was one of the earliest people to look into the development of an object-oriented design method. In 1986, he published a paper [Boo86] which states that an object-oriented design method should consist of the following steps:

i. identifying the objects and their actions.

ii. identifying the operations that may be acted upon the objects.

iii. identifying the relationships between objects and operations.

iv. examining the detailed design to give implementation descriptions for objects.

These steps are then reiterated recursively in order to obtain a complete design. In fact, this set of steps has summarised the essential procedures for object-oriented design and is regarded as the basic framework for most of the other object-oriented design methods [Lor86]. Booch's method also provides some graphical notations for system designers to represent the program components and packages (see Figure 2.2).

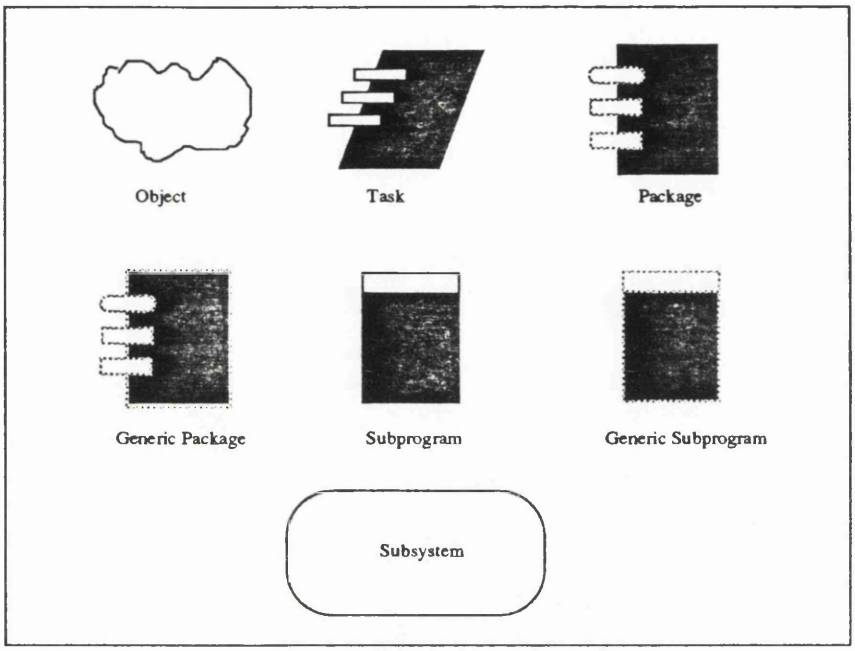


Figure 2.2: Graphical Notations for the Booch's Diagram

HOOD was developed by the European Space Agency (ESA) as a design method in the ESA Software Engineering Lifecycle. It is very similar to the Booch's design method, in fact, HOOD is usually regarded as an extension of the Booch's method. The design method is interfaced with SADT in the analysis/requirement phase and is targeted at the Ada programming language in the implementation phase. It is divided into four stages:

i. Definition and analysis stage.

ii. Revise the definition and analysis stage in natural language.

iii. Identify the nouns and verbs from the natural language specification to be objects and actions of the system.

iv. Refine the design and produce a formal object description skeleton to be ready for coding. The object description skeleton concerns interfacing in Ada syntax and semantics.

HOOD, also incorporates a graphical notation to allow systems designers to express their idea in diagrams (see Figure 2.3). There are at least three computer-aided software engineering (CASE) toolsets which have been developed to support HOOD [Rob89].

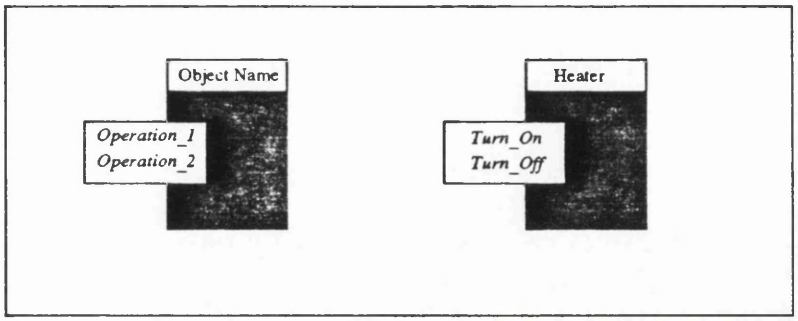


Figure 2.3: Graphical Notations for the HOOD Diagrams

As it stands, the fundamental problem with these two methods is that they are targeted at systems which are to be implemented in the programming language Ada [Bar82]. Now whilst whether Ada is an object-oriented programming language is still a debatable issue [Tou87, Weg87], it is generally felt that design methods which are geared towards Ada implementation do not fully support the features found in object-oriented programming. As Pressman [Pre87] mentions in his book, "The object-oriented design method of Booch is oriented towards software development in programming languages such as Ada. The method does not address a number of important object-oriented concepts (e.g., inheritance, messages) that can serve to make object-oriented design even more powerful". Besides, a good design method should not be closely bound to a particular programming language, these two methods have already demonstrated inflexibility in this sense. In the case of HOOD design method, its inflexibility is even more obvious. Robinson, himself stated in his paper [Rob89], "ESA has decided to use SADT for requirements analysis for the Columbus Space Station project software, and Ada had been selected as the main programming language for onboard and ground software. Some sort of PDL was planned for the detailed Design Phase, so *HOOD had to fit between SADT and PDL with Ada as the target programming language*"†.

There are also criticisms about the graphical notation employed in the Booch's method. Some people have expressed concern about the program components and package notations. They say that these notations require certain details which are not easy to use with automated graphical editing system [Som89].

All these points imply that Booch's and HOOD design methods are not true object-oriented design methods and do not fully support object-oriented system developments.

† This author's italics

2.2.6. Mittermeir Object Oriented Software Design

Mittermeir described an object-oriented method for large scale software design in 1986 [Mit86]. His object-oriented design method adopted the conceptual modelling language, CML [Mit82] which is based on a semantic data base method to support the requirement analysis and conceptual design. The design method is divided into three steps:

i. Object identification step.

ii. Network step.

iii. Disaggregration/specification step.

The object identification step consists of three substeps:

i. reality capture to identify objects of reality which are relevant for the system to be developed

ii. attribute capture to identify the properties of each object

iii. normalisation to rule out functional dependencies within an object

The network stage is used to identify the interaction between the objects in the system. It is also responsible for setting up the message passing communication amongst objects in the system. This step results in specifying objects' interfaces and guiding designers as to how to actually implement the object interactions' connection structure.

The disaggregation/specialisation step consists of two substeps:

i. the object disaggregation substep is to decompose the objects into subobjects.

ii. the object specialisation substep determines the speciality classes of the generalised objects that are defined earlier in the design phase.

As it stands, the object-oriented design method suggested by Mittermeir addresses important object-oriented features such as message passing and inheritance. Although the basic framework of the Mittermeir design method is similar to that of Booch, Mittermeir has refined each step in the framework in more detail. For example, the object identification step is further divided into three substeps. It suggests that the identified objects should be categorised into input/output, initiating/passive, and referenced/consumed objects. Hence, the Mittermeir design method is more an object-oriented design method than Booch's design method.

However, the method suggested by Mittermeir is still inadequate for object-oriented system development. As a design method, it defines what should be done in each step but does not specify the interface between each step. For example, how to present the information

obtained in the object identification step to the network step is not specified. Hence, again, the design method lacks a means of continuity which allows system designers to go from one step to the next comfortably.

Furthermore, although inheritance is regarded as an important issue in object-oriented programming, this issue is not emphasised or supported enough in the design method.

2.2.7. ObjectOry

ObjectOry [Jac87] is a development method for large systems. It is divided into two phases:

i. system analysis.

ii. system design.

The system analysis phase is used to set up the conceptual model for the system design phase. The system analysis phase is divided into three sub-factories:

i. Entity modelling - this is used to model the static aspects of the system. The result of the modelling is shown in a conceptual diagram in which nodes correspond to entities and arcs correspond to relations.

ii. Use cases modelling - "A use case is a special sequence of transaction, performed by the user or the system in a dialogue". The result of the use cases modelling is a kind of conceptual diagram which denotes the relation of use cases and entities. There are two kinds of relations:

   - the built-on relation which is found between different use cases. It indicates a use case is a specialisation of a more general use case. The built-on relation, in fact, highlights the inheritance relation between two use cases.

   - the access relation which is found between entities and use cases. It highlights those entities which can be accessed by a use case.

   The conceptual diagrams generated from use cases modelling is mainly used as the specification in discussions between developers and clients (see Figure 2.4). They are also used internally to check whether the developed system meets the use requirements.

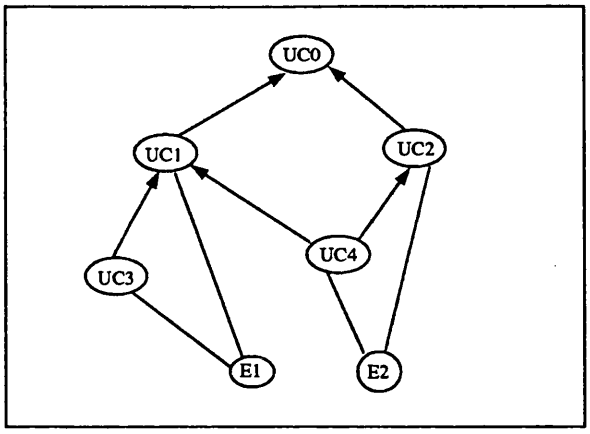


Figure 2.4: A Schematic Conceptual Model of the System's Use Cases

iii. Service modelling - A service is a cluster of similar parts of use cases. Service modelling is important to identify a suitable structure for the system. As a result, three types of relation are identified between services:

- the IS-A relation means that a service is a specialisation of another, this indicates the inheritance relation between two services.
- the extension relation simply indicates that a service is an extension of another.
- the interaction relation denotes the communication between two services.

The result of the system analysis phase yields three kinds of conceptual diagrams, those for entities, use cases and services. These diagrams are then used as the input for the system design phase. The system design phase is again divided into three sub-factories:

i. system level design

This is used to identify the system structure. It is further sub-divided into four sub-factories: system structuring, use case design, use case testing and system testing.

ii. block design

A block in the system is constructed as a factory. Note that a block is said to be implemented as a class in object-oriented programming.

iii. component level design

This is where system components are built.

As it stands, ObjectOry can be regarded as a promising approach for object-oriented system development. The method covers both the analysis and the design phase. Each phase is

sub-divided into several stages. The details of each stage are clearly defined and the input and output of each stage is specified. The design method highlights some object-oriented features, for example, the blocks used in the design phase will be implemented as classes using object-oriented programming. Also, the method encourages the inheritance relations to be expressed in the analysis stage. Further, ObjectOry is a user-oriented design method. The use cases concept in the method help to develop a user-driven system.

However, ObjectOry is not a design method which is specially developed for object-oriented programming. In fact, the method incorporates three independently developed techniques:

i. block design.

ii. conceptual modelling.

iii. object-oriented programming.

This is probably the reason why ObjectOry looks so complicated. Further, as Jacobson mentions in his paper [Jac87], the framework of this design method originates from Ericsson Telecom and is widely used in the telecommunication industry. Thus, one can easily find some phases and descriptions in the design method that are specially aimed at telecommunication software. For example, a service is defined as an indivisible ordering packet of functions for a specific system. The term 'packet' is one oriented to telecommunication. In addition to this, there are a large number of terms and concepts used in the design method such as use cases, services, factories, etc which are new to system designers. The consequence of this is that system designers who are not in the telecommunication industry may find the method difficult to understand and follow. This also means that system designers who are familiar with current structured design methods such as Yourdon's and JSD will have a difficult transition to object-oriented design.

The method encourages the expression of inheritance relations in the analysis phase but there is no general guideline of how it should be handled, especially in the design phase. It is strongly felt that some sort of support should be given in the design phase to ease the job of designers in handling inheritance and designing class hierarchies.

2.2.8. Summary

In software engineering, design is the process of applying various techniques and principles for the purpose of defining a system to solve a given problem. The system should be defined in sufficient detail to permit its physical realisation. A design method should help system designers to organise the design activities towards such a definition. It should decompose the design activities into distinct and ordered tasks, providing tools appropriate to each task and criteria for the correct completion of the task. More importantly, it should set up the

appropriate implementational model for the implementation phase. Hence, the design method should embody the same model as the implementation language so that the transition from the design phase to implementation phase is straight forward.

With conventional structured design methods, the design method organises the design activity towards an implementation in a procedural programming language. It targets an implementational model in which modularity and abstraction lies within an operation; functional abstraction. The decomposition of the system is based on its functionality. The graphical constructs and notations provided by these design methods are geared towards procedural programming. In object-oriented programming, the modularity and abstraction lies within an object. This kind of mismatch, therefore, forms the basis of why these conventional design methods are inappropriate to be used in object-oriented programming.

Although transformation techniques have been developed to allow specifications generated from conventional design methods to be used in object-oriented design, as the fundamental philosophy of these design methods and object-oriented programming are different, they do not seem to provide a satisfactory solution.

In the case of the existing 'object-oriented' design methods, it was found that these existing design methods do facilitate the development of object-oriented software to a certain extent. They all agree on a common framework which emphasises the identification of objects and their corresponding actions within the system. However, most of the development of these design methods are not dedicated to object-oriented programming. Whilst some of them are developed as a design method to fit in a pre-defined software environment, others were developed for a particular programming language. For example, the development of HOOD had to be consistent with a software environment which takes SADT as the requirement analysis method and Ada as the programming language. In this case, the development of a true object-oriented design method becomes less important than a design method that fits in a pre-defined environment. The consequence of this is that not only the resulted design method is inflexible to use, but also that it does not fully support the main features of object-oriented programming.

One of the important feature in object-oriented programming is inheritance. A design method for object-oriented programming should fully support this feature. It should have guidelines and even software tools to help system designers to construct the inheritance hierarchies. However, as one can see, the four existing object-oriented design methods do not support inheritance to an acceptable extent, for example Booch's design method and HOOD which are bound to the programming language Ada. Since Ada itself is not a true object-oriented programming language, it does not support inheritance. Hence, the design methods which are

targeted at Ada as the implementation language do not support inheritance†. The Mittermeir method and ObjectOry have discussed inheritance in their design process but do not provide full support or guidelines for system designers to handle inheritance.

2.3. Conclusion

This chapter has defined clearly the term 'object-oriented' as used in this research. It has analysed why existing design methods are inadequate for object-oriented programming. The result of the analysis highlights that an alternative design method for object-oriented programming has to be developed. Such a design method should emphasises object-orientation awareness in the early stage of the design phase. The method should contain adequate support for system designers to handle inheritance. The method would guide system designers to organise the design activities towards an object-oriented implementation. Chapter 3 of this thesis gives a detailed description of the design method which has been developed according to this specification.

† The importance of inheritance, in fact, has recognised by some of the authors of the existing object-oriented design methods recently. In June, 1989, Booch gave a seminar in London and said that he realised that a shortcoming of his design method was having no support for inheritance. Hence, he has extended his method to cover this issue so that it caters for more orthogonal object-oriented programming languages such as Smalltalk-80 and C++.

"A systematic approach to design simplifies the process and results in software which is understandable, verifiable and reliable without stifling the creativity of the software engineer."

~ Ian Sommerville ~

Chapter 3

The Design Method

The previous chapter highlighted the deficiencies of existing design methods when applied to object-oriented programming. Therefore, the primary goal of this research is to develop a design method which is specially targeted towards an implementation in an object-oriented language. Such a design method would form the core of an integrated object-oriented software development environment.

This chapter serves the purpose of describing the primary framework for the design method resulting from this research. It starts off by discussing the goals and defining the scope of such a design method. It then gives an overview of the design method prior to a detailed discussion. This chapter concludes with a discussion of other issues which are related to object-oriented systems design.

3.1. Basic Principles of the Design Method

The design method developed in this research is specifically tailored for object-oriented programming. Before describing the design method itself, it is important to first define the design principles of this method. This also serves to explain how certain design decisions are made with respect to this method.

3.1.1. Attitudes towards Design

Like most design, software design is a creative process. It requires a certain amount of experience and intuition from the software engineers. Software engineers seldom find that they can attain a good design at the first attempt. A good design is always the result of a

number of preliminary designs. As Sommerville says [Som89], "Design cannot be learned from a book - it must be practised and learned by experience and study of existing systems". A design method cannot be expressed as a formula which when followed it, guarantees a successful design. What a design method can do is to provide enough guidelines for system designers to organise the design activities towards a particular goal. It guides the designer in getting from one point to another in the design phase. It provides clear instructions about what to do next whenever there are a large number of choices.

Therefore, the main principle of the design method in this research is to provide a set of guidelines to help system designers to organise the design process towards an implementation in an object-oriented programming language. The design method does not impose rules to hinder the creativeness of the designer.

Most system designers have some knowledge about existing structured design methods. Although it has been found that these design methods are inadequate when applied to object-oriented programming, it is better to have the design method bear some similarities to existing design methods when possible. In this way, system designers are familiar with the design method and this will bring down the learning curve. Therefore, the design method from this research tries to use constructs which are already be familiar to system designers.

Further, the design method is designed to serve as a teaching aid for the novice in object-oriented programmer. It helps them to understand the philosophy and characteristics of the object-oriented programming paradigm. For the experts in object-oriented programming, the design method also serves as a documentation tool. It aims to assist them to document and realise each stage in the design process.

3.1.2. Analysis and Design

Software development always begins with a problem thought to be solvable by a computer system. It then goes through a process in which the solution is presented in software. During the development process, a sequence of decisions have to be made which leads to a target system that can be assembled and used.

The classic life cycle for software development is known as the 'waterfall model'. This is illustrated in Figure 3.1. The 'waterfall model' demands a systematic, sequential approach to software development which starts off with the requirement analysis phase and progresses through design, coding and testing.

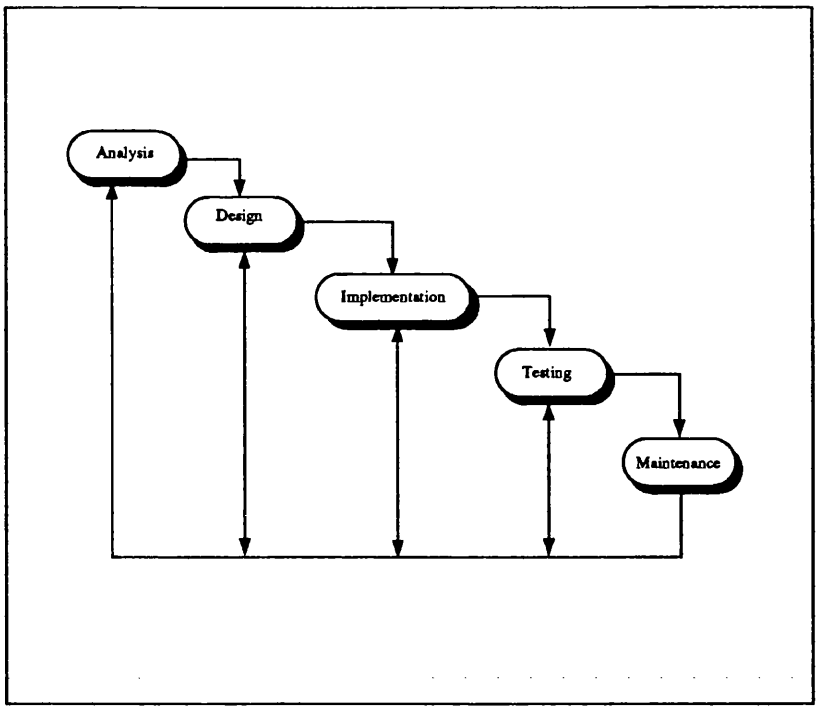


Figure 3.1: The 'Waterfall Model' for the Software Development Cycle

de Marco [deMar78] said, "In the specific domain of computer systems development, analysis refers to the study of some applications, usually leading to the specification of a new system". System analysts need to understand the problem domain and its development environment. Although it is often suggested that the requirement analysis phase is to find out 'what' is to be implemented and the design phase is to figure out 'how' to implement it, it is impossible to carry out a detailed analysis and to produce a precise requirement specification without some design activities. System analysts not only interact with the clients and the end-users †, but also communicate a lot with software engineers. Hence, the requirement analysis phase is always found to overlap with the design phase.

In fact, the distinction between the analysis and the design phases is even more blurred in object-oriented programming. It is generally agreed that object-oriented programming is more suitable for prototyping [DiM87, Mey88, Som89] (see Figure 3.2). Hence, the classic 'waterfall model' is no longer applicable in the development of object-oriented programs. The software development cycle for object-oriented programming has been described as a

† In this thesis, clients are people who want to construct the computer systems, end-users are people who actually use the computer systems when it is constructed.

'recursive/parallel' cycle in which the various phases in the development cycle overlap each other. The requirement analysis phase and the design phase are no longer independent.

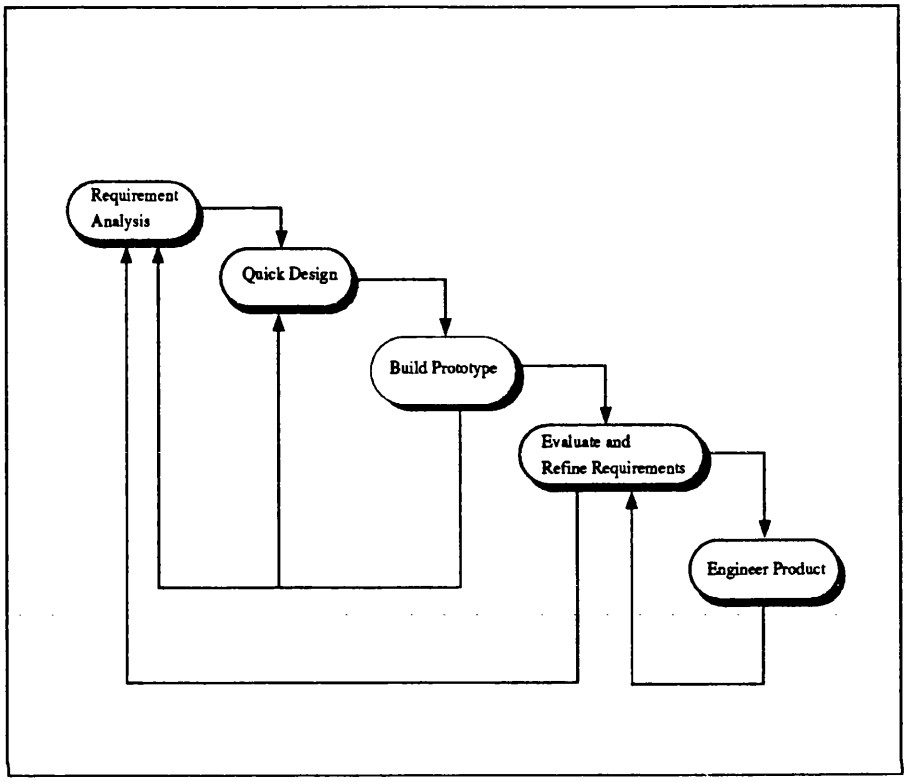


Figure 3.2: The 'Prototype Model' of the Software Development Cycle

This characteristic has, in fact, been recognised in this research. The design method developed encourages interactions between the analysis and the design phases. The design method assumes that the requirement analysis and the feasibility study of the system have been carried out and a set of requirement specifications is ready prior to using the design method. However, the first part of the design method specifies how the conceptual model should be set up and used in the rest of the design phase.

3.1.3. Modelling

A model is a representation, usually on a smaller scale, of a device, system, structure etc. Modelling is regarded as an effective mechanism for the technical analysis of software systems. In fact, the different phases in software development can be viewed as the different processes required to construct different models.

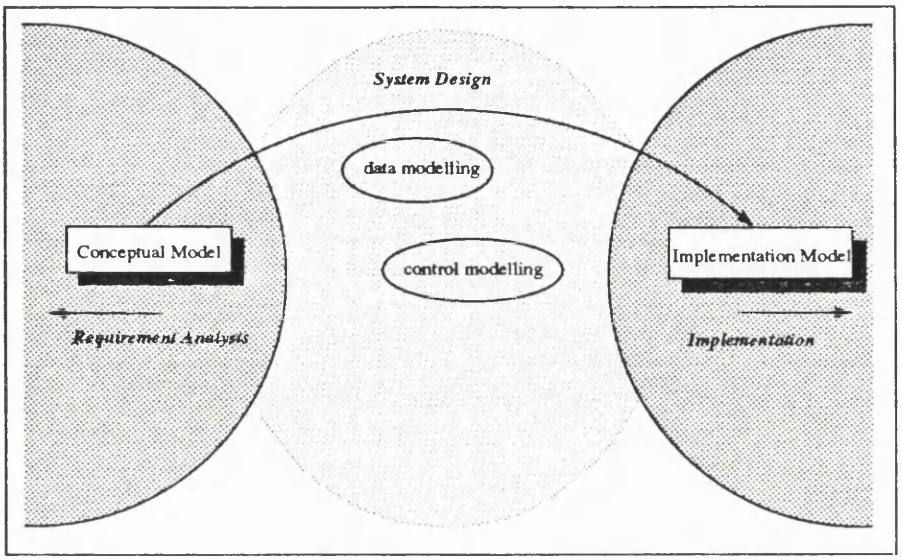


Figure 3.3: Modelling in Software Development Cycle

As shown in Figure 3.3, the software development process begins with the requirement analysis phase. The aim of the requirement analysis phase is to find out what exactly the client wants. This leads to the construction of a conceptual model of the software system. A conceptual model is basically a high level abstract representation of the system. It presents a system view which is understood by clients and expected by end-users. In the later stage of the software development, such a conceptual model has to be transformed into an implementation model. An implementation model is a representation of the system with respect to the implementation environment of the system, e.g., the kind of implementation language one is going to use. The implementation model generally acts as a reference model for programmers to implement the system.

To transform the conceptual model to the implementation model is, therefore, the objective of the design phase. In a way, the design process can also be viewed as a collection of modelling processes. For example, it may involve data modelling, control modelling and process modelling of the system.

In view of this, the design method of this research specifies the kind of conceptual model required in the first stage of the design phase. The design method then provides guidelines for system designers to transform the conceptual model into an implementation model. Of course, the implementation model accomplished here is one which is suitable for an implementation in an object-oriented programming language.

3.1.4. Design Description Language

As mentioned earlier, the software development process involves constructing appropriate models at different stages. These models are normally specified using a design description language. The design description language can either be textual, graphical or both. As it is widely agreed that the human mind acquires information at a significantly higher rate from notations in pictures than by reading text [Cha80, Rae85], models are usually specified in a graphical form.

Besides graphical and textual description languages, specifications can be specified in a very formal, mathematical approach, i.e. formal specification languages, which is very popular in the formal methods community. The main advantage of formal specifications is that they have a precise meaning [LiG86]†. With a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specification. This can demonstrate that certain classes of errors are absent [Som89]. However, formal specifications have not been widely used because they require system developers to have a familiarity with discrete maths and logic. Besides, it is hard to demonstrate that the development of a formal system specification will reduce software development costs.

Nevertheless, it has been decided in this research that to attain visibility in a design is more important. Hence the design method developed tends to use graphical constructs in specifying the models. These graphical constructs are easy to learn. With the help of graphical editors, users should not find that they have to spend too much time on manually drawing the graphical constructs and can spend most of their time on the main system design.

3.1.5. User Interface Design

In the early days, the user interface was not regarded as part of the system design. Often, system designers either treated the user interface design as a separate issue or left it untill the end of the system development. This observation can be further supported by the fact that most of the traditional design methods do not include user interface design as part of the design method. However, in order to attain a successful and usable system, system designers should focus on user interface design at an early stage of the design phase [GoL85, Kli77]. Besides, a user-oriented approach to system design helps to reduce conflict and promote cooperation between users and designers [Luc71].

† Formal semantics can sometimes found in graphical notations such as Petri Nets [Pet77].

Therefore, the design method developed in this research brings in user interface issues at an early stage of the system design. It emphasises that user interface design is not only important but also has to be looked at at an early stage.

3.1.6. Design in Different Problem Domains

Just like all other design methods, it is unwise to think that the design method developed in this research can be applied to all cases of object-oriented systems design. Software systems can be grouped into different domains depending on their characteristics. There are specific systems such as concurrent systems, distributed systems, real-times systems. Each of these specific systems emphasises different aspects in the system development. There is no one method which can cater for systems in all problem domains. In fact, when one looks into the development history of structured design methods, one finds that a general purpose structured design method such as Yourdon's method was first introduced, then modified and further developed to fit different problem domains. For example, Post [Pos86, War89] has modified the structured method to apply in the real-time industrial software development. Gomaa [Gom84] has extended the Structured Analysis/Structured Design method (SASD) for real-time systems.

Therefore, the primary aim of this research is to obtain the general framework of the design method for object-oriented programming. It is believed that the design method will have to be developed further in order to fit in special domains such as building concurrent systems.

3.2. An Overview of the Design Method

This section gives an overview of the design method before a more detailed discussion is given†.

As shown in Figure 3.4, the design method is basically divided into three levels:

i. the conceptual level,

ii. the system level and

iii. the specification level.

† A summary of this section can be found in [PuW89b, PuW90a].

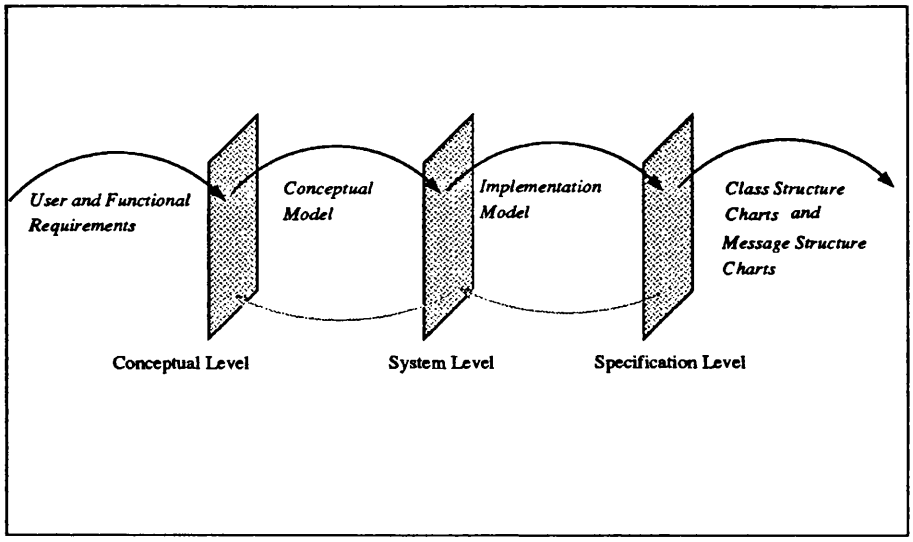


Figure 3.4: The Overview of the Design Method

The design method has been developed to fit into a 'prototyping' software development environment. Thus different phases in the process overlap with each other. Because of this, the design method not only covers the design phase but also part of the analysis phase. A set of requirement specifications needs to be ready prior to using the design method. With the requirement specifications, system designers go through the stages suggested in the method to produce a set of design specifications. This set of design specifications will then be taken by the programmers and implemented in an object-oriented language. Due to the characteristic of the 'prototyping' model, the three levels mentioned in the design method are re-iterated throughout the design phase.

The design method starts off with the conceptual level. The main objective of the conceptual level is to identify the objects and operations occuring in the application layer of the system. The designers obtain information from the clients, analyse it and construct the conceptual model for the application. Such a conceptual model is presented by object-interaction diagrams.

Once the conceptual model of the application is set up, system designers enter the system level. The system level is where the implementation model of the application is assembled. There are three issues which are of interest at this level:

i. object inclusion,

ii. object interaction and

iii. class inheritance.

The system level is mainly concerned with how an object is constructed and since an object

in object-oriented programming is created as an instance of a class, it is also concerned with how to construct a class. This leads to the identification of the set of variables and the set of operations that belong to a class. As well as this, the interactions between classes, especially existing system classes must be taken into account when constructing classes. For example, a class can be created as a subclass of an existing system class. In some cases, it may even lead to the construction of new class hierarchies. Further, it is also necessary to understand the interactions amongst the objects at the system level. All this information can be recorded in the object-interaction diagrams.

With the information obtained from the conceptual and the system levels, system designers should be able to specify the objects and the interaction between objects. This kind of information is presented in the 'class structure chart' and the 'message structure chart' at the specification level. In fact, these charts will be taken as design specifications by the programmers to implement the system.

3.3. Detailed Description of the Design Method

This section of the thesis presents the details of the individual stages of the design method. It discusses the objective of each stage and the steps required in order to achieve the objective in a comprehensive manner. Throughout the discussion, examples drawn from the development of a 'GP Surgery Notes System' and a 'Home Heating System' are used to illustrate the ideas central to the design method. Both these systems were developed with this object-oriented design method. The usage of the design method in developing these two systems can be found in Appendix A and Appendix B.

3.3.1. The Conceptual Level

The conceptual level is regarded as the front end of this design method. Its main objective is to set up the conceptual model of the application. The objects and interactions involved in the application layer of the system are identified and presented as the conceptual model of the system. The conceptual model, as defined earlier, is a representation of the system which is mutually agreed between system analysts and clients. It contains very little information about the implementation of the system.

The conceptual level can also be viewed as the back-end of the requirement analysis phase in the software development process. The requirement analysis phase itself is a complicated process. It ranges from interviewing clients and end-users and analysing their needs to specifying both the functional behaviour of the proposed system and non-functional requirements that must be met [Bor85]. As the requirement analysis phase is a complex

one, various methods and techniques have been developed to ease the task of analysts. Some of the more popular ones are Structured Analysis [Mar78], SADT [RoS76], SREM [SSR85] and SSADM [AsG90]. There are also requirement analysis methods which are specially for user-interface design such as TAKD [Joh85]. Obviously, to attain a complete and consistent software development process, it is better to have the requirement analysis method compatible with the design method. Therefore, a better candidate for a requirement analysis method to complement this design method would be one which is object-oriented.

One of the earliest attempts to introduce object-orientation into requirement analysis and specifications is probably that of Borgida [Bor85]. In his paper, he suggests using an object-oriented framework to construct the requirement model. Recently, serious effort has been made to try to attain an object-oriented analysis method. For example, Shlaer and Mellor [ShM88], Coad and Yourdon [CoY90] have published work on object-oriented analysis. Bailin [Bai89] has proposed a method of analysing requirements for object-oriented software which evolved from structured analysis and serves as an alternative to structured analysis when the use of object-oriented design is foreseen. Although, these requirement analysis methods are not widely used yet, it is generally agreed that an object-oriented requirement analysis method is needed to complement the object-oriented design method.

As stated earlier, most of the activities in the requirement analysis will not be considered in this design method. Hence, the details about object-oriented analysis will not be discussed here. The design method assumes that the functional and the non-functional requirement specifications are ready. With the requirement specifications, the design method starts off by identifying the proper objects and interactions to construct the desired conceptual model. In a way, the relationship between the analysis and the design phases suggested by this design method is that the design method makes use of the information obtained in the analysis phase to construct the conceptual model. The model is specified in the language defined by the method.

3.3.1.1. Identification of Objects

Most of the literature about object-oriented programming [Boo86, Cox86, Pre87] agrees that the initial step in object-oriented programming is to identify the objects involved in the system. Although everyone knows that this is the first step to carry out, no one knows exactly how to proceed.

In 1983, Abbot [Abb83] proposed a method to identify objects and actions from informal requirement specifications written in English language. He suggested that the nouns and noun phrases in an informal English description of the system are good indicators of the

objects and their classifications. Abbott's method, in fact, has been included in certain object-oriented design methods such as Booch's method and HOOD. However, the general feeling about Abbott's method is that it may be taken as the starting point to identify objects but the pinpointing of the relevant objects still relies very much on the intuition and experience of the system designers themselves, i.e., the 'human magic' of design.

Hence, it seems that there is no one formula which can be applied to identifying objects in a system. It all depends on ones understanding of what an object is. Therefore, the primary step taken in this method is to state clearly what an object is at the conceptual level.

In object-oriented programming, an object represents an entity in the user's mental conception of the real world. An object is an encapsulation of:

i. a set of variables which describe its state, and

ii. a set of operations that can act upon the object to alter its state.

An object is a model of an entity in reality, it is not a value, an action or a time. To determine whether an entity in the application should be regarded as an object in the system, one has to make sure that the entity has a state that undergoes some actions in the application. For example, in the GP system of the Appendix A, 'the Patient Card Database' is an object. It embeds operations such as 'add', 'delete' etc, which can be invoked by other objects in the application. Also, as mentioned in Chapter 2, an object does not have a time factor, i.e., the order of the actions within an object is irrelevant. This distinguishes the object defined in object-oriented programming and an entity defined in the Jackson System Method.

Further, in this design method, objects are classified into two different types: application objects and implementation objects. At the conceptual level, system designers are interested in identifying the application objects. Here, application objects refer to objects which are found in the application layer of the system. These are objects which are understood by the clients and end-users in the system. For example, the 'Patient Card Database', and the 'GP Menu' are application objects identified at the conceptual level.

The application objects at the conceptual level are different from the implementation objects at the system level. Briefly, implementation objects are identified at the system level and are concerned more with the implementation model. This is discussed in more detail in Section 3.3.3.1.

3.3.1.2. Identification of Actions

In addition to identifying objects in the system, system designers have to identify the appropriate interactions between these objects. Abbott [Abb83] again suggests that the

verbs in the informal specifications of the system are good indicators of the actions involved in the system but like identifying objects, it also depends a lot on the intuition of the system designers.

Besides, in order to find out the interactions in the system, the developer may need to consider the functionality of the application stated in the requirement specifications. For example in the 'GP Surgery System' found in Appendix A, the requirement specification mentions that the system should allow users to update, and retrieve information about a patient. Hence, the object 'Patient Card Database' should have operations such as 'add card', 'read card' to act upon it. Although there are suggestions which said that functional decomposition is not an appropriate strategy for object-oriented programming [Mey88], some of the interactions can be deduced from the functional approach. For example, by examining the 'read card' operation and thinking about how the operation 'read card' works, the developer will identify that 'search card' should also be an operation acting upon the 'Patient Card Database'.

3.3.1.3. The Two Layers of the Conceptual Level

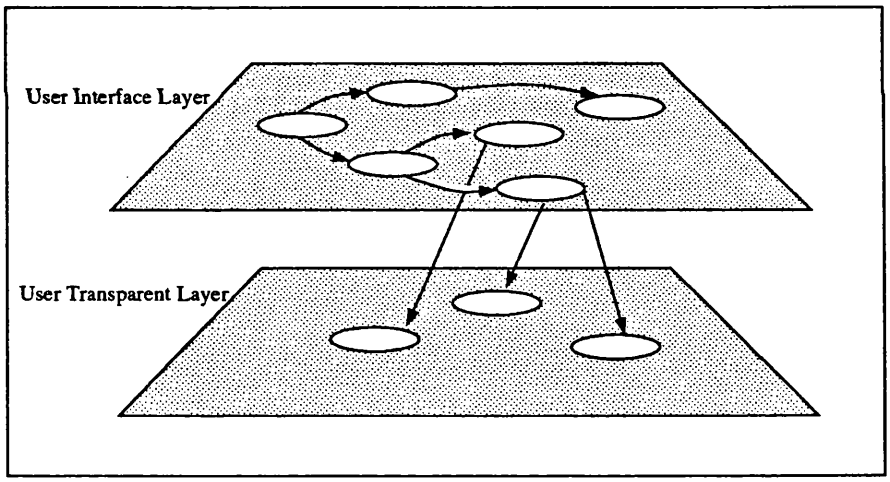


Figure 3.5: The Two Layers of the Conceptual Level

Almost all systems developed involve interactions with end-users at some stage, thus the user-interface design is actually part of the design phase. In order to highlight this issue and bring in user-interface design at the early stage of the design phase, the conceptual level is deliberately divided into two layers, the user-interface and the user-transparent layers (see Figure 3.5).

i. The User Interface Layer

This layer contains application objects which communicate and interact directly with end-users. It provides a visual presentation of what the system is to the

end-users. Objects such as forms and menus are typical interfacing objects.

ii. The User Transparent Layer
This layer contains application objects which are transparent to the end-users. As far as the end-users are concerned, they do not directly interact with these objects. For example, in the GP system, the 'Patient Card Database' is transparent to end-users.

Since the conceptual level is divided into two layers, system designers are obliged to arrange the identified objects into these two layers. The connection between these two layers is via message communication amongst objects in the two layers.

The separation of the conceptual level into two layers not only highlights the importance of user-interface design in general system design, but also gives more flexibility to system designers. As the importance of user-interface design becomes more and more recognised, methods have emerged to assist system designers in the design of better interfaces. The object-oriented paradigm has been introduced in this particular area. For example the Model-View-Controller (MVC) [Tre84], Presentation, Abstraction and Control (PAC) [Cou87] are two different frameworks which apply the object-oriented paradigm to user-interface management. By separating the conceptual level into two layers, the user-interface layer can be extracted and handled by interface experts. As long as the user-interface layer provides a proper interface, i.e., the message communication to interact with objects in the user-transparent layer, the conceptual model of the application is still correct. This approach certainly gives a better result as experts are handling that part of the design at which they are good.

3.3.1.4. Object Interaction Diagrams

Once objects and their interactions have been identified and classified, the conceptual model of the application can be set up. The design method uses object interaction diagrams to specify the conceptual model. The object interaction diagram is a kind of graphical notation used at the conceptual level. It allows designers to express the results of systems analysis diagrammatically. The graphical notation of the diagram is very simple. There are basically two different constructs (see Figure 3.6):

i. 'circles' denote objects which are involved in the system,

ii. arcs indicate that there are interactions between the objects. Note that these interactions would be further analysed at the system level and the information about what interactions, i.e., which message, is then realised. At that point, arrows are added to the arcs to show the direction of the flow of messages (see Figure 3.7).

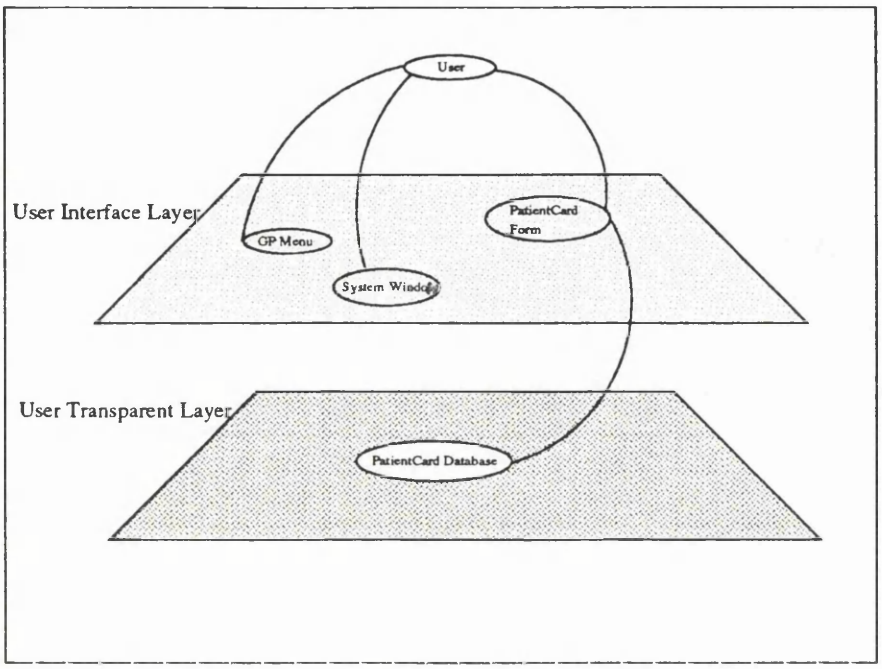


Figure 3.6: An Object Interaction Diagram of the GP System

3.3.1.5. Levelled Object Interaction Diagrams

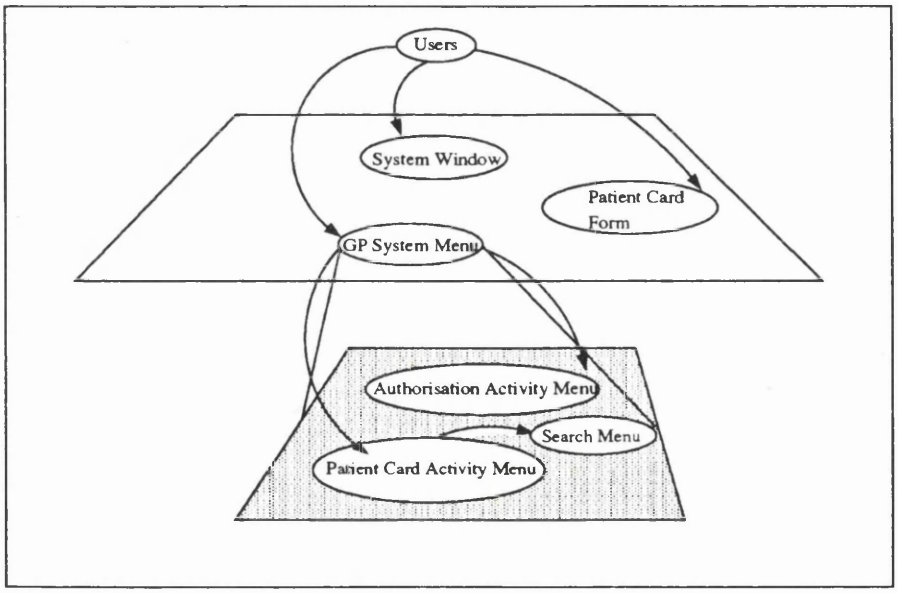


Figure 3.7: A Levelled Object Interaction Diagram

Very often, an object is actually an abstraction of several other objects. For example, a 'GP Menu' may not be a single-level menu but one which contains several sub-menus. It may

contain a sub-menu called 'Patient Card Activity Menu' which allows users to select an option to interrogate the patient card database. It may also contain another sub-menu called 'Authorisation Activity Menu' which allows users to select an option to enquire about who can do what with the GP system.

