Object-Oriented Design Methodologies

for

Software Systems

by

Luiz Fernando Capretz

Ph.D. Thesis

UNIVERSITY OF
NEWCASTLE UPON TYNE

University of Newcastle upon Tyne

Computing Laboratory

November 1991

Abstract

In the last few years, demand for object-oriented software systems has increased dramatically, and it is widely accepted that present software engineering methodologies are unable to cope with the needs of that demand. The object-oriented paradigm has promised to revolutionise software development, and it has been seen as an attempt to extend and apply the techniques of encapsulation and inheritance, not only in the implementation phase but also during the design and system analysis phases of the software development process. As a result, several methodologies have recently arisen to support software development based on an object-oriented approach.

This thesis is concerned with object-oriented design methodologies for software systems and addresses four points. First, a classification scheme for object-oriented development methodologies is proposed and their problems and limitations are pointed out. Second, a general methodology for object-oriented design (called **MOOD**) is presented. MOOD is unrelated to any programming language, yet is capable of being used to design a variety of object-oriented software systems. In particular, MOOD allows the creation of a design mainly in terms of classes, objects and inheritance, and the representation of a design graphically by a set of class hierarchy diagrams, composition diagrams, object diagrams and operation diagrams. Third, the thesis puts software development into a new perspective, by proposing an alternative software life cycle model which links system analysis, domain analysis, design and implementation to form a coherent object-oriented software development life cycle model that takes reusability into account during the design phase. Lastly, a prototype of an environment which supports MOOD has been developed and is described.

Acknowledgements

It is my pleasure to acknowledge some people who have influenced and contributed to the research reported in this thesis.

First and foremost, I am indebted to my supervisor, Professor Peter A. Lee, for many reasons. He has been a constant source of useful advice and guidance over the last few years. He has challenged me to form many ideas in this thesis and encouraged me throughout their development. His detailed readings, numerous comments and constructive criticisms on the early drafts of this thesis have been invaluable and added many improvements to the content of the thesis. I am also extremely grateful for the tremendous effort that Pete has made so that this thesis could be finished on time. His efforts are greatly appreciated and will not be forgotten.

I would like to thank several staff members of the Computing Laboratory. In particular, I wish to acknowledge useful technical discussions with Dr. Lindsay Marshall, Dan McCue, Ron Kerr and Jim Wight. I am thankful to Shirley Craig for her patience and efficiency in searching out many relevant references for this thesis.

I have also been very fortunate to receive support and assistance from my wife, Miriam, during some very difficult periods (for both of us) throughout this research. If I had been her, I would not have put up with myself.

Special thanks are due to my family, in Brazil, for understanding my absence during the moments they most needed me. Their endurance has been really admirable. My mother, especially, has given me strength and motivation throughout my studies, and for teaching me the meaning of perseverance.

Financial support for this research has been provided by grants from CAPES (Brazilian Federal Agency for Postgraduate Education).

Table of Contents

Abstract .... *i*

Acknowledgements .... *ii*

Table of Contents .... *iii*

List of Figures .... *vi*

List of Tables .... *viii*

1 INTRODUCTION 1

1.1 THESIS AIMS 7

1.2 THESIS OUTLINE 8

2 OBJECT-ORIENTED DESIGN 12

2.1 APPLICATION DOMAINS AND SOLUTION DOMAINS 13

2.2 THE PROFILE OF PRESENT SOFTWARE SYSTEMS 15

2.3 TOWARDS AN OBJECT-ORIENTED APPROACH 17

2.3.1 Abstract Data Types 19

2.3.2 Evolution of Abstraction in Programming Languages . 22

2.3.3 Comparison between "Object-Oriented" Languages ... 26

2.4 CHARACTERISATION OF AN OBJECT-ORIENTED MODEL 34

2.4.1 Objects 37

2.4.2 Classes 37

2.4.3 Inheritance 39

2.5 PHILOSOPHY OF OBJECT-ORIENTED DESIGN 41

2.5.1 Domain Analysis 44

2.5.2 Reuse of Software 46

2.5.3 Software Life Cycle Models 51

2.6 SUMMARY 59

3 CLASSIFICATION OF OBJECT-ORIENTED METHODOLOGIES 61

3.1 CLASSIFICATION OF METHODOLOGIES 62

3.1.1 Classification of Existing Object-Oriented Methodologies 67

3.2 SURVEY OF EXISTING METHODOLOGIES 69

3.2.1 The Booch Methodologies and Their Influences 70
3.2.2 OOSD 81
3.2.3 Responsibility-Driven Design 83
3.2.4 OORA 85
3.2.5 OMT 88
3.2.6 OOSA 89
3.2.7 Other Methodologies 91
3.3 FINAL REMARKS ON CLASSIFICATION 98