To reveal the structure of this abstraction, an object can be expanded into another set of object interaction diagrams as shown in Figure 3.7.

This kind of multi-level diagram is particularly useful in specifying large systems. In a way, it is similar to that of levelled data flow diagrams [deMar78]. In the structured analysis and design method, when a system is too large for its data flow diagrams to be shown on a single page, the system is partitioned into sub-systems as shown in Figure 3.8.

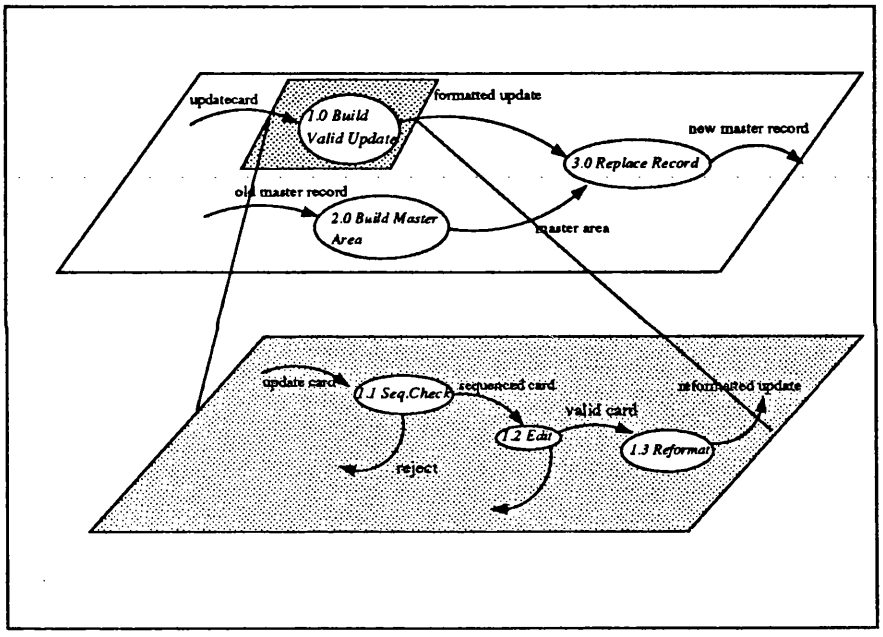


Figure 3.8: A Levelled Data Flow Diagram

A levelled data flow diagram allows the system to be partitioned in a process-oriented fashion whereas a levelled object interaction diagram partitions the system in an object-oriented fashion. The levelled object interaction diagrams are used more often at the system level when more implementation objects are revealed by identifying the 'contain' relationship in the system (see Section 3.3.2.2).

3.3.2. The System Level

The system level is regarded as the core of this design method. As stated earlier, the objective of a design method is to transform the conceptual model of an application to

its implementation model. The system level in this design method is responsible for carrying out the necessary tasks to accomplish this goal. Briefly, it has to identify the set of implementation objects and how they interact in the implementation model. In order to achieve this, system designers begin by considering the three important relationships associated with an object.

i. the 'contain' relationship,

ii. the 'use' relationship and

iii. the 'inherit' relationship.

This section describes what must be done at the system level. It first explains the concept of the implementation objects appear at this level. It then talks about the three relationships associated with an object and their importance in constructing the implementation model. The identification of the implementation objects and the three relationships contribute to the construction of the class structure. After presenting what have to be done at this level, it then suggests how to proceed with these tasks in an effective way.

3.3.2.1. The Concept of Implementation Objects

In section 3.3.1, it is mentioned that objects are classified into application objects and implementation objects in this design method. Application objects are mainly objects found in the application layer of the system and are fairly obvious to be identified. Implementation objects are objects which are going to appear in the implementation model. They have to be constructed as instances of a class in the implementation stage.

Although the objects which are identified at the conceptual level are very likely to be implementation objects at the system level, this is not always the case. For example, 'user' is an object identified in the conceptual model but such an object becomes transparent in the implementation model. Besides, additional objects may be needed in order to complete the set of implementation objects for the implementation model. For instance, one may need an object called 'error state' to determine which type of error has occured and its corresponding actions. Probably, it is better to view application objects as a small subset of the implementation objects. In order to obtain the complete set of implementation objects, system designers have to work on identifying relationships such as 'contain', 'use' and 'inherit' which are discussed in full detail in the following sections.

3.3.2.2. The 'Contain' Relationship

The 'contain' relationship is a one to many relationship amongst objects. As mentioned earlier, an object may be an abstraction of a set of objects, hence an object may contain a

set of other objects. For example, in the 'GP Surgery System' of Appendix A, although the 'GP System' itself is an object, it contains objects such as 'GP Menu' and 'Patient Card Database'. Thus, there is a 'contain' relationship between the 'GP System' and the 'GP Menu' as well as the 'Patient Card Database'. Another example is found in the 'Home Heating System' of Appendix B, the object 'Room' which contains a few objects such as the 'WaterValve', the 'TempSensor', the 'DesiredTemp' and the 'OccupiedSensor'. All these objects bear a 'contain' relationship with the object 'Room'.

The main objectives of realising the 'contain' relationship are:

i. to highlight some more implementation objects that are required for the implementation model,

ii. to reveal the underlying structure of an object, hence the structure of its corresponding class.

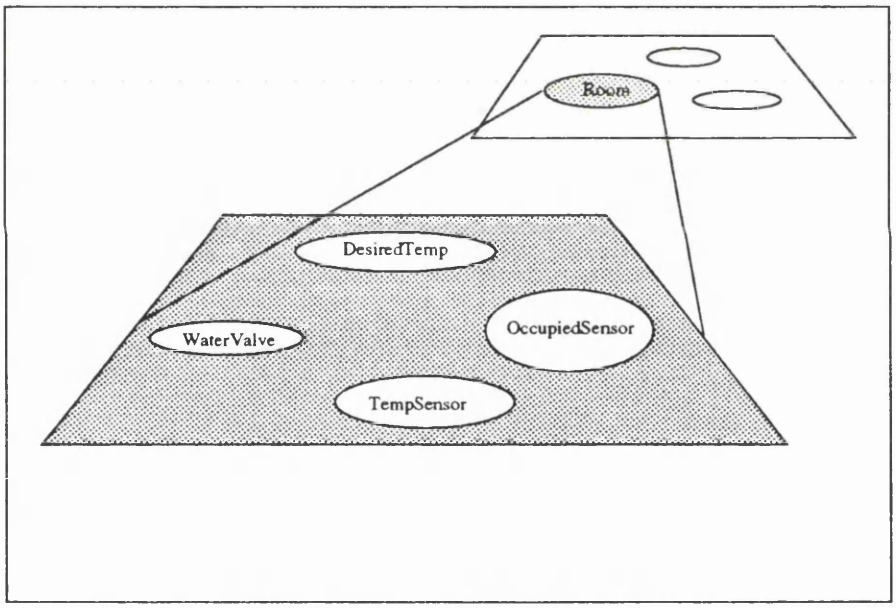


Figure 3.9: The 'Contain' Relationship of the Object 'Room'

The 'contain' relationship is extremely useful in constructing a class. As stated earlier, a class is a template which contains a set of instance variables and a set of operations. The objects which are identified in the 'contain' relationship are variables in the class structure. This piece of information is particularly useful at the specification level when specifying the class structure chart.

The identified 'contain' relationships have to be expressed explicitly in the object interaction diagram. It is expressed in a levelled object interaction diagram as mentioned in Section 3.3.1. For example, the 'contain' relationships found in the 'room' object mentioned

above is denoted as Figure 3.9.

3.3.2.3. The 'Use' Relationship

The 'use' relationship is a one to one relationship between two objects. In fact, the 'use' relationship is denoted by the message passing between two objects. In object-oriented programming, objects communicate by sending messages to each other. When objectA sends a message to objectB, one can say that objectA uses objectB to achieve some goal and a 'use' relationship is then established. The 'use' relationship also highlights the direction of the message flows, it distinguishes which is the sender and which is the receiver.

Again, the 'use' relationship can be used to identify more implementation objects for the implementation model. At the same time, it also highlights which operations/messages † are required for a particular object. For example, in the 'GP Surgery System' of Appendix A, a 'use' relationship is found between the 'Patient Card Form' and the 'Patient Card Database'. Once the end-user finishes filling the 'Patient Card Form', the 'Patient Card Form' then sends a message, 'add patient', to the 'Patient Card Database'. Here, it highlights that 'add patient' has to be defined as an operation which acts upon the object 'Patient Card Database'. In other words, the class 'Patient Card Database Class' from which the object 'Patient Card Database' is instantiated, has to define the operation 'add patient'.

The 'use' relationship is usually found between an object and its contained objects, i.e., the sender always contains the receiver. For example, in the 'Home Heating System' of Appendix B, the 'Room' contains a 'WaterValve'. When the system wants to open the water valve, it has to send a message 'openWaterValve' to the object 'Room'. The object 'Room' then sends a message 'openValve' to the 'WaterValve' which it contains. Hence, within the method of the message 'openWaterValve', a 'use' relationship is detected between the object 'Room' and the object 'WaterValve'.

Also, it is possible that an object has a 'use' relationship with an object which is not contained within that object, i.e., a sender can send messages to a receiver without physically containing the receiver. In this case, the sender has to know the reference of the receiver. In the example of the 'GP System' of Appendix A, the 'Patient Card Database' and the 'Patient Card Form' are two objects that are physically created and reside in the 'GP System'. In order to allow the 'Patient Card Form' to send the message 'add patient' to the 'Patient Card Database',

† The term 'message' originates from Smalltalk-80 [Gol83b]. Messages represent the interactions between the components of the system.

the 'Patient Card Form' has to know the reference of the 'Patient Card Database'. In most programming languages, this is done by passing the references of the receiver to the sender.

In addition, the 'use' relationship can also occur between an object and itself, i.e., an object can send a message to itself to perform certain tasks (See Figure 3.10).

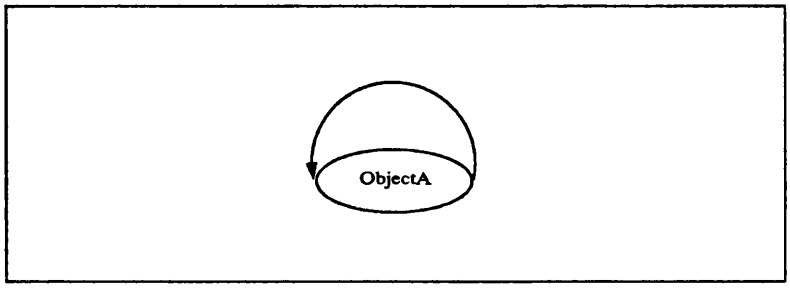


Figure 3.10: An Object which Sends a Message to Itself

This situation usually occurs when the set of operations cannot be accessed by outside objects. These operations/messages are introduced to support the implementation of other operations/messages. For example, error messages that are used more than once can be specified as private messages in order to create the literal message string only once. These operations are named differently in different programming languages, e.g. they are called private member functions in C++ [Str86b], private message category in Smalltalk [Gol83b] and non-export features in Eiffel [Mey88]. As messages can be either private or public, when a 'use' relationship between an object and itself is identified, one has to decide whether such a message should be implemented as a private message or not.

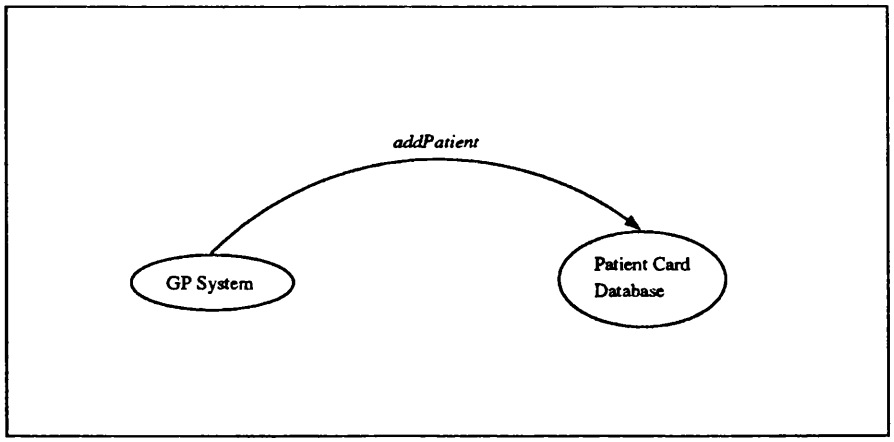


Figure 3.11: The 'Use' Relationship between Objects

Like the 'contain' relationship, the 'use' relationship has to be expressed in object interaction

diagrams. It is denoted by an arc and arrow in which the arrow shows the direction of the message flow. This is illustrated in Figure 3.11.

3.3.2.4. The 'Inherit' Relationship

The realisation of the 'contain' and the 'use' relationships assists system designers in highlighting the objects and interactions found in the implementation model. At the same time, it helps to model the structure of the required classes. However, all the information obtained from these two relationships is application-oriented. The information is derived from the characteristics and functionality of the application. As this information is very specific to a particular application, it is unlikely that the classes generated directly from the two relationships can be reused in other applications. Hence, this design method brings about a third relationship, the 'inherit' relationship, which has to be realised by system designers to attain reusable classes.

When classA is related to classB and classA is more general than classB, i.e., classB needs a larger set of attributes to describe its behaviour, then it is said that classA has an 'inherit' relationship with classB. Here, classA is the superclass and classB is the subclass in this 'inherit' relationship and classB inherits all the attributes that are defined for classA. The 'inherit' relationship of a particular class can either be a one to one or a many to one relationship depending on whether single inheritance or multiple inheritance is considered.

The 'inherit' relationship is particularly important in object-oriented programming because it distinguishes object-oriented programming from class-based and object-based programming [Joh88, Weg88a]. As the 'inherit' relationship allows a subclass to reuse the properties defined in the superclass, it enhances the reusability of software which is regarded as important in software engineering [Fis87, Joh88, Mey87].

In Johnson's paper [Joh88], he says, "Software reuse does not happen by accident, even with object-oriented languages. System designers must plan to reuse old components and must look for new reusable components". Handling the 'inherit' relationship is indeed the responsibility of the system designers. In fact, with the 'inherit' relationship, system designers cannot simply construct a class structure just by realising the 'contain' and the 'use' relationship. They have to compare the class structure with those which already exist in the development environment and see whether an 'inherit' relationship can be established. They may even need to design a whole class hierarchy in order to cater for future reuse. To carry out these tasks is not easy. Beck et al. [OBH86] accentuates the difficulty in handling the 'inherit' relationship by saying, "Even our researchers who use Smalltalk everyday do not often come up with generally useful abstractions from the code they use to solve problems. Useful abstractions are usually created by programmers with an obsession for simplicity, who

are willing to rewrite code several times to produce easy-to-understand and easy-to-specialise classes".

As the 'inherit' relationship is so crucial and difficult to handle in object-oriented programming, a design method specially developed for object-oriented programming has to provide some kind of guidelines or procedures to help system designers deal with inheritance. One of the deficiencies of existing object-oriented design methods is that they do not have enough support for inheritance, thus extra attention is paid to this area in this research. In order to help system designers handle inheritance, a manipulation process which is called the 'inheritance factorisation process' has been developed to assist system designers in constructing class hierarchies. The manipulation process is supported by a formal algebraic structure to ensure its correctness. Such a process can also be automated and allows system designers to specify a set of class specifications for which an optimal class hierarchy will be constructed automatically.

The 'inheritance factorisation process' is very important in this design method. It highlights one of the most important features which makes this design method different from the other object-oriented design methods. Therefore, the details about the inheritance factorisation process are to be found in Chapter 4.

3.3.2.5. How to Proceed?

Up until now, this section has mainly described what should be done at the system level in order to accomplish the implementation model. However, the sequence of how these tasks should proceed has not yet been discussed.

Actually, it is difficult to specify the sequence of these tasks particularly in a prototyping approach. One cannot really say that system designers should identify all the implementation objects before they start with realising the 'contain', 'use' and 'inherit' relationships of the system. Somehow, these tasks seem to occur simultaneously. Probably, the best approach is to apply the 'recursive/parallel' cycle to these tasks. The conceptual model which is obtained from the conceptual level provides a good starting point for this cycle. With the application objects identified in the conceptual model, system designers can begin to evaluate each application object and see whether it is indeed an implementation object. Then, system designers can examine each implementation object in turn and try to identify the 'contain' and the 'use' relationships associated with it. On the identification of each relationship, some more implementation objects and interactions may be picked out. These objects are then joined in the queue of the implementation objects to be examined later. This procedure is illustrated in Figure 3.12.

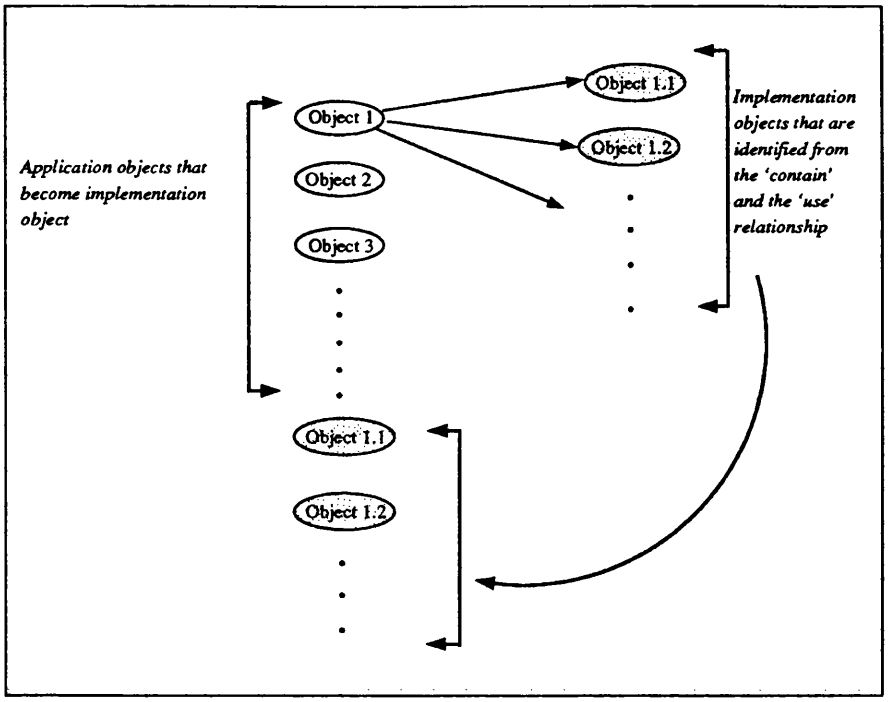


Figure 3.12: The Procedure suggested for the System Level

A concrete example of such a procedure can be demonstrated with the 'Home Heating System' of Appendix B. The conceptual model of the 'Home Heating System' contains objects such as 'Furnace', 'Timer', 'Heat Flow Regulator' and a number of rooms. These objects are also confirmed as implementation objects for the implementation model. First of all, the system designer examines the 'contain' relationship of the object 'Room'. It is found that the object 'Room' contains other objects, 'WaterValve', 'TempSensor', 'DesiredTemp', and 'OccupiedSensor'. These objects once identified are then grouped into the set of implementation objects which are going to be inspected later. The next thing to do is to pick out the 'use' relationship of the 'Room' object. From the conceptual model, the system designers notice that the 'Heat Flow Regulator' sends a message, 'getCurrentTemp', to the 'Room' object to obtain its current temperature. Hence, the operation, 'getCurrentTemp' has to be defined in the object 'Room', i.e., the class 'Room'. The system designer then looks into how 'getCurrentTemp' is constructed and finds that to accomplish such a task, the object 'Room' has to send a message 'getTemp' to the object 'TempSensor'. If more implementation objects are identified in this step, they are again included into the group of implementation objects which have to be examined later. This process is repeatly executed until all existing implementation objects are examined. Once this is completed, the system designer can look at the 'inherit' relationship of individual implementation objects.

3.3.3. The Specification Level

At the system level, system designers have to identify the set of implementation objects and realise the three relationships associated with each object. Although most of this information is recorded in the object interaction diagrams, when comes to the specification level, it is better to present it in a form in which programmers can easily visualise its implementation structures. As in object-oriented programming, there are two types of implementation structures, the class structure and the message structure. The class structure mainly concerns the set of variables, the set of operations and the class hierarchy associated with it. The message structure mainly concerns the method of a particular operation/message in a particular class.

In order to specify these two structures for the implementation phase, the design method has a specification level in which system designers have to present the implementation model in two individual kinds of design specifications, the 'class structure chart' and the 'message structure chart'.

3.3.3.1. The Class Structure Chart

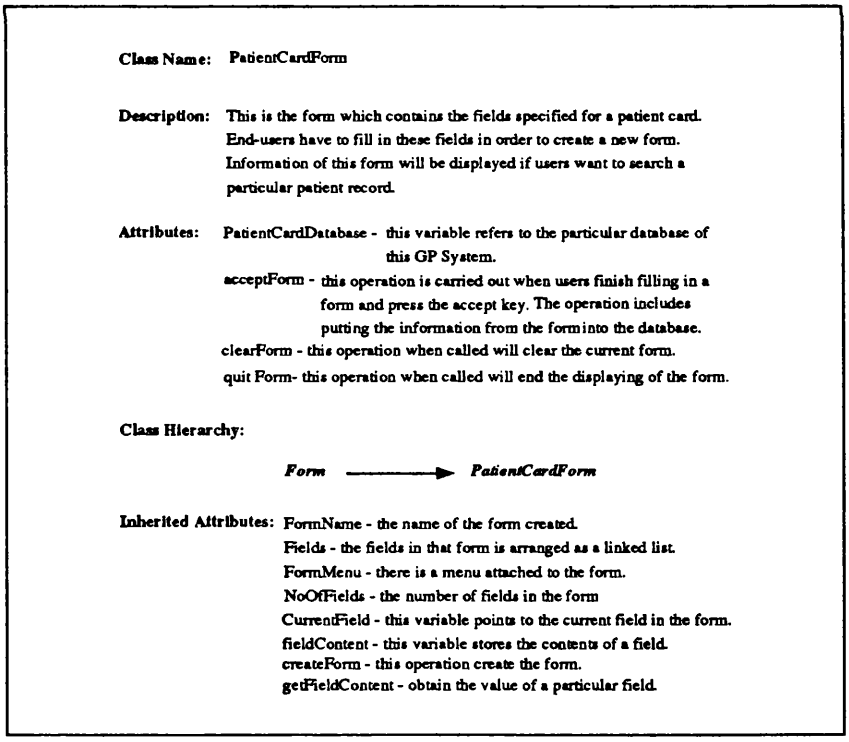


Figure 3.13: An Example of a Class Structure Chart

The class structure chart is a document which records detailed information about a particular

class. The information required in the chart includes the class name, the description of the class, the set of variables and the set of operations that belong to that class. In addition, the class structure chart also contains the corresponding class hierarchy graph. The graph shows the superclass and subclass relationships associated with the class described by the chart.

As it stands, the class structure chart is a summary of the information obtained from the system level in the design method. At the system level, system designers have to identify the set of implementation objects for the implementation model. This set of objects determines which class structure charts one has to construct. In addition, the system designer has to identify the 'contain' relationship which gives information about the set of variables belonging to a particular class. By realising the 'use' relationship, information about the set of operations belonging to that class is found and by working on the 'inherit' relationship, the corresponding class hierarchy graph for the chart is developed.

3.3.3.2. The Message Structure Chart

The message structure chart is a graphical notation which allows system designers to specify the method of a message/operation in a particular class. It is used as a visual program description to define the sequence of message passing and the control information within a method. The graphical notations of the message structure chart are:

i. Rectangular Boxes

A rectangular box represents an individual object/class. The box which is of current interest is divided into two compartments. The first compartment denotes the class name, the second contains the current operation/message for which a method is going to be defined†. Other rectangular boxes in the message structure chart denotes the message receivers of some messages and only contain the objects' name.

ii. Dashed Rectangular Boxes

Sometimes, the message receiver belongs to the same class as the message sender. In order to denote this, a dashed rectangular box is used for such a message receiver.

iii. Curved Rectangular Boxes

This kind of box denotes the superclass of the class which is of current interest.

iv. Interaction Connections

An interaction connection between boxes is denoted by an arc with an arrow which points from a sender to a receiver.

† In this thesis, this kind of rectangular boxes is shown in black to distinguish with the rest of the boxes which denote message receivers.

v. A Diamond

A diamond denotes an alternative path. It is a selector construct and used to represent an 'if' statement or a 'case' statement.

vi. An Ellipse Loop

An ellipse loop is an iteration construct which denotes a repetition of execution.

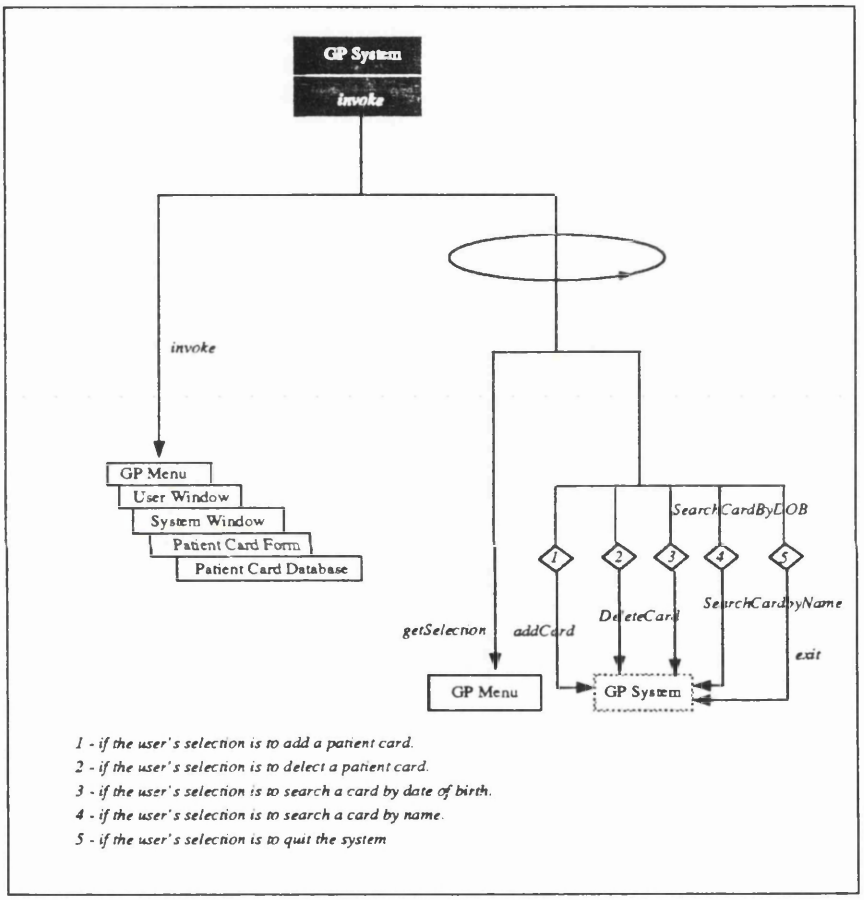


Figure 3.14: A Message Structure Chart in the GP Surgery System

The information recorded as object interaction diagrams gives knowledge about how the message structure charts should be constructed. The 'use' relationships which are identified at the system level denote the messages which are sent between the sender and the receiver. Very often, the identified 'use' relationship is part of the method in a particular operation/message of a particular class. Figure 3.14 illustrates the method of the operation, 'invoke' for the 'GP System' class.

As mentioned earlier, an object can send a message to itself. Very often, this situation usually occurs when such a message is actually a private message, i.e., it is not an operation/message

in which other objects can invoke. It is useful to mention that such an operation/message is a private one by putting the word 'private' after the operation name in the class structure chart. Programmers who read this message structure chart will immediately know that such operation has to be included in the private message category. This is illustrated in Figure 3.15. These message structure charts are the design specifications which programmers use in order to implement the system.

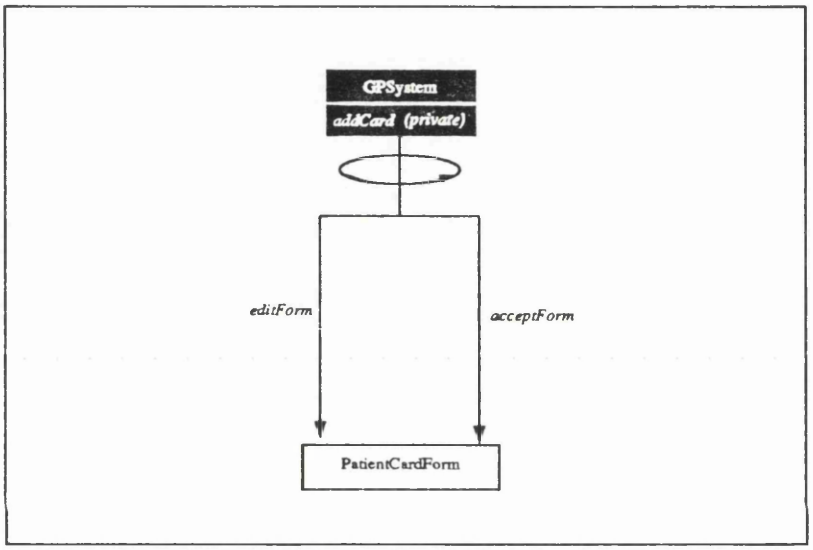


Figure 3.15: An Example of Message Structured Chart with Private Message

3.4. Other Issues

Now that the details of the different levels of the design method developed in this research have been discussed, the following sections talk about some of the issues which are also important in object-oriented systems design.

3.4.1. Data Modelling in Object Oriented Programming

Data modelling plays an important role in traditional system developments. Many software systems require a large and efficient database to store data for the systems. A database system is concerned with the manipulation of data in the system. Besides the fundamental operations on data, such as insertion and deletion, a database system is also responsible for managing the integrity of data, reducing redundancy and avoiding any data inconsistency [Dat75]. Because of its nature, the development of a large database is always treated as a separate design issue from the rest of the system design. The rest of the system design

interacts with the database via the data manipulation language provided by the database system, e.g., SQL [Dat75]. Furthermore, the different arrangements of data in the database always result in different database systems. Generally speaking, they can be categorised into three approaches.

i. the relational approach,

ii. the hierarchical approach and

iii. the network approach.

Each of these approaches has its own good points and bad points. The data modelling process is, therefore, to model the data in a software system using one of these approaches.

One of the general problems found in traditional software is the mismatch between the programming language of the application program and the query language of the database system [Ban88, LRV88]. As Bancilhon says, "Application development requires the communication between a relational query language and a programming language. These two types of languages do not mix well: they have different types, they have different computational models, relational systems are set-at-a-time while programming languages are record-at-a-time. Solving the mismatch requires integrating database and programming language technology" [Ban88].

It has been suggested that the solution of such a problem can be found in object-oriented programming. In fact, intensive research has been carried out in object-oriented databases for the past few years [Ala89, BCG87]. The discussion about the development of object-oriented databases and its important issues are outside the scope of this thesis. However, the important point which is related to this research is that object-oriented programming bridges the gap between application programs and database systems which is found in traditional software systems. In object-oriented databases, data is treated as an object. What used to be the data modelling in the conventional system development process now becomes object modelling. Since object modelling is already part of the system development in an object-oriented software development, one does not need to specially emphasise data analysis and data modelling as found in traditional development methods.

3.4.2. Cohesion and Coupling

In conventional system developments, there are two fundamental elements which contribute to 'good' software. These are:

i. high degree of cohesion,

ii. low degree of coupling.

Cohesion is a measure of the strength of association of the elements inside a unit. A high degree of cohesion means that the elements in that unit are strongly associated with the unit in order to achieve a goal. Coupling is related to cohesion. It is a measure of the interdependence of program units. A low degree of coupling indicates that the program units are highly independent of each other.

In conventional software development, system designers have to work hard in order to ensure their designs exhibit a high degree of cohesion and a low degree of coupling. However, it is found that in object-oriented software development, these two features almost come free with the paradigm [Som89]. In object-oriented software, the basic unit is an object. An object is an entity which contains a set of variables and a set of operations which can act upon it. Since the operations defined are going to be embedded in the object, it naturally demonstrates a high degree of cohesion. Every object is an abstraction in itself. The representation of the object and the implementation of its operations are hidden from external components. Therefore, it displays a low degree of coupling.

3.4.3. Factors for a Good Object-Oriented Design

The main theme in object-oriented software revolves around the construction of classes. Hence the design of the class interface and the class hierarchies are the main factors that contribute to a good object-oriented design.

Concerning the design of the class interface, Lieberherr et al. [LHR88] have defined the Law of Demeter which provokes a good style of object-oriented programming. The Law states that:

For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:

i. the argument classes of M (including C),

ii. the instance variable classes of C.

It is said that by following the Law of Demeter, one can naturally achieve coupling control, information hiding, information restricting, localisation of information, narrow interfaces and structural induction. In fact, the design method of this thesis has included the basic idea of the Law of Demeter in the design phase. The 'use' relationship mentioned in this design method indirectly achieves what the Law of Demeter has proposed. The 'use' relationship mentions that a sender can send messages to its receivers if:

i. the sender physically contains the receiver,

ii. the sender knows the reference of the receiver,

iii. the sender and the receiver are the same object.

Considering the class hierarchy construction, it is believed that a good class hierarchy design should be a design which has many reusable classes. Hence a good design is normally one which contains a fair number of smaller classes and the class hierarchy is deep and narrow. Of course, one should not take it to an extreme. As indicated by Linton et al. [LCV87], a large class hierarchy may overwhelm programmers, especially when there are many levels of subclasses. Users of this class hierarchy have trouble grasping the many different classes and their inherited behaviour. Hence, a deep and narrow class hierarchy may achieve more reuse but it is better to keep the depth level to about seven so that users will not find it difficult to use [Mil56]. Besides a sensible depth, Johnson [Joh88] has identified that a well-developed class hierarchy should have the following characteristics:

i. the top of the class hierarchy should be abstract and

ii. the subclasses should be specialisations.

These issues have been taken into considerations when developing the inheritance factorisation process which helps system designers in constructing class hierarchies. This is discussed in details in Chapter 4.

3.5. Conclusion

What has been described in this chapter is the preliminary framework of a design method which is specially targeted towards object-oriented programming. The design method is divided into three levels:

i. The conceptual level assists system designers to analyse and examine the application in an object-oriented fashion. It is further divided into two layers, the user-interface and the user transparent layers.

ii. The system level concentrates on some of the important issues in the construction of an implementation model. For example, the identification of the implementation objects, the realisation of the 'contain', 'use' and 'inherit' relationships.

iii. The specification level concerns the production of design specifications of the system. These specifications are passed to the programmers to implement the system in the implementation phase.

The design method has also specified the design description language which is used by system designers for communication during the design phase. At the conceptual and the system levels, system designers express their design ideas in object-interaction diagrams. At

the specification level, system designers are requested to specify their design specifications in class structure charts and message structure charts.

Now that the framework of the design method has completed, one can look into the support of handling inheritance in this design method. As it stands, an inheritance factorisation process has been developed to be incorporated into the design method which assists system designers in constructing class hierarchies. The details of this process is discussed in Chapter 4.

"The major point, which almost doesn't need stating, is that you must not do anything which is outside the rules. We might call this restriction the Requirement of Formality."

~ Douglas Hofstadter ~

Chapter 4

The Inheritance Factorisation Process

It has been reiterated several times that inheritance is very important in object-oriented programming. Inheritance enhances reusability of software and provides a simple and elegant organisational discipline for objects. Although inheritance is a useful feature, it is also the subject of a great deal of controversy [MiR87, TGP89]. System designers always encounter problems in constructing class hierarchies. In order to provide a more adequate design method for object-oriented programming, a formal manipulation process which is called the inheritance factorisation process (IFP), has been developed within the design method. The IFP is developed to help system designers in constructing class hierarchies. This manipulation process can be automated so that system designers need only specify the related class specifications and an optimal class hierarchy will be generated.

This chapter is mainly about the inheritance factorisation process. It starts off with a description about the background of the inheritance factorisation process. It then describes the formal model which lies behind the inheritance factorisation process. Once the basic of the formal model is set up, this chapter further explores how such a model can be extended to cater for different inheritance models. It also investigates how the inheritance factorisation process can be used in the system design phase and how to incorporate the process into the design method described in Chapter 3. In addition to this, one of the attractive feature of developing the inheritance factorisation process is that it can be automated. This chapter, hence, also discusses how the process can be automated. The chapter is concluded with a discussion about what exactly IFP is and how the software development in object-oriented programming benefits from it.

4.1. Background

This section starts off with a discussion about the background behind the factorisation process. It discusses some controversial issues concerning inheritance, for example, the difference between the term 'class' and the term 'type'; should inheritance be an implementation issue or a design issue; the various inheritance models in different domains, etc. After that, it describes some of the general problems encountered by system designers in handling inheritance. It defines what the problems are and discusses the attitudes taken in this research towards these problems.

4.1.1. Classes and Types

The term 'class' is introduced in object-oriented programming languages such as Simula and Smalltalk. It is used to denote the specification of classes of objects with common behaviour. The term 'type' has long been known in conventional programming languages such as Pascal. It is used to categorise objects according to their usage and behaviour. As the term 'type' and 'class' are similar, it causes confusion when they co-exist in a programming language.

In weakly-typed object-oriented programming languages such as Smalltalk, the concept of 'type' is transparent. An object is an instance of a class and there is only the class hierarchy to be considered concerning inheritance. However, in strongly-typed object-oriented programming languages such as Eiffel [Mey88] or Solve [RWW88], an object is not only an instance of a class but also associated with a type. For example, in Eiffel, there are two kinds of types in its type system: there are the four simple types, namely integer, boolean, character and real; any other type must be defined by a class declaration and will be called a class type. In this case, there may be two kinds of hierarchies: the class hierarchy and the type hierarchy, i.e., subclassing and subtyping. The difference between these two kinds of hierarchies is that the class hierarchy is defined in terms of template modifications whereas the type hierarchy is defined in terms of constraints that determine a subset of the set defined by the parent predicates [Weg88b]. However, classes can be viewed as a special kind of type. Class-based languages automatically have an associated type system and class hierarchies of object-oriented languages automatically have associated type hierarchies. The concept of 'type' is mainly motivated by type checking. Hence, in this thesis, inheritance is primarily defined as a mechanism for template modification rather than subtyping.

4.1.2. Inheritance as a Design Issue

There is always some concern about whether inheritance should be regarded as a design issue

or an implementation issue, i.e., whether it should be handled in the design phase or the implementation phase. Johnson [Joh88] believes that it is the responsibility of the system designers to plan the class hierarchy to achieve maximum reuse. However, not everyone agrees with this. Brachman [Bra83] suggests that inheritance is purely an implementation issue.

In fact, it is difficult to decide which side gives a correct view. As mentioned earlier, object-oriented programming applies a 'prototyping' software development cycle. Not only is the boundary between the requirement analysis and the design phase not clearly defined but the boundary between the design and the implementation phase is also not well separated. As LaLonde et al. say, "Traditionally, design is segregated as an activity separate from implementation. This has served to clarify the two activities and create two classes of people: designers and programmers. In object-oriented systems, the two groups must be integrated; i.e., a designer-programmer, say to be called design engineers, is needed to effect a proper design; lower level programmers can still be used for the simpler parts" [BRL88]. If one applies such a view about design and implementation phase, inheritance should be taken care of in the design phase. In addition to this, the design method resulting from this research requires that the output of the design phase should be a set of design specifications. These design specifications should have all the implementation details so that when programmers take away the specifications, they can implement the system with confidence. Therefore, it is decided that working on the inheritance hierarchies should be an activity in the design phase.

4.1.3. Inheritance in Different Domains of Discourse

It is generally agreed that the fundamental problem of inheritance is: what exactly is inherited in an inheritance relation. The answer to this problem is, in fact, found to be different in different domains of discourse.

As mentioned in Carnese's thesis [Car84], inheritance systems are used in domains other than general programming. One of the domain in which inheritance is widely used is 'knowledge representation' in artificial intelligence [Hut89, Tou88].

In general, when people talk about inheritance systems, they seldom distinguish which domain they are referring to. However, there are significant differences between inheritance systems in different domains which may affect the answer of 'what exactly is being inherited'. For instance, the main difference between inheritance in general programming and inheritance in knowledge representation lies in the abstractions which are associated with the objects that are defined. In general programming, the abstraction includes two sets: the set of variable names which describes the state of the object and the set of operation names which

describes the operations that can act upon the object. Whereas in knowledge representation, the abstraction encodes the normative information about the object [Tou88]. For example to illustrate this in general programming, the class of elephants may contain variable names such as 'colour', 'weight' and some operations such as 'changeColour' to update the colour of an elephant. In knowledge representation, the class of elephants contains some true facts about an elephants, e.g., 'its colour is grey' and 'it has four legs'.

In fact, this key difference leads to different areas of interest concerning inheritance systems. A classical area of research in inheritance systems in artificial intelligence and database systems is 'how to handle inheritance with exception'. In the above example of the class of elephants, 'its colour is grey' is a piece of information which is generally true. However, there are cases in which the colour of an elephant is white, say if the elephant is a royal elephant. In this case, researchers working on such an inheritance system have to develop mechanisms to handle exception cases like this [Bor88, Tou88]. This problem, however, does not occur in the general programming domain. For example, 'grey' and 'white' are just values of the variable 'colour' and may vary with different elephants.

As one can see, the answer to 'what should one inherit in an inheritance relationship?' depends on which domain one is referring to. In the general programming domain, one inherits the structure of a class and the structure is expressed in a set of variables and a set of operations. As far as this research concerns, it is limited to look at the inheritance systems in the general programming domain. Hence, one tends to inherit the structure of a class.

4.1.4. Inheritance from Different Perspectives

The above section has analysed the different inheritance models in different problem domains. In fact, the inheritance model may vary even in the same problem domain.

In the general programming domain, the subclass is said to inherit the structure of the superclass. The structure of a class is expressed in either variable names or operation names. These names are simply some symbols and the meaning of these symbols depends on how one perceives inheritance. As it is mentioned in Chapter 2, there are basically two perspectives in viewing inheritance: non-strict inheritance and strict inheritance [Sak89, Sny87, Weg88b].

Non-strict inheritance simply implies code-sharing. The main objective of non-strict inheritance is to reuse as much of existing implementation as possible. With such an inheritance model, the symbol which describes an operation denotes a particular piece of code, i.e., one is inheriting a piece of code. Non-strict inheritance tends to encourage a casual attitude

towards class hierarchies constructions. As the main purpose of this inheritance model is to reuse as much code as possible, it allows two conceptually unrelated classes to be put in the same hierarchy. It may even allow an inheritance relationship to be established between two classes provided there is one common piece of code that these two classes can share.

Strict inheritance not only concerned with inheriting the code but also the formal meaning associated with the class. By formal meaning, one refers to the abstract data type specification description which incorporates the sort, the signature and the equation of the class. With such an inheritance model, the subclass not only inherits the code but also the formal semantic of the structure of its superclass. Further, strict inheritance does not allow a casual inheritance relationship to be set up. A subclass has to share all the properties defined in its superclass.

When determining which perspective should be considered in the inheritance factorisation model, it is found that whether it is the piece of code and/or the formal semantics that is inherited is not that important. By varying the representation of an attribute and a class specification, the inheritance factorisation process can support various kinds of inheritance models. The details of this is discussed in Section 4.3. What is more important is to discourage the casual construction of class hierarchies. Hence, an inheritance relationship cannot be established if two classes only share a few properties. A subclass can only be created if the set of attributes that describes its structure is a proper superset of attributes of its superclass.

4.1.5. The Problem in Constructing Class Hierarchies

In object-oriented programming, software reuse is mainly attained via inheritance. In order to achieve maximum reuse, the fundamental step is to construct a well-defined class hierarchy that reflects maximum reuse.

The construction of a class hierarchy probably seems trivial on the surface. When system designers have to construct a class hierarchy from two related classes, class A and class B, what they usually do is to factorise out the common properties between these two classes to construct the superclass, class AB [Boo86, Bor88, Weg88b]. This 'ad hoc' method of constructing class hierarchies seems to work fine when there are only a small number of classes involved and when the description of the classes are specified in a monotonic incremental fashion. For example, suppose one is asked to form a hierarchy between two classes, 'Point' and 'HistoryPoint' which are specified as below†:

† Details for this example can be found in Carnese's thesis [Car84].

Point: *create, location, move, display;*
HistoryPoint: *create, location, move, display, history;*

In this case, system designers can easily identify that the class 'Point' is going to be the superclass and the class 'HistoryPoint' is its subclass.

However, the problem of constructing class hierarchies becomes non-trivial when it involves many classes and even more prominent if the classes are specified in a way which is not monotonic incremental. For example, if a class hierarchy has to be constructed not only from the class 'Point' and the class 'HistoryPoint' but also with two other classes, 'BoundedPoint' and 'BhPoint' which are specified as follows:

BoundedPoint: *create, move, display, location, max, min;*
BhPoint: *create, move, display, location, history, max, min, boundHistory;*

In this case, the 'ad hoc' method alone is beginning to be inadequate to construct the class hierarchy successfully. System designers may find that they have to try several times before a satisfactory solution is obtained. In order to provide a better way for system designers to construct class hierarchies, a more 'algorithmic' approach to handle the construction of class hierarchies is introduced in this research. A mechanism called the 'inheritance factorisation process' (IFP) has been developed to assist system designers to construct class hierarchies.

The IFP has been developed to assist system designers in generating an optimal class hierarchy in a quicker and more efficient way than using the 'ad hoc' method. The class hierarchies generated from the IFP accentuates a hierarchical organisation of maximum reusability and hence minimum duplication of common properties defined amongst the classes in the hierarchy. It is believed that with the IFP, system designers need only specify the conceptual class specifications involved in a hierarchy and an optimal class hierarchy graph will be generated.

4.2. The Algebraic Model for the Inheritance Factorisation Process

The manipulation process provided in the IFP is based on an algebraic structure which is specially defined for constructing class hierarchies. The fundamental principle of the IFP is, in fact, inspired by the 'ad hoc' method described in the previous section. If one examines the 'ad hoc' method more closely, it is easy to see that the essence of constructing class hierarchies is to identify the appropriate superclasses and the corresponding 'inherit' relationship. The identification of superclasses involves factoring out common attributes amongst the involved classes [Boo86, Bor88, Weg88b]. The factorised attributes are then grouped together to form the superclass. The IFP uses this idea as the basis and extends

it further to obtain the basic framework of a systematic manipulation process. This section serves to discuss the formal aspects which lie behind the IFP†.

4.2.1. Basic Assumptions

It is assumed that the inheritance model in the inheritance factorisation process has the following characteristic:

Assumption 4.1

Inheritance with cancellation is not allowed in the inheritance model of the IFP. Hence a subclass has to inherit all the attributes defined in its superclass.

Assumption 4.2

An optimal class hierarchy is one which refers to maximum reusability of properties defined in the superclass, i.e., minimum replication of properties in class specifications.

Assumption 4.3

The inheritance model applied to the IFP is only suitable for use in the general programming languages domain.

4.2.2. Attributes

Definition 4.1

An *attribute* is a symbol which denotes a feature or a property found in a class specification. In general programming languages domain, the label which describes the state of an object or the operation which can act upon an object is regarded as an attribute.

It is not enough just to look at the label of an attribute, it is also necessary to examine the semantic of the attribute to determine its class hierarchy. However, the semantic of an attribute varies with different inheritance models. This is further discussed in Section 4.3.

Assumption 4.4

To simplify the model, it is assumed that there exists a mechanism to check the semantic equivalence of two attributes. When system designers use the same label for two attributes, these two attributes are said to be semantically equivalent.

† A summary of this section can be found in [PuW90].

Assumption 4.5

The ordering of the attributes in a class specification is not significant. For example, the class 'PointA' is the same as the class 'PointB' in the following example.

PointA: *create, display, location, move;*
PointB: *display, location, create, move;*

For most programs this assumption is completely valid; the order of members of a data structure can be rearranged, the program recompiled and it will work just as before. However, there is a class of programs – low level systems software – for which this assumption might not be valid. An example might be a disc driver. The control/status register structure is fixed by the hardware so the order of the members of the data structure describing this is fixed. However, it is believed that the way in which an object-oriented program would deal with this would not involve exhibiting this fixed structure at the language level.

Definition 4.2

The *universe of attributes*, $\mathcal{A}$, is the set of all attributes.

$$\mathcal{A} = \{a_i\}$$

4.2.3. Class Specification

Definition 4.3

A *class specification* is a set of mutually exclusive attributes.

$$c_i = \{a_1, a_2, a_3, \ldots, a_n\}$$

where $a_i \in \mathcal{A}$ and $c_i \in \mathcal{P}(\mathcal{A})$.

Definition 4.4

The power set of the universe of attributes, $\mathcal{P}(\mathcal{A})$, is the set of all possible subsets of $\mathcal{A}$. In the model, $\mathcal{P}(\mathcal{A})$ represents the *universe of class specifications* and is denoted by $\mathcal{C}$. $\phi_c \in \mathcal{C}$ is the empty class specification, i.e., the class specification with no attributes.

Definition 4.5

A *conceptual class specification* is a class specification which contains all the attributes necessary to describe the class. There is no inheritance to be taken into account.

Definition 4.6

An *implementation class specification* is a class specification which contains only the attributes over and above those that will be drawn from superclasses. It is assumed that the inheritance relations are known in order to be able to construct the conceptual class specification from the implementation class specification.

Example 4.1

The following example highlights the difference between conceptual and implementation class specifications:

The conceptual class 'vehicle' contains attributes age and maxspeed. The conceptual class 'car' contains attributes age, maxspeed and fuel. If the class 'car' inherits from the class 'vehicle' then the implementation class 'i-vehicle' is the same as the conceptual class 'vehicle' and the implementation class 'i-car' contains only the attribute fuel.

Definition 4.7

The power set of the universe of class specifications, $\mathcal{P}(\mathcal{P}(\mathcal{A}))$, is the set of all possible subsets of $\mathcal{C}$. $\mathcal{P}(\mathcal{P}(\mathcal{A}))$ is the *universe of all possible sets of class specifications* and is denoted by $\mathcal{S}$. $\phi_s \in \mathcal{S}$ is the empty set of class specifications.

4.2.4. The Class Hierarchy Construction Problem

Definition 4.8

An *initial set of class specifications*, $s_i \in \mathcal{S}$, contains all the conceptual class specifications involved in a particular class hierarchy construction problem.

Definition 4.9

A *class hierarchy construction problem* for an initial set of class specifications, s_i, is to generate the corresponding optimal class hierarchy which reflects maximum reusability of attributes and hence minimum duplication of attributes for the set s_i. The optimal class hierarchy which is generated is an implementation class hierarchy. The inheritance relationship identified in this hierarchy are between implementation superclasses and subclasses.

4.2.5. Axioms for the Inheritance Factorisation Process

This subsection describes the inheritance factorisation process and presents the essential axioms and operators. The characteristic of the operators are discussed here as well.

Axiom 4.1

Individual sets of class specifications, $s_1, \cdots, s_n$ can be grouped into a particular set, s'.

$$s' = s_1 + \cdots + s_n$$

or in terms of class specifications, it is

$$\{c_1, c_2, \cdots, c_n\} = \{c_1\} + \{c_2\} + \cdots + \{c_n\}$$

The combine operator, +, can be viewed as the set union operator on any set of class specifications, s_i.

The signature of the combine operator, +, is,

$$+ : \quad \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$$

or in terms of $\mathcal{A}$, it is:

$$+ : \quad \mathcal{P}(\mathcal{P}(\mathcal{A})) \times \mathcal{P}(\mathcal{P}(\mathcal{A})) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{A}))$$

Since the combine operator, +, is in fact the set union operator, it automatically possesses the properties of the set union operator. Hence, the combine operator, +, is associative, commutative and has an identity, ϕ_s.

$$\{c_0\} + \{c_1\} = \{c_1\} + \{c_0\}$$

and

$$\{c_0\} + (\{c_1\} + \{c_2\}) = (\{c_0\} + \{c_1\}) + \{c_2\}$$

and

$$\phi_s + \{c_0, \cdots, c_n\} = \{c_0, \cdots, c_n\} + \phi_s = \{c_0, \cdots, c_n\}$$

Axiom 4.2

Common attributes in a combinator subexpression can be factorised out to the left and form the superclass of the original classes. Attributes which are common in the most number of classes should be factorised out first.

$$\{c_1\} + \cdots + \{c_n\} = c_0' \lhd_s (\{c_1'\} + \cdots + \{c_n'\})$$

$$iff \quad \forall a_i. \quad a_i \in c_0' \rightarrow a_i \in c_1 \wedge \cdots \wedge a_i \in c_n$$

The semantics of this operation are in terms of set union and set intersection:

$$\{c_1\} + \cdots + \{c_n\} = c_0' \lhd_s (\{c_1'\} + \cdots + \{c_n'\})$$

$$iff \quad c_0' = \bigcap_{i=1}^{n} c_i \quad and \quad c_1 = c_0' \cup c_1', \cdots, c_n = c_0' \cup c_n'$$

This axiom introduces the *single inherit operator*, $\triangleleft_s$. This operator describes the single inheritance relationship. Here, the axiom shows that the class specification c_1' inherits from c_0' to give c_1, c_2' inherits from c_0' to give c_2, etc. At a first glance, the single inherit operator, $\triangleleft_s$, appears to be simply the set union operator on a set of attributes. However, the single inherit operator, $\triangleleft_s$, is more than this. It identifies which is the superclass and which is the subclass via its signature. In the example above, c_0' is the superclass specification. $c_1', \cdots, c_n'$ are the implementation subclass specifications and $c_1, \cdots, c_n$ are the conceptual subclass specifications.

The signature of the single inherit operator, $\triangleleft_s$, is:

$$\triangleleft_s : \quad \mathcal{C} \times \mathcal{S} \rightarrow \mathcal{S}$$

or in terms of $\mathcal{A}$,

$$\triangleleft_s : \quad \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{P}(\mathcal{A})) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{A}))$$

Proposition 4.1

The single inherit operator, $\triangleleft_s$, is non-commutative.

Proof:

This follows from the fact that the right and left operands are from different sets, hence the single inherit operator, $\triangleleft_s$, is non-commutative.

QED

It is important that the single inherit operator, $\triangleleft_s$, is non-commutative since it is this that distinguishes a superclass from a subclass in a subexpression.

Proposition 4.2

The single inherit operator, $\triangleleft_s$, is non-associative.

Proof:

Again, the signature of the single inherit operator, $\triangleleft_s$, implies the non-associative property of this operator.

QED

Proposition 4.3

The empty class specification, ϕ_c, is the left identity of the single inherit operator, $\triangleleft_s$.

$$\phi_c \triangleleft_s \{c_1, \cdots, c_n\} = \{c_1, \cdots, c_n\}$$

Proof:

If $\phi_c \lhd_s \{c_1, \cdots, c_n\} = \{c_1, \cdots, c_n\}$ is false then ϕ_c contains at least one attribute. Since ϕ_c is an empty class specification, there is a contradiction.

QED

Proposition 4.4

A set of class specifications which contains only the empty class specification, $\{\phi_c\}$, inherits from a class specification, c_i, always gives a set of class specification that contains one element $\{c_i\}$.

$$c_i \lhd_s \{\phi_c\} = \{c_i\}$$

Proof:

If $c_i \lhd_s \{\phi_c\} = \{c_i\}$ is false, then ϕ_c must contains at least one attribute. Since ϕ_c is an empty class specification, there is a contradiction.

QED

Proposition 4.5

An empty set of class specification, ϕ_s, inherits from a class c_i giving a set of class specifications that contains only one element c_i.

$$c_i \lhd_s \phi_s = \{c_i\}$$

Proof:

If $c_i \lhd_s \phi_s = \{c_i\}$ is false, then ϕ_s contains at least one class specification. Since ϕ_s is an empty set of class specifications, there is a contradiction.

QED

It should be noted that although such an expression is valid, it is redundant and would not normally occur in an inheritance expression.

With these axioms and propositions, one can handle any class hierarchy construction problems which involves only single inheritance. However, multiple inheritance has become more and more popular these days. Multiple inheritance allows a subclass to have more than one superclass. To support multiple inheritance in the inheritance factorisation model, another axiom is needed.

Axiom 4.3

For multiple inheritance, the subclass is inherited from two or more superclasses.

$$\{c_0\} = (\{c_1'\} + \cdots + \{c_n'\}) \lhd_m c_0'$$

$$iff \quad n > 1 \quad and \quad \forall a_i . \cdots a_m . \quad a_i \in c_1' \wedge \cdots \wedge a_m \in c_n' \rightarrow a_i, \cdots, a_m \in c_0$$

This axiom can also be looked at in terms of set operators.

$$\{c_0\} = (\{c_1'\} + \cdots + \{c_n'\}) \lhd_m c_0' \quad iff \quad c_0 = \bigcup_{i=0}^{n} c_i' \quad and \quad n > 1$$

By definition, the result obtained from a multiple inheritance subexpression is always a singleton set, s_i, containing only one class specification.

Axiom 4.3 introduces the *multiple inherit operator*, $\lhd_m$, which describes the multiple inheritance relationship. As with the single inheritance operator, the multiple inherit operator distinguishes which are the superclasses and which is the subclass in the inheritance expression. For example,

$$\{c_3\} = \{c_1', c_2'\} \lhd_m c_3'$$

Here, the class specification c_3' multiply inherits from the class specifications c_1' and c_2' to give c_3. c_1', c_2' and c_3' are the implementation class specifications. c_3 is a conceptual class specification.

The signature of the multiple inherit operator, $\lhd_m$, is:

$$\lhd_m : \quad \mathcal{S} \times \mathcal{C} \rightarrow \mathcal{S}$$

or in terms of $\mathcal{A}$,

$$\lhd_m : \quad \mathcal{P}(\mathcal{P}(\mathcal{A})) \times \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{A}))$$

Proposition 4.6

The multiple inherit operator, $\lhd_m$, is non-commutative.

Proposition 4.7

The multiple inherit operator, $\lhd_m$, is non-associative.

Proof:

Again, the signature of the multiple inherit operator, $\lhd_m$, implies the non-commutative and non-associative properties of the operator. ***QED***

Proposition 4.8

A class specification, c_i, multiple inheriting from an empty set of class specifications gives a set of class specifications containing only c_i.

$$\phi_s \lhd_m c_i = \{c_i\}$$

Proof:

If $\phi_s \lhd_m c_1 = \{c_1\}$ is false then ϕ_s contains at least one class specification. Since ϕ_s is an empty set of class specifications, there is a contradiction.

QED

Such an expression is valid but redundant and would not normally occur in the manipulation process.

4.2.6. Class Hierarchy Expression

Definition 4.10

A *class hierarchy expression*, ξ, is an expression constructed from any sets of class specifications, the combine operator, $+$, the single inherit operator, $\lhd_s$, and the multiple inherit operator, $\lhd_m$. Parentheses are used to group into subexpressions. For example,

$$\xi = c_0 \lhd_s (\{c_1\} + \{c_2\})$$

Definition 4.11

An *initial class hierarchy expression*, ξ_I, of a particular class hierarchy construction problem, is a class hierarchy expression constructed from the corresponding initial set of class specifications, s_i, alone.

Definition 4.12

A *normalised class hierarchy expression*, ξ_N, is a class hierarchy expression to which no further factorisation can be applied (See Definition 4.13). The interpretation of this expression is that it provides maximum reusability and minimum duplication of attributes. For example,

$$\xi_N = \{a_1, a_2\} \lhd_s (\{\{a_3, a_4\}\} + \{\{a_5\}\})$$

IFP =

Sorts : $\mathcal{A}$
$\mathcal{P}(\mathcal{A})$
$\mathcal{P}(\mathcal{P}(\mathcal{A}))$

Opns : $+ : \mathcal{P}(\mathcal{P}(\mathcal{A})) \times \mathcal{P}(\mathcal{P}(\mathcal{A})) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{A}))$
$\lhd_s : \mathcal{P}(\mathcal{A}) \times \mathcal{P}(\mathcal{P}(\mathcal{A})) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{A}))$
$\lhd_m : \mathcal{P}(\mathcal{P}(\mathcal{A})) \times \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{A}))$

Eqns : $c_0, c_1, \cdots, c_n, c_0', \cdots, c_n', \phi_c \in \mathcal{P}(\mathcal{A})$
$\{c_0\}, \cdots, \{c_n\}, \{c_0, \cdots, c_n\}, \phi_s \in \mathcal{P}(\mathcal{P}(\mathcal{A}))$

1. $\{c_0\} + \cdots + \{c_n\} = \{c_0, \cdots, c_n\}$
2. $\{c_1\} + \cdots + \{c_n\} = c_0' \lhd_s (\{c_1'\} + \cdots + \{c_n'\})$
   iff $\forall a_i.\ \ a_i \in c_0' \rightarrow a_i \in c_1 \wedge \cdots \wedge a_i \in c_n$
   i.e., $c_0' = \bigcap_{i=1}^{n} c_i$ *and* $c_1 = c_0' \cup c_1', \cdots, c_n = c_0' \cup c_n'$
3. $\{c_0\} = (\{c_1'\} + \cdots + \{c_n'\}) \lhd_m c_0'$ *where* $n > 1$
   iff $\forall a_i. \cdots a_m.\ \ a_i \in c_1' \wedge \cdots \wedge a_m \in c_n' \rightarrow a_i, \cdots, a_m \in c_0$
   i.e., $c_0 = \bigcup_{i=0}^{n} c_i'$
4. $\{c_0\} + \{c_1\} = \{c_1\} + \{c_0\}$
5. $\{c_0\} + (\{c_1\} + \{c_2\}) = (\{c_0\} + \{c_1\}) + \{c_2\}$
6. $\{c_0\} + \phi_s = \phi_s + \{c_0\} = \{c_0\}$
7. $\phi_c \lhd_s \{c_1, \cdots, c_n\} = \{c_1, \cdots, c_n\}$
8. $c_1 \lhd_s \phi_s = \{c_1\}$
9. $c_1 \lhd_s \{\phi_c\} = \{c_1\}$
10. $\{c_1, \cdots, c_n\} \lhd_m \phi_c = \{c_0\}$ *where* $c_0 = \bigcup_{i=1}^{n} c_i$

Figure 4.1: The Algebraic Structure of the IFP

Figure 4.1 is a summary of the algebraic structure of the IFP. Although there are other interesting mathematical properties concerning the algebraic structure of the IFP, it is outside the scope of this thesis to investigate them all. The main objective of developing the algebraic structure is to support the development of a tool to assist system designers in constructing class hierarchies. The structure as presented above is sufficient for this purpose so further investigation will not be carried out here.

Definition 4.13

The *inheritance factorisation process* is a process which takes an initial class hierarchy expression and applies only the definitions, propositions and Axiom 4.1 to Axiom 4.3 for manipulation. The factorisation process continues until a normalised class hierarchy expression is obtained.

4.2.7. Detecting Multiple Inheritance

Handling multiple inheritance in the IFP is not quite as straight forward as handling single

inheritance. First of all, one needs to know whether multiple inheritance exists in the class hierarchy construction problem and then transform the current subexpression into one which reflects the multiple inherit relationship. The following is a summary of how multiple inheritance is detected in the manipulation process and what should one do when such a situation comes up.

During the inheritance factorisation process using only single inheritance, one may encounter subexpressions in which common attributes can be factorised in two or more different ways. For example,

$$\begin{aligned}
\xi &= \{a_1\} \lhd_s (\{\{a_2, a_3, a5\}\} + \{\{a_3, a_6, a_8\}\} + \{\{a_4, a_6, a_8\}\} + \{\{a_2, a_7\}\}) \\
&= \{a_1\} \lhd_s (\{a_3\} \lhd_s (\{\{a_2, a5\}\} + \{\{a_6, a_8\}\}) + \{\{a_4, a_6, a_8\}\} + \{\{a_2, a_7\}\}) \quad [\xi_{N1}] \\
\text{or} &= \{a_1\} \lhd_s (\{a_2\} \lhd_s (\{\{a_3, a5\}\} + \{\{a_7\}\}) + \{a_6, a_8\} \lhd_s (\{\{a_3\}\} + \{\{a_4\}\})) \quad [\xi_{N2}]
\end{aligned}$$

Here, there are two valid ways to factorise out common attributes. This, in fact, indicates that multiple inheritance has been detected.

When multiple inheritance is detected, one can identify the superclasses involved in each multiple inheritance relationship. In the above example, from the expressions ξ_{N1} and ξ_{N2}, one can deduce that $\{a_3\}, \{a_2\}$ will be the superclasses involved in the multiple inheritance relationship of the set of class specification $\{\{a_2, a_3, a_5\}\}$, and $\{a_3\}, \{a_6, a_8\}$ will be the superclasses involved in the multiple inheritance relationship of the set of class specification $\{\{a_3, a_6, a_8\}\}$. Now, instead of factorising out the common attributes, one has to expand the subexpression to capture the corresponding single inheritance and multiple inheritance subexpressions. The steps to follow are:

For each individual class specification involved in the subexpression, do the following:

1. If the class specification can inherit from more than one class specification, transform it to a subexpression which involves the multiple inherit operator.
2. Otherwise, transform it to a subexpression which involves the single inherit operator.

Hence, the resulted expression from the above example will be:

$$\begin{aligned}
\xi_N = &\{a_1\} \lhd_s ((\{\{a_2\}\} + \{\{a_3\}\}) \lhd_m \{a_5\} + (\{\{a_3\}\} + \{\{a_6, a_8\}\}) \lhd_m \phi_c + \{a_6, a_8\} \lhd_s \{\{a_4\}\} \\
&+ \{a_2\} \lhd_s \{\{a_7\}\})
\end{aligned}$$

4.2.8. Class Hierarchy Graphs

The above discussion provides a manipulation process which allows one to form an initial class hierarchy expression from the set of class specifications involved in a particular class

hierarchy construction problem and transform it into a normalised class hierarchy expression. Although this normalised class hierarchy expression reflects the single inheritance and multiple inheritance relationships in the optimal class hierarchy, these relations are often best expressed in a graphical form [Bol79, BoM76]. This section describes how to transform a normalised class hierarchy expression to its normalised class hierarchy graph.

Definition 4.14

A *graph*, G, is a pair $G = (V, E)$ where V is a finite set of vertices and E is a set of unordered pairs of distinct vertices.

Definition 4.15

A *directed graph* is a pair $G = (V, E)$, where V is a finite set of vertices and E is a set of ordered pairs of vertices.

Definition 4.16

A *class hierarchy graph* is a directed graph, $G_h = (V, E)$ where

$$V = \{v_1, \cdots v_n\}$$

$$f: V \rightarrow \mathcal{C}$$

$$E = \{(v_i, v_j)\} \quad where \quad v_i, v_j \in V$$

$$and \quad f(v_i) \lhd_s s_j \quad and \quad f(v_j) \in s_j \quad where \quad s_j \in \mathcal{S}$$

$$or \quad s_i \lhd_m f(v_j) \quad and \quad f(v_i) \in s_i \quad where \quad s_i \in \mathcal{S}$$

The normalised class hierarchy graph stems from the normalised class hierarchy expression which is generated from the IFP. Such a class hierarchy graph is for implementation and is different from the conceptual class hierarchy graph.

Obtaining the set of vertices is straight forward. One just need to gather the individual class specifications found in the normalised class hierarchy expression.

To obtain the set of edges is only straight forward if the normalised class hierarchy expression is simple. For example, a trivial normalised class hierarchy expression such as, $c_0' \lhd_s (\{c_1'\} + \{c_2'\})$, can be transformed to $c_0' \lhd_s \{c_1', c_2'\}$. Hence from the semantics of the single inherit operator and the definition of the class hierarchy graph, one can deduce the set of vertices, V, and the set of edges, E, are:

$$V = \{c_0', c_1', c_2'\}$$

$$E = \{(c_0', c_1'), (c_0', c_2')\}$$

Hence, the class hierarchy graph, $G_h = (V, E)$ is generated.

However, most of the normalised class hierarchy expressions are more complicated. They are made up of various kinds of subexpressions. In this case, one needs to have a function, $\Psi(\xi_N)$, which takes in the normalised class hierarchy expression and generates the set of vertices and the edges for the graph. The following is an overview of the algorithm applied by the function, $\Psi(\xi_N)$.

Algorithm 4.1

In order to obtain the set of edges from the class hierarchy expression, one need to define a grammar which acts as the basis for a syntax analyser of the expression. The syntax analyser recognises the elements of a class hierarchy expression and generates the set of edges accordingly. The following shows the grammar of the syntax analyser.

ClassHierExpr	:	*ClassSpec* '$\triangleleft_s$' *SubExpr* '$'
	|	*SubExpr* '$\triangleleft_m$' *ClassSpec* '$'
SubExpr	:	'(' *SubExpr* ')'
	|	*SubExpr* '+' *SubExpr*
	|	*ClassSpec* '$\triangleleft_s$' *SubExpr*
	|	*SubExpr* '$\triangleleft_m$' *ClassSpec*
	|	'{' *ClassSpec* '}'
ClassSpec	:	'{' *attributes* '}'
attributes	:	*attributes* ',' *ATTRIBUTE*
	|	'{' *ATTRIBUTE* '}'

Figure 4.2: The Grammar for the Parser

The grammar above shows that the symbol '$' is the termination symbol. A class hierarchy expression is made up of subexpressions. A subexpression can be one which contains the combine operator and/or the single inherit operator and/or the multiple inherit operator. Each of these subexpressions has a list of active class specifications attached to it. An active class specification is a class specification which would be involved in the generation of edges once a single/multiple subexpression is recognised.

The whole objective of the syntax analyser is to recognise the different types of subexpressions and perform the necessary tasks. This is described by the following.

1. *ClassSpec* '$\triangleleft_s$' *SubExpr*

   When a subexpression with a single inherit operator, i.e., the single inherit

subexpression is recognised, a set of edges, $(v_i, v_{j_1}), (v_i, v_{j_2}), \cdots, (v_i, v_{j_n})$ are generated in which v_i is the class specification on the left hand side of the single inherit operator and $v_{j_1}, v_{j_2}, \cdots, v_{j_n}$ are the active class specifications of the SubExpr on the right hand side of the single inherit operator.

$$\underbrace{ClassSpec}_{v_i} \text{ '}\lhd_s\text{' } \underbrace{SubExpr}_{v_{j_1},\cdots,v_{j_n}}$$

Such a single inherit subexpression is then reduced to another SubExpr itself. The list of active class specifications attached to this resulting SubExpr contains only v_i because semantically v_i is the superclass of $v_{j_1}, v_{j_2}, \cdots, v_{j_n}$ and the graph is constructed in a bottom-up fashion, hence what is involved in the next construction is the superclass in this single inherit subexpression.

2. *SubExpr* '$\lhd_m$' *ClassSpec*

   When a subexpression involves a multiple inherit operator, i.e., the multiple inherit subexpression is recognised, a set of edges $(v_{i_1}, v_j), \cdots, (v_{i_n}, v_j)$ are generated in which $v_{i_1}, \cdots, v_{i_n}$ are the active class specifications from the SubExpr on the left hand side of the multiple inherit operator and v_j is the ClassSpec on the right of the operator.

$$\underbrace{SubExpr}_{v_{i_1},\cdots,v_{i_n}} \text{ '}\lhd_m\text{' } \underbrace{ClassSpec}_{v_j}$$

   Such a multiple inherit subexpression is then reduced to another SubExpr itself. The list of active class specifications attached to this resulting SubExpr contains all the superclasses $v_{i_1}, \cdots, v_{i_n}$ derived from the multiple inherit subexpression.

3. *SubExpr* '+' *SubExpr*

   When a subexpression involves a combine operator, i.e., the combine subexpression is recognised, it is reduced to another SubExpr. The list of active class specifications of the new SubExpr is simply a concatenation of the active class specifications found in the two SubExprs of the combine subexpression.

4. '{' *ClassSpec* '}'

   'ClassSpec' is a class specification. When a ClassSpec is recognised, it is reduced to a SubExpr with a list of active specifications containing only that class specification.

4.2.9. Examples to use the IFP

Now that the details of the formal manipulation process have been presented, it is time to give some examples showing how the manipulation process is used to construct the class hierarchy

graph. This section presents two examples that show how the IFP is applied to different class hierarchy construction problems. The examples also illustrate how the normalised class hierarchy expression obtained is transformed to a graph. Example 4.2 shows its action for single inheritance and Example 4.3 shows its action for multiple inheritance. More concrete examples of the usage of the IFP can be found in Chapter 6.

Example 4.2

In this example, a class hierarchy is constructed for the class specifications set, $s_i = \{c_0, c_1, c_2, c_3\}$ where $c_0 = \{a_1, a_2, a_3, a_4, a_5\}$, $c_1 = \{a_1, a_9, a_{10}\}$, $c_2 = \{a_1, a_2, a_8\}$ and $c_3 = \{a_1, a_2, a_6, a_7\}$, the inheritance factorisation process will be:

$$
\begin{aligned}
& \xi_I \\
= \quad & \{c_0, c_1, c_2, c_3\} \\
= \quad & \{c_0\} + \{c_1\} + \{c_2\} + \{c_3\} \\
= \quad & \{\{a_1, a_2, a_3, a_4, a_5\}\} + \{\{a_1, a_9, a_{10}\}\} + \{\{a_1, a_2, a_8\}\} + \{\{a_1, a_2, a_6, a_7\}\} \\
= \quad & \{a_1\} \lhd_s (\{\{a_2, a_3, a_4, a_5\}\} + \{\{a_9, a_{10}\}\} + \{\{a_2, a_8\}\} + \{\{a_2, a_6, a_7\}\}) \\
= \quad & \{a_1\} \lhd_s (\{\{a_9, a_{10}\}\} + (\{a_2\} \lhd_s (\{\{a_3, a_4, a_5\}\} + \{\{a_8\}\} + \{\{a_6, a_7\}\}))) \qquad [\xi_N]
\end{aligned}
$$

By introducing $c_4' = \{a_1\}$, $c_1' = \{a_9, a_{10}\}$, $c_5' = \{a_2\}$, $c_0' = \{a_3, a_4, a_5\}$, $c_2' = \{a_8\}$ and $c_3' = \{a_6, a_7\}$ as the implementation class specifications of this hierarchy, the normalised class hierarchy expression, ξ_N, can be re-written as:

$$\xi_N = c_4' \lhd_s (\{c_1'\} + (c_5' \lhd_s (\{c_0'\} + \{c_2'\} + \{c_3'\})))$$

The set of vertices for the hierarchy graph is,

$$V = \{c_0', c_1', c_2', c_3', c_4', c_5'\}$$

and the set of edges, E, is

$$E = \{(c_4', c_1'), (c_4', c_5'), (c_5', c_0'), (c_5', c_2'), (c_5', c_3')\}$$

The corresponding class hierarchy graph is shown in Figure 4.3.

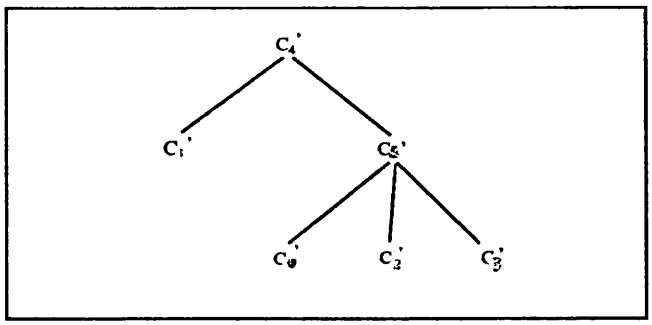


Figure 4.3: The Class Hierarchy Graph of Example 4.2

Example 4.3

In this example, a class hierarchy is constructed for the set of class specifications, $s_i = \{c_0, c_1, c_2, c_3\}$. If $c_0 = \{a_1, a_2, a_3, a_5\}$, $c_1 = \{a_1, a_3, a_6, a_8\}$, $c_2 = \{a_1, a_4, a_6, a_8\}$ and $c_3 = \{a_1, a_2, a_7\}$ then the factorisation process will be:

$$
\begin{aligned}
& \xi_i \\
= \; & \{c_0, c_1, c_2, c_3\} \\
= \; & \{c_0\} + \{c_1\} + \{c_2\} + \{c_3\} \\
= \; & \{\{a_1, a_2, a_3, a_5\}\} + \{\{a_1, a_3, a_6, a_8\}\} + \{\{a_1, a_4, a_6, a_8\}\} + \{\{a_1, a_2, a_7\}\} \\
= \; & \{a_1\} \lhd_s (\{\{a_2, a_3, a_5\}\} + \{\{a_3, a_6, a_8\}\} + \{\{a_4, a_6, a_8\}\} + \{\{a_2, a_7\}\}) \\
= \; & \{a_1\} \lhd_s (\{a_3\} \lhd_s (\{\{a_2, a_5\}\} + \{\{a_6, a_8\}\}) + \{\{a_4, a_6, a_8\}\} + \{\{a_2, a_7\}\}) \quad [\xi_{N1}] \\
= \; & \{a_1\} \lhd_s (\{a_2\} \lhd_s (\{\{a_3, a_5\}\} + \{\{a_7\}\}) + \{a_6, a_8\} \lhd_s (\{\{a_3\}\} + \{\{a_4\}\})) \quad [\xi_{N2}]
\end{aligned}
$$

Now since two valid normalised expessions are obtained, we know that multiple inheritance has been detected. In this case, Axiom 4.3 is used to obtain the appropriate expression.

$$
\begin{aligned}
\xi_N = & \{a_1\} \lhd_s ((\{\{a_2\}\} + \{\{a_3\}\}) \lhd_m \{a_5\} + (\{\{a_3\}\} + \{\{a_6, a_8\}\}) \lhd_m \phi_c + \{a_6, a_8\} \lhd_s \{\{a_4\}\} \\
& + \{a_2\} \lhd_s \{\{a_7\}\})
\end{aligned}
$$

By introducing $c_4' = \{a_1\}$, $c_5' = \{a_2\}$, $c_6' = \{a_3\}$, $c_0' = \{a_5\}$, $c_7' = \{a_6, a_8\}$, $c_2' = \{a_4\}$, $c_3' = \{a_7\}$, as the implementation class specifications of this hierarchy, the normalised class hierarchy expression, ξ_N, can be re-written as:

$$
\xi_N = c_4' \lhd_s ((\{c_5'\} + \{c_6'\}) \lhd_m c_0' + (\{c_6'\} + \{c_7'\}) \lhd_m \phi_c + c_7' \lhd_s \{c_2'\} + c_5' \lhd_s \{c_3'\})
$$

The following set of vertices and edges are obtained:

$$
V = \{c_4', c_5', c_6', c_0', c_7', c_2', c_3, \phi_c'\}
$$

$$E = \{(c_4', c_5'), (c_4', c_6'), (c_4', c_7'), (c_5', c_0'), (c_6', c_0'), (c_6', \phi_c), (c_7', \phi_c), (c_5', c_3'), (c_7', c_2')\}$$

The corresponding class hierarchy graph is shown in Figure 4.4.

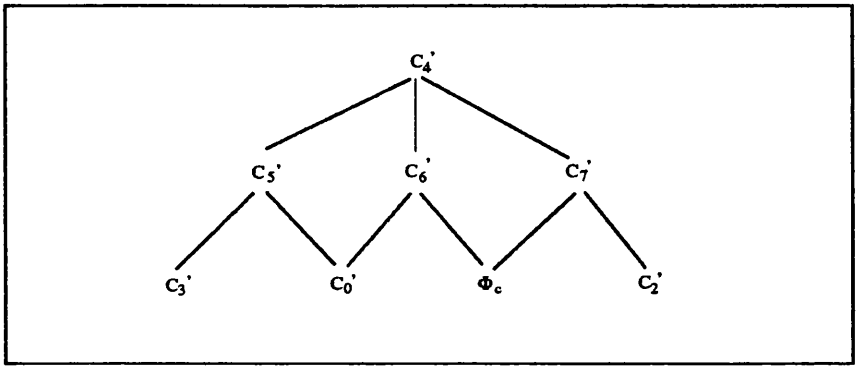


Figure 4.4: The Multiple Inheritance Graph for Example 4.3

4.3. The IFP in Various Inheritance Models

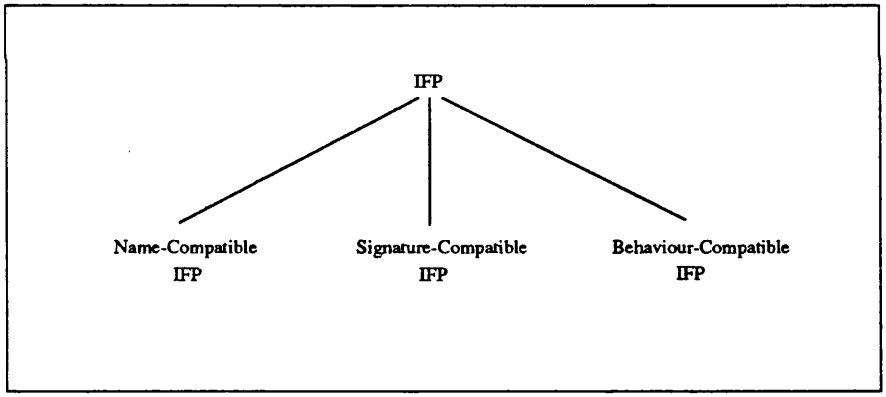


Figure 4.5: The IFP in Various Inheritance Models

The previous section has discussed in detail the formal model which lies behind the inheritance factorisation process. As it stands, the formal model is deliberately generalised to allow further modifications and extensions. This is especially true when comes to the definition of an attribute.

Currently, the model of the IFP states that a class specification is specified in terms of a collection of attributes. An attribute, in its simplest form, is a name of an operation. Two attributes are said to be semantically equivalent and can be factorised out if their names are the same. It is assumed that there exists a mechanism to check for the equivalence of the attributes. All these, of course, have over-generalised the model and have not taken into

considerations of the meaning behind the names. As it stands, besides name-compatibility, there are other meanings which can be used to give a more precise definition for the semantic equivalence of two attributes e.g. signature-compatibility and behaviour-compatibility (see Figure 4.5).

These various definitions of the equivalence of two attributes are, in fact, related to what exactly one inherits in an inheritance model, i.e., the nature of the inheritance model. This section gives a full account of the semantic of an attribute with respect to different inheritance models. It defines the definition of the semantic equivalence for attributes in different inheritance models. It also discusses about how the IFP can accommodate different models.

4.3.1. The IFP with Name-Compatibility

The name-compatible IFP is the simplest kind of model that one can obtain in the IFP universe. It requires only the name of two attributes to be the same for a valid factorisation. It is assumed that each name has an unique meaning attached to it and the meaning is intuitively understood by people. Hence, one can attach the same name to two attributes if they refer to the same meaning. Although this kind of IFP is very flexible to use, it induces various ambiguities in usage.

The ambiguities arise as a name seldom refers to only one meaning and a connotation can always be presented by several names. For example, one may use the word 'node' to refer to a vertex in a graph. At the same time, one may use the word 'vertex' to mean the same thing. Although the name 'vertex' and 'node' are different in syntax, they mean the same thing and should be factorised out in the IFP. However, if one only goes for a mechanism which checks for the syntax, the names 'vertex' and 'node' will not be factorised out. Besides, ambiguities can arise when people use abbreviations. For example, some people use 'phone-no' instead of 'telephone-number'. Again, simply checking the syntax of the names leads to a failure in the factorisation process.

There is not yet a perfect solution to solve such an intuitive mismatch. Nevertheless, there are a few precautions which system designers can take to reduce the ambiguities involved. Firstly, a good choice of names for the attributes contributes significantly to readability and understandability of the meaning of an attribute. It is suggested that one should strive for names which are clean, direct and avoid using dummy names such as 'foo' which either means nothing or everything. At the same time, software tools can be developed to provide common definitions for attributes. One can develop an attribute dictionary which is analogous to the traditional data dictionary. The attribute dictionary stores the definitions of the attributes and their aliases. When specifying a new class specification, system designers

are required to consult the attribute dictionary to avoid any confusions in the meaning of an attribute.

The name-compatible IFP is normally found in the non-strict inheritance model which implies code-sharing. What is being inherited here is a piece of code or a memory location. Hence, an attribute is a name that refers to a piece of code. Two attributes are said to be semantically equivalent and hence can be factorised out if their names denote the same piece of code. This seems to give more concrete grounds to compare two attributes. Unlike the above discussion which states that a name indicates an intuitive meaning and is too abstract to compare, a name in the non-strict inheritance model refers to a piece of code. One can only use the same name for the attributes if the attributes refer to the same memory location and the same piece of code. However, the name-compatible IFP is only straight forward in a non-strict inheritance model when code is not allowed to be overwritten. When the model allows code to be overwritten to preserve the name being used, a more complicated IFP results. The IFP with overwriting implies that the IFP should allow users to specify the priority of attributes and this is discussed in more detail in Section 4.3.4.

4.3.2. The IFP with Signature-Compatibility

The name-compatible IFP is more suitable for weakly-typed languages such as Smalltalk [Gol83b] and LOOPS [WeQ84]. However, for strongly-typed languages such as Eiffel [Mey88], Trellis/Owl [Kil89, OHK87] and SOLVE [RWW88], name-compatibility alone becomes inadequate. To rectify this situation, one has to take into consideration of the signature corresponding to each name as well, hence the signature-compatible IFP.

An attribute in the signature-compatible IFP not only embodies the intuitive meaning of the name and the implementation that is attached to the name but also takes into account the signature associated with that attribute. The signature conveys information about the types involved in an attribute. For example, the attribute '*int foo*' means that 'foo' has a type 'int'. Also, '*int add(int a)*' indicates that 'add' is a function that takes in an object of type 'int' and returns an object with a type 'int'.

In the signature-compatible IFP, two attributes are said to be semantically equivalent if the syntax of the attributes denotes the same intuitive meaning, the same piece of code and their corresponding signatures are compatible. Note that when signature is considered, one tends to say that two signatures are compatible instead of two signatures are equal. This is because two signatures need not be equal to be compatible. There are two situations when signatures can be said to be compatible:

i. Subtyping

The concept of subtyping has been discussed earlier in this chapter. Subtyping is similar to that of subclassing; the elements of a subtype also belong to its supertype. Therefore, 'typeA' is said to be compatible to 'typeB' if there is a direct inherit relationship between 'typeA' and 'typeB'. For example, if one examines the following two operations:

'add: int int → int'

'add: real real → real'

The signature of these two operations are not equal because they expect different types of input and output arguments. However, the signature of these two operations are said to be compatible because the type 'int' is a subtype of the type 'real'. This kind of compatibility is also known as inclusion polymorphism [CaW85].

ii. Conformance

The concept of subtyping is often considered restricted in the sense that it requires two types to have a direct 'inherit' relationship before they are regarded as compatible. For example, Figure 4.6 shows that 'typeB' is compatible with 'typeA' because it is a direct descendent of 'typeA'. However, 'typeC' can never be compatible to 'typeA' because they are in different branches in the hierarchy.

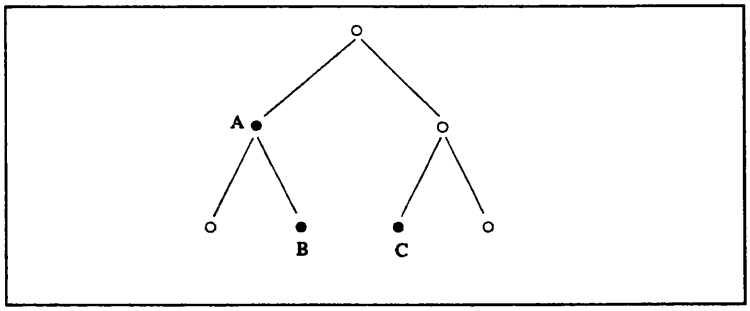


Figure 4.6: Type Compatibility and Conformance Rules

In order to relax this situation, one can introduce appropriate conformance rules to the type system. The conformance rules define when and how a type can be conformed to another type. With the correct conformance rules, two types do not need to have a direct inherit relationship in order to be compatible. This kind of compatibility is also known as coercion polymorphism [CaW85].

With signature-compatible IFP, it seems that in order to decide whether two attributes are semantically equivalent and hence can be factorised out, one has to take considerations of the conformance rules defined for the type system.

Now that the definition of semantic equivalence of two attributes in the signature-compatible

IFP is defined, one can examine how to support a mechanism to check for the equivalence in the IFP. There are basically two ways to handle it. An easy way in which the current IFP model still applies is to require system designers to do the checking. Two attributes can only be labelled with the same name if:

i. they share the same intuitive meaning,

ii. they refer to the same piece of code and

iii. the corresponding signatures of the attributes are compatible.

In this case, system designers are responsible for attaching the right label to the attributes. The checking mechanism in the IFP only needs to check for the syntactic equivalence of the two names.

The other more efficient way to handle the signature-compatible IFP is to modify the syntax of an attribute to capture the signature of the attribute as well. An attribute is no longer represented by a name alone but also the types of its input and output arguments. For instance, the previous example about the operation 'add' can have the syntax, *int add(int)*. In this case, the checking mechanism in the IFP has to be modified so that it does not only check for the syntactic equivalence of the names but also look into the types defined for such a name. The type-checking mechanism found in compilers would be useful in implementing the checking mechanism of the IFP.

As the signature-compatible IFP concerns type hierarchies as well as class hierarchies, the signature-compatible IFP is less flexible than the name-compatible IFP since it involves type-checking of the attributes. However, type checking is more supportive of the programmer's intentions and yields a more expressive and structured hierarchy.

4.3.3. The IFP with Behaviour-Compatibility

The behaviour-compatible IFP can be considered as the ultimate IFP. It is related to the strict inheritance model mentioned earlier in this chapter. Here, a class specification should specify the behaviour of a particular class. To define the precise behaviour of a class is rather difficult [Weg88b]. One of the mechanisms for specifying behaviour is by algebraic specification. In order to achieve strict inheritance in the IFP, a class specification should, therefore, be defined in terms of algebraic specifications [EhM85]. An example of such algebraic specification is illustrated in Figure 4.7.