4 A METHODOLOGY FOR OBJECT-ORIENTED DESIGN 100
4.1 THE CONTEXT OF MOOD 101
4.2 THE STEPS FOR OBJECT-ORIENTED DESIGN 105
4.2.1 Representation of the Design Model 105
4.2.2 Identification of Components 107
4.2.3 Identification of Classes 108
4.2.4 Identification of Inheritance 110
4.2.5 Identification of Objects 117
4.2.6 Identification of Software System Behaviour 121
4.3 FIRST EXPERIENCE WITH MOOD 124
4.3.1 An Electronic Mail System 124
4.3.2 Identifying and Representing the EMS Classes 125
4.3.3 Identifying and Representing Inheritance 129
4.3.4 Comments on Inheritance within MOOD 132
4.4 A CASE ENVIRONMENT FOR MOOD 134
4.5 FINAL REMARKS ON MOOD 137

5 REUSABILITY AND SOFTWARE LIFE CYCLE ISSUES 140
5.1 REUSABILITY WITHIN MOOD 141
5.1.1 Relationships between Components 142
5.1.2 The Process of Component Reuse 145
5.1.3 The Lifetime of Reusable Components 146
5.2 MOOD WITHIN A SOFTWARE LIFE CYCLE MODEL 150
5.2.1 The Role of the Knowledge about the Application Domain 150
5.2.2 The Proposed Software Development Life Cycle 151
5.3 COMMENTS ON THE SOFTWARE DEVELOPMENT PROCESS 158

6 THE MOOD PROTOTYPE 160
6.1 OVERALL IMPLEMENTATION ISSUES 161
6.1.1 InterViews 163
6.2 THE MOOD INTERFACE 166
6.2.1 The Interface of a Diagram Editor 168
6.2.2 The Interface of the Checkers 169
6.3 THE MOOD DATABASE 171
6.3.1 Persistence 171
6.3.2 Manipulation of the Design Information 172
6.4 THE TOOLS 173
6.4.1 The Class Hierarchy Diagram Editor 174
6.4.2 The Object Diagram Editor 174
6.4.3 The Operation Diagram Editor 175
6.4.4 The Composition Diagram Editor 176
6.4.5 The Checkers 177
6.5 CONSIDERATIONS ON THE MOOD PROTOTYPE 182

7 EXPERIENCE WITH MOOD 184
7.1 THE SUITABILITY OF MOOD 184
7.2 GENERAL ISSUES RELATED TO MOOD 188

8 CONCLUSIONS 193
8.1 DISCUSSION 193
8.2 DIRECTIONS FOR FUTURE RESEARCH 198
8.3 IN CONCLUSION 201

References 203

List of Figures

Figure 2.1 Mapping between Application and Solution Domains 14
Figure 2.2 The Background of the Object-Oriented Paradigm 18
Figure 2.3 Language Evolution 25
Figure 2.4 Concepts Related to the Object-Oriented Paradigm 38
Figure 2.5 Hierarchy Representing Inheritance 40
Figure 2.6 The Classic Waterfall Software Life Cycle Model 52
Figure 2.7 A Spiral Software Life Cycle Model 55

Figure 3.1 Some Combinations of Approaches 65
Figure 3.2 Classes as Packages 75
Figure 3.3 Seniority Hierarchy in Object Diagrams 77
Figure 3.4 Class Representation in OOSD 82
Figure 3.5 A Class Card 85
Figure 3.6 Object Representation in OORA 87
Figure 3.7 An Object Model Diagram 88
Figure 3.8 An Information Structure Diagram 90
Figure 3.9 An ObjectOry System 94
Figure 3.10 Representing Classes and Methods 96
Figure 3.11 Notation for Classes, Objects, Methods and Inheritance 97