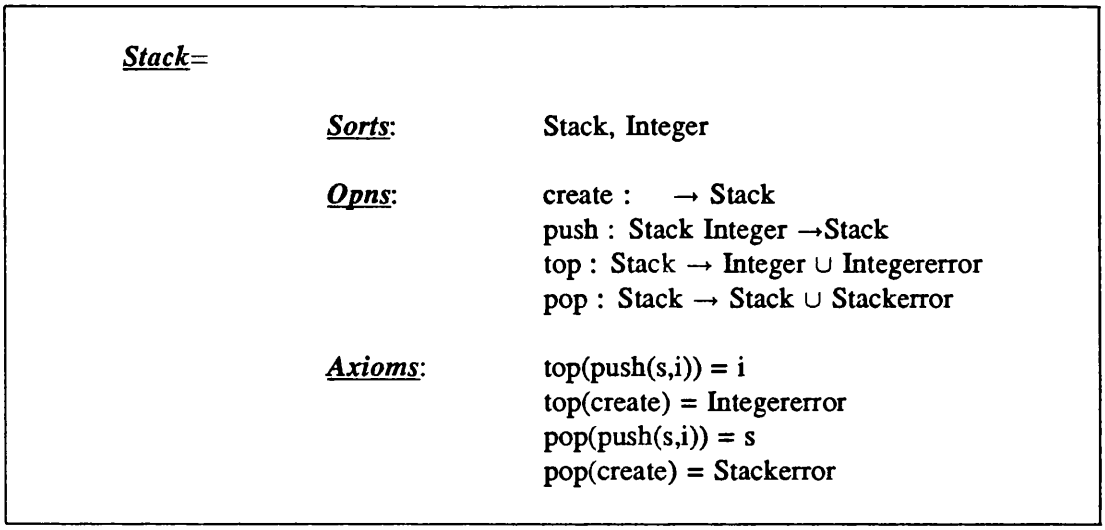


Figure 4.7: The Algebraic Specification for the Class 'Stack'

As one can see, when a class specification is defined algebraically, it involves defining the sorts, the operations and the equations of that class specification. Here, sorts simply refer to the types involved in the class specification. Operations are the actions which are defined for the class specification. Each operation is denoted by an operation name and the signature associated with it. Equations define the relationships between the operations within the class specification.

Behaviour-compatible IFP is very different from name-compatible and signature-compatible IFP. In fact, there are complications in extending the basic IFP model to cover the behaviour-compatible IFP model. In the basic IFP model, a class specification is defined as a collection of attributes which are of the same kind, i.e., they can all be viewed as operations defined for a particular class. This definition applies both to the name-compatible and the signature-compatible IFP. Although in the signature-compatible IFP one may need a richer syntax to include the signature as well as the name for an attribute, the attributes described in a class specification still belong to the same kind. However, in the behaviour-compatible IFP, a class specification involves three different kinds of attributes namely, sorts, operations and equations.

Class Specification = (Sorts, Operations, Equations)

As one can imagine, some of the definitions in the basic IFP model have to be modified to cater for the behaviour-compatible IFP. First of all, the definition of a class specification has to be re-defined.

Definition 4.17

A *behaviour-compatible IFP class specification* is a collection of three different types of attributes,

$$c_i = \{st_1, \cdots, st_l\} \cup \{op_1, \cdots, op_m\} \cup \{eq_1, \cdots, eq_n\}$$

where $st_i \in \mathcal{ST}_i$, $op_i \in \mathcal{OP}_i$ and $eq_i \in \mathcal{EQ}_i$

and $\mathcal{ST}_i \subset Sorts, \mathcal{OP}_i \subset Operations$ and $\mathcal{EQ}_i \subset Equations$.

Further, the axiom which governs the factorisation of common attributes has to be modified.

Axiom 4.4

For behaviour-compatible IFP,

$$\{c_1\} + \cdots + \{c_n\} = c_0' \lhd_s (\{c_1'\} + \cdots + \{c_n'\})$$

$$iff \quad \mathcal{ST}_0 = \bigcap_{i=1}^{n} \mathcal{ST}_i \quad and \quad \mathcal{OP}_0 = \bigcap_{i=1}^{n} \mathcal{OP}_i \quad and \quad \mathcal{EQ}_0 = \bigcap_{i=1}^{n} \mathcal{EQ}_i$$

It is generally agreed that writing a formal specification is a difficult task [LiG86]. Hence although the behaviour-compatible IFP presents a precise and non-ambiguous inheritance model, it is hard to achieve. Concerning the class specifications, there is research going on investigating how to use algebraic specifications to specify a class, such as OBJ2 [GoM82, Shu89]. Concerning the mechanism to check whether a factorisation can be taken place between two class specifications, one basically needs to check the three kinds of attributes associated with each class specification instead of one as in the name-compatible and the signature compatible IFP. It is outside the scope of this thesis to give the details of how to attain the behaviour-compatible IFP. However, this thesis has demonstrated such a model is plausible.

4.3.4. The IFP with Priority Attributes

The basic IFP model treats all the attributes as having the same priorities when it comes to factorisation. However, it is more practical to grant different priority values to different attributes according to the knowledge system designers have about the attribute and the nature of the development environment. In this case, system designers can specify that a particular attribute must not be factorised under certain circumstances. This serves to give more flexibility to the inheritance factorisation model. Such flexibility is found to be very useful with inheritance models that allow code to be overwritten.

Inheritance models which allow code to be overwritten are common in the inheritance world. They permit the names to be preserved while the implementation attached to that name is altered. For example, with the code overriding, the class 'circle' and the class 'rectangle' in the same class hierarchy can have an operation named 'draw' and yet the algorithm to draw a circle and a rectangle is different. In the basic IFP model, in order to distinguish that the 'draw' operation in the class 'circle' has a different implementation from that of the class 'rectangle', one needs to name the operation with different names such as 'drawCircle' and 'drawRectangle' respectively. This, of course, is valid but clumsy. With priority attributes, in order to highlight the fact that the operation 'draw' in the class 'circle' has a different implementation from that of the class 'rectangle', one can assign a lower priority value to the operation 'draw' so that when factorisation takes place, it will not be factorised out. In this case, the name of the operation is still preserved and yet they can have different implementations.

Moreover, if one considers the inheritance model which supports only single inheritance, the ability to define the priority of the attributes in factorisation becomes extremely useful. As indicated in Section 4.2, it is possible for the inheritance factorisation to yield more than one normalised class hierarchy expression. If multiple inheritance is supported then this indicates that a multiple inheritance hierarchy exists. However, in some cases, multiple inheritance is not accommodated. Here, one has to choose a more suitable class hierarchy expression amongst various normalised hierarchy expressions obtained. Again, with priority attributes, one can define the priority value of factorisation for each attribute according to the nature of the development environment. In this case, system designers can obtain a unique normalised class hierarchy expression by taking into the account the priority of the attributes. In order to illustrate the usefulness of the priority attributes, Example 4.4 is a revisit of Example 4.3 but this time it is assumed that the IFP does not support multiple inheritance.

Example 4.4

In this example, a class hierarchy is constructed for the set of class specifications, $s_i = \{c_0, c_1, c_2, c_3\}$. If $c_0 = \{a_1, a_2, a_3, a_5\}$, $c_1 = \{a_1, a_3, a_6, a_8\}$, $c_2 = \{a_1, a_4, a_6, a_8\}$ and $c_3 = \{a_1, a_2, a_7\}$ then the factorisation process will be:

$$
\begin{aligned}
\xi_i &= \{c_0, c_1, c_2, c_3\} \\
&= \{c_0\} + \{c_1\} + \{c_2\} + \{c_3\} \\
&= \{\{a_1, a_2, a_3, a_5\}\} + \{\{a_1, a_3, a_6, a_8\}\} + \{\{a_1, a_4, a_6, a_8\}\} + \{\{a_1, a_2, a_7\}\} \\
&= \{a_1\} \lhd_s (\{\{a_2, a_3, a_5\}\} + \{\{a_3, a_6, a_8\}\} + \{\{a_4, a_6, a_8\}\} + \{\{a_2, a_7\}\}) \\
&= \{a_1\} \lhd_s (\{a_3\} \lhd_s (\{\{a_2, a_5\}\} + \{\{a_6, a_8\}\}) + \{\{a_4, a_6, a_8\}\} + \{\{a_2, a_7\}\}) \; [\xi_{N1}] \\
\text{or} \quad &= \{a_1\} \lhd_s (\{a_2\} \lhd_s (\{\{a_3, a_5\}\} + \{\{a_7\}\}) + \{a_6, a_8\} \lhd_s (\{\{a_3\}\} + \{\{a_4\}\})) \; [\xi_{N2}]
\end{aligned}
$$

Here, the IFP has generated two valid normalised expressions, ξ_{N1} and ξ_{N2}. Suppose the

attribute a_3 denotes an operation in which its corresponding code would be overwritten by the inheriting class. If the degree of reusability is measured by how many pieces of code are being reused then the class hierarchy generated from ξ_{N1} has a lower degree of reusability than that of ξ_{N2} because the code of a_3 is not going to be reused. As the basic IFP model defines that an optimal class hierarchy is one which achieves maximum reusability, a better choice of normalised expression for an optimal class hierarchy, in this case, is ξ_{N2}.

To modify the IFP model so that it can recognise ξ_{N2} as a more suitable candidate for a normalised class hierarchy expression, one has to install the capability to specify priority values in the IFP. If system designers can specify that the attribute a_3 has a lower priority value, a_3 will not be factorised out when an ambiguity is detected. In this way, ξ_{N2} is automatically selected as the normalised class hierarchy expression. The current IFP, though, has not been developed to cater for this. This section has demonstrated such an improvement of the IFP is possible and useful.

4.4. Applying the IFP to Systems Design

The main objective of developing the IFP is to help system designers handle inheritance in object-oriented programming. With respect to inheritance in object-oriented programming, one of the main task one needs to do is to identify the 'inherit' relationship of the class one wants to construct. This usually involves two kinds of activities.

i. System designers have to check whether there exists a suitable class hierarchy for the new class to attach to. If a suitable class hierarchy is not found, a new class hierarchy has to be constructed from scratch.

ii. If a suitable class hierarchy for the class one wants to construct is identified, the class is then added to such an existing class hierarchy.

This section discusses how the IFP can be applied in these two design activities. After that, it discusses how the design behaviour of the system designers may be changed as a consequence of using the IFP. It also mentions how the IFP is incorporated into the design method proposed in Chapter 3.

4.4.1. In Building a Class Hierarchy from Scratch

In the application system design domain, system designers seldom find that they have to construct new class hierarchies from scratch [Joh88]. Normally, the system development environment already has a rich set of class hierarchies in which system designers can find a suitable class hierarchy for the new class to attach to. Even if one cannot find a suitable

class hierarchy, application system designers tend to create the new class as the subclass of the universal superclass '*Object*' instead of working on a new class hierarchy from scratch.

The construction of new class hierarchies from scratch is found to be more common in composing the system development environment for applications to build on. Here, system designers have to provide a good environment for software development. This involves supplying adequate class hierarchies in which classes for the application can be built. As it stands, the basic model of the IFP is actually more appropriate for system designers in constructing class hierarchies from scratch. With the IFP, system designers do not need to work out all the abstract classes† involved in the class hierarchy nor do they need to compare and contrast the classes to identify the superclasses. They only need to specify the conceptual class specifications involved in a class hierarchy construction problem and the corresponding optimal class hierarchy is then generated.

4.4.2. In Adding a New Class to an Existing Class Hierarchies

As mentioned above, it is more common for system designers to add a new class to an existing class hierarchy. Hence, it is important to examine what the system designers need to do with the IFP when adding a new class to an existing class hierarchy.

In the IFP, a class hierarchy is presented as a graph. The addition of a new class means that a new node is added to a graph. The new node can be added as an external node or an internal node. The structure of the graph may be changed according to whether it is added as an internal or an external node. Consequently, system designers have to use IFP differently depending on whether the structure of the graph is going to be changed or not.

A good object-oriented software development environment should provide a set of well-defined class hierarchies which reflects maximum reuse. These hierarchies should have the abstract classes at the top of the class hierarchy and concrete classes at the bottom [Joh88]. Thus, when a new concrete class is added to a well-defined class hierarchy, it is always added as an external node. This is illustrated as Figure 4.8.

When a class is simply added as an external node, no restructuring of the graph is required. The new class specification is usually a simple incremental modification of an existing class specification in the hierarchy. All this implies that there is no need to perform a new factorisation on the conceptual class specifications in the existed graph and the new class specification. A simple graph-walking algorithm, based on comparing the attributes of the

† In Johnson's paper [Joh88], he defines an abstract class as one that seldom has instances, only its subclasses have instances. The root of a class hierarchy is usually abstract classes while the leaves are never abstract.

class specifications, can be developed to append the new class to the existing class hierarchy graph.

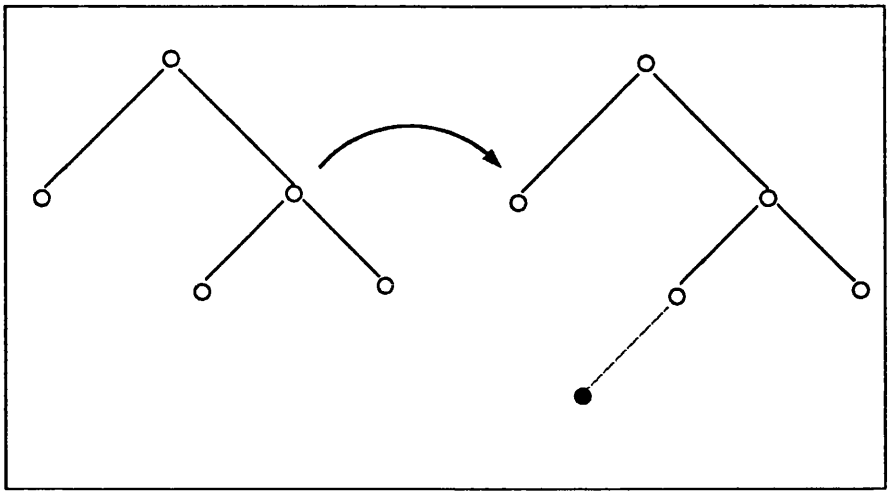


Figure 4.8: Adding a New Class as an External Node

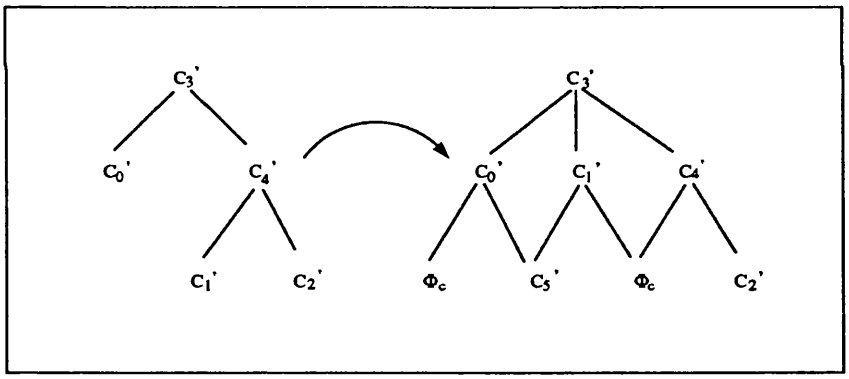


Figure 4.9: A Restructuring of a Class Hierarchy

However, the addition of a new class may sometimes lead to a restructuring of an existing class hierarchy. This happens when the class hierarchy itself is not yet well-defined, i.e., the addition of the new class simply identifies more abstract classes at the top of the hierarchy to give a better hierarchy structure with higher degree of reusability. This is especially true if the addition of the new class means that the class hierarchy will transform from a single inheritance hierarchy to a multiple inheritance hierarchy as shown in Figure 4.9.

The class hierarchy on the left hand side of Figure 4.9 is constructed from conceptual class

specifications, $c_0 = \{a_0, a_1\}$, $c_1 = \{a_0, a_2, a_3\}$ and $c_2 = \{a_0, a_2, a_4\}$.

$$\begin{aligned}\xi_i &= \{c_0, c_1, c_2\} \\ &= \{\{a_0, a_1\}\} + \{\{a_0, a_2, a_3\}\} + \{\{a_0, a_2, a_4\}\} \\ &= \{a_0\} \lhd_s (\{\{a_1\}\} + \{\{a_2, a_3\}\} + \{\{a_2, a_4\}\}) \\ &= \{a_0\} \lhd_s (\{\{a_1\}\} + \{a_2\} \lhd_s (\{\{a_3\}\} + \{\{a_4\}\}))\end{aligned}$$

If $c_0' = \{a_1\}$, $c_1' = \{a_3\}$, $c_2' = \{a_4\}$, $c_3' = \{a_0\}$ and $c_4' = \{a_2\}$, then the corresponding hierarchy graph is shown on the left hand side of Figure 4.9. Now, the new class specification $c_5 = \{a_0, a_1, a_3, a_5\}$ is added to this class hierarchy. As c_5 is not a simple incremental modification of any existing class specifications, the factorisation process has to be repeated with c_0, c_1, c_2, c_5.

$$\begin{aligned}\xi_i &= \{c_0, c_1, c_2, c_5\} \\ &= \{\{a_0, a_1\}\} + \{\{a_0, a_2, a_3\}\} + \{\{a_0, a_2, a_4\}\} + \{\{a_0, a_1, a_3, a_5\}\} \\ &= \{a_0\} \lhd_s (\{\{a_1\}\} + \{\{a_2, a_3\}\} + \{\{a_2, a_4\}\} + \{\{a_1, a_3, a_5\}\}) \\ &= \{a_0\} \lhd_s (\{a_1\} \lhd_s (\{\phi_c\} + \{\{a_3, a_5\}\}) + \{a_2\} \lhd_s (\{\{a_3\}\} + \{\{a_4\}\})) \\ \text{or} \quad &= \{a_0\} \lhd_s (\{\{a_1\}\} + \{a_3\} \lhd_s (\{\{a_2\}\} + \{\{a_1, a_5\}\}) + \{\{a_2, a_4\}\})\end{aligned}$$

The factorisation indicates that multiple inheritance is detected and by Axiom 4.3, the normalised class hierarchy expression is:

$$\{a_0\} \lhd_s (\{a_1\} \lhd_s \{\phi_c\} + (\{\{a_2\}\} + \{\{a_3\}\}) \lhd_m \phi_c + \{a_2\} \lhd_s \{\{a_4\}\} + (\{\{a_1\}\} + \{\{a_3\}\}) \lhd_m \{a_5\})$$

By having $c_5' = \{a_5\}$, the new graph is presented on the right hand side of Figure 4.9.

As it stands, the restructuring of a class hierarchy should not affect the objects which have been created based on the old class hierarchy. The structure of the class from which the objects have been instantiated should remain the same. The only thing which has altered is the inheritance path. The new class hierarchy should provide a more efficient inheritance path which gives a higher degree of reusability. Because of this, when there is a restructuring of a class hierarchy, all the inheritance paths of the defined classes of the previous hierarchy have to be updated.

4.4.3. The Importance of Specifications in the IFP

The development of the inheritance factorisation process has highlighted the point that system designers should look at class hierarchy manipulations from a different angle. Up till now, system designers have applied 'ad hoc' methods and intuition in constructing class hierarchies. They spend more time comparing the similarities and differences amongst various classes than working on the precise properties and attributes of these classes. This has led to the construction of ill-defined class hierarchies with little reuse. Since the class

hierarchies are not well-defined, it is more likely for these class hierarchies to be reorganised whenever a new class is added to them.

Unlike the 'ad hoc' method, the IFP emphasises the importance of having correct class specifications. As Liskov and Guttag once said [LiG86], "The art of writing a specification sheds light on the abstraction being specified by focusing attention on the properties of that abstraction. This use can be enhanced by careful attention to properties that might be overlooked". This is indeed the essence of the IFP. By relieving system designers of any mechanical processes in constructing class hierarchies, the IFP encourages system designers to pay more time and attention on identifying the correct properties and characteristics of individual classes. The generation of a reasonable class hierarchy relies very much on specifying the classes correctly in the first place. This point is currently overlooked by most system designers.

4.4.4. Incorporating the IFP into the Proposed Design Method

The inheritance factorisation process is developed as part of the design method mentioned in Chapter 3. The design method requires system designers to identify the 'inherit' relationship at the system level. This can be done by using the inheritance factorisation process. When the system designers have confirmed the set of implementation objects, they have to decide whether these objects can be instantiated using existing classes or not. Very often, new classes have to be constructed in which case inheritance factorisation process can be used. The inheritance factorisation process would help them to construct the necessary class hierarchies and hence to identify the required 'inherit' relationships. This information is required to construct the class structure chart and the message structure chart at the specification level.

Further, this design method forms the backbone of a CASE environment for object-oriented programming. In this case, the inheritance factorisation process lays down the foundation of a tool which helps system designers in constructing class hierarchies. The tool which is called the factorisation engine can be incorporated into the CASE environment. This is discussed briefly in the next section but in more detail in Chapter 5.

4.5. Automating the IFP

Automation in CASE has become more and more important in recent years. It has attained dramatic productivity advances in software engineering. Automation not only concerns code generation in the implementation phase but also comes as the front-end of the system analysis phase [Mar88]. As it stands, the inheritance factorisation process is only a manual manipulation process. System designers will benefit more if such a process can be automated.

The automation process can eventually be incorporated into the CASE environment for object-oriented programming. This section describes the various stages which are required to develop such an automation process.

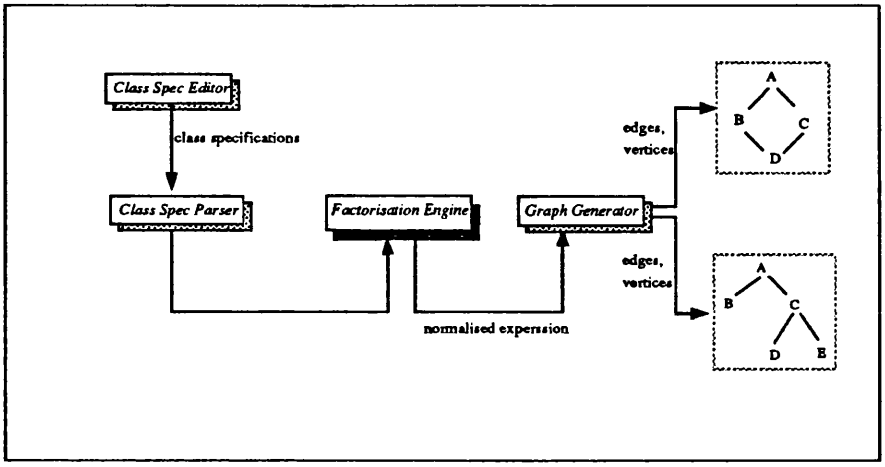


Figure 4.10: The Automation Process of the Inheritance Factorisation Model

The automation of the IFP involves five components:

i. Class Specification Editor

The class specification editor allows system designers to define the set of class specifications involved in a particular class hierarchy construction problem. The editor consists of a template which prompts users to fill in the required information about a class specification. Basically, users have to specify the individual class name and the attributes belonging to that particular class. The following is an example of a class specification in the name-compatible IFP.

Point: create, display, move, locate;

ii. Class Specification Parser

Once the set of class specifications is defined, it is passed through the class specification parser. The responsibility of the parser is three-fold.

- It has to verify the correctness of the defined specifications, e.g., whether the attributes found within a class specification are mutually exclusive.
- When the verification process is finished, the parser has to generate a symbol table for the corresponding set of class specifications. The symbol table contains the name of the classes involved and the description of the individual attributes.
- However, the most important task of the parser is, in fact, to generate an initial expression from the set of class specifications for the inheritance

factorisation process.

iii. Inheritance Factorisation Engine

The inheritance factorisation engine is the main component in the whole automation process. Its construction is based on the formal model mentioned in Section 4.2. The inheritance factorisation engine takes in the initial expression and applies an algorithm based on the inheritance factorisation model to factorise out common attributes. When a set of common attributes is factorised out, it indicates that a new superclass in the class hierarchy has been generated. The factorisation process continues until the expression cannot be factorised further. The output of the factorisation engine is a normalised class hierarchy expression. The normalised class hierarchy expression will then go into the graph generator in which an optimal graph is generated.

iv. The Graph Generator

The main function of the graph generator is to take the normalised expression and generate the corresponding set of vertices and edges. By having this set of vertices and edges, the corresponding class hierarchy graph can be produced.

As one can see, the core component of the automation of the IFP is the development of the inheritance factorisation engine. In order to demonstrate such automation and examine the performance of the IFP, a prototype of the inheritance factorisation engine was constructed. The implementation details of the inheritance factorisation engine is found in Chapter 5.

4.6. The Truth of the IFP

When the idea of the inheritance factorisation process was first introduced, a number of people showed enthusiasm for it. Most of them recognised that it could be useful, contributing to the design and being a part of the development environment for object-oriented programming. A few of them suggested that the IFP is similar to the techniques of cluster analysis. Hence, it is important to have this section to define what exactly the IFP is and is not.

4.6.1. The Techniques of Cluster Analysis and the IFP

The technique of cluster analysis [Eve80] is a classification scheme for grouping objects into a number of classes such that objects within classes are similar in some respect and unlike those from other classes. The objects or individuals are described by a set of numerical measures. There are various methods of cluster analysis, e.g., Q-analysis, typology, grouping, classification, numerical taxonomy and unsupervised pattern recognition. Such a variety of

names is due to the popularity of cluster analysis in various fields such as psychology, zoology, biology, artificial intelligence and information retrieval. The technique of cluster analysis is regarded as a useful tool to search for natural groupings in the data.

The details about the technique of cluster analysis are outside the scope of this thesis. However, the basic idea of cluster analysis is to find the similarity coefficient or distance measures of each variable between two entities. The grouping of the entities depends totally on these coefficients. For example, the following table shows how the three entities are specified by some numerical values in a set of chosen variables.

Item/Variables	height	weight
1. M. Smith	66	120
2. W. Blogg	76	130
3. M. Storm	70	150

Hence, by obtaining the similarity coefficients†:

$$S_{12} = 0.334, \quad S_{13} = 0.466, \quad S_{23} = 0.200$$

one can then group similar items together by comparing the similarity coefficient.

As it stands, the technique of cluster analysis is very different from that of the IFP, as it serves as a classification of similar objects into classes. The classification depends on the quantitative factors assigned to each variable. Whereas in the IFP the objective is not about grouping similar objects into classes but to identify how the classes are related to each other in the class hierarchy. Although each class is described in terms of attributes which is similar to that in the cluster analysis, the comparison does not rely on any quantitative measure of the attributes. There is no similarity coefficient in the IFP. A class either has an attribute or does not have it. Besides, the result of the IFP not only highlights the 'inherit' relationships between different classes in the hierarchy but also identifies all the necessary abstract classes which are required to build the hierarchy. Therefore, there is little similarity between the technique of cluster analysis and the IFP.

4.6.2. The Benefits brought by the IFP

After declaring what the IFP is not, this section states what the IFP is by discussing the objectives of developing the IFP and the benefits which is brought by the IFP to object-oriented programming

† The details of how to obtain the coefficient can be found in [Eve80].

The IFP allows system designers to specify the related class specifications involved in a class construction problem and the corresponding optimal class hierarchy is then generated. The generation of the optimal class hierarchy means that the following goals have been achieved:

i. The IFP has identified all the abstract classes required for a class hierarchy.

ii. The description of these abstract classes are obtained as a result of the IFP.

iii. The 'inherit' relationships amongst these abstract classes are also identified.

The development of the IFP has been found to be beneficial to the construction of class hierarchies in various ways:

i. The IFP has relieved system designers of any mechanical process so that they can concentrate more on specifying the correct properties and characteristics of individual classes in the course of constructing class hierarchies.

ii. As indicated by Johnson [Joh88], "Finding new abstraction is difficult. ··· Humans think better about the concrete examples then about abstractions". Hence, the identification of the abstract classes is a non-trivial task. However with the IFP, system designers do not need to worry about finding the correct abstract classes. All the abstract classes required in a particular class hierarchy and their descriptions are automatically generated as a result of the IFP.

iii. The IFP has a formal model as its basis. This feature of the IFP is very important because it means that provided the class specifications given by the system designers are complete and correct, the class hierarchies generated from the IFP should be consistent and well-defined. The result can help to resolve some of the problems that occur in the construction of class hierarchies, such as minimising the restructuring of class hierarchies when a new class is added to an existing class hierarchy.

iv. The 'ad hoc' method tends to encourage system designers to apply a top-down design approach. When system designers have to create new class hierarchies, they have to start with the root node, the internal nodes and then the leaf nodes. This limitation imposes unnecessary constraints on system designers in designing the class hierarchy especially in the case of reuse in which a middle-out or bottom-up approach is required. The IFP, however, offers more flexibility to system designers in constructing class hierarchies. It allows system designers to apply bottom-up and middle-out approach to design class hierarchies. In fact, system designers need only to specify the conceptual class specifications and an optimal class hierarchy is then generated. The IFP is not aware of whether a top-down, bottom-up or middle-out approach is being applied.

4.7. Conclusion

This chapter has discussed the details of the inheritance factorisation process. The inheritance factorisation process has been developed to help system designers construct class hierarchies in object-oriented programming. It provides a systematic manipulation which system designers can apply in constructing class hierarchies, guaranteering the generation of an optimal class hierarchy.

Besides giving the details of the formal model which lies behind the manipulation process, this chapter has also demonstrated the flexibility of the inheritance factorisation model. It shows how the basic model can be extended to cover both the non-strict inheritance and the strict inheritance. It has examined how the IFP can be used in constructing class hierarchies from scratch and adding new classes to an existing class hierarchies. As these two activities are found in the system design process, hence it has established the value of the IFP in system designs. In addition to this, this chapter has also showed that such a process can be automated. The implementation of the automation process is found in Chapter 5.

"On the other hand, we cannot ignore efficiency."

~ Jon Bentley ~

Chapter 5

A Prototype of The IFE

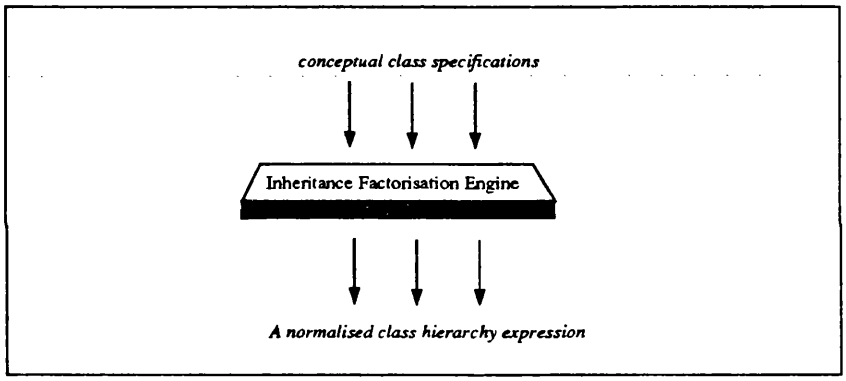


Figure 5.1: The Input and Output of the IFE

Chapter 4 has described the formal model which lies behind the inheritance factorisation process. Such a process is a manual manipulation process which provides a systematic approach to constructing class hierarchies. Although this manual process helps system designers attain an optimal class hierarchy in an efficient manner, designers will benefit more if such a process can be automated. As suggested in Chapter 4, such an automation relies heavily on the implementation of a factorisation engine. As part of this research, the first prototype inheritance factorisation engine has been assembled in the C++ programming language on a SUN 3. It was constructed from the basic model of the inheritance factorisation process. Hence, it only supports name-compatible IFP with no priority of factorisation for attributes. The prototype engine was designed to take in a collection of conceptual class specifications as input, and generate a normalised class hierarchy expression as output (see Figure 5.1). This chapter, therefore, discusses various implementations of the factorisation engine and compares their efficiency.

5.1. An Overview of the Inheritance Factorisation Engine

The algorithm which is suggested from the inheritance factorisation process for the factorisation engine can be summarised in Figure 5.2.

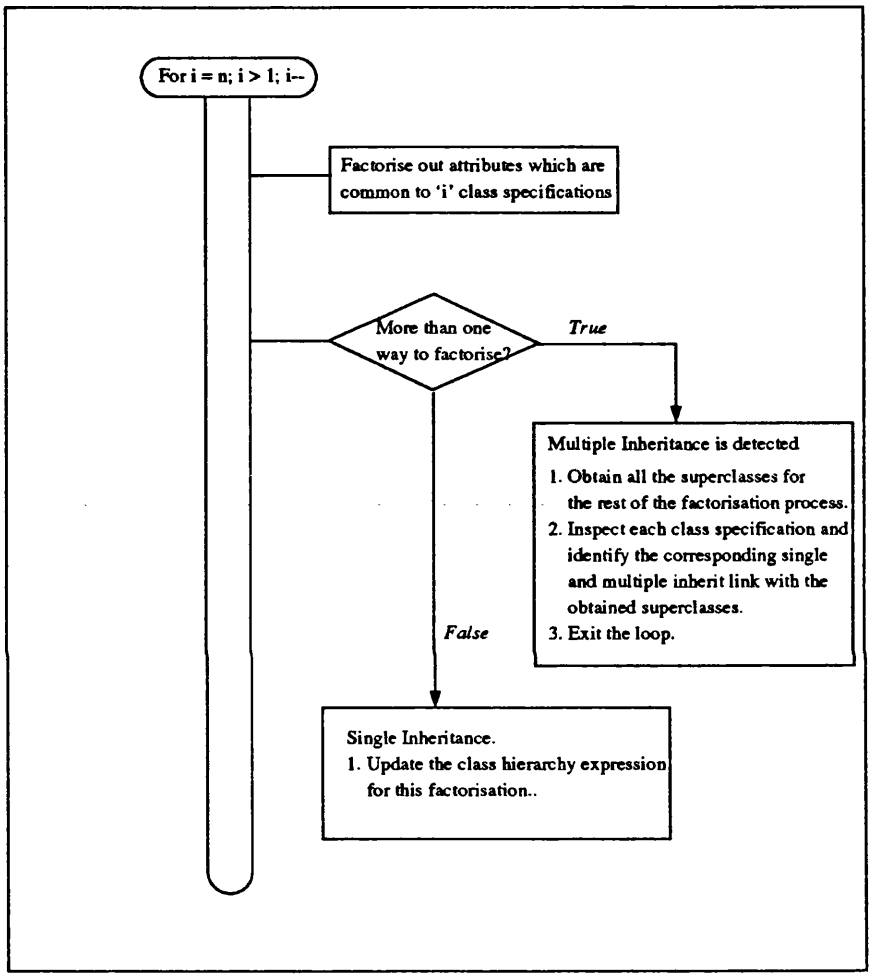


Figure 5.2: An Overview of the Algorithm for the Factorisation Engine

Here, the factorisation engine has to factorise out common attributes and check for multiple inheritance. This procedure is iterated for $i = n, \cdots, i = 2$, where n is the number of conceptual class specifications involved in the class hierarchy construction problem. This ensures that attributes which are common to more class specifications will be factorised out first. The procedure halts when no more factorisation can be carried out, i.e., $i = 2$ or multiple inheritance is detected. When multiple inheritance is detected, the factorisation process employs another method to identify the rest of the inherit links and superclasses of the hierarchy graph (see Axiom 4.3). The details of the implementation of these methods can be found later in this chapter.

From the algorithm, it is apparent that in order to implement the factorisation engine, the following issues have to be tackled :

i. Define the data structure of :

- the class specification and
- the class hierarchy expression

ii. Define the method :

- to carry out the factorisation process,
- to detect multiple inheritance,
- to update expressions which involve single inheritance alone and
- to update expressions which involve single and multiple inheritance.

The following sections discuss these issues in details.

5.2. Data Structures

5.2.1. The Class Specification

A class specification is defined as a set of attributes. The desirable data representation for a class specification is in terms of boolean values. Each possible attribute of a class specification is represented by a boolean value to indicate whether or not the attribute is in a particular class specification. For example, in a particular class hierarchy construction problem which involves three class specifications, c_0, c_1, c_2, where $c_0 = \{a_0, a_1, a_2, a_3\}$, $c_1 = \{a_0, a_1, a_8, a_9\}$ and $c_2 = \{a_0, a_7\}$. The data representation of these class specifications would be:

		a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
c_0	=	1	1	1	1	0	0	0	0	0	0
c_1	=	1	1	0	0	0	0	0	0	1	1
c_2	=	1	0	0	0	0	0	0	1	0	0

As shown later, such a boolean representation actually makes the identification of superclasses and the update of the class hierarchy expression very easy.

5.2.2. The Class Hierarchy Expression

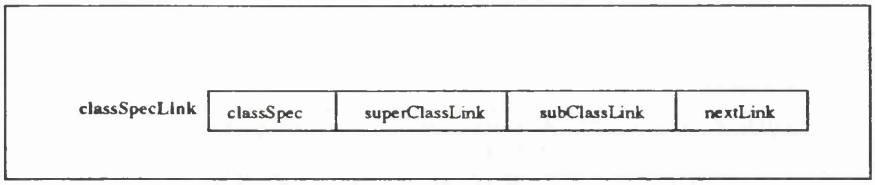


Figure 5.3: A Typical classSpecLink

A class hierarchy expression denotes the 'inherit' relationship between superclasses and subclasses. There are basically two kinds of data representation one can use. The straight forward one is to implement the class hierarchy expression as a linked list. The linked list would be made up of links called 'classSpecLink'. Each classSpecLink is composed of four fields: classSpec, superClassLink, subClassLink and nextLink as shown in Figure 5.3.

The field 'classSpec' contains the data representation of a particular class specification and as discussed above, it is a boolean representation. The fields 'superClassLink' and 'subClassLink' are pointers that point to the immediate superclass and the immediate subclass links of that class specification respectively. The field 'nextLink' points to the next classSpecLink of that class specification. Hence, the data representation of a typical class hierarchy expression which involves single inheritance only, such as $c_0' \lhd_s (\{c_1'\} + c_2' \lhd_s (\{c_3'\} + \{c_4'\} + \{c_5'\}))$ is shown in Figure 5.4.

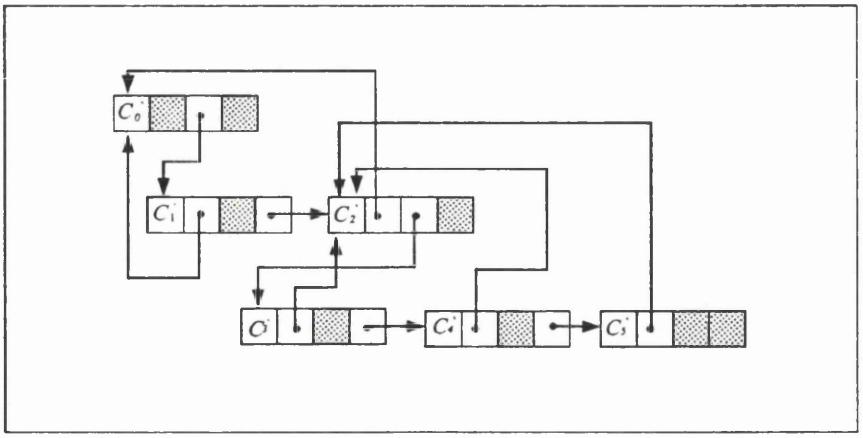


Figure 5.4: The Data Representation of a Single Inherit Expression

The data representation for a class hierarchy expression which also involves multiple inheritance as well as single inheritance, such as $c_0' \lhd_s (c_1' + (\{c_2'\} + \{c_3'\}) \lhd_m c_4')$ is shown in Figure 5.5.

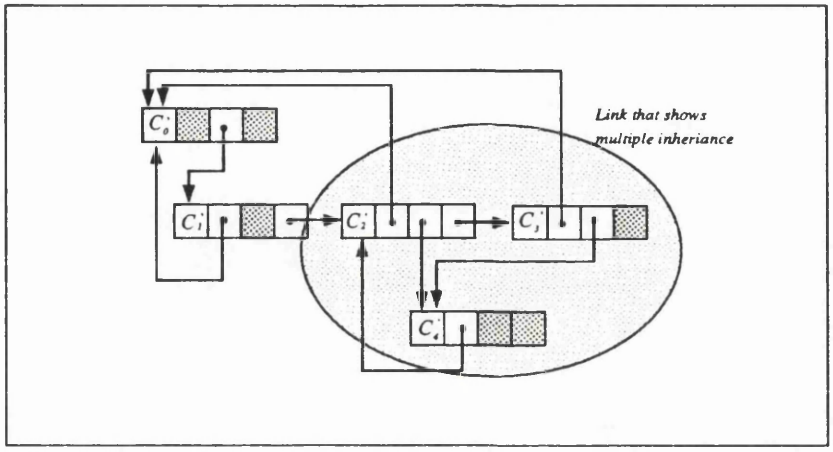


Figure 5.5: The Data Representation of a Multiple Inherit Expression

The data representation of a class hierarchy expression in the form of a linked list works well. One can generate the set of edges of the corresponding graph by transversing the linked list. However, the updating of the linked list can be quite cumbersome. Each time a factorisation is carried out, all the links of the linked list have to be updated to reflect the result of the factorisation. The following example attempts to show the complexity involved in updating the linked list when factorisation takes place.

Example 5.1

This example assumes that a class hierarchy has to be constructed from conceptual class specifications c_0, c_1, c_2, c_3 where $c_0 = \{a_0, a_1, a_2, a_3\}$, $c_1 = \{a_0, a_4, a_5\}$, $c_2 = \{a_0, a_2, a_6\}$ and $c_3 = \{a_0, a_7\}$. The factorisation process of these class specifications are:

$$
\begin{aligned}
& \xi_I \\
= \quad & \{c_0, c_1, c_2, c_3\} \\
= \quad & \{c_0\} + \{c_1\} + \{c_2\} + \{c_3\} \\
= \quad & \{\{a_0, a_1, a_2, a_3\}\} + \{\{a_0, a_4, a_5\}\} + \{\{a_0, a_2, a_6\}\} + \{\{a_0, a_7\}\} \\
= \quad & \{a_0\} \lhd_s (\{\{a_1, a_2, a_3\}\} + \{\{a_4, a_5\}\} + \{\{a_2, a_6\}\} + \{\{a_7\}\}) \\
= \quad & \{a_0\} \lhd_s (\{a_2\} \lhd_s (\{\{a_1, a_3\}\} + \{\{a_6\}\}) + \{\{a_4, a_5\}\} + \{\{a_7\}\})) \qquad [\xi_N]
\end{aligned}
$$

These various stages of the factorisation process when represented in a linked list are shown in Figure 5.6, 5.7 and 5.8 respectively.

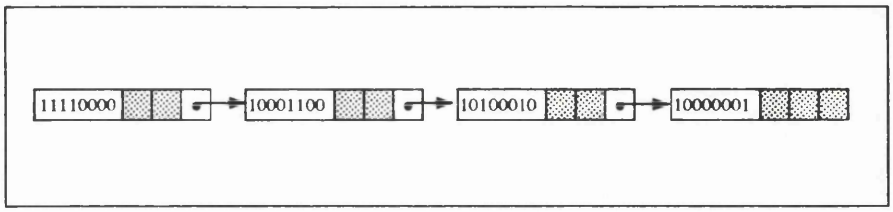


Figure 5.6: The Data Representation for the Initial Class Hierarchy Expression

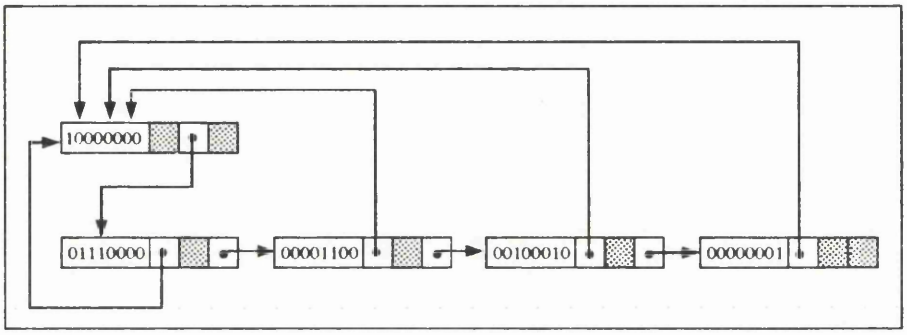


Figure 5.7: The Data Representation after the 1st Factorisation

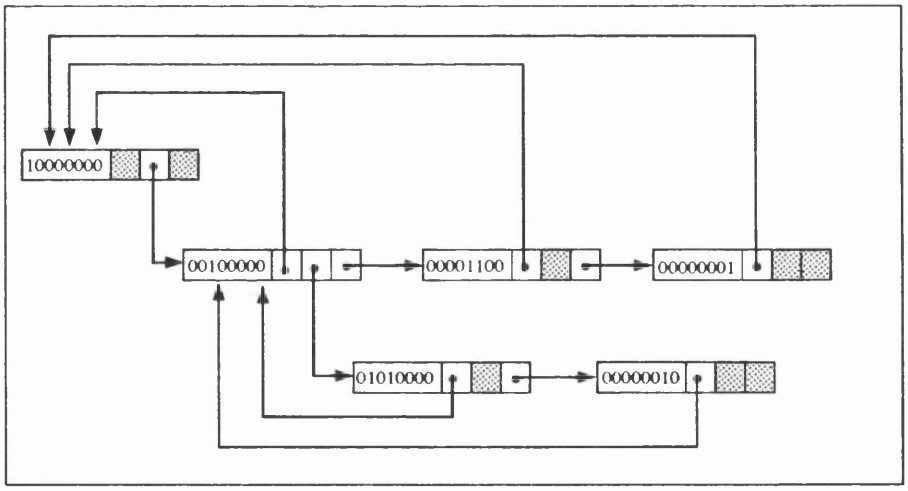


Figure 5.8: The Data Representation of the Normalised Expression

The data representation for the initial class hierarchy expression as shown in Figure 5.6 indicates that no inherit relationship is expressed. The first factorisation of these class specifications factorises out the superclass for c_0, c_1, c_2, c_3. Hence the linked list has to be updated to show this superclass/subclass link. Also the contents of the class specifications have to be updated to reflect the corresponding subclasses. This is illustrated in Figure 5.7

The next factorisation factorises out common attributes of the first and the third link. Hence, a superclass is found and the contents of the first and the third link have to be updated as well as the link of the linked list. This is shown in Figure 5.8.

As one can see, the update of the linked list representation can be quite complicated especially if the factorisation itself is non-trivial. Therefore, it is necessary to look for a simpler data representation for the class hierarchy expression. An alternative to represent the data structure of the class hierarchy expression is in terms of an adjacency matrix. As mentioned in Chapter 4, a class hierarchy expression embeds the structure of the hierarchy graph. Hence, it is reasonable to examine the data structure of a graph when looking for the data representation of the class hierarchy expression. A graph is expressed in terms of a set of vertices and a set of edges. The set of vertices is always represented as an array, e.g., vertices[]. The set of edges can be represented as an adjacency table, e.g., edges[][]. Edges[v][w] has a boolean value 'true' if and only if an edge is detected from node[v] to node[w], i.e., node[v] is the immediate superclass of the node[w].

When the class hierarchy expression is directly expressed in terms of a graph structure, the update of the expression or the graph itself becomes trivial. To illustrate this, the previous example used to show the complexity involved in updating the linked list representation is reused. Before any factorisation takes place, the adjacency table contains the four conceptual class specification, c_0, c_1, c_2, c_3 and they are stored in v_0, v_1, v_2 and v_3 as shown in Table 5.1.

vertices	v_0	v_1	v_2	v_3
$v_0 = 11110000$	0	0	0	0
$v_1 = 10001100$	0	0	0	0
$v_2 = 10100010$	0	0	0	0
$v_3 = 10000001$	0	0	0	0

Table 5.1: The Adjacency Table before Factorisation

The first factorisation factorises out the superclass from these four specifications. This superclass is then stored in v_4, the contents of v_0, v_1, v_2 and v_3 have to be updated to contain the subclasses yielded by the factorisation. The adjacency table also has to be updated to show the superclass/subclass link generated from such factorisation (see Table 5.2).

The next factorisation factorise out the superclass from v_0 and v_2. The corresponding superclass of this factorisation is stored in v_5 and the contents of v_0 and v_2 have to be updated. Again the adjacency table has to be updated to show the new edges resulted from the factorisation. This is shown in Table 5.3.

vertices	v_0	v_1	v_2	v_3	v_4
$v_0 = 01110000$	0	0	0	0	0
$v_1 = 00001100$	0	0	0	0	0
$v_2 = 00100010$	0	0	0	0	0
$v_3 = 00000001$	0	0	0	0	0
$v_4 = 10000000$	1	1	1	1	0

Table 5.2: The Adjacency Table after the First Factorisation

vertices	v_0	v_1	v_2	v_3	v_4	v_5
$v_0 = 01010000$	0	0	0	0	0	0
$v_1 = 00001100$	0	0	0	0	0	0
$v_2 = 00000010$	0	0	0	0	0	0
$v_3 = 00000001$	0	0	0	0	0	0
$v_4 = 10000000$	0	1	0	1	0	1
$v_5 = 00100000$	1	0	1	0	0	0

Table 5.3: The Adjacency Table of a Normalised Class Expression

As it is illustrated above, updating the adjacency table is easier than that of a linked list. Although adopting the adjacency table representation requires more memory space, the ease in updating makes it worthwhile.

5.3. The Method of Factorisation

Having decided upon the data structures for class specifications and class hierarchies, one can concentrate on the method of factorisation.

5.3.1. Factorisation Process

The obvious way to implement the whole factorisation process is to follow the manual manipulation process closely. Hence, factorisation is performed to identify common attributes for $i = n$ class specifications, then $i = n - 1$, $i = n - 2$, $\cdots$, $i = 2$. Since the algorithm does not know which 'i' class specifications to factorise, various combinations of 'i' class specifications have to be tried out. For example, if a class hierarchy construction problem that involves six class specifications, $c_0, c_1, c_2, c_3, c_4, c_5$, the algorithm starts off by looking for a factorisation that involves all the six class specifications:

1st factorisation, $i = n$; i.e., $i = 6$,

$$\underbrace{c_0, c_1, c_2, c_3, c_4, c_5}$$

Once the factorisation amongst all the class specifications is complete, the algorithm is reiterated but this time it looks for factorisation for any five class specifications. Since the number of r combinations from n objects is, ${}_nC_r$, i.e., $\frac{n!}{(n-r)!r!}$, the number of factorisations one has to carry out here is ${}_6C_5$ which is 6.

2nd factorisation, $i = n - 1$; i.e., $i = 5$,

$$\underbrace{c_0, c_1, c_2, c_3, c_4}\, c_5$$
$$\underbrace{c_0, c_1, c_2, c_3, c_5}\, c_4$$
$$\underbrace{c_0, c_1, c_2, c_4, c_5}\, c_3$$
$$\underbrace{c_0, c_1, c_3, c_4, c_5}\, c_2$$
$$\underbrace{c_0, c_2, c_3, c_4, c_5}\, c_1$$
$$\underbrace{c_1, c_2, c_3, c_4, c_5}\, c_0$$

As the factorisation progresses, the number of combinations which have to be checked increases sharply. With this algorithm, the worst case indicates that all the possible combinations have to be tried out. Therefore, the order of complexity of this algorithm tends to be exponential.

$$\begin{aligned} \text{Order of complexity} &\simeq \sum_{r=2}^{n} {}_nC_r \\ &\simeq 2^n \end{aligned}$$

For this reason, this algorithm which directly implements the inheritance factorisation process is almost certainly too costly to implement. In order to automate the factorisation process, one has to look for a more efficient algorithm.

When one examines the problem more closely, it is not difficult to find that the inefficiency of the algorithm lies in the fact that it has to try out all possibilities, i.e., the algorithm does not know which class specifications to factorise. Therefore, it seems that a promising approach to obtain a better algorithm is to work on eliminating the redundant factorisation attempts that occur. In order to do this, one needs to find a way to obtain the information about the exact class specifications involved in a factorisation. As mentioned earlier, a set of conceptual class specifications is presented in terms of boolean values.

Example 5.2

For example, a class hierarchy construction problem which involves class specifications, $c_0 = \{a_0, a_1, a_2, a_3\}$, $c_1 = \{a_0, a_1, a_4\}$, $c_2 = \{a_0, a_5, a_6, a_7\}$, and $c_3 = \{a_0, a_5, a_8\}$ have the

data structures as follows:

		a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
C_0	=	1	1	1	1	0	0	0	0	0
C_1	=	1	1	0	0	1	0	0	0	0
C_2	=	1	0	0	0	0	1	1	1	0
C_3	=	1	0	0	0	0	1	0	0	1

By inspecting this data structure horizontally, one can obtain information about which attributes are found in a particular class specification. Also, by inspecting the data structure vertically, one can obtain information about what class specifications contain a certain attribute. If one extends the data structure by adding an extra row which stores the frequency of the occurrence of each attribute amongst the class specifications, one can directly obtain information about which attributes occur in m class specifications.

		a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
C_0	=	1	1	1	1	0	0	0	0	0
C_1	=	1	1	0	0	1	0	0	0	0
C_2	=	1	0	0	0	0	1	1	1	0
C_3	=	1	0	0	0	0	1	0	0	1
freq	=	4	2	1	1	1	2	1	1	1

When one examines the factorisation problem again, to factorise out common attributes amongst m class specifications is, in fact, to obtain the attributes that occur in m class specifications. With the above extended data structure, by inspecting the frequency row in the table, one can find out the attributes which occur in m class specifications and by going horizontally, one can identify which class specifications are involved in a factorisation of m class specifications. For example, if one wants to factorise out attributes that are common to two class specifications, one examines the row that contains the frequency value and pick out those which has a value of 2. As shown in the above extended data structure, attributes a_1 and a_5 have a frequency of 2. Now, in order to find out which class specifications are involved in such a factorisation, one can examine the column in which the attribute a_1 and a_5 reside and identify those class specifications which have a boolean value of 'true'. In the above example, factorisations of two class specifications occur between class specification c_0, c_1 and c_2, c_3. Note that the order of which set of class specifications is being factorised first is not important.

Although this way of using the extended class specification does allow the identification of the class specifications that are involved in a particular factorisation and hence reduce the complexity of the algorithm, such an algorithm can be improved further. The current data representation actually contains redundant information which is never used in the factorisation

process. For example, as the factorisation process is only interested in attributes which occur in more than one class specification, all the attributes which have a frequency value of '1' are found to be redundant. Besides, the data representation contains indirect information for a factorisation. In order to find out which class specifications are involved in a factorisation, one has to examine the boolean value of each class specification before knowing which class specifications are relevant. In order to improve the efficiency further, one can specially generate the relevant information of the factorisation by scanning the data representation of the class specifications. The relevant information can be stored as a linked list which is made up of links called 'attInfoLink'. Each 'attInfoLink' contains four fields: attribute, frequency, nextAtt and csLink. This is illustrated in Figure 5.9.

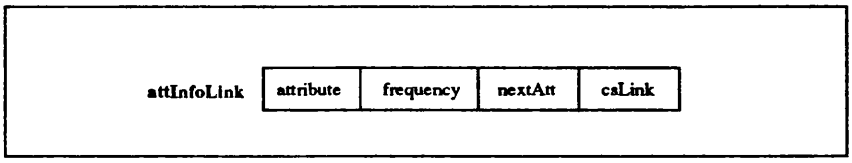


Figure 5.9: The Data Structure of 'attInfoLink'

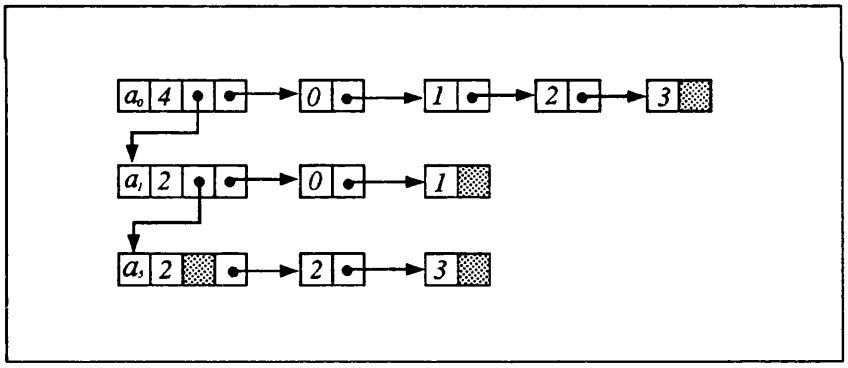


Figure 5.10: The attInfoLink List of Example 5.2.

The field 'attribute' stores the attribute of that attInfoLink. The frequency field stores the number of times such an attribute occurs in the corresponding set of conceptual class specifications. The nextAtt is a pointer which points to the next attInfoLink of the list. The csLink is a pointer which points to the corresponding list of class specifications that contain such attributes. Note that this csLink list is made up of links called csLink. A csLink contains two fields, the first field stores the index of the class specification it refers to and the second field is a pointer points to the next csLink. Hence, the attInfoLink list of Example 5.2 has the data structure shown in Figure 5.10.

Furthermore, the links can be arranged in a decreasing order of frequency value. This

fulfils the requirement that the attributes which are common in most class specifications are factorised first.

5.3.2. Detecting Multiple Inheritance

As the IFP supports multiple inheritance as well as single inheritance and the method employed to factorise an expression with multiple inheritance is different from that of single inheritance, one needs a mechanism to check whether multiple inheritance exists in a particular expression before factorisation is carried out. This subsection describes an algorithm which is used to detect multiple inheritance.

Multiple inheritance occurs when a class specification has more than one immediate superclass. In the factorisation process, this is indicated by the fact that a class specification can be factorised in two or more different ways simultaneously.

Say if $s_i = \{c_0, c_1, \cdots, c_n\}$ is the set of class specifications involved in a factorisation process. If at a particular instant, s_i can be factorised in two ways: one way of factorisation involved the set of class specifications $s_{i_1} = \{c_0, c_1, \cdots, c_i\}$, and the other way involved $s_{i_2} = \{c_0, c_1, \cdots, c_j\}$ then multiple inheritance has been detected. To describe it more formally, multiple inheritance occurs if:

$$s_{i_1} \cap s_{i_2} \neq \phi \quad and \quad s_{i_1} \not\subseteq s_{i_2} \quad and \quad s_{i_2} \not\subseteq s_{i_1}$$

The algorithm for detecting multiple inheritance employed by the IFE is, therefore, based on the above description.

Figure 5.11 shows an attrInfoLink that involved multiple inheritance. Here, the first factorisation involved factorising attributes that are common in five class specifications. Before one can perform this factorisation, one has to check whether multiple inheritance occurs in this factorisation, i.e., examine whether any class specifications involved in this factorisation also occur in other factorisations later on. As indicated in Figure 5.11, the set of class specifications involved in factoring out a_0 is $s_{a_0} = \{c_0, c_1, c_2, c_3, c_4\}$. This set of class specifications is then used to check against the other sets of class specifications in the attrInfoLink list, for example, $s_{a_1} = \{c_0, c_1, c_3\}$ is the set of class specifications involved in the factorisation of the attribute a_1, $s_{a_2} = \{c_1, c_2, c_3\}$ is the set of class specifications involved in the factorisation of the attribute a_2 and $s_{a_3} = \{c_1, c_2\}$ is the set of class specifications involved in the factorisation of the attribute a_3. Since $s_{a_1}, s_{a_2}, s_{a_3}$ are subsets of s_{a_0}, then according to the previous specified conditions for multiple inheritance, no multiple inheritance occurs when factorising out the attribute a_0.

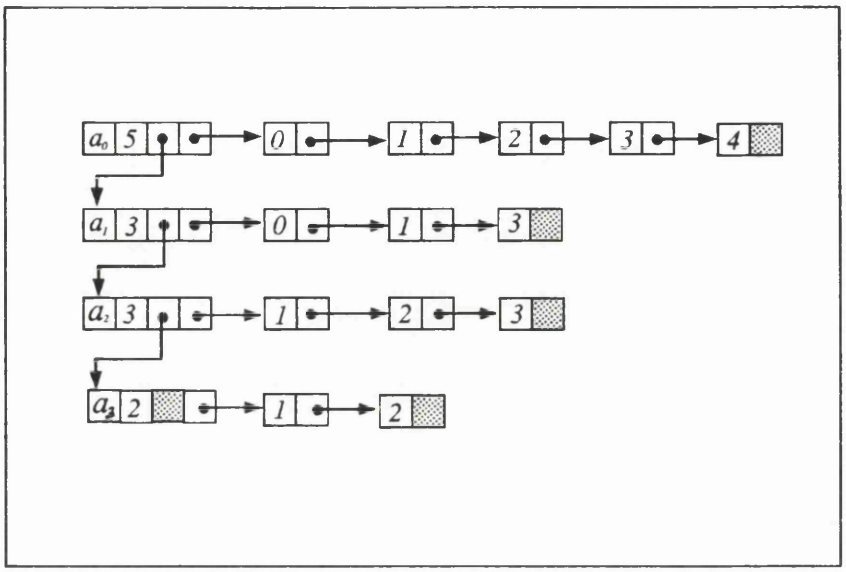


Figure 5.11: An AttrInfoLink that involved Multiple Inheritance

After factorising the attribute a_0, one goes down the attrInfoLink list attempting to factorise out the attribute a_1. Again, before the factorisation, one has to check whether multiple inheritance occurs. This time, it is found that $s_{a_1} \cap s_{a_2} \neq \phi$ and $s_{a_1} \not\subseteq s_{a_2}$ and $s_{a_2} \not\subseteq s_{a_1}$, therefore multiple inheritance is detected at this point.

5.3.3. Updating Expressions with Single Inheritance Alone

After one has checked whether multiple inheritance occurs in a particular factorisation, the next step is to perform the factorisation and update the expression. The method to update an expression that involved single inheritance is straight forward. It consists of two steps: obtaining the superclass and updating the contents of the subclasses.

i. Obtaining Superclass

In each factorisation, the factorisation engine has to factorise out the common attributes for a collection of class specifications as their superclass. Since a class specification is a set of attributes, the superclass of a particular factorisation can be obtained by finding the intersection set of attributes amongst a collection of class specifications. As the class specifications are represented as boolean values, in order to find the intersection of a collection of class specifications, one has to carry out a boolean 'AND' operation on these class specifications.

$$Super(c_0, \cdots, c_m) = \bigwedge_{i=0}^{m} c_i$$

where $Super(c_0, \cdots, c_m)$ is the superclass for the class specifications $c_0, \cdots, c_m$.

ii. Updating the Contents of the Subclasses
Once the superclass is identified, its corresponding subclasses can be obtained by carrying out a boolean '$EXCLUSIVE \quad OR$' operation on the corresponding class specification. Hence,

$$\begin{array}{lcll} c_0' & = & c_0 & xor \quad Super(c_0, \cdots, c_m) \\ c_1' & = & c_1 & xor \quad Super(c_0, \cdots, c_m) \\ \cdots & & \cdots & \\ \cdots & & \cdots & \\ \cdots & & \cdots & \\ c_m' & = & c_m & xor \quad Super(c_0, \cdots, c_m) \end{array}$$

where c_i' is the subclass for c_i.

After carrying out these two steps, one has to go back to the attrInfoLink list and continue with the next factorisation.

5.3.4. Updating Expressions with Multiple Inheritance

If multiple inheritance is detected in a particular factorisation, one has to apply a different method to update the expression (See Axiom 4.3). Such a method involves identifying all the superclasses found in the rest of the factorisation process. These superclasses are then checked against every class specification to obtain the proper single inherit and multiple inherit links amongst them.

Figure 5.11 is again used to illustrate this. When multiple inheritance is detected, one wants to factorise out the attribute a_1, the method to update the expression is as follows. Firstly, the superclass of the class specifications, $SS_1 = Super(c_0, c_1, c_3)$, involved in factorising the attribute a_1 is obtained and stored in the list of superclasses called *SuperClasses*. Then, one goes down to the next link in the attrInfoLink list and extracts out the superclass of the class specifications involved in factorising the next attribute which in this case is a_2. If this superclass, $SS_2 = Super(c_1, c_2, c_3)$, has not yet been stored in *SuperClasses*, one then stores that in the list. The same procedure is repeated with each link for the rest of the attrInfoLink list. In the end, *SuperClasses* contains all the superclasses that can possibly be obtained in the factorisation process. Now that the list of superclasses is obtained, the next step is to find out which class specification has a single inherit and/or multiple inherit relationship with which superclass. This can be done by simply checking every class specification in turn with the list of superclasses, *SuperClasses*. For example, by checking c_0 with *SuperClasses*, one finds that c_0 has a single inherit link with SS_1, but when checking c_1 with *SuperClasses*, c_1 is found to have superclasses SS_1 and SS_2, therefore c_1 has a multiple link with SS_1 and

SS_2 and the expression is updated accordingly. Since this method of updating an expression with multiple inheritance embeds the factorisation for the rest of the process, the expression obtained with this update is the normalised expression and the process is terminated at this point.

5.4. Complexity of the Algorithm

As the individual methods required to implement the factorisation engine have been discussed, this section analyses the complexity of the algorithm. The main body of the algorithm can be summarised as the following program extract which is written in C++ syntax in Figure 5.12:

```
genAttrInfo();

for (i=1; ((i < = noattrInfoLink) && (Not Finish)); i++)
{
        if (chkMultipleInherit(i) != True)
                updateSingleInherit(i);
        else
        {
                updateMultipleInherit(i);
                Finish = True;
        }
}
```

Figure 5.12: The Main Body of the Algorithm

Let n be the number of class specifications and m be the number of attributes involved in a particular class hierarchy construction problem. The function 'genAttrInfo()' scans the set of class specifications in a class hierarchy construction problem and generates the corresponding 'attInfoLink' list. Hence, the order of complexity of this function,

$$\Theta(genAttrInfo) \simeq m \times n$$

The function 'chkMultipleInherit()' checks whether multiple inheritance exists. It has to check before each factorisation actually takes place, hence it is inside the 'for' loop. This function has to check the current factorisation with the rest of the factorisation recorded in the attrInfoLink list, therefore, depending on i which is the current number of iterations of the loop, the order of this function is,

$$\Theta(chkMultipleInherit) = m - i$$

The 'updateSingleInherit()' involves finding out the superclass of a particular factorisation and updating the corresponding subclasses. This procedure has an order of complexity of n. The 'updateMultipleInherit()' involves finding out the superclasses for the rest of the factorisation process and then identifying the single and multiple inherit links for each class specification. This routine has an order of complexity of $(m-i)+(m-i)\times n$.

Hence, in the worst case, the order of complexity of this algorithm used to implement the IFP is:

$$\Theta(IFP) \simeq m \times n$$

and is therefore reasonably efficient.

In order to show the significant improvement of this algorithm, selected class hierarchies found in the InterViews library are used to illustrate this†.

InterViews [LCV87, LVC89, VlL88] is a library of C++ classes that defines common interactive objects and composite strategies. There are about 150 classes involved in the construction of the library. The three main class hierarchies found in the library are:

i. Interactors - this class hierarchy contains interactive classes such as buttons and scenes.

ii. Resources - this class hierarchy contains resource classes such as fonts and brush.

iii. Graphics - this class hierarchy contains classes such as circle and polygon.

The value of n, i.e., the number of classes required to construct the class hierarchy can be obtained by counting how many external nodes and internal nodes that contain only one subclass in the class hierarchy. The reason why one does not need to count all the internal and external nodes of the class hierarchy as the value of n is because the IFP is capable of identifying the abstract classes which tend to be situated at the top of the hierarchy. These abstract classes do not need to be specified for the class hierarchy construction problem. The value of m, i.e., the number of attributes involved in a class hierarchy can be obtained by examining the definition of the classes. Assuming the external node of the longest branch of a class hierarchy contains the largest number attributes, one can have a rough idea of what 'm' is by summing up all the public and protected member functions of all the superclasses of the longest branch and add this value to the number of member functions found in the

† The reason why InterViews is chosen in this case is because InterViews presents an average size of a system in which class hierarchies are constructed from scratch. Besides, information about the classes and attributes of InterViews can be easily accessed.

external node. Table 5.4 gives a rough idea of the order of complexity obtained for the three class hierarchies found in InterViews.

	Interactors	Resources	Graphics
n	37	27	14
m	134	28	177
$\sum_{r=2}^{n} \simeq 2^n$	$\simeq 1.37 \times 10^{11}$	$\simeq 1.34 \times 10^{7}$	$\simeq 16000$
$\Theta(IFP) \simeq n \times m$	$\simeq 5000$	$\simeq 750$	$\simeq 2500$

Table 5.4: The Complexity of an Average Size Problem

The value obtained above has shown that even for an average size problem, there is a significant improvement of this algorithm compared with the exponential complexity of the algorithm suggested in section 5.3.1.

5.5. Other Components in the Automation Process

The automation of the inheritance factorisation process is constructed around the inheritance factorisation engine. Now that the implementation of the factorisation engine has been discussed, one can look at the other components and how they are assembled in the automation process.

The class specification editor mainly allows system designers to input the conceptual class specifications of a particular class hierarchy construction problem. The class specification description is then passed through the class specification parser in which the required data structures are generated for the inheritance factorisation engine. The factorisation engine takes these data structures and performs the necessary tasks to yield a set of vertices and edges for the corresponding optimal class hierarchy graph. These sets of vertices and edges are then directed to a graph generator in which the corresponding graph is generated. Currently, the prototype of the editor, the parser and the factorisation engine are completed. Although the graph generator has not been constructed, there is existing software which can be used to build the graph generator [Dea89].

5.6. Conclusion

This section has demonstrated that the inheritance factorisation process can be automated. The automation relies heavily on assembling the inheritance factorisation engine. The

implementation details of such an engine have been discussed, and the efficiency of such an implementation has been analysed. The evaluation of the inheritance factorisation process and the design method as a whole is examined in the next chapter.

"Does this apply always, sometimes, or never?"

~ Sidney Harris ~

Chapter 6

Evaluation

Now that the design method for object-oriented programming has been developed, it is time to assess what has been achieved in this research. As the development of an inheritance factorisation process plays a significant role in this design method, the evaluation process is actually divided into two parts. First of all, it assesses the performance of the inheritance factorisation process. Then it evaluates the design method as a whole. The evaluation process emphasises two aspects:

i. It examines whether the IFP and the design method perform as intended.

ii. It investigates the usability of the IFP and the design method.

6.1. The Inheritance Factorisation Process

The original aim of the inheritance factorisation process was to assist system designers to construct class hierarchies. Therefore, the evaluation process for the IFP is mainly concerned with whether such an objective has been fulfilled or not.

The evaluation process is divided into three parts. The first part is to examine the functionality of the factorisation engine. Here, arbitrary examples are randomly chosen to feed into the engine. The output of the engine which is a class hierarchy graph is then examined. As the examples involved in the first part of the evaluation process are abstract examples, one can only examine the syntax or the structure of the generated graph. In order to examine the semantics of the generated graphs, concrete examples which are drawn from reality are used in the second part of the evaluation process. Here, three concrete examples are used to compare the class construction process in using the traditional 'ad hoc' method, the manual IFP and the automated IFP. The aim of this part of the evaluation process is to highlight the

efficiency of using the automated IFP. The third part of the evaluation process is to examine the current limitations of the IFP.

6.1.1. Introductory Examples

Example 6.1

Example 6.1 is a simple example that examines the functionality of the factorisation engine. The conceptual class specifications, c_0, c_1, c_2 and c_3 can be obtained from tracing the inheritance path which ends with vertex[0], vertex[1], vertex[2] and vertex[3].

```
Inputs are:
-----------
                    c0:a0,a1,a2,a3,a4;
                    c1:a0,a3,a5;
                    c2:a0,a1,a4,a6,a7;
                    c3:a0,a3,a8;


Outputs are:
------------

  Vertices are:                              Edges are:

          vertex[0] = {a2}                       (vertex[4],vertex[5])
          vertex[1] = {a5}                       (vertex[4],vertex[6])
          vertex[2] = {a6,a7}                    (vertex[5],vertex[0])
          vertex[3] = {a8}                       (vertex[5],vertex[1])
          vertex[4] = {a0}                       (vertex[5],vertex[3])
          vertex[5] = {a3}                       (vertex[6],vertex[0])
          vertex[6] = {a1,a4}                    (vertex[6],vertex[2])
```

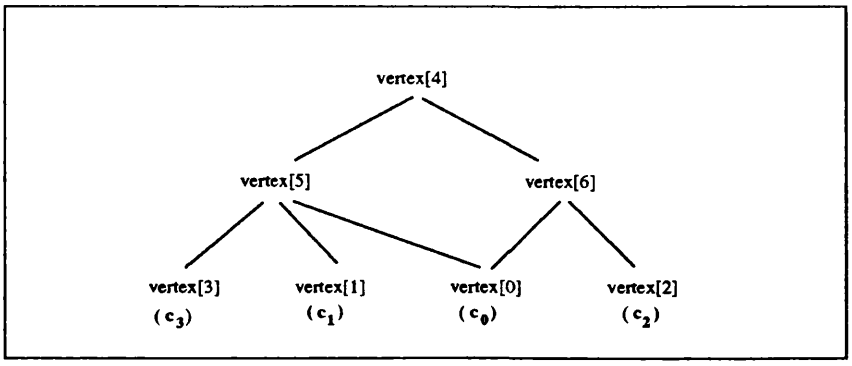


Figure 6.1: Input and Output Of Example 6.1

Example 6.2

Example 6.2 contains a large number of class specifications and attributes.

```
Inputs are:
----------
      c0:a0,a1,a2,a3,a4,a5,a7,a9,a10;        c3:a0,a1,a3,a5,a7,a8,a9;
      c1:a0,a2,a4,a6,a8,a12;                 c4:a0,a1,a2,a4,a5,a10;
      c2:a0,a1;                              c5:a0,a1,a2,a3,a4,a5,a9;

Outputs are:
------------

  Vertices are:

        vertex[0] = {}                 vertex[7] = {a1}
        vertex[1] = {a6,a12}           vertex[8] = {a2,a4}
        vertex[2] = {}                 vertex[10] = {a5}
        vertex[3] = {}                 vertex[11] = {a3,a9}
        vertex[4] = {}                 vertex[12] = {a7}
        vertex[5] = {}                 vertex[13] = {a10}
        vertex[6] = {a0}               vertex[14] = {a8}
  Edges are :

   (vertex[6],vertex[7])   (vertex[7],vertex[4])    (vertex[8],vertex[5])
   (vertex[6],vertex[8])   (vertex[13],vertex[0])   (vertex[10],vertex[0])
   (vertex[6],vertex[10])  (vertex[7],vertex[5])    (vertex[10],vertex[3])
   (vertex[6],vertex[11])  (vertex[13],vertex[4])   (vertex[10],vertex[4])
   (vertex[6],vertex[12])  (vertex[8],vertex[0])    (vertex[10],vertex[5])
   (vertex[6],vertex[13])  (vertex[14],vertex[1])   (vertex[11],vertex[0])
   (vertex[6],vertex[14])  (vertex[8],vertex[1])    (vertex[11],vertex[3])
   (vertex[7],vertex[0])   (vertex[14],vertex[3])   (vertex[11],vertex[5])
   (vertex[7],vertex[2])   (vertex[8],vertex[4])    (vertex[12],vertex[0])
   (vertex[7],vertex[3])   (vertex[12],vertex[3])
```

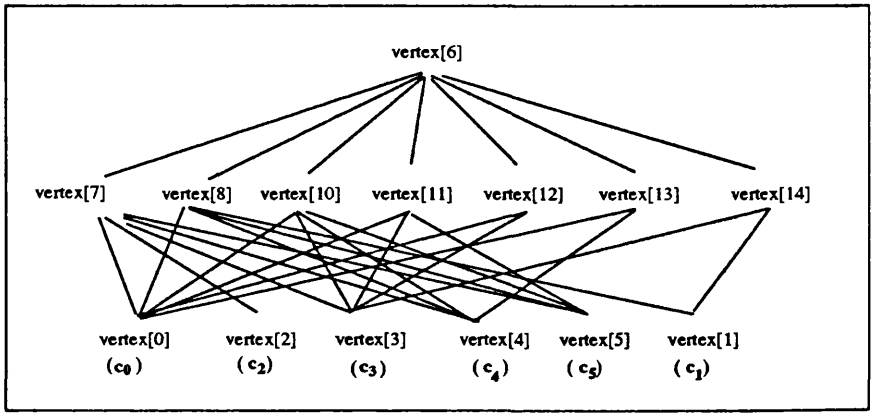


Figure 6.2: Input and Output of Example 6.2

In this example, the conceptual class specifications, $c_0, c_1, c_2, c_3, c_4, c_5$ can be obtained by

tracing the inheritance path that ends with vertex[0], vertex[1], vertex[2], vertex[3], vertex[4], vertex[5]. This example, particularly, demonstrates the efficiency of the factorisation engine as such a hierarchy is difficult to construct by hand.

Example 6.3

Example 6.3 involves the construction of a class hierarchy which has no common attributes to form the root of the graph.

```
Inputs are:
-----------
                    c0:a0,a2,a4,a6;
                    c1:a0,a1,a2,a4,a5;
                    c2:a0,a1,a5,a7,a8;
                    c3:a1,a2,a3;


Outputs are:
------------

Vertices are:           Edges are :

  vertex[0] = {a6}        (vertex[4],vertex[0])     (vertex[6],vertex[3])
  vertex[2] = {a7,a8}     (vertex[4],vertex[1])     (vertex[7],vertex[0])
  vertex[3] = {a3}        (vertex[4],vertex[2])     (vertex[7],vertex[1])
  vertex[4] = {a0}        (vertex[5],vertex[0])     (vertex[8],vertex[1])
  vertex[5] = {a2}        (vertex[5],vertex[1])     (vertex[8],vertex[2])
  vertex[6] = {a1}        (vertex[5],vertex[3])
  vertex[7] = {a4}        (vertex[6],vertex[1])
  vertex[8] = {a5}        (vertex[6],vertex[2])
```

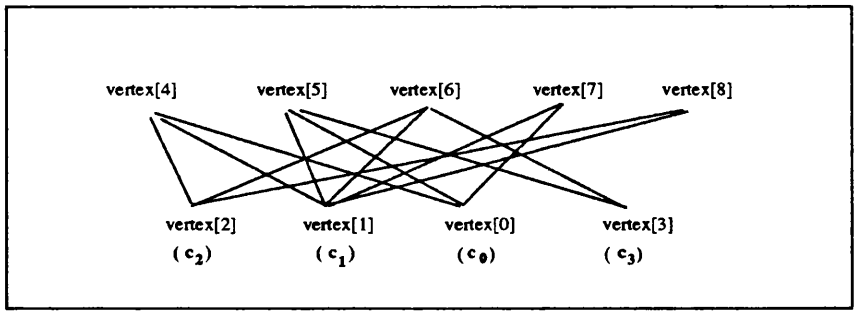


Figure 6.3: Input and Output for Example 6.6

Example 6.4

Example 6.4 is the last example in this part of the evaluation process. The example contains

class specifications that have no common attributes, therefore no inherit relationships, i.e., no information about edges is generated because there are none.

```
Inputs are:
-----------
                    c0:a0,a1,a2,a3,a4;
                    c1:a5,a6,a7;
                    c2:a8,a9,a10;
                    c3:a11,a12,a13;


Outputs are:
------------

Vertices are:

          vertex[0] = {a0,a1,a2,a3,a4}
          vertex[1] = {a5,a6,a7}
          vertex[2] = {a8,a9,a10}
          vertex[3] = {a11,a12,a13}
```

Figure 6.4: Output for Example 6.4

As shown from the above four examples, the factorisation engine works properly. It has fulfilled its task of taking in a set of class specifications and generating the corresponding hierarchy graph.

6.1.2. Miscellaneous Examples

In this part of the evaluation process, concrete examples are shown so that not only the structure but the semantics of the graph can be examined. There are three examples involved. The first two are trivial examples involving single inheritance and multiple inheritance respectively. The third example is a more complicated one. All three examples are used to compare the construction process when using the 'ad hoc', the manual IFP and the automated IFP.

Example 6.5

Example 6.5 is a simple class hierarchy construction problem with single inheritance alone. The problem is to construct an optimal class hierarchy for three classes, the 'Student', the 'Undergraduate' and the 'Graduate' classes. With traditional 'ad hoc' approach, system designers already have some conceptual ideas about the three classes when they encountered the problem. Intuitively, they know that the 'Graduate' class and the 'Undergraduate' class are more specific than the 'Student' class. Hence, even before specifying the properties of

these classes, system designers could confidently say that the 'Student' class is going to be the superclass for the 'Graduate' and the 'Undergraduate' class.

However, with the IFP, one needs proper specifications to start with (see Figure 6.5).

```
Student:StudentId,Name,Age,Address,Phone,Level,EnrollStudent,DropStudent,
        FindStudent,ChangeAddrs,ChangePhone,ChangeLevel;

Graduate:Program,Name,Age,Advisor,StudentId,Address,Department,Phone,
         Level,EnrollStudent,DropStudent,FindStudent,ChangeAddrs,
         ChangePhone,ChangeLevel,EnrollGraduate,DropGraduate,ChgProgram,
         ChgAdvisor,ChgDept;

UnderGraduate:Status,School,Year,StudentId,Name,Age,Address,Phone,Level,
              EnrollStudent,DropStudent,FindStudent,ChangeAddrs,ChangePhone,
              ChangeLevel,EnrollUg,DropUg,ChgStatus,ChgYear,ChgSchool;
```

Figure 6.5: The Definition of a set of Class Specifications

With the manual IFP, the procedure to obtain the optimal class hierarchy is shown in Figure 6.6.

$$
\begin{aligned}
& \{\texttt{Student},\texttt{Graduate},\texttt{Undergraduae}\} \\
= \; & \{\texttt{Student}\}+\{\texttt{Graduate}\}+\{\texttt{Undergraduate}\} \\
= \; & \{\{\texttt{StudentId},\texttt{Name},\texttt{Age},\texttt{Address},\texttt{Phone},\texttt{Level},\texttt{EnrollStudent},\texttt{DropStudent},\texttt{FindStudent}, \\
& \texttt{ChangeAddrs},\texttt{ChangePhone},\texttt{ChangeLevel}\}\}+\{\{\texttt{Program},\texttt{Name},\texttt{Age},\texttt{Advisor}, \\
& \texttt{StudentId},\texttt{Address},\texttt{Department},\texttt{Phone},\texttt{Level},\texttt{EnrollStudent},\texttt{DropStudent},\texttt{FindStudent}, \\
& \texttt{ChangeAddrs},\texttt{ChangePhone},\texttt{ChangeLevel},\texttt{EnrollGraduate},\texttt{DropGraduate},\texttt{ChgProgram}, \\
& \texttt{ChgAdvisor},\texttt{ChgDept}\}\} + \{\{\texttt{Status},\texttt{School},\texttt{Year},\texttt{StudentId},\texttt{Name},\texttt{Age},\texttt{Address},\texttt{Phone}, \\
& \texttt{Level},\texttt{EnrollStudent},\texttt{DropStudent},\texttt{FindStudent},\texttt{ChangeAddrs},\texttt{ChangePhone}, \\
& \texttt{ChangeLevel},\texttt{EnrollUg},\texttt{DropUg},\texttt{ChgStatus},\texttt{ChgYear},\texttt{ChgSchool}\}\} \\
= \; & \{\texttt{StudentId},\texttt{Name},\texttt{Age},\texttt{Address},\texttt{Phone},\texttt{Level},\texttt{EnrollStudent},\texttt{DropStudent},\texttt{FindStudent}, \\
& \texttt{ChangeAddrs},\texttt{ChangePhone},\texttt{ChangeLevel}\} \lhd_s (\{\{\texttt{Status},\texttt{School},\texttt{Year},\texttt{EnrollUg},\texttt{DropUg}, \\
& \texttt{ChgStatus},\texttt{ChgYear},\texttt{ChgSchool}\}\} + \{\{\texttt{Program},\texttt{Advisor},\texttt{Department},\texttt{EnrollGraduate}, \\
& \texttt{DropGraduate},\texttt{ChgProgram},\texttt{ChgAdvisor},\texttt{ChgDept}\}\}+\{\phi_c\})
\end{aligned}
$$

Figure 6.6: The Result obtained with Manual IFP

One can see that using the manual IFP is not very efficient and the manipulation process can be fairly tedious when there are a large number of attributes involved. However, with the help of the factorisation engine, the result can be obtained in a more efficient manner. System designers only need to specify the involved class specifications as input to the factorisation

```
Vertices are:

vertex[1] = {Program,Advisor,Department,EnrollGraduate,DropGraduate,
             ChgProgram,ChgAdvisor,ChgDept}
vertex[2] = {Status,School,Year,EnrollUg,DropUg,ChgStatus,ChgYear,
             ChgSchool}
vertex[3] = {StudentId,Name,Age,Address,Phone,Level,EnrollStudent,DropStudent,
             FindStudent,ChangeAddrs,ChangePhone,ChangeLevel}

Edges are:

          (vertex[3],vertex[1])
          (vertex[3],vertex[2])
```

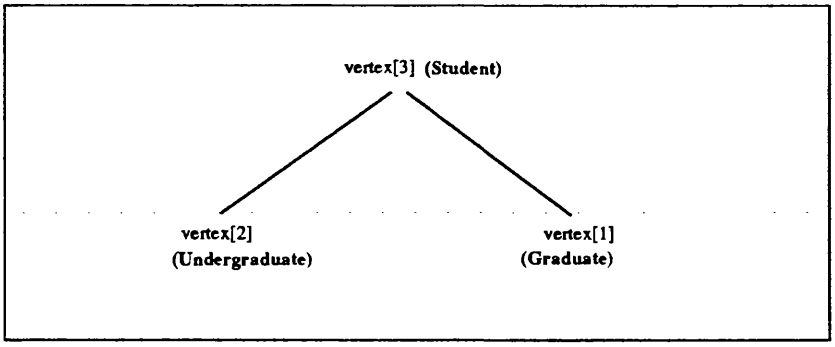


Figure 6.7: Result Obtained from the Automated IFP for Example 6.5

engine and the set of vertices and edges of the corresponding hierarchy graph are then generated (see Figure 6.7).

Example 6.6

In Example 6.6, the class hierarchy construction problem involves multiple inheritance. It contains four class specifications, BoundedPt, Point, HistoryPt and BhPoint. In this example, the names of the classes involved do not convey obvious enough meanings or concepts about the definitions of the class specifications. Hence system designers cannot construct the hierarchy graph just by looking at the names but have to specify the involved classes.

With the 'ad hoc' method, system designers probably start with the class 'Point' because it seems to be a very general class. The class 'Point' should be able to create, locate, move and display a point. Hence, these would be the attributes belong to the 'Point' class. A 'BoundedPt' is simply a 'Point' with two attributes: 'min' and 'max' to define the boundary of a point. A 'HistoryPt' is also a 'Point' but has an attribute 'history' which gives the history information. A 'BhPoint' is a 'Point' with both the capability of the 'BoundedPt' and the 'HistoryPt' (see Figure 6.8). Therefore, it is logical to construct the class 'Point'

to be the superclass in which the 'BoundedPt' and the 'HistoryPt' inherited from. Since the 'BhPoint' should contain the capabilities of both the 'BoundedPt' and the 'HistoryPt', hence, it should be multiply inherited from the 'BoundedPt' and the 'HistoryPt' classes.

```
BoundedPt:min,max,create,locate,move,display;

Point:create,locate,move,display;

HistoryPt:locate,move,create,display,history;

BhPoint:create,locate,move,min,display,max,history,boundHistory;
```

Figure 6.8: The Definition of the set of Class Specifications of Example 6.6

With the manual IFP, the manipulation is shown in Figure 6.9

$$
\begin{aligned}
& \{\texttt{BoundedPt,Point,HistoryPt,BhPoint}\} \\
= \; & \{\texttt{BoundedPt}\}+\{\texttt{Point}\}+\{\texttt{HistoryPt}\}+\{\texttt{BhPoint}\} \\
= \; & \{\{\texttt{min,max,create,locate,move,display}\}\}+\{\{\texttt{create,locate,move,display}\}\}+ \\
& \{\{\texttt{locate,move,create,display,history}\}\}+\{\{\texttt{create,locate,move,min,display,}\\ & \texttt{max,history,boundHistory}\}\} \\
= \; & \{\texttt{locate,create,display,move}\}\lhd_s(\{\{\texttt{min,max}\}\}+\{\phi_c\}+\{\{\texttt{history}\}\}+\{\{\texttt{min,max,history,}\\ & \texttt{boundHistory}\}\}) \\
= \; & \{\texttt{locate,create,display,move}\}\lhd_s(\{\phi_c\}+\{\texttt{min,max}\}\lhd_s(\{\phi_c\}+\{\{\texttt{history,boundHistory}\}\})+ \\ & \{\{\texttt{history}\}\}) \quad [\xi_{N1}] \\
= \; & \{\texttt{locate,create,display,move}\}\lhd_s(\{\phi_c\}+\{\texttt{history}\}\lhd_s(\{\phi_c\}+\{\{\texttt{min,max,boundHistory}\}\})+ \\ & \{\{\texttt{min,max}\}\}) \quad [\xi_{N2}]
\end{aligned}
$$

since multiple inheritance is detected, hence the normalised expression is:

$$
= \{\texttt{locate,create,display,move}\}\lhd_s(\{\phi_c\}+\{\texttt{history}\}\lhd_s\{\phi_c\}+\{\texttt{min,max}\}\lhd_s\{\phi_c\}+ (\{\{\texttt{history}\}\}+\{\{\texttt{min,max}\}\})\lhd_m\{\texttt{boundHistory}\})
$$

Figure 6.9: The Results obtained from the Manual IFP

With the factorisation engine, the class specifications are input into the engine to generate the set of vertices and edges for the corresponding graph (see Figure 6.10).

The results from Example 6.5 and Example 6.6 indicate that the same hierarchy graph is obtained with the 'ad hoc', the manual IFP and the automated IFP methods. The 'ad hoc' method is found to be a better choice if the names of the classes actually convey some meanings that contributes to the identification of the superclass/subclass link. However,

```
Vertices are:  vertex[3] = {boundHistory}
               vertex[4] = {create,locate,move,display}
               vertex[5] = {history}
               vertex[6] = {min,max}

Edges are:  (vertex[4],vertex[5])
            (vertex[4],verter[6])
            (vertex[5],vertex[3])
            (vertex[6],vertex[3])
```

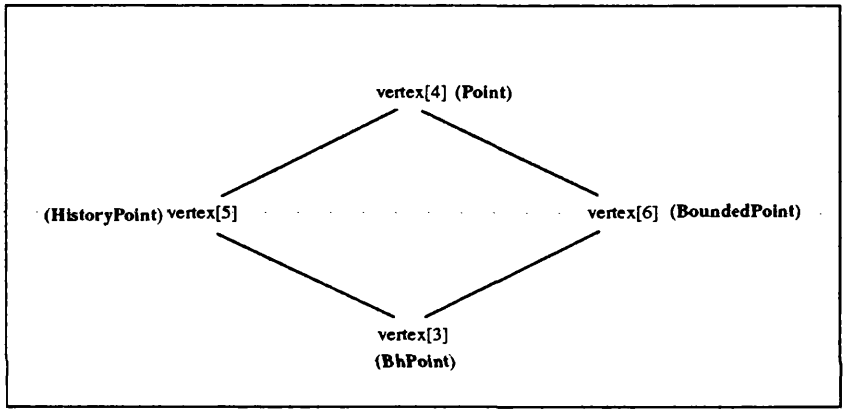


Figure 6.10: The Results Generated for Example 6.6

depending on individual experience, a name may mean different things to different people. Hence, with the same collection of classes, different hierarchy graphs may be constructed by different people. The IFP provides a more systematic and consistent approach in constructing class hierarchies. With the IFP, the same graph is generated everytime with the same set of class specifications. When the manual IFP is compared with the automated one, it is obvious that if the problem involves a large number of classes and attributes, the manual manipulation becomes tedious and mistakes may easily be introduced. The automated IFP is an isomorphism of the manual IFP with the manipulation process carried out automatically by the factorisation engine. System designers benefit more with the automated IFP.

Example 6.7

The above two examples are carefully chosen such that the conceptual classes specifications are all the classes needed for the hierarchy graph. However in reality, system designers may find that they have to identify the appropriate abstract classes before they can build the class hierarchy. The following example tries to illustrate this point and also demonstrates how the

procedure to identify the correct abstract classes is actually carried out automatically by the IFP.

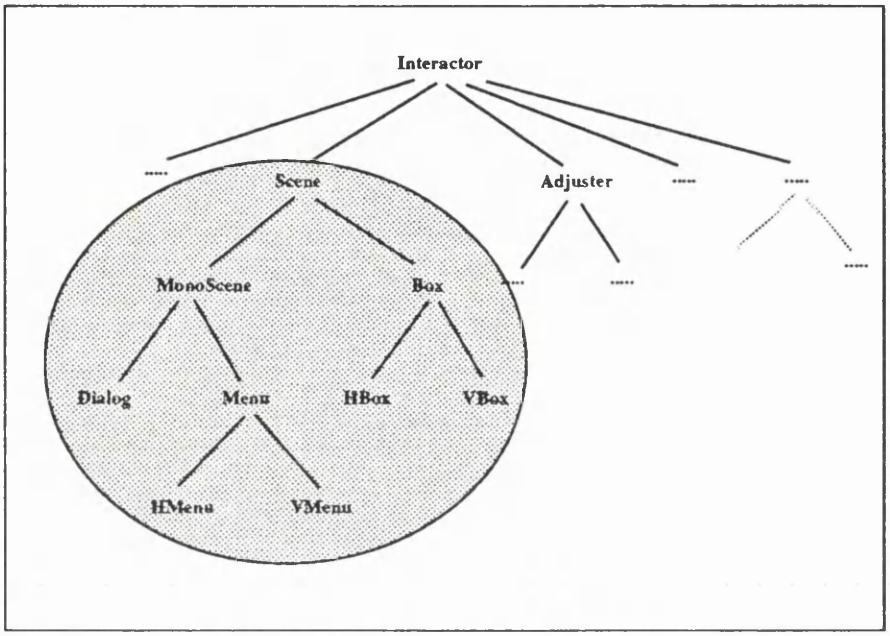


Figure 6.11: An Extract from the InterViews System Class Hierarchy

This example is based on the library classes found in InterViews Version 2.4 [Lin88, LVC89]. InterViews is a library which provides classes that support the design and implementations of user interfaces. It is object-oriented and is written in C++ [Str86b]. The library runs on top of the X window system [Sch86]. In InterViews, part of the system class hierarchy contains the Menu class and the Dialog class as shown in Figure 6.11. This example shows that by giving the conceptual class specifications, HBox, VBox, Dialog, HMenu and VMenu alone. The IFP generates a class hierarchy structure which matches the one given in the InterViews library. The descriptions of these classes and their attributes are summarised in Figure 6.12.

Note that the specifications defined for this example are modified from that of InterViews. This is because the class specifications found in InterViews contain implementation details which are unknown to the system designers at the stage of using the IFP. For example, the attributes of the specifications found in InterViews are categorised into private, protected and public category as required by the C++ programming language. This is unnecessary when using the IFP. Therefore, the class specifications defined for this experiment is a simplified version of that found in InterViews itself.

The trial starts off by specifying the class specifications for the IFP. This is shown in Figure 6.13.

```
The class 'HMenu' is a menu in which its menu items are arranged horizontally.
It contains the following attributes:

     HMenuCreate - creates an instance of the horizontal menu.
     Insert - insert a menu item to the menu.
     Compose - build the box representation of the menu.
     GetSelection - return the current selected item.
     Handle - the handle operation reads events in until a button is released,
              passing the events to the appropriate menu item.
     Change - notify there is a change of the shape.
     Remove - remove the object from the scene.
     Draw - draw the object.
     Lower - put the object behind any objects that overlapping it.
     Raise - raise the object to the front, i.e., makes it visible.
     Resize - change the size of the object.

The class 'VMenu' is similar to that of 'HMenu' except that it has a different
create operation called 'VMenuCreate' and it creates vertical menus.

The class 'Dialog' is a dialogbox which handles input from the user. Like the
'HMenu' and the 'VMenu' class, the 'Dialog' class contains attributes Change,
Remove, Lower, Raise, Draw and Resize. Besides these, it contains two other
attributes:

     CreateDialog - create the dialog box.
     buttonState - this is to indicate the state of the button.
     Accept - This sets the dialog button state to zero and loop reading events
              until the button state's value becomes non-zero, i.e., the user
              presses the button.

Then, there is the 'HBox' class. The 'HBox' class basically groups the interactor
objects into a horizontal box. It has attributes Change, Remove, Lower, Raise
as mentioned earlier. Besides, it has the following operations:

     CreateHBox - create the horizontal box.
     DrawBox - draw all the interactor objects within the box.
     Align - aligning all the interactor objects within the box.
     align - an instance variable which specifies how to align
             the interactor objects.
     ResizeBox - change the size of the box.
     ComputeHBoxShape - obtain the shape of the horizontal box.
     PlaceHBox - put the horizontal box on the scene.

The 'VBox' class is similar to that of 'HBox' except that it has a different
create operation, compute box shape operation and place box operation which is
named as CreateVBox, ComputeVBox and PlaceVBox.
```

Figure 6.12: The Description of the Class Specifications for Example 6.7

When these class specifications are fed into the factorisation engine, the set of vertices and edges are then produced. This is shown in Figure 6.14. If the class hierarchy graph of Figure 6.14 is compared to that of Figure 6.11, it is found that they share the same graph structure. Further, when the contents of the vertices in Figure 6.14 are checked against those

```
HMenu:HMenuCreate,Insert,Compose,GetSelection,Change,Handle,Remove,Lower,Raise,
      Draw,Resize;

VMenu:VMenuCreate,Insert,Compose,GetSelection,Change,Handle,Remove,Lower,Raise,
      Draw,Resize;

Dialog:CreateDialog,Accept,buttonstate,Change,Remove,Lower,Raise,Draw,Resize;

HBox:CreateHBox,Align,align,DrawBox,ResizeBox,ComputeHBoxShape,Change,Remove,
     Lower,Raise,PlaceHBox;

VBox:CreateVBox,Align,align,DrawBox,ResizeBox,ComputeVBoxShape,PlaceVBox,
     Change,Remove,Lower,Raise;
```

Figure 6.13: The Definitions of the Class Specifications for Example 6.7

```
Vertices are:
          vertex[0] = {HMenuCreate}
          vertex[1] = {VMenuCreate}
          vertex[2] = {CreateDialog,Accept,buttonstate}
          vertex[3] = {CreateHBox,ComputeHBoxShape,PlaceHBox}
          vertex[4] = {CreateVBox,ComputeVBoxShape,PlaceVBox}
          vertex[5] = {Change,Remove,Lower,Raise}
          vertex[6] = {Draw,Resize}
          vertex[7] = {Align,align,DrawBox,ResizeBox}
          vertex[8] = {Insert,Compose,GetSelection,Handle}

Edges are:
  (vertex[5],vertex[6])   (vertex[7],vertex[3])   (vertex[5],vertex[7])
  (vertex[7],vertex[4])   (vertex[6],vertex[2])   (vertex[8],vertex[0])
  (vertex[6],vertex[8])   (vertex[8],vertex[1])
```

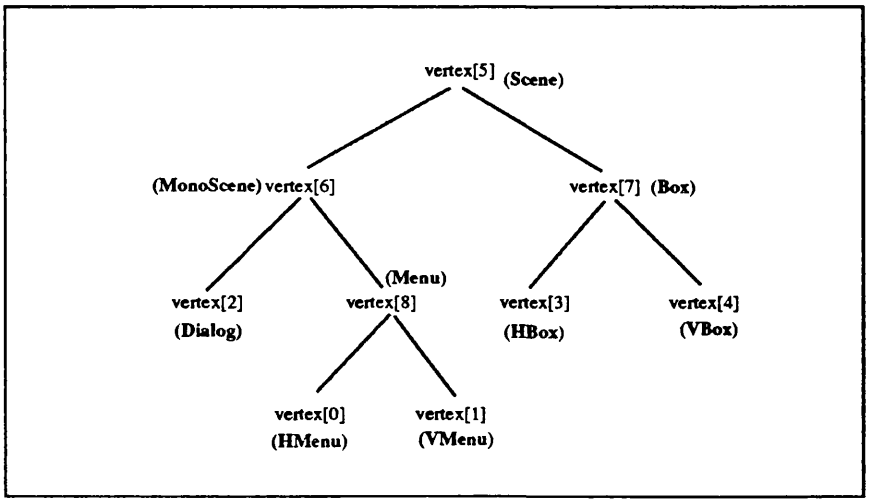


Figure 6.14: The Graph Structure obtained from Example 6.7

in Figure 6.11, the vertices in Figure 6.14 are found to be matched to that in Figure 6.11. For example, vertex[5] is found to be matched to the class Scene and the vertex[6] is found to be matched to the class MonoScence. This indicates that the same graph that is found in InterViews, which is constructed with the 'ad hoc' method by experienced object-oriented programmers, can be generated by the IFP.

More importantly, this example has highlighted an advantage of using the IFP in constructing class hierarchies. To construct the same graph with the 'ad hoc' method, system designers have to apply a top-down design approach. They have to identify the necessary abstract classes in constructing class hierarchies. Abstract classes are classes that seldom have instances and are always situated higher up in the class hierarchy [Joh88]. As humans think better about the concrete examples than abstractions, the identification of the abstract classes is never an easy task [Joh88].

For instances, 'Scene' and 'MonoScene' are two abstract classes found in the class hierarchy (see Figure 6.11) and they are not as apparent as classes such as 'Box' or 'Menu'. Hence, system designers may encounter difficulties in constructing the hierarchy. However, with the IFP, system designers do not need to worry about identifying abstract classes. As shown in Figure 6.14, the required abstract classes for the class hierarchy such as vertex[5], vertex[6] and vertex[7] are automatically highlighted in the factorisation process. This also means that the IFP allows more flexibility in constructing class hierarchies. Not only the top-down design approach can be applied but also the middle-out and bottom-up. In this sense, the IFP has demonstrated its contribution in constructing class hierarchies.

6.1.3. Limitations of the IFP

Of course, it is unwise to be dogmatic about the usage of the IFP. As it stands, there are some shortcomings found in the IFP which limit its usage. These inadequacies are, in fact, more to do with the philosophical issues than the technical issues of the IFP.

6.1.3.1. The Requirement for a Precise Specification

The factorisation engine is simply a tool that takes in a set of class specifications and generates the corresponding hierarchy graph as output. The structure of the generated graph is governed by the manipulation rules defined for the factorisation engine. As these rules are pre-defined, the generated graph solely depends on the input data, i.e., the set of class specifications.

If Example 6.7 is carried out with a slightly different set of class specifications, e.g., 'Menu',

'Dialog', 'HBox' and 'VBox' alone, a different class hierarchy graph is then obtained. This is shown in Figure 6.15.

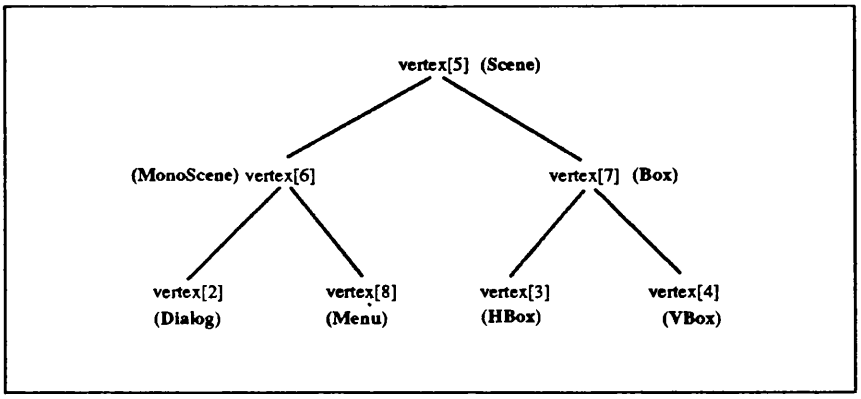


Figure 6.15: The Class Hierarchy Graph without the 'HMenu' and 'VMenu'

As one can see, the basic structure of the graph is still preserved but the class 'HMenu' and 'VMenu' are not identified. This indicates that the IFP is capable of identifying classes which are higher up in the class hierarchy but not those which are below the given conceptual classes. In other words, the IFP can identify the abstract classes in the class hierarchy but not the conceptual classes other than those which are given for the IFP in the first place. However, with the 'ad hoc' method, system designers may know that since a 'Menu' class is in the hierarchy, intuitively, a 'HMenu' and a 'VMenu' class will be the subclasses of it.

Not only is the set of the class specifications important to the construction of the hierarchy graph but also the individual specifications for each class, i.e., the attributes which belong to a particular class. Say in the previous trial, if in Example 6.7, the system designer forgot to define the 'Remove' and the 'Draw' operation for the 'HMenu' class (see Figure 6.16), a different class hierarchy graph is then generated from the IFP. This is illustrated in Figure 6.17.

In order to obtain a reasonable class hierarchy, system designers have to provide a reasonable set of class specifications for the factorisation engine in the first place. These class specifications have to be carefully constructed so that they contain all the properties of a particular class. However, to understand a class thoroughly and be able to identify all the properties of that class is not an easy task. This is the reason why the concrete examples used in this part of the evaluation process are comparatively trivial than those used in the first part of the evaluation process. It is always difficult to come up with a set of fully specified class specifications.

```
HMenu:HMenuCreate,Insert,Compose,GetSelection,Change,Handle,Lower,Raise,
      Resize;

VMenu:VMenuCreate,Insert,Compose,GetSelection,Change,Handle,Remove,Lower,Raise,
      Draw,Resize;

Dialog:CreateDialog,Accept,buttonstate,Change,Remove,Lower,Raise,Draw,Resize;

HBox:CreateHBox,Align,align,DrawBox,ResizeBox,ComputeHBoxShape,Change,Remove,
     Lower,Raise,PlaceHBox;

VBox:CreateVBox,Align,align,DrawBox,ResizeBox,ComputeVBoxShape,PlaceVBox,
     Change,Remove,Lower,Raise;
```

Figure 6.16: The Class Specifications with a Different 'HMenu' Class

```
Vertices are:
  vertex[0] = {HMenuCreate}                          vertex[1] = {VMenuCreate}
  vertex[2] = {CreateDialog,Accept,buttonstate}      vertex[6] = {Remove}
  vertex[3] = {CreateHBox,ComputeHBoxShape,PlaceHBox} vertex[7] = {Resize}
  vertex[4] = {CreateVBox,ComputeVBoxShape,PlaceVBox} vertex[10] = {Draw}
  vertex[5] = {Change,Lower,Raise}
  vertex[8] = {Insert,Compose,GetSelection,Handle}
  vertex[11] = {Align,align,DrawBox,ResizeBox}

Edges are:
  {vertex[5],vertex[6]}    {vertex[5],vertex[7]}    {vertex[5],vertex[8]}
  {vertex[5],vertex[10]}   {vertex[5],vertex[11]}   {vertex[6],vertex[1]}
  {vertex[6],vertex[2]}    {vertex[6],vertex[3]}    {vertex[6],vertex[4]}
  {vertex[7],vertex[0]}    {vertex[7],vertex[1]}    {vertex[7],vertex[2]}
  {vertex[8],vertex[0]}    {vertex[8],vertex[1]}    {vertex[10],vertex[1]}
  {vertex[10],vertex[2]}   {vertex[11],vertex[3]}   {vertex[11],vertex[4]}
```

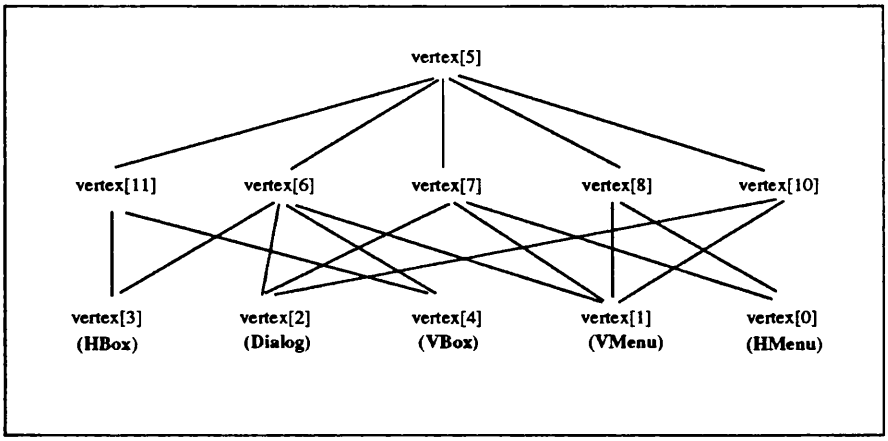


Figure 6.17: Result Generated with Different Class Specifications

6.1.3.2. The Practicability of the IFP

Although the basic model of the inheritance factorisation process is soundly based, whether it is generally applicable to systems design in object-oriented programming is still uncertain. The uncertainty concerning the practicability of the IFP is three fold:

i. The IFP requires a set of well-defined class specifications in order to obtain a well-defined class hierarchy and this is not easy to achieve. This may put off system designers from using the IFP.

ii. The conventional behaviour in constructing class hierarchies is based on intuition. A top-down approach which identifies the superclass first and then appends sub-classes later is employed. However, with the IFP, different attitudes in constructing class hierarchies are introduced. The IFP encourages system designers to concentrate on specifying the characteristics of the classes. It suggests the hierarchy be constructed at one time instead of a 'build a little and append the rest' approach. As it is always difficult for human to abandon their old habits and take up a new approach, there may be difficulties in persuading them to use the IFP.

iii. It is discussed in Chapter 4 that most system designers find that they have to append a new class to an existing class hierarchy. As the current IFP does not cater for the activity of adding a new class to an existing class hierarchy, the current IFP may not be as useful as it should be.

Besides, the graph obtained from the IFP is, in fact, governed by the manipulation rules defined in the formal model. At the moment, the formal model is fairly rudimentary. It defines an optimal graph as a graph that represents minimum duplication of attributes, i.e., maximum reusability. It also assumes that multiple inheritance is more desirable than single inheritance. Hence, if multiple inheritance is detected in a class hierarchy construction problem, the graph that exhibits the multiple inheritance relationship is chosen as the optimal graph. All these assumptions have restricted the structure of the graph that is generated from the IFP. In reality, there are situations when system designers prefer to have a single inheritance graph than a multiple inheritance one. Further, a higher degree of resuability does not always give graph that is optimal to everyone.

When comparing the IFP to the 'ad hoc' method, the 'ad hoc' method tends to give more flexibility to system designers constructing class hierarchies in this aspect. The intuitive knowledge and the experience of the system designers are employed in constructing class hierarchies using the 'ad hoc' method so that system designers can tailor the graph to their own definitions of 'optimal'. It is believed that the IFP will be more practical if it is modified to allow users to exercise their intuitive knowledge and desire in constructing the

class hierarchy. Perhaps, an expert-system like front-end is required for the IFP so that system designers can specify the rules about how the graph should be generated. This kind of extension and modification has to be part of the future work.

6.2. The Design Method

As indicated by Rosson and Gold [RMK88], too little is known about the complexities of design practices, thus it is difficult to set out to test hypothesis about the effects of particular design methods, i.e., to evaluate a design method. The evaluation of the design method developed in this research, therefore, tends to be informal. First of all, it checks whether this design method has satisfied the general goal of a design method. Then, it examines whether the objectives of developing this design method have been achieved or not. It also compares this design method to existing design methods to highlight the improvement of the design method. Of course, an important part of the evaluation of the design method has to be based on user performance. Currently, two case studies have been carried out to demonstrate the usage of this design method. These two case studies can be found in Appendix A and B. Other plans have been set up to involve other system designers besides the developer to examine the usability of the design method. These evaluation plans are discussed later in this section.

6.2.1. General Assessment

As has been reiterated several times, design is a creative process. The design process involves generating ideas and trying them out. A design method can be regarded as a general guideline which helps system designers to develop these ideas at different stages in the design process. The design method also helps in organising the design activities to capture and express these ideas. The design method arising from this research certainly has achieved this basic goal of a design method. The design method extends from the last part of the analysis phase to the beginning of the implementation phase. This gives a continuity notion in the system development. The design method splits the design phase into three stages, the conceptual level, the system level and the specification level. Each stage has a specific goal to achieve. The tasks which are assigned for individual stages are defined clearly in the design method. The accomplishment of each task at individual stages and the proceeding from one stage to the next adds more details to the implementation model. The results of individual design stages are to be expressed in design documents such as the object interaction diagrams and the class structure charts. The graphical notations of these documents are well-defined so that system designers can communicate in a semi-formal and unambiguous manner.

Beside the general objective, this design method has a specific objective to achieve. The design method has been specially developed for object-oriented implementations. Therefore, the design method has to encompass guidelines that help system designers solve problems which are specific to object-oriented programming. For example, the design method should discuss how to construct objects and class hierarchies. As it stands, this design method has also achieved this specific aim. There are detailed descriptions about how to identify objects and the appropriate relationships which lead to the construction of objects. Further, the design method has incorporated an algorithm which is called the 'inheritance factorisation process' that guides system designers to construct well-defined class hierarchies. The development of the inheritance factorisation process has made this design method more adequate than other existing object-oriented design methods in supporting object-oriented programming.

The success of a good design method needs the complement of appropriate software tools for recording and managing the results of analysis, design and implementation. Hence, the next step of this research is to develop the necessary tools to support the design method. The future work concerning this aspect is discussed in Chapter 7.

6.2.2. Comparison with Other Existing Design Methods

Most of the existing design methods are function-oriented and were developed long before the flourishing of object-oriented programming. Therefore, they are inadequate for giving guidelines in designing systems which are targeted at object-oriented programming. This design method, having achieved the general and specific goals as mentioned in the above section, has demonstrated its suitability in serving as a design method for object-oriented programming.

When comparing this design method with other object-oriented design methods, it is found that this design method has remedied the inadequacies that are identified in other object-oriented design methods. These can be classified into the following four areas:

i. Some of the existing object-oriented design methods such as the earlier version of the Booch and the HOOD methods have been developed for a specific programming language, Ada. This results in design methods which are not general enough to be used by other systems that are targeted at other programming languages. Moreover, according to this research, Ada is not regarded as an object-oriented programming language. Therefore the design methods which are specially developed for it, are inadequate for object-oriented programming. The design method which emerges from this research, however, is not targeted at any particular programming language but a programming paradigm. Therefore, it is comparatively more flexible than the Booch and the HOOD design methods.

ii. One of the criticism of existing object-oriented design methods is that they do not have enough details to guide system designers to design their systems in an object-oriented fashion. Most of them only contain graphical notations and not the procedures which tell system designers when to do what in the design phase. This design method gives enough guidelines for system designers to proceed through the whole design phase. It begins with emphasising the construction of a conceptual model for the application and ends with the design specifications which the programmers can take away and implement the system with.

iii. Also, one or two object-oriented design methods introduce unnecessary new terms and diagrammatic notation to the system designers. This does not help them to grasp the philosophy of the object-oriented design in a reasonable short time. The development of this design method recognised that many system designers already have a strong background in structured design methods. In order to bring down the learning curve and provide a smooth transition for these system designers, this design method employs graphical constructs which are similar to those in structured design methods. For example, the object interaction diagrams are similar to the data flow diagrams and the class structure charts remind system designers of the structure charts found in structured design methods.

iv. However, a more important improvement of the design method over other object-oriented design methods is that it provides substantial support in handling inheritance. None of the existing object-oriented design methods have enough support to handle inheritance in the design phase. This design method provides a formal manipulation process to assist system designers to construct class hierarchies. This has confirmed the reason why this design method is better in supporting systems design in object-oriented programming when compared to other existing design methods.

6.2.3. Evaluation by Other Users

The usage of this design method has been demonstrated with the two case studies which can be found in Appendix A and Appendix B. However, as these two case studies are carried out by the developer of the design method, it may give a biased judgement on the usability of the design method. Therefore, it is decided that the design method must be assessed by system designers other than the developer. As the evaluation which involved other users especially those from industry will be a long process, it was decided that such an evaluation process will not be included in this research. Nevertheless, the plan to carry out this kind of evaluation is discussed Chapter 7 as part of the future work.

6.3. Conclusion

This chapter discusses the evaluation on the inheritance factorisation engine and the design method. The evaluation attempts to examine the functionality and the usability of both the factorisation engine and the design method.

Concerning the inheritance factorisation engine, various examples have been used to demonstrate the functionality of the factorisation engine. The pros and cons in using the engine to generate the required class hierarchy have also been discussed. It is believed that the factorisation engine forms the basis of a CASE tool which assists system designers to construct class hierarchies.

Concerning the design method, the usage of the design method in systems design is demonstrated with two case studies found in Appendix A and B. The virtues of the design method have been highlighted through a comparison with other design methods.

Due to the time constraint and the current status of this research, the evaluation of the usability with other users have not been carried out. However, it is realised that such an evaluation is important and plans to carry them out in future are discussed in Chapter 7.

"The same thrill, the same awe and mystery, comes again and again when we look at any question deeply enough. With more knowledge comes a deeper, more wonderful mystery, luring one on to penetrate deeper still. Never concerned that the answer may prove disappointing, with pleasure and confidence we turn over each new stone to find unimagined strangeness leading on to more wonderful questions and mysteries - certainly a grand adventure!"

~ Richard P. Feynman ~

Chapter
Chapter 7

Future Work

Now that the work concerning this piece of research has been presented and discussed, it is time to talk about the future work that should follow.

With respect to the immediate future work, there is the extension and modification of the inheritance factorisation process and the evaluation of the usability of both the factorisation engine and the design method. Such an evaluation will provide valuable feedback which can improve the method further. To design the experiments for such an evaluation is a long term process as it involves detailed study of design, how to specify usability in measurable terms and how to analyse the data obtained. Although giving a detail account of the design of these experiments is outside the scope of this research, it is possible to outline these experiments and discuss some important issues that these experiments have to take care of.

With respect to future work in the long term, the design method of this research has provided part of the backbone for a computer aided software engineering (CASE) for object-oriented programming. The design method has specified the tasks which have to be performed in the design phase. At the same time, it has highlighted the tools which are required to support the design phase in the CASE environment. To develop these tools will be part of the work in the future.

This chapter, therefore, explores the future work in these two categories.

7.1. Immediate Future Work

7.1.1. The Inheritance Factorisation Process

As mentioned both in Chapter 4 and Chapter 6, the current status of the factorisation process only supports the construction of class hierarchies from scratch. However, very often, system designers find that they want to add new classes to existing class hierarchies. Therefore, the prime interest of the immediate future work is to extend the current model to cover this aspect. Further, by introducing priority attributes factorisation gives more flexibility to the users. Hence, the prototype of the factorisation engine has to be modified to support these two features before any tests concerning the usability of the model could be carried out.

The tests on whether the factorisation process is practical or not can, in fact, be incorporated in the evaluation of the usability of the design method which is discussed in section 7.1.2. If the result of the evaluation proves to be positive, more work can be carried out with the factorisation process. This includes extending the model to support signature-compatible IFP and behaviour-compatible IFP as mentioned in Chapter 4. In addition, one can start looking into how to modify the IFP so that it can automatically generate the required data structures with respect to the implementation language.

7.1.2. Usability of the Design Method

The design method of this research has defined the procedure of the design phase which leads to the construction of an object-oriented implementation model. Before this method can be refined in more detail and extended further, one needs to set up experiments to test the usability of the design method.

It is difficult to define what usability means exactly and hence specify usability in measurable terms. However, the criteria for usability can be summarised into two areas [Goo87]:

i. Ease of use,

ii. The functions provided match the task requirements.

Concerning the ease of use, there are a number of issues governing it:

i. Martin [Mar73] has written that a user's ability to use a system depends on the ease with which he/she can communicate with it, i.e., whether the design method has provided sufficient means for communication.

ii. Very often, the means of communication of a design method are graphical and textual constructs. The design method specifies the rules that govern the usage of

these constructs. To examine whether these graphical and textual constructs are easy to use is another test to check the usability of the design method.

iii. One of the aims of a design method is to divide the design activity into manageable sub-tasks. Hence, to see whether a design method is easy to use, one has to check whether the users feel comfortable and confident about how to achieve these sub-tasks.

iv. Besides dividing the sub-tasks, the design method also specifies the procedures to achieve these sub-tasks. To check whether these procedures are clearly defined and easy to follow also leads to an answer of whether the design method is easy to use.

Considering the functionality of a design method, again there are a few issues governing it:

i. The main objective of a design method is to set up the correct implementation model. Therefore, a way to check whether the design method has achieved its required functionality is to examine how well the design method helps in setting up an implementation model for object-oriented programming. The following are a few questions which may lead to answers about whether the design method helps to set up the correct implementation model for object-oriented programming.

   - Does the design method help to identify the objects and actions of a system?
   - Has the design method provided enough guidelines in constructing a class?
   - Does the design method provide enough facilities to help system designers in building class hierarchies?
   - Do the users feel that they have the confidence to take away the design specifications and implement the system directly?

ii. Besides checking whether the design method helps in setting up the required implementation model, another area needs to be examined is whether the design method helps in modelling the problem.

In order to compare the usability of this design method and other design methods with respect to setting up the correct implementation model for object-oriented programming, the experiments should be designed to include other design methods, for example, a few popular conventional design methods and some object-oriented design methods.

The above description of the experiment is summarised into Table 7.1.

	This Method	*Yourdon*	*JSD*	*HOOD*	*ObjectOry*
Ease of Use 1. ease of communication 2. graphical and textual constructs easy to use 3. manageable and easy to follow sub-tasks 4. clearly defined procedure					
Functionality 1. help to identify objects and actions 2. help to construct classes 3. help to build class hierarchies 4. problem modelling					

Table 7.1: The Result of the Experiment in table format

In addition, usability is also affected by the types of tasks to be accomplished. Therefore, different types of problems must be set for the experiments, for example, typical object-oriented problems, information-oriented problems and real-time systems etc. Further, the characteristics that make a design method usable for one set of users may be unusable for another. Hence, there will be four groups of subjects for each design method, the novice users of both the object-oriented programming and the design method (Group A), the experts of both the object-oriented programming and the design method (Group B), the novice users of the object-oriented programming but experts in the design method (Group C) and lastly the experts of the object-oriented programming and novice of the design method (Group D).

To compare the results obtained, subjects might be requested to give a score for each item. The score ranges from 4 which is perfect to 0 which is no evidence of any support (see Table 7.2).

4	Perfect
3	Good
2	Average
1	Poor
0	No evidence of any support

Table 7.2: Scoring Table

Obviously, the design method which scores highest in all the problems and for all the four different groups of subjects achieves most usability.

As indicated above, the experiments concerning the usability of the design method involve a large number of subjects and will require at least a year to carry out. The above discussion does not attempt to give a full-blown design of the experiments but to outline many of the issues that would have to be addressed.

Once the experiments are performed and the results are analysed, one can modify and improve the method accordingly. Currently, one may find that the design method is suitable for small, general purpose application domains. The appropriateness of such a design method in large applications and specific problem domains needs further investigations. In fact, the history of structured design development has reflected this trend of development. When structured design methods were first introduced in early 1970's, they did little in the area of such things as interrupts design and scheduling of concurrent processes. An extended, 'real-time' flavour of structured design methods only emerged in the early 1980's. Although it is said that object-oriented programming is good for implementing concurrent systems, extending this design method to cater for concurrent programming would be part of the future work.

7.2. Future Work in Long Term

Computer aided software engineering (CASE) has become increasingly important in recent years. It is now recognised that with the large range of automated facilities provided in CASE environments, the productivity of software engineers has improved dramatically [Som89]. In order to know what automated facilities to be provided in the design phase, one needs to know what tasks system designers have to perform during the design phase. As a design method defines the procedures one has to carry out in the design phase [McC89], it indirectly decides what tools one requires in the design phase of the CASE environment. Thus, the design method has a vital role to play in CASE environments. As it stands, the design method resulting from this research is tailored for object-oriented programming. It, therefore, forms the basis of a CASE environment for object-oriented programming. This section discusses possible future work necessary to achieve this objective. It starts off by presenting an overview of a CASE environment for object-oriented programming. It then briefly discusses the individual components that are found in such an environment. The components which are directly related to the design method are then investigated in more detail.

7.2.1. Computer Aided Software Engineering Environment

Software development comprises a number of activities which cover requirement analysis, design, implementation and testing [How82, McD85]. In a CASE environment, there are a large set of automated facilities which assist software engineers with the task of software development. Contemporary software development environments can be classified into four categories [DEF87]:

i. Language-centered Environments

These environments are built around one particular language and the tools supported

are suitable for that particular language only, e.g., Smalltalk-80 [Gol83b], Cedar [SZH85].

ii. Structure-oriented Environment

These environments employ techniques that allow users to manipulate structures directly, e.g., Cornell Program Synthesizer [Rep84], Pecan [Rei84].

iii. Toolkit Environment

These environments provide a set of tools for the coding phase of the software development cycle. e.g., PCTE [GMT87], Unix [DHM84].

iv. Method-based Environments

These environments are built around a particular development method. The tools provided support for a broad range of activities which are defined by the design method. e.g., SADT [Ros85], PSL/PSA [TeI77].

As each of these kinds of environment actually covers part of the development phase, it seems logical to merge these capabilities to achieve a highly interactive, tailorable, multiple-user, full life-cycle environment [DEF87]. Such an environment should support all stages of the software development process from initial feasibility studies to operations and maintenance.

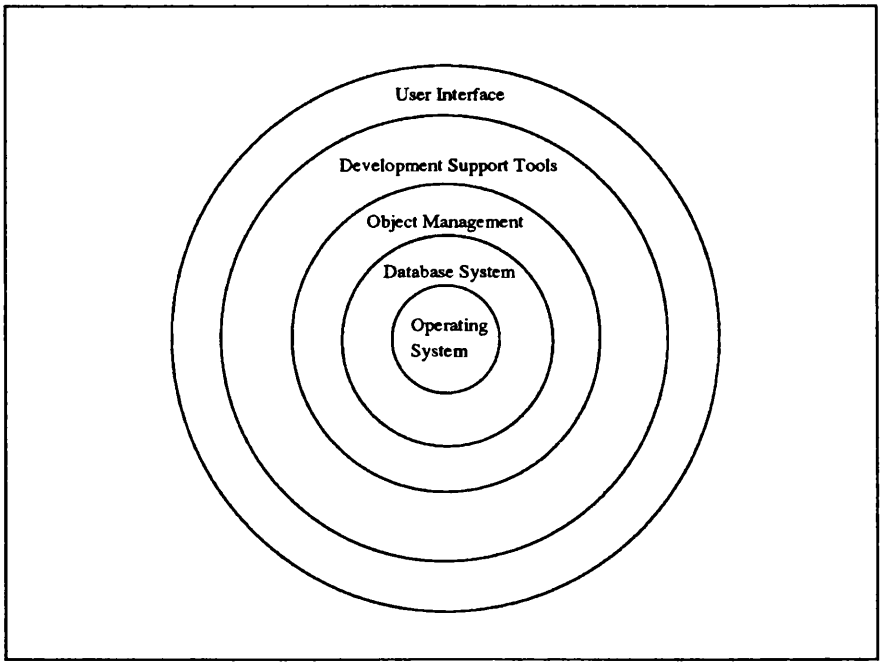


Figure 7.1: The Structure of the Traditional CASE Environment

Current CASE environments are generally built around a database management system [Som89] as illustrated in Figure 7.1. All the tools supported in such environments output

information to and collect information from the database. In addition, the database system supports information retrieval tools which can collate and present component information which is generated at different stages of the life-cycle.

As it stands, the structure of a CASE environment for object-oriented programming (see Figure 7.2) is similar to that of the traditional one. This is not surprising as object-oriented programming is a natural evolution of traditional programming. However, it is believed that certain components of this environment will have to be changed and modified to give a more consistent and sophisticated CASE environment for software engineers.

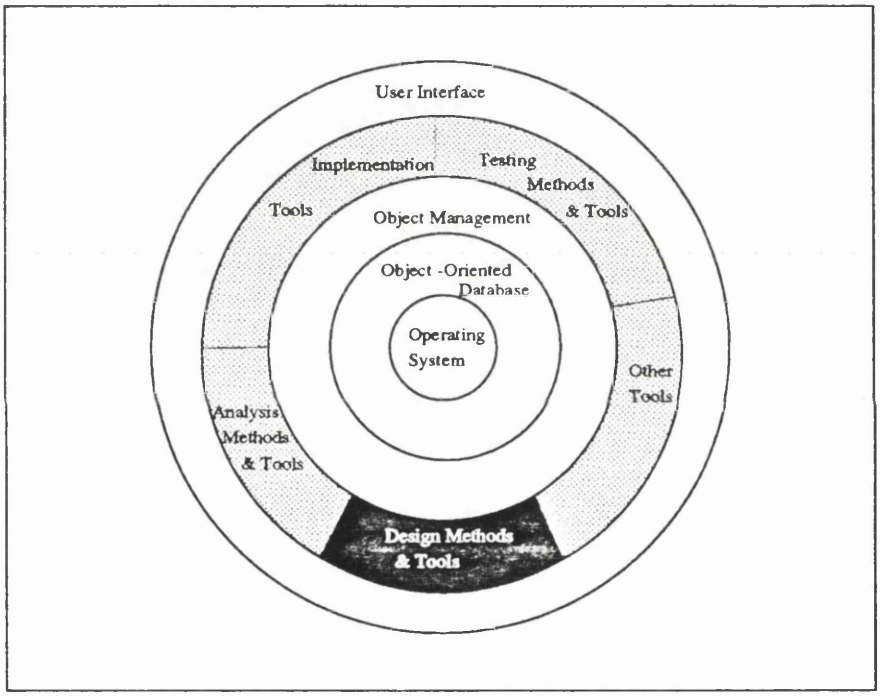


Figure 7.2: The Structure of the CASE Environment for Object-Oriented Programming

7.2.2. The Object-Oriented Database

In the traditional software engineering environment as shown in Figure 7.1, the inner layer is the database layer. It provides all the data storage facilities for project information and allows relationships between project entities to be defined and maintained. Although traditional databases such as hierarchical, network and relational have been widely used in this layer, they were designed to support alphanumeric data, formated into records. While this may be appropriate for typical information management system applications; they are not really appropriate for applications such as mechanical, computer-aided design and computer-aided software engineering, which relies on complex and non-record oriented data [Atw89].

Therefore, an object-oriented database is more likely to be used in the next generation of environments [AnH87, Som89]. This is especially true if one is aiming towards a CASE environment for object-oriented programming.

Object-oriented databases are considered to be more powerful than traditional databases. Traditional databases such as the relational database are inadequate in representing the complexity of the real world [DKL86], as there are difficulties in matching the real world complexity to flat data structures. The object-oriented database, however, emphasises persistent storage of objects and provides a higher level of abstraction. This gives the capability to express the complexity involved in modelling the real world. Besides, in conventional software development, programming languages and databases are not fully integrated. 'Embedded languages' are always required to allow the application, which is written in general purpose programming languages, to access the databases systems. These kinds of bridges are usually awkward, and provide limited functionality. The object-oriented database, however provides a tighter coupling between data management and programming language facilities which results in a safer and higher performance system [AnH87, Kil89].

Currently, object-oriented database are not yet commercially used. However, there is a lot of research going on in this area [Atw85, Loc87]. Some of these works are concerned with how to modify relational databases to support object-oriented features [BPR88, Rum87]. It is strongly believed that the object-oriented database is going to be a crucial component of CASE environments for object-oriented programming and hence will play an important part in future work concerning this research.

7.2.3. The Object Management System

The object management system [Zdo86] is responsible for providing control over name spaces and the configuration control of objects within the development environment. With the object management system, the user may simply refer to environment objects using local names with the system handling the mapping to the database layer. Configuration control or version control is very important within the environment. Designers always need to generate and experiment with multiple versions of a design, before selecting one that satisfies the design requirements [ChK86]. The object management system should also provide concurrency control for cooperative work among a team of software engineers.

7.2.4. The Development Toolkit for the Design Phase

The object-oriented database and the management system described above are used to store and manage the objects created by the software tools in the development process. As has

been reiterated several times, the development process generally covers the requirement analysis phase, the design phase, the implementation phase and the testing phase. Each of these different phases require different methods and tools to support it. For example, the implementation phase may need debuggers, compilers, browsers, and syntax-directed editors to support the programming task. As it stands, any future work involved in developing the complete toolkit is quite substantial. Firstly, it would entail the development of proper methods for object-oriented requirement analysis, design, implementation and testing. These methods, as mentioned earlier, will then define the sorts of toolkit needed for the different phases of the development process. Since this research concerns mainly the design phase in the development process, there is not sufficient information to start discussing the toolkits for other phases and the integration of these toolkits into the CASE environment. Nevertheless, this section examines the software tools for the design phase which are derived from the design method discussed in this research.

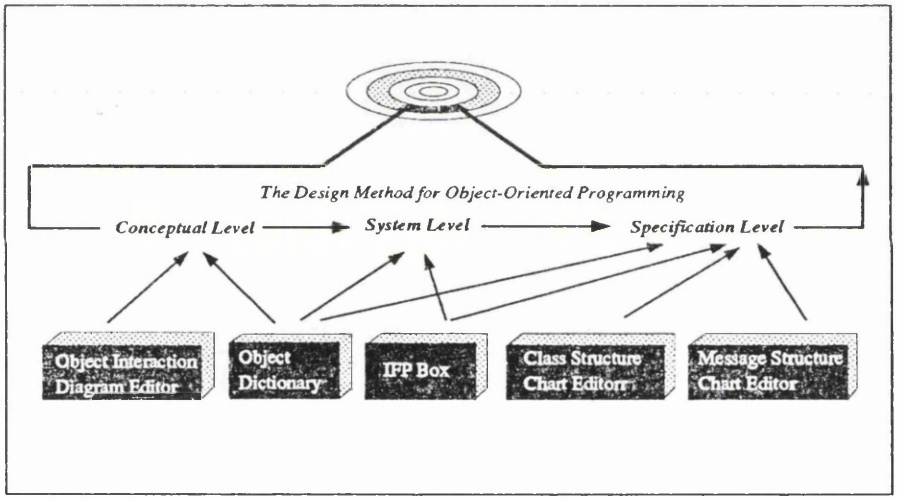


Figure 7.3: The Primary Software Tools which are derived from the Design Method

The result of this research has produced a design method which is specially targeted for object-oriented programming. The design method encourages the usage of graphical notations to document the design process. It introduces the inheritance factorisation process to assist the construction of class hierarchies. Hence, the tools which have to be developed to support the design process defined by this design method can be divided into two categories:

i. Graphical Tools

The design method suggests that the conceptual model of the design phase should be presented as object-interaction diagrams. Besides, the design specifications have to be presented as class structure charts and message structure charts. This suggests that an object-interaction diagram editor, a class structure chart editor and a message

structure chart editor have to be developed. These editors should have some syntax checking facilities. The documents and specifications created with these editors can be regarded as objects which will be managed by the object management system and stored in the object-oriented database in the environment.

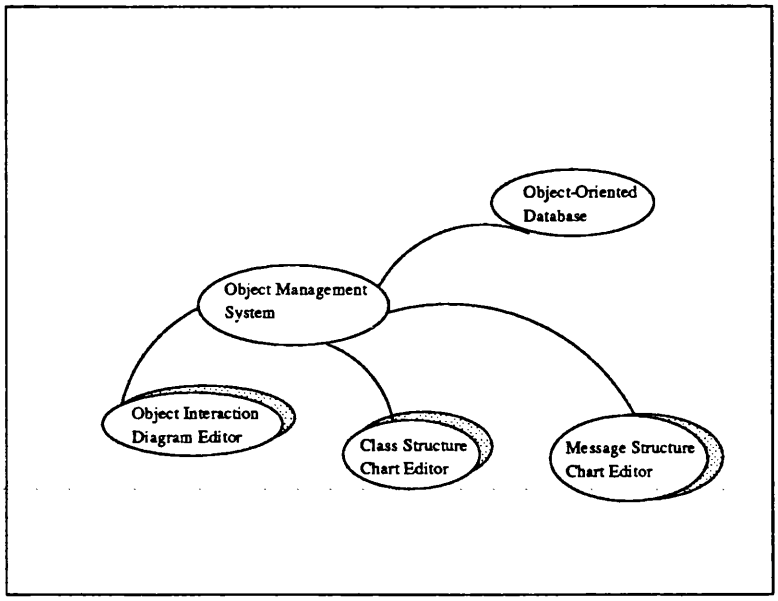


Figure 7.4: The Graphical Tools in the Design Phase

As suggested in Chapter 3, the object-interaction diagrams are not only the documents found in the design phase but also the analysis phase. Therefore, it is believed that the object-interaction diagram editor should be used in the analysis phase of the development process.

ii. Tools for the inheritance factorisation process

The inheritance factorisation process plays a vital role in the design method which has emerged from this research. The tools required to support this process can be grouped into a black box called the IFP box. The box will contain an editor to specify the class specifications needed, the factorisation engine that generates the normalised hierarchy expression and the graph generator which produces the class hierarchy graph. As these tools are used to help system designers design the class hierarchies, they would not interact with the object-oriented database nor the object management system. However, the IFP box will interact with the class structure chart editor to automatically generate the required class structure chart. Besides, there might be a possibility that the IFP can be improved to generate the required class data structures automatically. In this case, the IFP box may interact with the tools found in the implementation phase (see Figure 7.5).

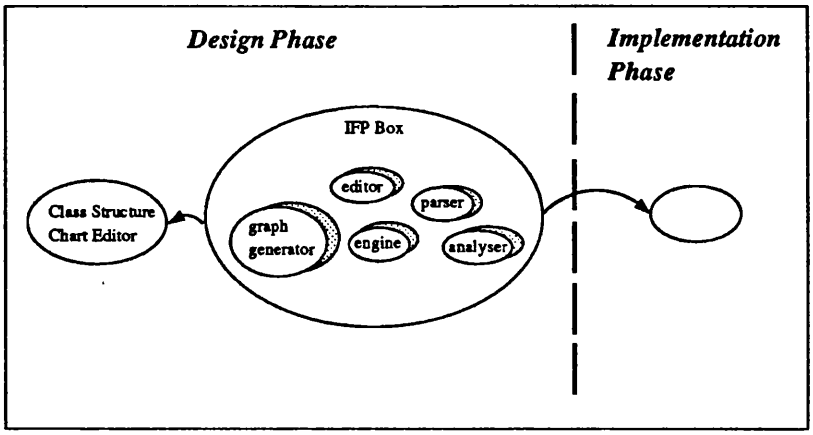


Figure 7.5: The IFP Box

In addition to these tools that are obviously required for the design method, there may be other auxiliary tools that are helpful for the system designers. For example, an object dictionary which allows system designers to examine and enter the definition of a particular object; filter-browser in the IFP box to give information about available classes in the system; design analyser tool which analyses the efficiency of a particular design by examining the object-interaction diagrams.

7.2.5. Other Tools

Besides tools that are required to support the different stages of the development phase, project management tools are also essential in the CASE environment. The objective of the project management is to provide a framework that enables the manager to make reasonable estimates of resources, cost and schedule. Most of these tools, such as those that help engineers to estimate budgets, to generate status reports and to plan project schedules already exist in traditional CASE environments. Part of the future work may involve investigating whether these tools can be reused in an environment for object-oriented programming.

7.3. Conclusion

This chapter has outlined the immediate future work with respect to this research. The immediate future work involves the extension and modification of the inheritance factorisation process and examining the usability of the design method. An overview of the experiments which are required to be carried out to test the usability of the design method has been discussed.

With the current research in object-oriented system analysis, object-oriented implementation and the fact that this research has developed a design method for object-oriented programming, the natural direction for any long term future work has to be conducted towards an object-oriented CASE environment. This chapter has also discussed some of the future work that would be necessary to achieve this goal. Such a discussion is by no means a full account of what one can do in the future. As most of the related work concerning this research is still in its infancy, it is believed that there is a long way to go before a sophisticated and acceptable CASE environment for object-oriented programming will emerge.

"When a scientist doesn't know the answer to a problem, he is ignorant. When he has a hunch as to what the result is, he is uncertain. And when he is pretty darn sure of what the result is going to be, he is still in some doubt. We have found it of paramount importance that in order to progress we must recognise our ignorance and leave room for doubt. Scientific knowledge is a body of statements of varying degrees of certainty - some most unsure, some nearly sure, but none absolutely certain."

~ Richard P. Feynman ~

Chapter
Chapter 8

Conclusion

When examining programming development history, it is found that modular programs were first introduced in the early 1970's via the technique of stepwise refinement. This kind of technique was then developed into structured programming which emphasised functional decomposition, i.e., the modularity of these programs was function-oriented. In the early 1980's, object-oriented programming emerged as the next phase in the evolution of programming. If one takes the term 'structured programming' in its more general sense, one can even view object-oriented programming as a part of the evolution of structured programming. Instead of being function-oriented, object-oriented programming provides new ways to structure programs, such as classes and class hierarchies. These provide a higher level of abstraction which increases the expressive power of software engineers in modelling the real world. Besides providing a higher level of abstraction for modularising a system, object-oriented programming also has other desirable features such as message passing, class inheritance and dynamic binding which secure it an important place in the programming evolution.

Just as its ancestor, structured programming with the function-oriented approach, object-oriented design methods are in great demand as object-oriented programming becomes more and more popular. The need for a design method is even more apparent when developing large systems. A design method generally lays down the ground rules for software engineers to organise design activities towards a particular implementation. It defines the tasks involved in the design phase and specifies the design description language for the engineers to

communicate with each other. As it stands, the kinds of design methods which are currently well-defined, popularly used and widely supported with CASE tools are those which target traditional structured programming such as data flow and data structure design methods.

This research has revealed that these design methods are inadequate for supporting programming activities which are targeted towards object-oriented implementation. The main reason for this is that they tend to apply 'functional decomposition' in modularisation. Modules are generated around operations and data structures are distributed between resulting routines. However, for object-oriented programming, the reverse occurs and the emphasis is on data structures. Modularisation and operations are generated around important data structures. Further, these structured design methods do not support the distinguishing features of object-oriented programming such as class inheritance.

As traditional design methods are inadequate to support design for object-oriented programming, work has been carried out to develop a design method which is suitable for object-oriented programming. Although a few examples of such design methods have emerged, they are shown in this research to be unsatisfactory. The main criticism lies in the fact that most of them are targeted at the programming language Ada. In addition, all of them are found to be inadequate in supporting the class inheritance feature of object-oriented programming. Therefore, the primary aim of this research has been to develop a better design method for object-oriented programming. Such a design method should have the following characteristics:

i. The design method should support designs for object-oriented implementation.

ii. The design method should have sufficient guidelines to assist system designers to handle class inheritance.

iii. The design method should reuse appropriate ideas from traditional design methods, if possible, to bring down the learning curve.

As revealed in this thesis, the primary framework of such a design method has been attained by this research. The design method which has emerged from this research aims at assisting system designers to organise the design activities towards an implementation model for object-oriented programming. It is divided into three levels:

i. The conceptual level assists system designers analyse and examine the application in an object-oriented fashion,

ii. The system level concerns the construction of the implementation model. The confirmation of the implementation objects, the identification of the 'contain', 'use' and the 'inherit' relationships contribute in defining the structure of the classes involved in the system.

iii. The specification level is responsible for the production of design specifications which are going to be passed on to the implementation phase.

The design method has also specified the design description languages such as the object-interaction diagram, the message structure chart and the class structure chart which system designers use for communication.

Beside defining the design process and specifying the design description language for object-oriented programming, one of the major achievements of this design method is that it has identified a way to guide system designers in constructing class hierarchies. The construction of class hierarchies plays a very important role in designing an object-oriented system. In the past, system designers have relied very much on their intuition and experience in constructing class hierarchies. This design method, however, has introduced a more algorithmic approach to tackling the problem. A process called 'inheritance factorisation process' has been developed to assist system designers in constructing class hierarchies.

The inheritance factorisation process has an algebraic model which lies behind it. Such an algebraic model ensures that the output of the inheritance factorisation process is correct, consistent and well-defined. With the inheritance factorisation process, system designers are only required to provide the related class specifications and the corresponding optimised class hierarchy is then generated. Although the current algebraic model may be too general, the possibility to extend and improve the basic model has been discussed in this thesis. Further, the development of the inheritance factorisation process has suggested the construction of class hierarchies can be automated to a certain extent. Such an idea has been demonstrated by implementing an inheritance factorisation engine as part of the work of this research. The performance of using this inheritance factorisation engine in constructing class hierarchies has also been investigated.

The result of a piece of research of this sort is seldom conclusive. Very often, it triggers the beginning of another piece of research. Therefore, it comes as no surprise that the design method obtained from this research is found only to be the backbone of future research. This future research concerns the development of a computer-aided engineering environment for object-oriented programming. Some of this future work has been discussed briefly in the thesis.

Generally speaking, the contributions of the research reported in this thesis are two-fold:

i. It has introduced and explored a more algorithmic approach to constructing class hierarchies in object-oriented programming. This approach forms the basis of a CASE tools which assists system designers to handle inheritance in object-oriented programming.

ii. This new approach to constructing class hierarchies is part of the design method developed in this research. The design method is specially developed for systems design aimed at object-oriented implementation. Although the design method emerging from this research may not yet be perfect, it is a more adequate design method for object-oriented programming when compared with other existing design methods. Further, it provides the primary framework in which extensions and modifications can be carried out. It also lays down the ground work for the development of the CASE environment for object-oriented programming.

During this period of research, a lot of people have shown great enthusiasm for this work. A few of them have enquired about the possibility of using this design method in their developments. A student at the Dutch Open University has actually used this design method in developing a hypertext system for electronic courseware in Smalltalk-80 and found the design method is useful. Therefore, there are reasons to believe that given more time, this research could be extended and modified to be beneficial for software developers. After all, design methods as popular and well-supported as Yourdon's and Jackson's methods have taken more than ten years to become established and adopted in the industrial world.

Appendix A

The GP Surgery Notes System

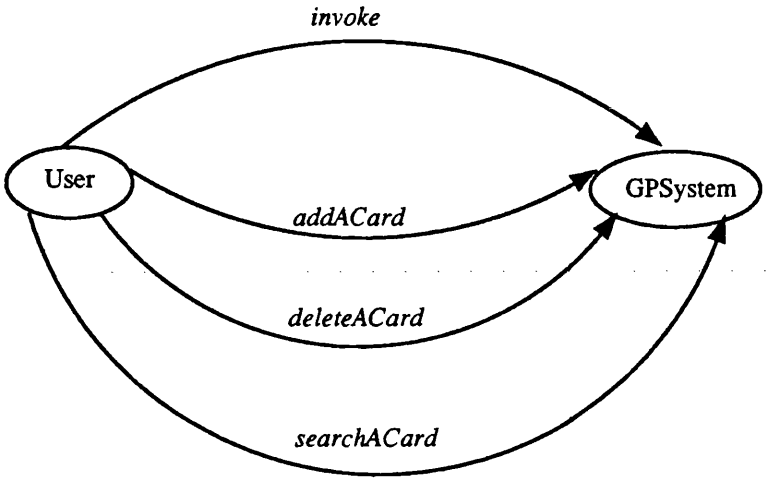
This appendix contains one of the two case studies which have been carried out in this research. This particular case study is about the development of a GP Surgery Notes System. The requirement specification for the system is shown below. Examples of the design documents which are generated at different stages of the design phase are shown in the rest of this appendix.

The Requirement Specification

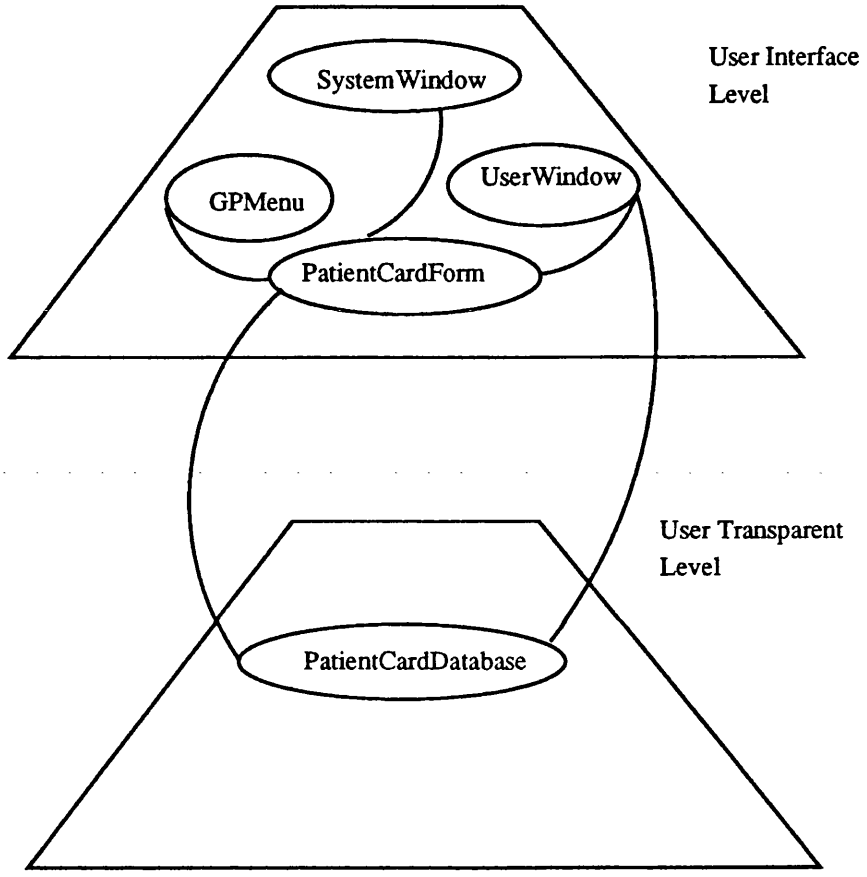
The GP Surgery Notes System is a computerised version of the current card system. The cards have fields for surname, other names, address, date of birth, house telephone, work telephone, allergies, prescriptions, history, illness history and general notes.

As well as searching for a given person, one often needs to search the file by date of birth, address.

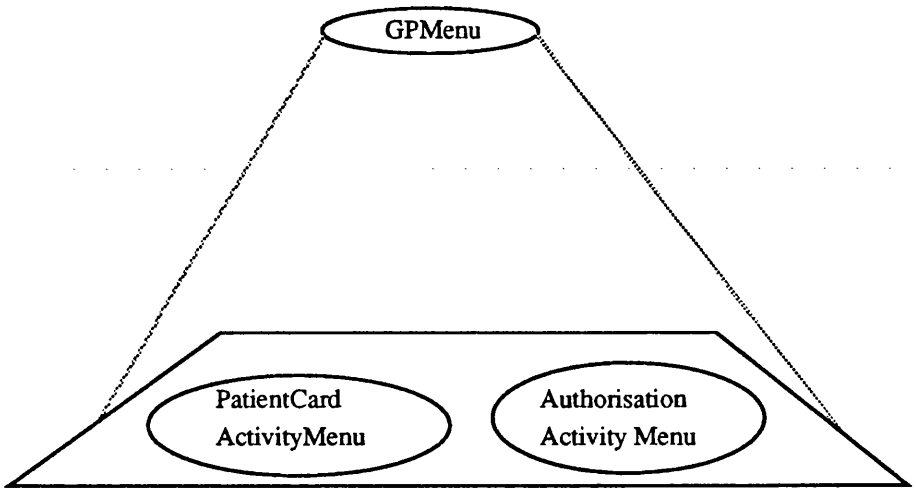
The specification given above though not in detail, provides enough information to actually design and implement a system.



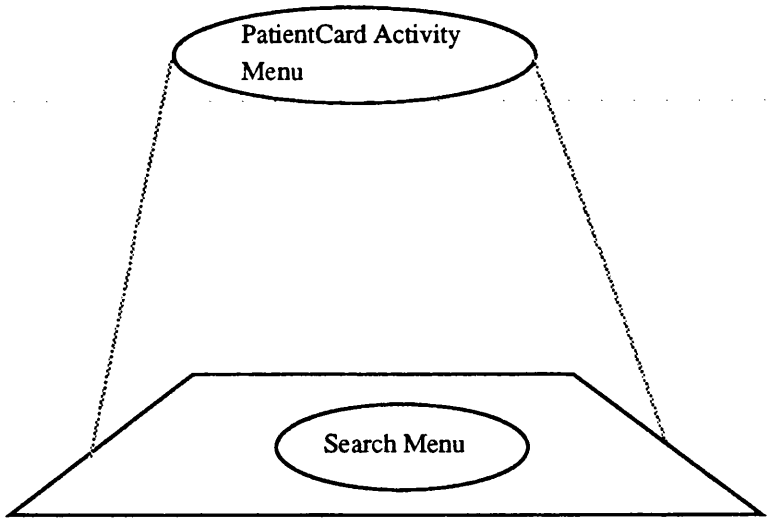
Conceptual Level: The Overview of the GP System



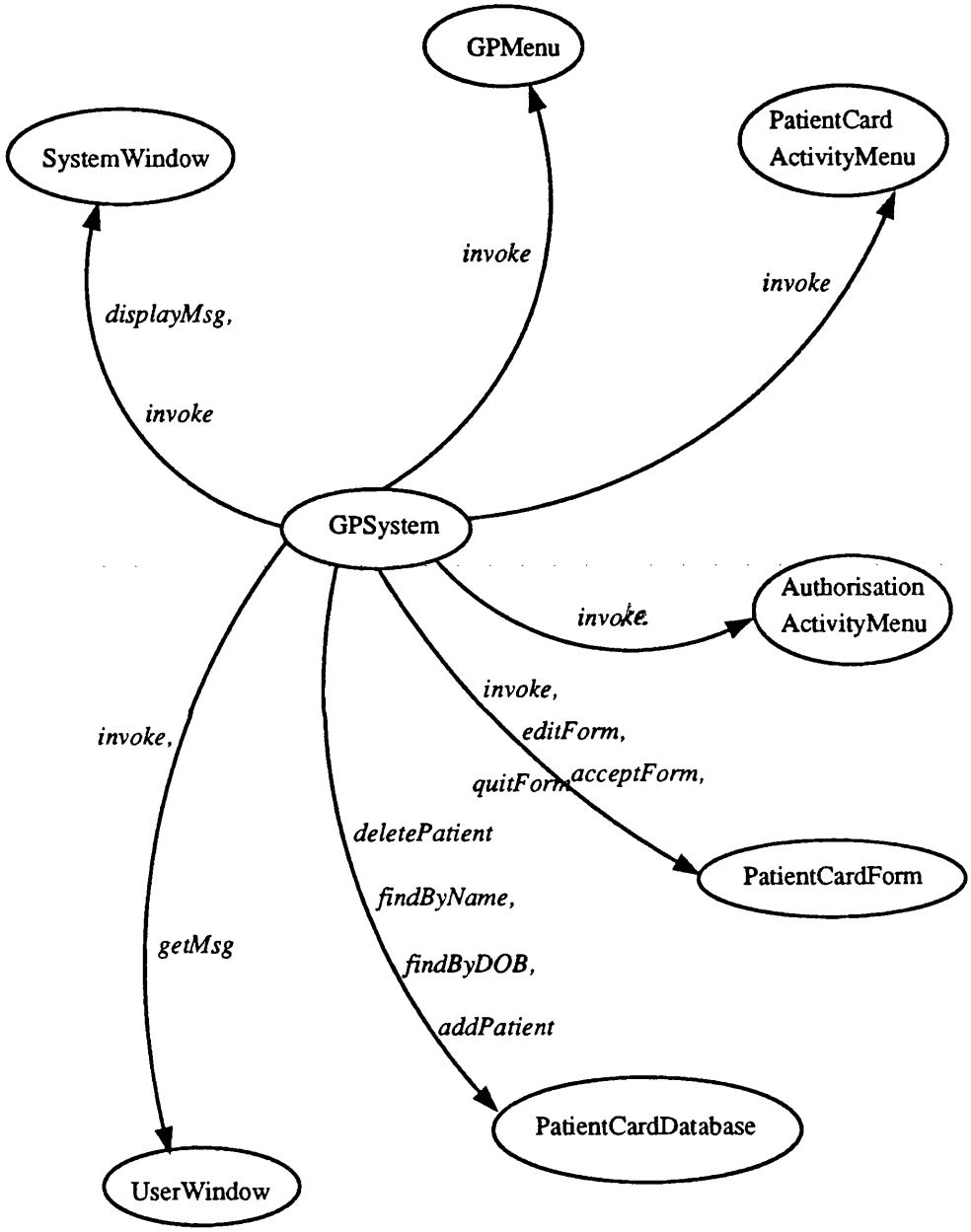
Conceptual Level: The Identified Application Objects of the GP System



System Level: The 'Contain' relationship of the GPMenu



System Level: The 'Contain' relationship of the PatientCardActivityMenu



System Level: The 'Use' relationships of the GPSystem

Class Name: GPSystem

Description: GPSystem is a computerised version of the traditional note card system. The card contains details of a patient. The system allows users to add a patient card, search a patient card by name and date of birth. The system also allows users to delete a card and update a card.

Attributes:

SystemWindow - the window space to display any message from the system.

UserWindow - the window space to read in any input from users.

PatientCardDatabase - this is the database which stores the patient cards.

PatientCardForm - this is the form which users have to fill in when they want to add a new card or display the information of a patient card.

GPMenu - this is the menu which users select their action options.

invoke - this is the operation to invoke the GP System.

addACard - this is the operation to add a new patient card.

deleteACard - this is the delect an exist card.

searchCardbyName - search a patient card by the surname,

searchCardByDOB - search a patient card by date of birth.

Class Hierarchy:

GPSystem

Inherited Attributes: NONE

Specification Level: The Class Structure Chart (CSC) of the GPSystem

Class Name: GPMenu

Description: GPMenu is the system menu of the GPSystem. It contains two submenus which allow users to access the PatientCardDatabase and to access the Authorisation of the PatientCardDatabase.

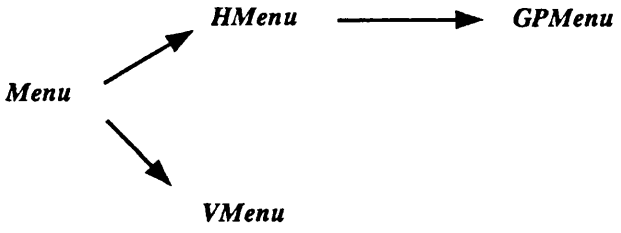
Attributes:

PatientCardActivityMenu -this is the submenu which contains menu options that allows end-users to consult the PatientCardDatabase.

AuthorisationActivityMenu -this is the submenu which contains menu options that allows authorised users to access the authorisation of the PatientCardDatabase.

HandleUserChoice - to execute the appropraite commands according to the users selection.

Class Hierarchy:



Inherited Attributes:

invoke - this is the operation to invoke the menu.
display - this is the operation to display the menu.
getSelection - this is the operation to obtain the menu option selected by users.
insertMenuItem - this is the opertaion to add a menu item to the menu.
compose - this is the operation to collect the menu items and compose the menu.

Specification Level: The Class Structure Chart of the GPMenu

Class Name: PatientCardForm

Description: This is the form which contains the fields specified for a patient card. End-users have to fill in these fields in order to create a new form. Information of this form will be displayed if users want to search a particular patient record.

Attributes: PatientCardDatabase - this variable refers to the particular database of this GP System.
acceptForm - this operation is carried out when users finish filling in a form and press the accept key. The operation includes putting the information from the form into the database.
clearForm - this operation when called will clear the current form.
quit Form- this operation when called will end the displaying of the form.

Class Hierarchy:

Form ⟶ *PatientCardForm*

Inherited Attributes: FormName - the name of the form created.
Fields - the fields in that form is arranged as a linked list.
FormMenu - there is a menu attached to the form.
NoOfFields - the number of fields in the form
CurrentField - this variable points to the current field in the form.
fieldContent - this variable stores the contents of a field.
createForm - this operation create the form.
getFieldContent - obtain the value of a particular field.

Specification Level: The Class Structure Chart of the PatientCardForm

Class Name: PatientCardDatabase

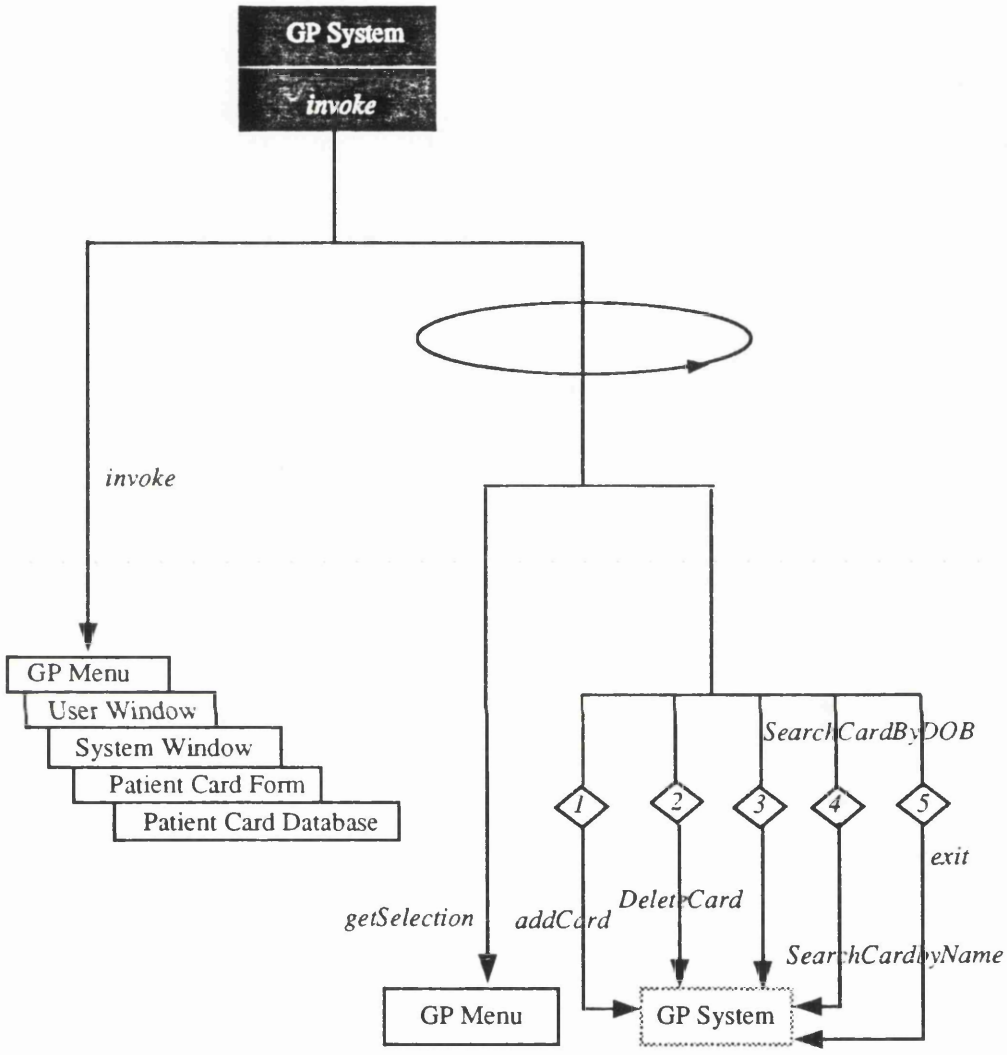
Description: This is the database which stores the records of the GP system.

Attributes:
createPCDatabase - this operation invokes the appropriate database.
findByName - this operation searchs the patiend card record by the patient's surname.
findByDob - this operation searchs the patient card records by the date of birth of
addPatient - this operation adds a patient card record to the database.
deletePatient - this operation delects a patient card from the database.
updatePatient - this operation updates a patient card in the database.

Class Hierarchy:

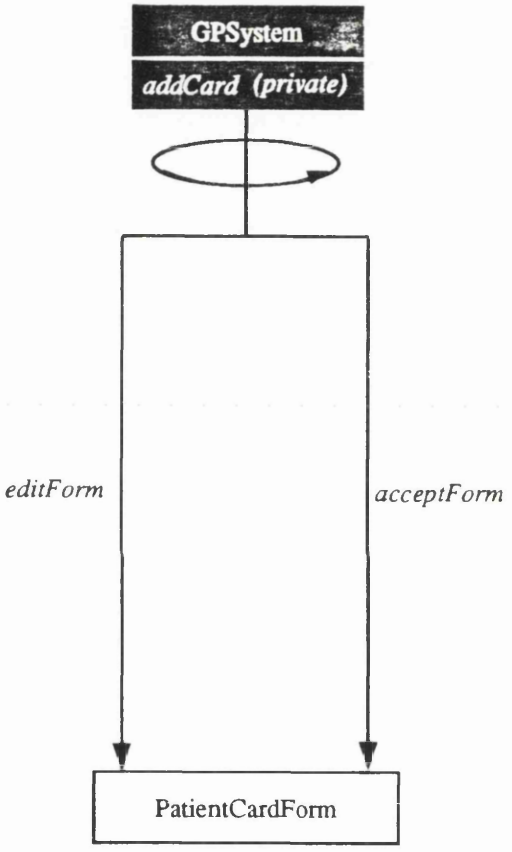
Inherited Attributes:
FileName - the name of the corresponding file.
CurList - the list of records.
RecSize - the size of each record.
openFile - this operation opens the appropriate file and read therecords from the file.
fileRead - this operation reads the next record.
fileWrite - this operation writes a record.
fileExist - this operation checks whether a given file exists or not.
fileClose - this operation close a file, this includes put back the list of recordsat run time into the corresponding files.

Specification Level: The Class Structure Chart of the PatientCardDatabase

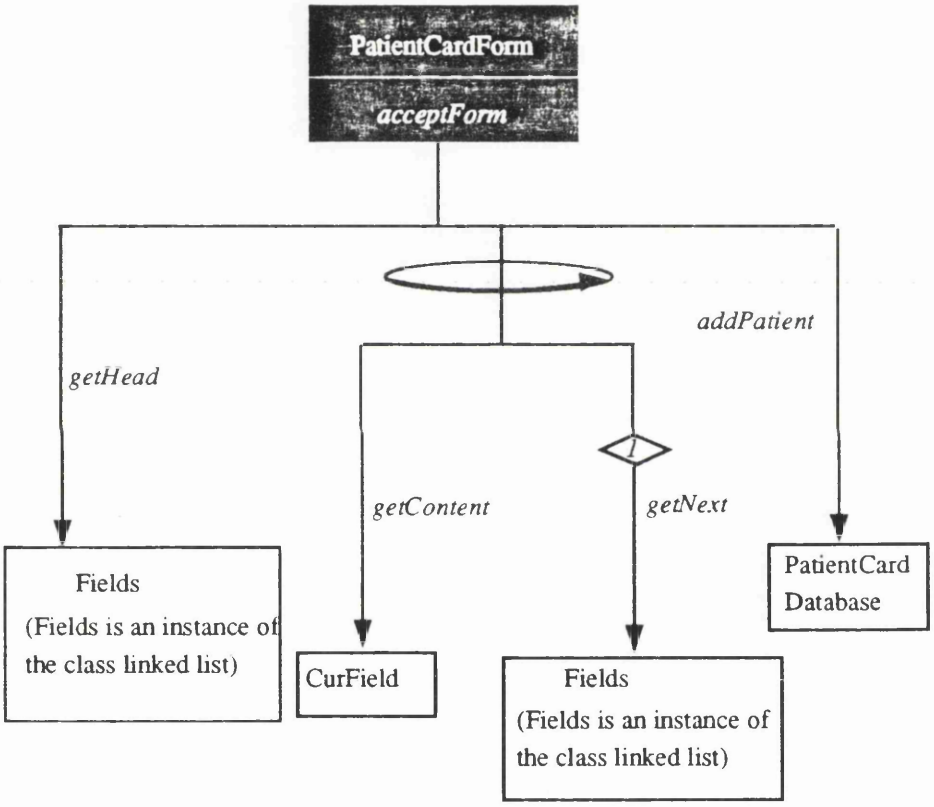


1 - if the user's selection is to add a patient card.
2 - if the user's selection is to delect a patient card.
3 - if the user's selection is to search a card by date of birth.
4 - if the user's selection is to search a card by name.
5 - if the user's selection is to quit the system

Specification Level: The Message Structure Chart (MSC) of the 'invoke' operation in the GPSystem

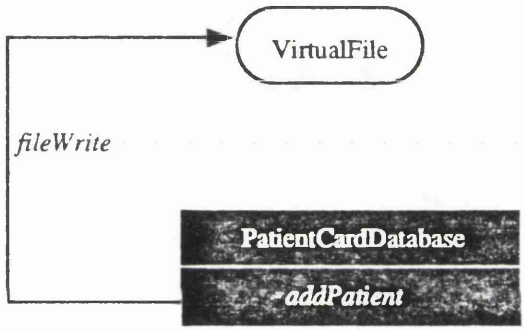


Specification Level: The MSC of the 'addCard' operation in the GPSystem



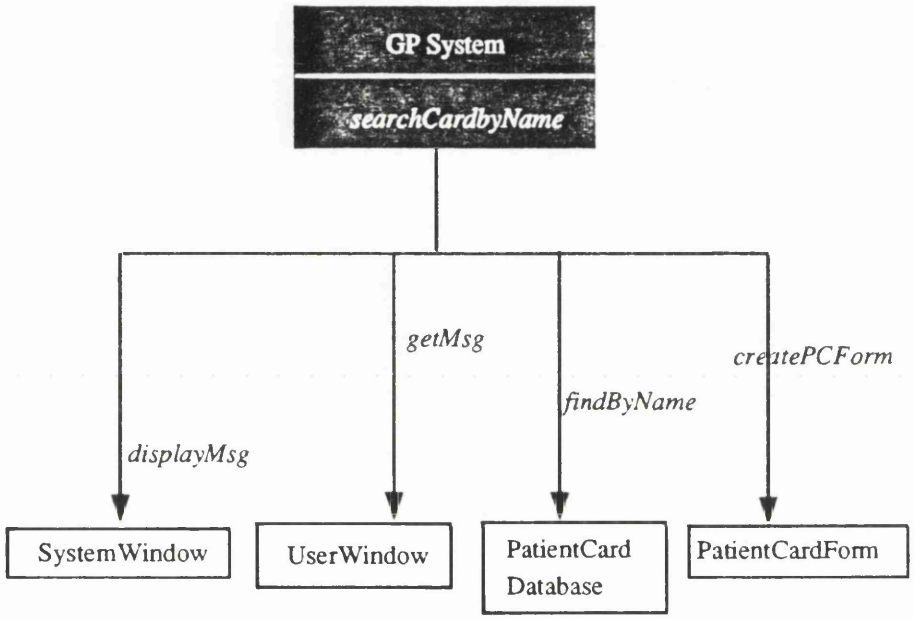
1 - if it is not the end of the field list

Specification Level: The MSC of the 'acceptForm' operation in the PatientCardForm



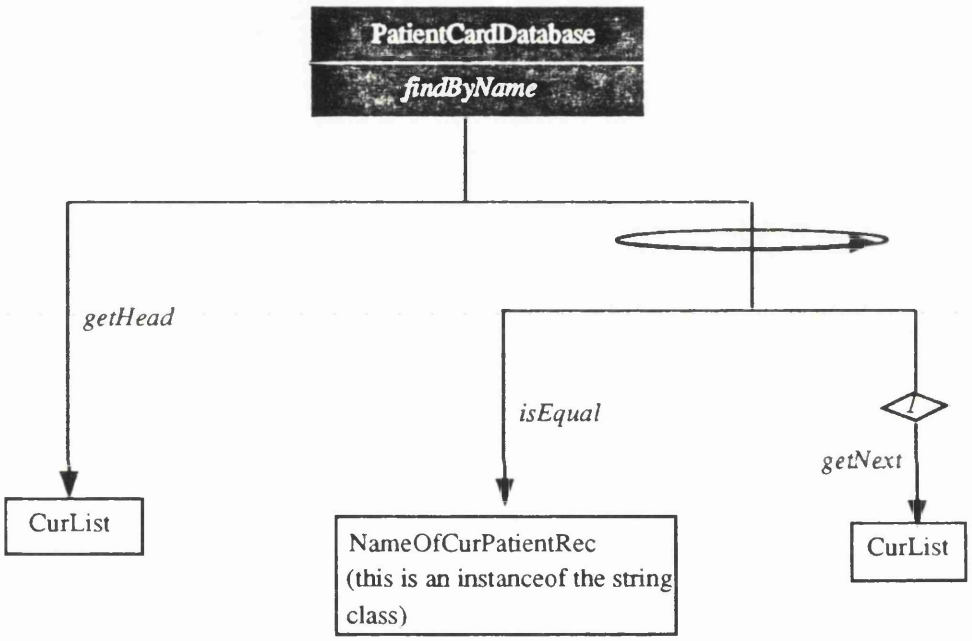
Note that the VirtualFile class is the superclass of the PatientCardDatabase class.

Specification Level: The MSC of the 'addPatient' operation in the PatientCardDatabase



Note: displayMsg is to display a message in the SystemWindow to request users to enter the patient's name to be searched.
getMsg is an operation to get the patient's name which is entered by the users in the UserWindow.

Specification Level: The MSC of the 'searchCardbyName' operation in the GPSystem



1 - if it is not the end of the list.

Specification Level: The MSC of the 'findByName' operation in the PatientCardDatabase

Appendix B

The Home Heating System

This appendix contains another case study of the design method. This case study concerns the development of a Home Heating System. The requirement specification of the system is below. The implementation of the system is not completed yet but examples of some of the design documents which are produced at different stages of the design phase are shown in this appendix.

The Requirement Specification

The Home Heating System described in this appendix was modified from the system proposed by S. White for the 1984 Embedded Computer System Requirements Workshop. The system has also been used as a case study in Booch's object oriented design method [Boo86].

The Home Heating System is responsible for regulating the in-flow of heat to individual rooms of the home in an attempt to maintain a working temperature established for each room. The working temperature for each room is calculated by the system as a function of a single desired temperature, which is set by the user, and whether or not the room is occupied. If the room is occupied, the working temperature is set to the desired temperature. If the room is vacant, the working temperature is set to 5 degrees F less than the desired temperature. Each room of the house is equipped with a sensor that continously measures temperature. Each room also has an infra-red heat sensor that continously determines whether or not the room is occupied.

The user interface provides input/output devices to permit the user to control and monitor the Furnace System. The following input devices are provided:

- heat switch

  The heat switch controls the status of the Furnace System. When the heat switch is turned to 'ON', and the fault reset switch is 'ON', and at least one room needs heat, the furnace system is activated. When the master switch is turned to 'OFF', the furnace system is deactivated.

- desired temperature input device
  The desired temperature input device continuously provides the value of the desired temperature set by the user.
- fault reset switch
  The fault reset switch is automatically turned 'OFF' by the heat flow regulator upon the detection of either a fuel flow or combustion state fault. The furnace cannot be activated when the fault reset switch is 'OFF'. The user can 'reset' a fault by setting the fault reset switch to 'ON'.

The following display devices are provided:

- fault indicator
  The fault indicator is turned on upon the detection of either a fuel flow or combustion state abnormality.
- furnace status indicator
  The furnace status indicator displays the on/off status of the furnace.

The system has a timer which provides a continuously incrementing count, one increment for every second of elapsed time.

Heat is provided to each room of the house by circulating hot water. The water is heated by the furnace system. Each room is equipped with a water valve that controls the flow of hot water into the room. The valve can be commanded to either full open or full closed.

The furnace system includes an oil combustion chamber, a combustion air blower, a water temperature sensor, and a boiler. The furnace heats water in the boiler which can then be circulated to one or more rooms of the house. The furnace is alternately activated and deactivated by the heat flow regulator as needed to maintain the required temperature for each room.

To activate the furnace:

1. The system activates the blower motor.
2. The system monitors the blower motor speed and when it reaches a predetermined RPM it opens the oil valve and ignites the oil.
3. When the water temperature reaches a predetermined value, the system opens the appropriate room water valves permitting the heated water to circulate through those rooms.
4. The furnace status indicator is turned on.

To deactivate the furnace:

1. The system closes the oil valve and then, after 5 seconds, stops the blower motor.
2. The system turns off the furnace indicator.
3. The sysem closes all the room water valves.

A fuel-flow status sensor and an optical combustion sensor signal the system if abnormalities occurs, in which case the system deactivates the furnace, turns the fault reset switch off, turns on the fault indicator, and closes all room water valves.

The heat flow regulator is a computer sysem that interacts with the other components of the home heating sysem to determine heating requirements for each room of the home and control the in-flow of heat necessary to satisfy those requirements.

The heat flow regulator maintains a working temperature which it establishes for each room. The working temperature for each room is calculated by the system as a function of a single desired temperature and whether or not the room is occupied. If the room is occupied, the working temperature is set equal to the desired temperature. If the room is occupied, the working temperature is set equal to the desired temperature. If the room is vacant, the working temperature is set equal to 5 degress F less the desired temperature. The heat flow regulator shall continue with this activity without regard to the state of the furnace.

The home flow regulator shall determine that a given room needs heat whenever:

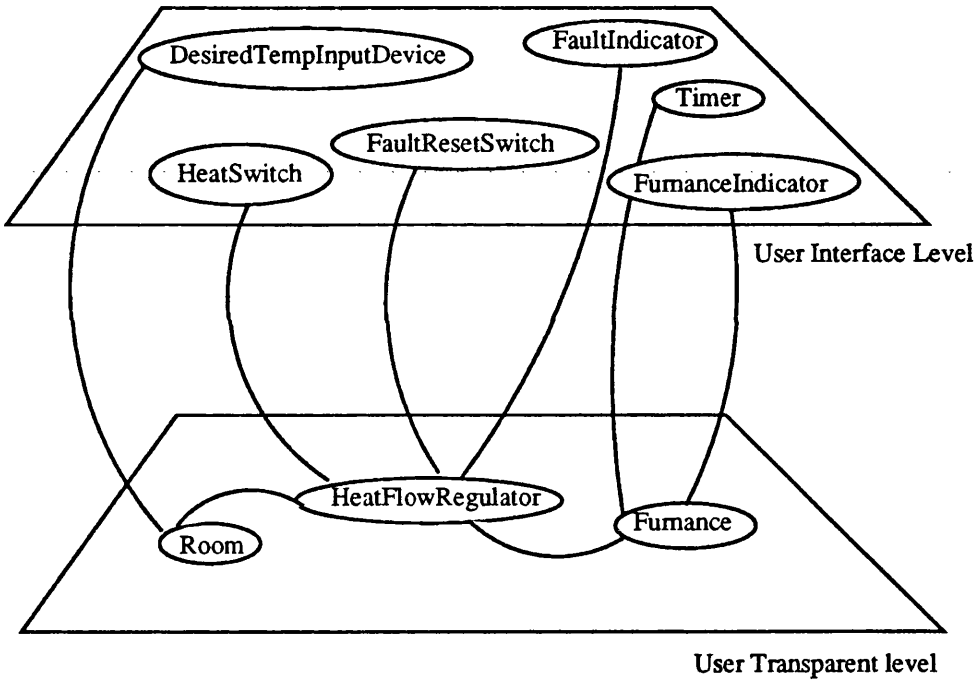
$$\text{room temperature} \leq \text{working temperature} - 2 \text{ degress F and}$$

shall determine that a given room does not need heat whenever:

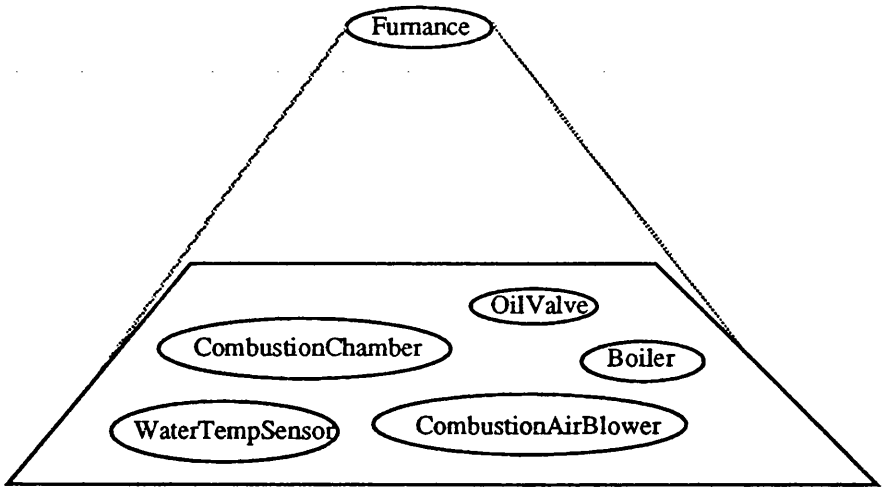
$$\text{room temperature} \geq \text{working temperature} + 2 \text{ degrees F}$$

The heat flow regulator will continue with this activity without regard to the state of the furnace. If the furnace is not active, the heat flow regulator shall activate the furnace and route heat whenever the heat switch and the fault reset switch are both 'ON' and at least one room needs heat. The activation procedure shall be as specified above in the furnace description. If the furnace is active, the heat flow regulator shall deactivate the furnace whenever either the heat switch or the fault reset switch is 'OFF' or no rooms need heat. The deactivation procedure shall be as specified above in the furnace description.

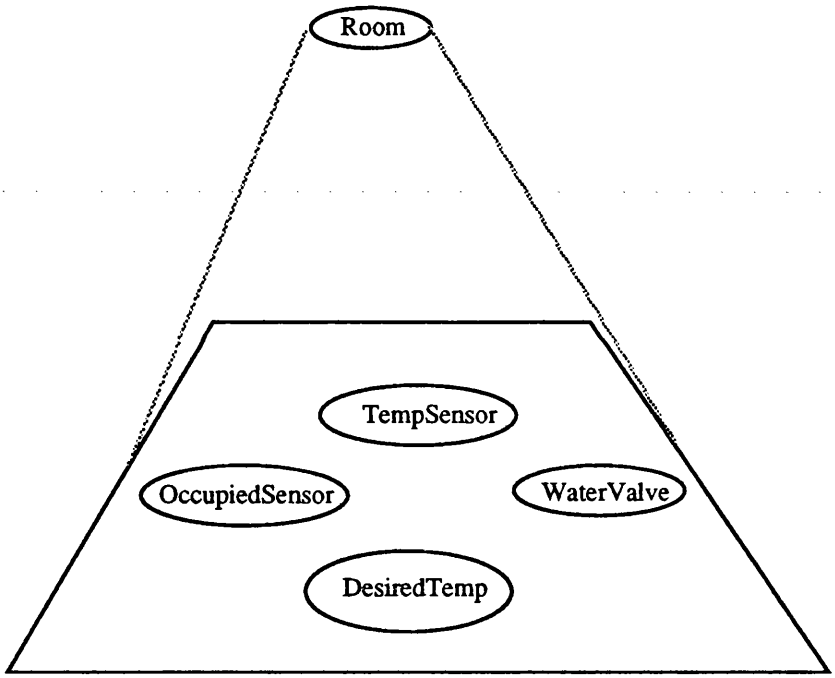
The heat switch can be turned 'ON' and 'OFF' by the user. The fault reset switch is turned 'OFF' by the heat flow regulator upon the detection of either a fuel flow or combustion state abnormality. The user can 'reset' a fault by setting the fault reset switch to 'ON'.



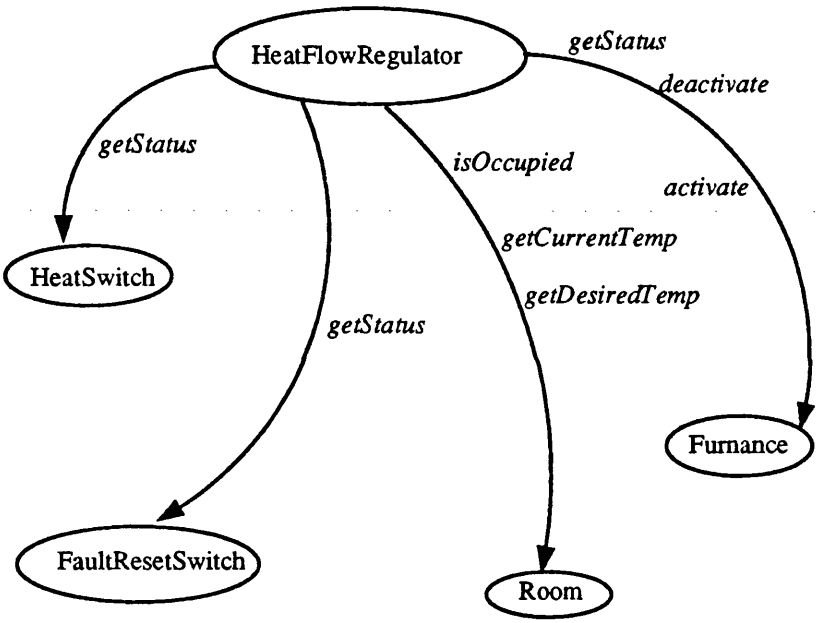
Conceptual Level: The Overview of the Home Heating System



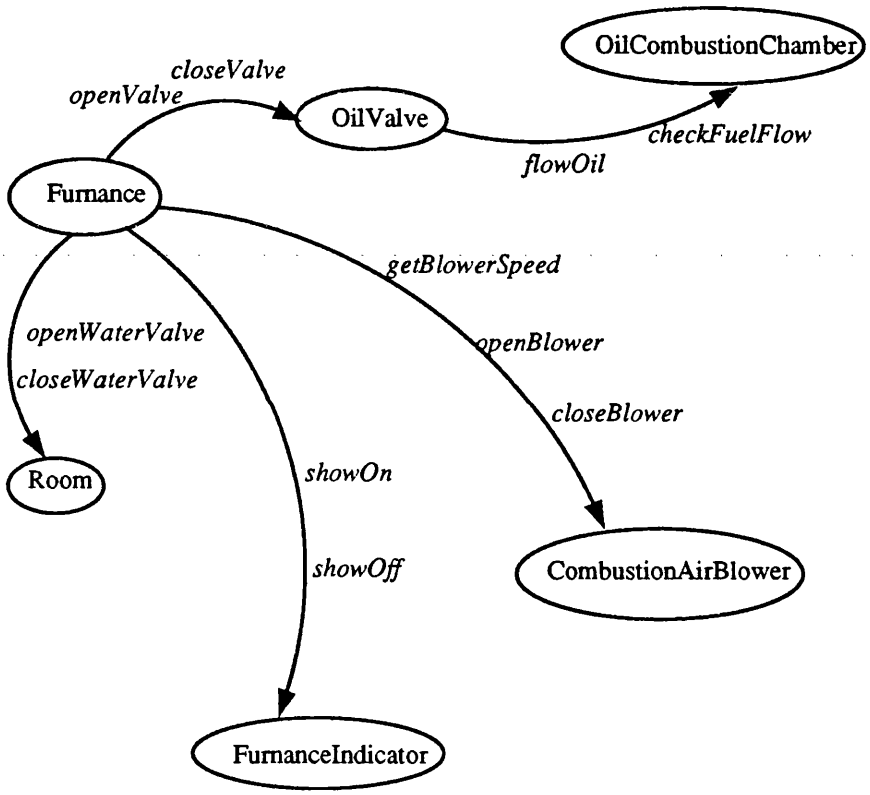
System Level: The 'Contain' relationship of the Furnace



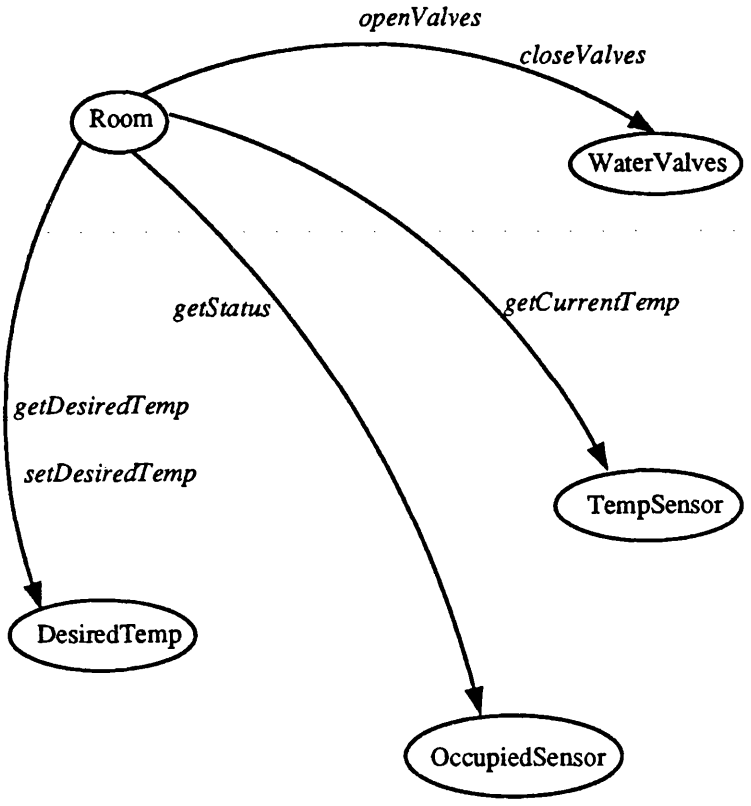
System Level: The 'Contain' relationship of the Room



System Level: The 'Use' relationship of the HeatFlowRegulator



System Level: The 'Use' relationship of the Furnace



System Level: The 'Use' relationship of the Room

Class Name: HomeHeatSystem

Description: The HomeHeatSystem regulates the in-flow of heat to individual rooms of the home in an attempt to maintain a working temperature established for each room. The working temperature for each room is calculated by the system as a function of a single desired temp, and whether or not the room is occupied. If the room is occuptied, the working temperature is set to the desired temperature. If the room is vacant, the room temperature is set to 5 degrees less than the working temperature.

Attributes: Rooms - this is the list of rooms within the Home.
Furnace - this is the furnace in the system.
HeatFlowRegulator - the heat flow regulator in the system.
HomeHeatInterface - this is the user interface of the system.
invoke - this opertaion invokes the system.
powerUp - this operation actually turns on the heat switch, i.e., it initialises the heat flow regulator.

Class Hierarchy:

HomeHeatSystem

Inherited Attributes: NONE

Specification Level: The Class Structure Chart of the HomeHeatSystem

Class Name: HeatFlowRegulator

Description: The heatflowRegulator is a computer system tht interacts with the other components of the home heat system to determine heating requirements for each room of the home control the in-flow of heat necessary to satisfy those requirements.

Attributes:
thisFurnace - the references to the furnace in the home heat system.
thisListOfRooms - the references to the rooms concerned in the home heat system.
chkHeatRequired - this may be a private operation which take a room as a parameter and check whether this room requires heating up or not.
run - the heat flow regulator will start running when this operation is invoked.

Class Hierarchy:

HeatFlowRegulator

Inherited Attributes: NONE

Specification Level: The Class Structure Chart of the HeatFlowRegulator

Class Name: Room

Description: A Room is part of a home that can be independently heated.

Attributes:

OccupiedSensor - a sensor which tells whether there is someone in the room.

DesiredTemp - this is a variable that stores the value of the desired temp. The temp is set by the users.

WaterValve- this is a valve which allows water to flow in when open.

TempSensor - this is a temperature sensor which gives the temp of the room.

isOccupied - this operation returns a boolean value to tell whether the room is occupied or not.

getDesiredTemp - obtain the desired temp that is set by users for this room.

setDesiredTemp - allow users to set the desired temp for a room.

closeWaterValve - close the water valve when the room is warm enough.

openWaterValve - open the water valve to allow hot water to flow into the room.

Class Hierarchy:

Room

Inherited Attributes: NONE

Specification Level: The Class Structure Chart of the Room

Class Name: Furnace

Description: A Furnace is part of the Home Heating System. When it is activated, it turn on the blower motor, monitor the motor speed and whne the speed reaches a predetermined speed, it opens the oil valve and ignites the oil. When the water tempeature reaches a predetermined value, the system opens the water valve of appropriate rooms to allow heated water to circulate through those rooms.

Attributes:
Boiler - the boiler in the furnace
Blower - the blower in the furnace
OilValve - the oil valve in the furnace
CurrentStatus - this is the variable which tells the current status of the furnace
WaterTempSensor - A temperature sensor which gives the temp of the water.

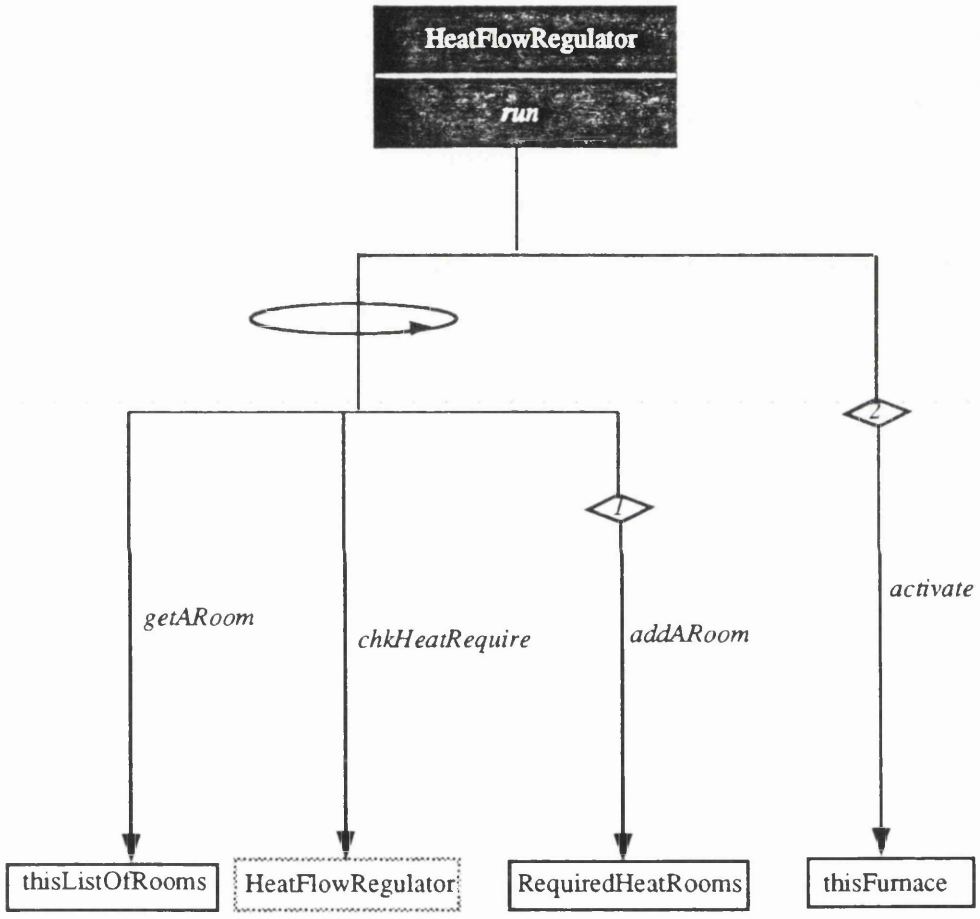
Operations:
invoke - this is the operation to invoke the furnace, it creates the corresponding boiler, blower and oilValve and initialise the status to 'off'
activate - this operation is to activate the furnace, it updates the status of the furnace accordingly. It turns on the boiler and the oilValve if necessary and keep track of the blower speed. It also keeps track of the water temperature and circulate heated water to the required rooms accordingly.
deactivate - this operation is to deactivate the furnace. It closes the oil valve, halts the blower and closes the water valves of all the rooms.
getStatus - obtain the current status of the furnace.

Class Hierarchy:

Furnace

Inherited Attributes: NONE

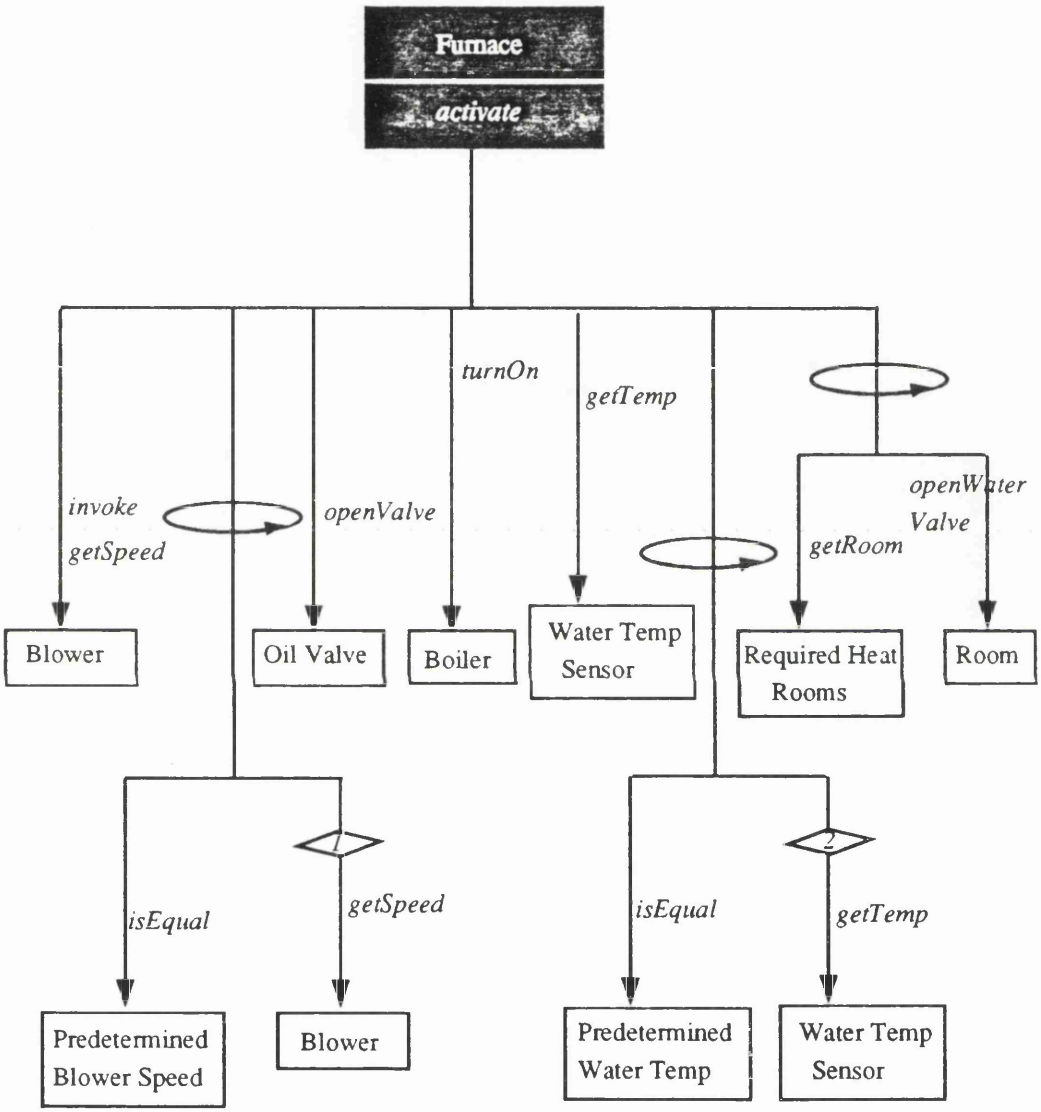
Specification Level: The Class Structure Chart of the Furnace



1 - if the room required to be heated up.

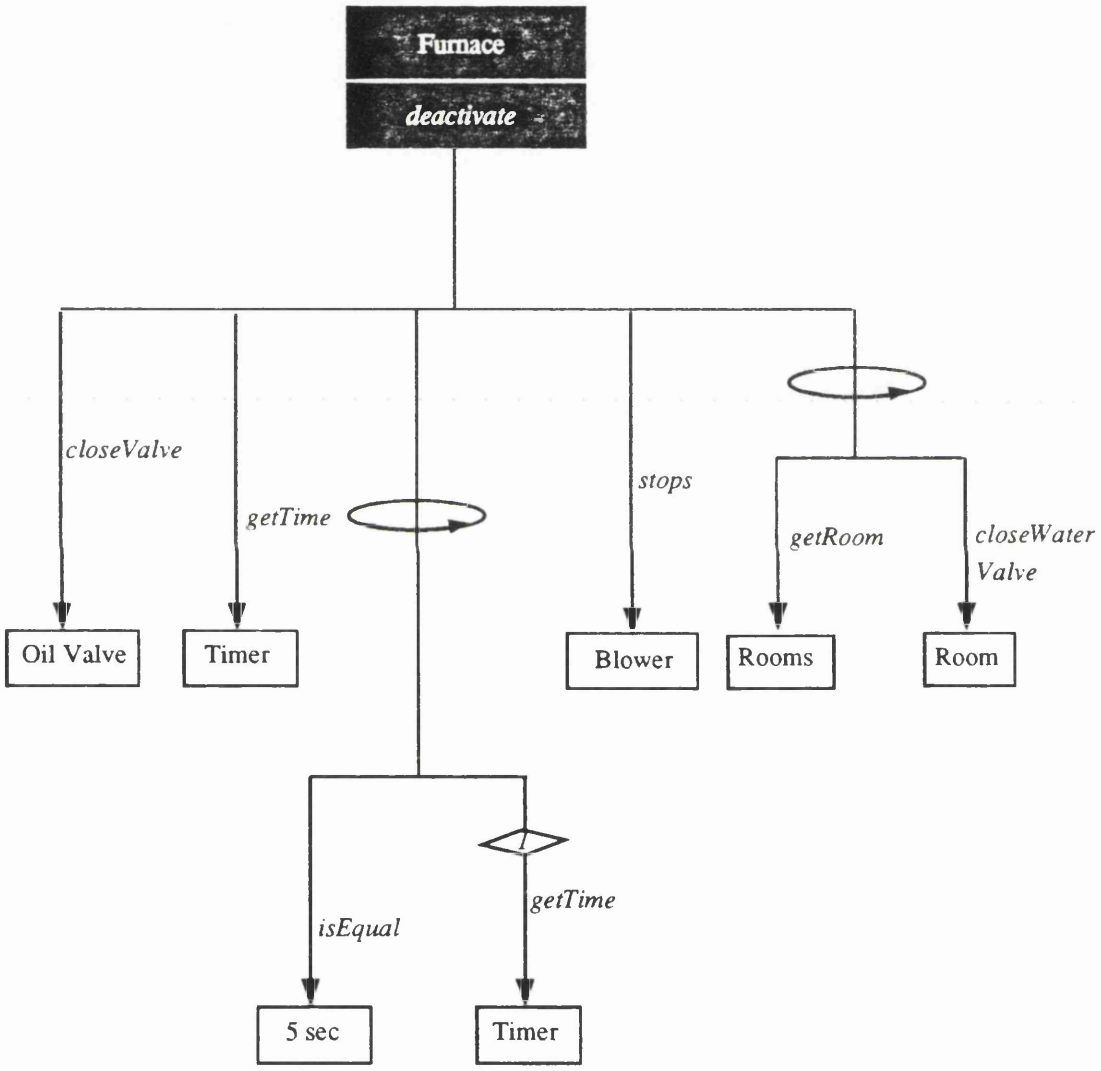
2 - after checking all the rooms in the system and if there is one room required to be heated up, the system has to activate the furnace.

Specification Level: The MSG of the 'run' operation in the HeatFlowRegulator Class



1 - if the speed obtained from the blower is not equal to the predetermined speed.
2 - if the temp from the water sensor is not equal to the predetermined temp.

Specification Level: The MSG of the 'activate' operation in the Furnace Class



1 - if the time obtained from the timer is not equal to 5 sec.

Specification Level: The MSG of the 'deactivate' operation in the Furnace Class

References

[Abb83] Abbott, R. J., "Program Design by Informal English Descriptions," *Communications of the ACM* 26 (11) (November 1983), pp. 882-895.

[Agh83] Agha, G., "An Overview of Actor Language," *SIGPLAN Notices* 18 (6) (June 1983), pp. 58-67.

[Ala88] Alabiso, B., "Transformation of Data Flow Analysis Models to Object-Oriented Design," in *Proceedings of the OOPSLA'88 Conference*, ACM Press, October 1988, pp. 335-353.

[Ala89] Alagić, S., *Object-Oriented Database Programming*, Springer-Verlag, 1989.

[ABF85] Alderson, A., Bott, M. F. & Falla, M. E., "An Overview of the ECLIPSE Project," in *IEEE Software Engineering Series 1: Integrated Project Support Environments*, John McDermid, ed., 1985, pp. 101-113.

[AnH87] Andrews, T. & Harris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment," in *Proceedings of the OOPSLA'87 Conference*, Norman Meyrowitz, ed., ACM Press, October 1987, pp. 430-440.

[AsG90] Ashworth, C. & Goodland, M., *SSADM: A Practical Approach*, McGraw Hill, 1990.

[Atw85] Atwood, T. M., "An Object-Oriented DBMS for Design Support Applications," in *Proceedings of the IEEE 1st International Conference on Computer Aided Technology 85*, 1985, pp. 299-301.

[Atw89] Atwood, T., "An Introduction to Object-Oriented Database Management System," *HOTLINE on Object-Oriented Technology* 1 (1) (Nov 1989), pp. 11–12.

[Aul89] Auld, W., "That Object of Design," *System International* 17 (6) (June 1989), pp. 83-88.

[Bai89] Bailin, S. C., "An Object-Oriented Requirements Specification Method," *Communications of the ACM* 32 (5) (May 1989), pp. 608-623.

[Ban88] Bancilhon, F., "Object-Oriented Database Systems," in *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database System*, Austin, Texas, March 1988, pp. 152-162.

[BCG87] BanerJee, J., Chou, H-T., Garza, J. F., Kim, W., Woelk, D. & Ballou, N., "Data Model Issues for Object-oriented Applications," *ACM Transactions on Office Information Systems* 5 (1) (Jan 1987), pp. 1-26.

[Bar82] Barnes, J. G. P., *Programming in Ada*, Addison-Welsey Publishing Company, 1982.

[BRL88] Beck, K., Raghavan, R., LaLonde, W. R. & Weinreb, D., "Panel: Experiences with Reusability," in *Proceedings of the OOPSLA'88 Conference*, ACM Press, September 1988, pp. 372-376.

[BiO85] Birrell, N. D. & Ould, M. A., *A Practical Handbook for Software Development*, Cambridge University Press, 1985.

[Bir73] Birtwistle, G. M., *Simula Begin*, Auerbach, 1973.

[BPR88] Blaha, M. R., Premerlani, W. J. & Rumbaugh, J. E., "Relational Database Design Using An Object-Oriented Methodology," *Communications of the ACM* 31 (4) (April 1988), pp. 414-427.

[BoS86] Bobrow, D. G. & Stefik, M., "Object-oriented Programming: Themes and Variations," *The AI Magazine* 6 (1986), pp. 40-62.

[Boe87] Boehm, B. W., "Improving Software Productivity," *IEEE Computer* 20 (9) (1987), pp. 43-58.

[Boe75] Boehm, B. W., "Software and its Impact: A Quantitative Assessment," *Datamation* 21 (5) (May 1975), pp. 48-59.

[Bol79] Bollobaś, B'ela, *Graph Theory An Introductory Course*, Springer-Verlag, 1979.

[BoM76] Bondy, J. A. & Murty, U. S. R., *Graph Theory with Applications*, The Macmillan Press Ltd, 1976.

[Boo86] Booch, G., "Object-Oriented Development," *IEEE Transactions on Software Engineering* 12 (2) (Feb 1986), pp. 211-217.

[Bor85] Borgida, A., "Knowledge Representation as the Basis for Requirements Specifications," *IEEE Computer* 18 (4) (April 1985), pp. 82-91.

[Bor88] Borgida, A., "Modeling Class Hierarchies with Contradictions," in *ACM Sigmod International Confernece on Managements of Data*, September 1988, pp. 434-443.

[BoI82] Borning, A. & Ingalls, D. H. H., "A Type Declaration and Inference System for Smalltalk," in *Conference Record of the 9th ACM Symposium on Principles of Programming Languages*, 1982, pp. 133-141.

[Bra83] Brachman, R. J., "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks," *IEEE Computer* 16 (10) (October 1983), pp. 30-36.

[BGM89] Blair, G. S., Gallaghen, J. J. & Malils, J., "Genericity vs Inheritance vs Delegation vs Conformance vs ...," *Journal of Object-Oriented Programming* 2 (3) (Sept/Oct 1989), pp. 11-17.

[Car84] Cardelli, L., "A Semantics of Multiple Inheritance," in *Proceeding of the Conference on the Semantics of Datatypes*, June 1984, pp. 51-67.

[CaW85] Cardelli, L. & Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism," *ACM Computer Surveys* 17 (4) (December 1985), pp. 471-522.

[Car84] Carnese, D., "Multiple Inheritance in Contemporary Programming Languages," Doctoral Dissertation TR-328, MIT Laboratory for Computer Science, Cambridge, MA., 1984.

[Cha80] Chapin, N., "Graphic Tools in the Design of Information Systems," in *System Anslysis and Design: A Foundation for the 1980's*, Cotterman, ed., North-Holland, 1980, pp. 121-162.

[Che76] Chen, P. P-S., "The Entity-Relationship Model - Towards a Unified View of Data," *ACM Transactions on Database Systems* 1 (1) (March 1976), pp. 9-36.

[ChK86] Chou, H-T. & Kim, W., "A Unifying Framework for Version Control in a CAD," in *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August 1986, pp. 336-344.

[CoY90] Coad, P. & Yourdon, E., *Object-Oriented Analysis*, Yourdon Press Computing Series, Englewood Cliffs, New Jersey, 1990.

[Coo86] Cook, S., "Languages and object-oriented programming," *Software Engineering Journal* 1 (2) (March 1986), pp. 73-80.

[Cou87] Coutaz, J., "The Construction of User Interfaces and the Object Paradigm," in *Proceedings of the ECOOP'87 Conference*, 1987, pp. 135-144.

[Cox86] Cox, B. J., *Object Oriented Programming: An Evolutionary Approach*, Addison Wesley, 1986.

[Cur82] Curtis, B., "A Review of Human Factors Research on Programming Languages and Specification," in *Human Factors in Computer Science Conference Paper*, 1982, pp. 212-218.

[DaT88] Danforth, S. & Tomlinson, C., "Type Theories and Object-Oriented Programming," *ACM Computing Surveys* 20 (1) (March 1988), pp. 29-72.

[DEF87] Dart, S. A., Ellison, R. J., Feiler, P. H. & Habermann, A. N., "Software Development Environments," *IEEE Computer* 20 (11) (Novemeber 1987), pp. 18-28.

[Dat75] Date, C. J., *An Introduction to Database Systems (3rd Edition)*, Addison Welsey, 1975.

[Dea89] Deal, S., "The Specification and Recognition of Optimal Layout Configurations for Graph Structures," University College London, University of London, Ph.D. Thesis, 1989.

[deMar78] deMarco, T., *Structured Analysis and System Specification*, Yourdon Inc., New York, December 1978.

[DKL86] Derrett, N., Kent, W. & Lyngback, P., "Some Aspects of Operations in an Object-Oriented Database," *IEEE Database Engineering* (1986).

[DiM87] Diederich, J. & Milton, J., "Experimental Prototyping in Smalltalk," *IEEE Software* 4 (3) (May 1987), pp. 50-64.

[DHM84] Dolotta, T. A., Haight, R. C. & Mashey, J. R., "Unix Time-Sharing System: The Programmer's Workbench," in *Interactive Programming Environments*, D R Barstow, H E Shrobe & E Sandewall, ed., McGraw Hill, New York, 1984, pp. 353-369.

[Eas86] Easteal, C., "Information Systems - Data Models," The Open University Press, Course Notes for M205 Fundamentals of Computer for the Open University, 1986.

[EhM85] Ehrig, E. & Mahr, B., *Fundamentals of Algebraic Specifications 1 Equations and Initial Semantics*, Springer-Verlag, 1985.

[EHZ89] Elizabeth, M., Hull, C., Zarea-Aliabadi, A. & Guthrie, D. A., "Object-Oriented Design, Jackson System Development (JSD) Specification and Concurrency," *Software Engineering Journal* 3 (2) (March 1989), pp. 79-86.

[Eve80] Everitt, B., *Cluster Analysis*, Halsted Press, 1980.

[Fis87] Fischer, G., "Cognitive View of Reuse and Redesign," *IEEE Software* 4 (4) (July 1987), pp. 60-72.

[GMT87] Gallo, F., Minot, R. & Thomas, I., "The Object Management System of PCTE as a Software Engineering Database Management System," *SIGPLAN Notices* 22 (1) (Jan 1987), pp. 12-15.

[Gib89] Gibson, M. L., "The CASE Philosophy," *Byte* 14 (April 1989), pp. 209-218.

[GoM82] Goguen, J. A. & Meseguer, J., "Rapid Prototyping in the OBJ Specification Language," *Software Engineering Notes* 7 (3) (1982), pp. 75-84.

[Gol83a] Goldberg, A., *Smalltalk-80: Interactive Programming Environment*, Addison Wesley, 1983.

[Gol83b] Goldberg, A., *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1983.

[Gom84] Gomaa, H., "A Software Design Method for Real-Time Systems," *Communications of the ACM* 27 (9) (September 1984), pp. 938-949.

[Goo87] Goodwin, N., "Functionality and Usability," *Communciations of the ACM* 30 (3) (March 1987), pp. 229-233.

[GoL85] Gould, J. D. & Lewis, C., "Designing for Usability: Key Principles and What Designers Think," *Communications of the ACM* 28 (3) (March 1985), pp. 300-311.

[Gre89] Green, T. R. G., "Cognitive Dimensions of Notations," in *Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group*, A Sutcliffe and L Macaulay, ed., Sept 1989, pp. 443- 460.

[Gri79] Gries, D., "Current Ideas in Programming Methodology," in *Research Directions in Software Technology*, Peter Wegner, ed., MIT Press, 1979, pp. 254-275.

[Han87] Hanks, W. T. ML. andP., ed., *The New Collins Concise Dictionary of the English Language*, Guild Publishing London, 1987.

[Hew77] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence* 8 (1977), pp. 323-364.

[Hof79] Hofstadter, D. R., *Gödel, Escher, Bach: An Eternal Golden Braid*, Penguin Book, 1979.

[How82] Howden, W. E., "Contemporary Software Development Environments," *Communications of the ACM* 25 (5) (May 1982), pp. 318-329.

[Hut89] Hutchinson, A., "Inheritance and kinds of slots," *The Computer Journal* 32 (1) (1989), pp. 63-67.

[Jac83] Jackson, M., *System Development*, Prentice Hall, 1983.

[JaK87] Jacky, J. P. & Kalet, I. J., "An Object-Oriented Programming Discipline for Standard Pascal," *Communications of the ACM* 30 (9) (September 1987), pp. 772-776.

[Jac87] Jacobson, I., "Object Oriented Development in an Industrial Environment," in *Proceedings of the OOPSLA'87 Conference*, ACM Press, October 1987, pp. 183-191.

[Joh85] Johnson, P., "Towards a Task Model of Messaging: An Example of The Application or TAKD to User Interface Design," in *Proceedings of the HCI'85 Conference*, Cambridge Press, 1985, pp. 46-62.

[Joh88] Johnson, R. E., "Designing Reusable Classes," *Journal of Object-Oriened Programming* 1 (2) (June/July 1988), pp. 22-35.

[Kah89] Kahn, K., "Objects - A Fresh Look," in *Proceedings of the Third European Conference on Object-Oriented Programming, 1989*, Stephen Cook, ed., Cambridge University Press, Cambridge, England, July 1989, pp. 207-223.

[Kay84] Kay, A., "Computer Software," *Scientific American* 251 (3) (September 1984), pp. 41-47.

[Kil89] Kilian, M., "A Model for Integrating Trellis with an Object-Oriented Database," in *Proceedings of the 2nd International Workshop on Distribution and Objects*, DECUS Müchen, April 1989, pp. 141-159.

[Kli77] Kling, R., "The Organizational Context of User-Centered Software Designs," *MIS Quarterly* 1 (1977), pp. 41-52.

[LRV88] Lécluse, C., Richard, P. & Velez, F., "O_2 an Object-Oriented Data Model," in *ACM SIGMOD International Conference on Management of Data*, Sept 1988, pp. 424-434.

[LHR88] Lieberherr, K., Holland, I. & Riel, A., "Object-Oriented Programming: An Objective Sense of Style," in *Proceedings of the OOPSLA'88 Conference*, ACM Press, September 1988, pp. 323-334.

[Lie86] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented System," in *Proceedings of the OOPSLA' 86 Conference*, 1986, pp. 214-223.

[Lin88] Linton, M., *InterViews Reference Manual (Version 2.4)*, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, August 1988.

[LCV87] Linton, M. A., Calder, P. R. & Vlissides, J. M., "InterViews: A C++ Graphical Interface Toolkit," *Proceedings of the C++ Workshop* 22 (2) (1987).

[LVC89] Linton, M. A., Vlissides, J. M. & Calder, P. R., "Composing User Interfaces with InterViews," *IEEE Computer* 22 (2) (Feburary 1989), pp. 8-22.

[LiG86] Liskov, B. & Guttag, J., *Abstraction and Specification in Program Development*, MIT Press, 1986.

[Loc87] "Special Issues on Object-Oriented Databases," *ACM Transactions on Office Information Systems* 5 (1) (January 1987).

[Lor86] Lorensen, W., "Object-Oriented Design," General Electric Co, CRD Software Engineering Guidelines, 1986.

[Luc71] Lucas, H. C., "A User-Oriented Approach to System Design," in *Proceedings of the National ACM Conference*, ACM Press, 1971, pp. 325-338.

[Mar88] Martin, C. F., "Second-Generation CASE Tools: A Challenge to Vendors," *IEEE Software* 5 (2) (March 1988).

[Mar73] Martin, J., *Design of Man-Computer Dialogues*, Prentice-Hall, Englewood Cliffs N.J., 1973.

[McC89] McClure, C., "The CASE Experience," *Byte* 14 (April 1989), pp. 235-246.

[McD85] McDermid, J., ed., *Integrated Project Support Environments*, IEE Software Engineering Series 1, 1985.

[Mey88] Meyer, B., *Object-oriented Software Construction*, Prentice-Hall, 1988.

[Mey87] Meyer, B., "Reusability: The Case for Object-Oriented Design," *IEEE Software* 4 (1) (March 1987), pp. 50-64.

[Mic88] Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages," *Journal of Object-Oriented Programming* 1 (1) (April 1988), pp. 12-35.

[Mil56] Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *The Psychological Review* 63 (2) (March 1956), pp. 81-97.

[MiR87] Minsky, N. H. & Rozenshtein, D., "A Law-Based Approach to Object-Oriented Programming," in *Proceedings of the OOPSLA'87 Conference*, ACM Press, October 1987, pp. 482-493.

[Mit82] Mittermeir, R. T., "CML-Graphs - A Notation for Systems Development," in *Cybernetics and System Research. Proceedings of the 6th European Meeting on Cybernetics and Sysems Research*, R Trappl, ed., North Holland, 1982, pp. 803-309.

[Mit86] Mittermeir, R. T., "Object-Oriented Software Design.," in *Software Engineering Environment - Proceedings of the international workshop on software engineering environment.*, China Academic publishers, ed., 1986, pp. 51-64.

[Nyg86] Nygaard, K., "Basic Concepts in Object Oriented Programming," *SIGPLAN Notices* 21 (10) (October 1986), pp. 128-132.

[OHK87] O'Brien, P. D., Halbert, D. C. & Kilian, M. F., "The Trellis Programming Environment," in *Proceedings of the OOPSLA'87 Conference*, ACM Press, October 1987, pp. 91-102.

[OBH86] O'Shea, T., Beck, K., Halbert, D. & Schmucker, K. J., "Panel On: The Learnability of Object-Oriented Programming Systems," in *Proceedings of the OOPSLA'86 Conference*, November 1986, pp. 502-504.

[Pas86] Pascoe, G. A., "Elements of Object-Oriented Programming," *Byte* 11 (8) (1986), pp. 139-144.

[Pet77] Peterson, J. L., "Petri Nets," *Computing Surveys* 9 (3) (September 1977), pp. 223-251.

[PeW90] Petre, M. & Winder, R., "On Languages, Models and Programming Styles," *The Computer Journal* 33 (2) (April 1990), pp. 173-180.

[Pok89] Pokkunuri, B. P., "Object Oriented Programming," *SIGPLAN Notices* 24 (11) (November 1989), pp. 96-101.

[PLK89] Poo, C-C. D., Loh, W. L. & Kazmi, P., "An Approach to Object-Oriented System Specification Based on the Jackson System Development Method," Department of Information Systems and Computer Science, National University of Singapore, Technical Report TR11/89, Nov 1989.

[Pos86] Post, J., "Application of a Structured Methodology to Real-Time Industrial Software Development," *Software Engineering Journal* 1 (6) (November 1986), pp. 222-235.

[Pow87] Power, L., "Workshop on the Specification and Design of Objects," in *OOPSLA'87 Addendum to the Proceedings*, 1987, pp. 7-16.

[Pre87] Pressman, R. S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill series in Software Engineering and Technology, 1987.

[PLT87] Pugh, J. R., LaLonde, W. R. & Thomas, D. A., "Introducing Object-Oriented Programming into the Computer Science Curriculum," *ACM SIGCSE Bulletin* 19 (2) (Feburary 1987), pp. 98-102.

[PuW90a] Pun, W. & Winder, R., "A Design Method for Object-Oriented Implementation," in *BIGRE (awaiting publication)*, Stephen Cook and J Bézivin, ed., 1990.

[PuW90b] Pun, W. & Winder, R., "A Model for Inheritance Factorisation in Object-Oriented Systems," *Formal Aspects of Computing (accepted to be published)* 2 (4) (1990).

[PuW89a] Pun, W. & Winder, R., "Using Data Flow Design Method in Object-Oriented Programming: It's Pros and Cons," *The OOPS Report* 2 (2) (April 1989), pp. 3-9.

[PuW89b] Pun, W. & Winder, R., "A Design Method for Object-Oriented Programming," in *Proceedings of the 3rd European Conference on Object-Oriented Programming*, Stephen Cook, ed., July 1989, pp. 225-240.

[Rae85] Raeder, G., "A Survey of Current Graphical Programming Techniques," *IEEE Software Engineering* 18 (August 1985), pp. 11-26.

[RPT84] Ramamoorthy, C. V., Prakash, A., Tsai, W-T. & Usuda, Y., "Software Engineering: Problems and Perspectives," *IEEE Computer* 17 (10) (October 1984), pp. 191-209.

[Rei84] Reiss, S. P., "Graphcial Program Development with PECAN," *SIGPLAN Notices* 19 (May 1984), pp. 30-41.

[Ren82] Rentsch, T., "Object-Oriented Programming," *SIGPLAN Notices* 17 (9) (1982), pp. 51-57.

[Rep84] Reps, T., "The Synthesizer Generator," *SIGPLAN Notices* 19 (5) (May 1984), pp. 42-48.

[RWW88] Roberts, G. A., Winder, R. L. & Wei, M., "The Solve Object-Oriented Programming System for Parallel Computers," Dept of Computer Science, University College London, RN/89/7, October 1988.

[Rob89] Robinson, P., "Hierarchic Object Oriented Design Method - HOOD," Seminar Notes of OOPS 23, London, UK, April 1989.

[Ros85] Ross, D. T., "Applications and Extensions of SADT," *IEEE Software* 2 (2) (April 1985), pp. 25-34.

[RoS76] Ross, D. T. & Schoman, K. E., "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering* SE-3 (January 1976), pp. 6-15.

[RoG89] Rosson, M. B. & Gold, E., "Problem-Solving Mapping in Object-Oriented Design," in *Proceedings of the OOPSLA'89 Conference*, October 1989, pp. 7-10.

[RMK88] Rosson, M. B., Maass, S. & Kellogg, W. A., "The Designer As User: Building Requirements for Design Tools from Design Practice," *Communications of the ACM* 31 (11) (November 1988), pp. 1288-1297.

[Rum87] Rumbaugh, J., "Relations as Semantic Constructs in an Object-Oriented Language," in *Proceedings of the OOPSLA'87 Conference*, ACM Press, October 1987, pp. 466-481.

[Sak89] Sakkinen, M., "Disciplined Inheritance," in *Proceedings of the Third European Conference on Object-Oriented Programming, 1989*, Stephen Cook, ed., Cambridge University Press, Cambridge, England, July 1989, pp. 39-56.

[Sau89] Saunders, J., "A Survey of Object-Oriented Programming Languages," *Journal of Object-Oriented Programming* 1 (6) (Mar/Apr 1989), pp. 5-11.

[SSR85] Scheffer, P. A., Stone, A. H. & Rzepka, W. E., "A Case Study of SREM," *IEEE Computer* 18 (4) (April 1985), pp. 47-54.

[Sch86] Schifler, R. W., "The X Window System," *ACM Transactions on Graphics* 5 (2) (April 1986), pp. 79-109.

[ShT83] Shapiro, E. & Takeuchi, A., "Object-Oriented Programming in Concurrent Prolog," *New Generation Computing* 1 (1983), pp. 25-48.

[Sha84] Shaw, M., "Abstraction Techniques in Modern Programming Languages," *IEEE Software* 1 (10) (Oct 1984), pp. 10-26.

[ShM88] Shlaer, S. & Mellor, S. J., *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

[Shu89] Shutt, R. N., "A Rigorous Development Strategy Using the OBJ Specification Language and the MALPAS Program Analysis Tools," in *Proceedings of the 2nd European Software Engineering Conference*, G Goos & J Hartmanis, ed., Springer-Verlag, University of Warwick, Conventry UK, September 1989, pp. 262-291.

[Sny87] Snyder, A., "Inheritance and the Development of Encapsulated Software Components," in *Research Directions in Object-Oriented Programming*, B. Shriver & Peter Wegner, ed., MIT Press, 1987, pp. 165-188.

[Som89] Sommerville, I., *Software Engineering (Third Edition)*, Addison-Welsey, 1989.

[Str86a] Strom, R., "A Comparison of the Object-Oriented and Process Paradigm," *SIGPLAN Notices* 21 (10) (October 1986), pp. 88-97.

[Str86b] Stroustrup, B., *The C++ Programming Language*, Addison Wesley, 1986.

[Str88] Stroustrup, B., "What is Object-Oriented Programming?," *IEEE Software* 5 (3) (May 1988), pp. 57-76.

[SZH85] Swinehart, D. C., Zellweger, P. T. & Hagmann, R. B., "The Structure of Cedar," *SIGPLAN Notices* 20 (7) (July 1985), pp. 230-244.

[TGP89] Taenzer, D., Ganti, M. & Podar, S., "Problems in Object-Oriented Software Reuse," in *Proceedings of the Third European Conference on Object-Oriented Programming*, Stephen Cook, ed., Cambridge University Press, 1989, pp. 25-38.

[TeI77] Teichroew, D. & III, E. A. H., "PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transaction of Software Engineering* 3 (Jan 1977), pp. 41-48.

[Tou87] Touati, H., "Is Ada an Object-oriented Programming Language?," *SIGPLAN Notices* 22 (5) (May 1987), pp. 23-26.

[Tou88] Touretzky, D. S., *The Mathemactics of Inheritance Systems*, Pitmans, 1988.

[Tre84] Trebaul, M. J., "Smalltalk: The User Interface (A Translated Paper)," *Actes des Journees Afcet-Informatique, Langages Orientés Objet* (1984).

[UnS87] Ungar, D. & Smith, R., "Self: The Power of Simplicity," in *Proceedings of the OOPSLA'87 Conference*, ACM Press, 1987, pp. 227-242.

[VlL88] Vlissides, J. M. & Linton, M. A., "Applying Object-Oriented Design to Structure Graphics," in *C++ Confernece Proceedings 1988*, 1988.

[War89] Ward, P. T., "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software* 6 (2) (March 1989), pp. 74-82.

[Weg87] Wegner, P., "Dimensions of Object-Based Language Design," in *Proceedings of the OOPSLA'87 Conference*, 1987, pp. 168-182.

[Weg88a] Wegner, P., "The Object-Oriented Classification Paradigm ," in *Research Directions in Object-Oriented Programming*, MIT Press, ed., 1988, pp. 479-560.

[Weg88b] Wegner, P., "Object-Oriented Concept Hierarchies (Draft Paper)," July 1988.

[WeQ84] Weil, W. & Quayle, M. A., "The Friendly LOOPS Primer," 1984.

[Wir71] Wirth, N., "Program Development by Stepwise Refinement," *Communications of the ACM* 14 (4) (April 1971), pp. 221-227.

[Yeh77] Yeh, R. T., *Current Trends in Programming Methodology. Volume 1: Software Specification and Design*, Prentice-Hall, New Jersey, 1977.

[YoC75] Yourdon, E. & Constantine, L., *Structured Design*, Yourdon, Inc., New York, 1975.

[Zdo86] Zdonik, S., "Object Management Systems for Design Environment," in *IEEE Database Engineering 1986*, 1986, pp. 259-266.