Figure 4.1 The Context of MOOD 103
Figure 4.2 A Composition Diagram for an Environment 109
Figure 4.3 Identification of Inheritance from Venn Diagrams 113
Figure 4.4 A Generic Class Hierarchy Diagram 115
Figure 4.5 Correspondence between Components and Classes 116
Figure 4.6 A State Transition Diagram for a Screen-Window Object .. 119
Figure 4.7 An Object Diagram for Objects in an Environment 120
Figure 4.8 An Operation Diagram for Drawing a Polygon 123
Figure 4.9 The Message Class for the EMS 127
Figure 4.10 A Venn Diagram for the EMS 130
Figure 4.11 The Class Hierarchy for the EMS Mailfile 131
Figure 4.12 The Class Hierarchy for the EMS Menus 132
Figure 4.13 The Static and Dynamic Layers of a Software System 137
Figure 4.14 The Steps Recommended by MOOD 138

Figure 5.1 Reusability within MOOD 142
Figure 5.2 A Procedure to Reuse a Component 147
Figure 5.3 The Lifetime of a Reusable Component 148
Figure 5.4 MOOD within a Software Development Life Cycle 152

Figure 6.1 The Interactor Class Hierarchy 164
Figure 6.2 A Scene Composition 165
Figure 6.3 The Tools of the MOOD Environment 167
Figure 6.4 The Class Hierarchy Diagram Editor Window 169
Figure 6.5 A Composition Diagrams for an Editor 170
Figure 6.6 The Interface for the Checkers 171
Figure 6.7 The Class Hierarchy Diagram Editor 175
Figure 6.8 The Object Diagram Editor 176
Figure 6.9 The Operation Diagram Editor 177
Figure 6.10 The Composition Diagram Editor 178
Figure 6.11 The Consistency Checker 179
Figure 6.12 The Class Browser 180
Figure 6.13 The Configuration Management Tool 182

Figure 7.1 Phases Versus Time 191

List of Tables

Table 2.1 Comparison among "Object-Oriented" Languages 32
Table 5.1 Input, Tasks and Output of each Phase 157
Table 7.1 Phases Versus Abstraction Mechanisms 190

Chapter 1

INTRODUCTION

The development of software systems is now regarded as among the most complex tasks performed by mankind. The problems that are caused by the scale of this complexity have been recognised for a long time. This complexity affects the costs and time expended on the construction of software systems. Moreover, after being built, software systems are often unreliable, difficult to use and, what is worse, they are frequently extremely difficult to maintain and evolve. These problems, together with the ever-increasing demand for software systems, have led to what has become known as *the software crisis.*

Most of the complexity found in software systems is left to be mastered by software engineers helped (or hindered!) by the discipline of software engineering. As its name suggests, software engineering is concerned with the establishment and use of sound engineering principles and good management practice. It involves the development of new techniques, methodologies and tools, and their use as an appropriate engineering approach to control and overcome the complexity inherent in software systems. The aim is to obtain, within adequate resource limitations, software which is of high quality in an explicitly defined sense.

For many years, it has been recognised that the use of methodologies has an important role to play in order to accomplish a well-engineered software system. Methodologies provide a set of rules, principles, guidelines and notational conventions which helps software engineers to understand, organise and decompose software systems, and hence manage their complexity. Such methodologies, therefore, facilitate the development of complex and/or large software systems and give the software engineers the feeling that technology is an extension of their capabilities. In the past few years, many methodologies have been proposed to support the engineering of software systems. These methodologies have addressed different aspects of software development ranging from requirements through to testing. Many of these methodologies have appeared in response to new ideas about how to handle software complexity.

A recent idea which has been receiving a great deal of attention from software engineers is the **object-oriented** paradigm. Currently, this paradigm is thought to be an important aspect of software development, so much so that it is now a major research area which is expected to bring significant benefits in the design of software systems. The rapid development of this paradigm during the past ten years can be attributed to several important reasons, which are discussed in more detail later in this thesis, but which include: better modelling of real-world applications; better structure for software systems based on abstract data type concepts; and the possibility of software reuse during the design of software systems.

Because of the rapid developments in the object-oriented field, it has become very fashionable to describe many kinds of software systems using object-oriented terminology. The term itself has become a buzzword, meaning different things to different people and hence has come to be somewhat of a minefield of contradictions and confusion. These issues are discussed in detail later in this thesis, but for the moment it is important to establish some object-oriented terminology.

In the scope of this thesis, an **object** embodies an abstraction characterised by an entity in the real-world. A **class** (or type) is a template description which specifies common properties and behaviour for a group of similar objects and an object is an instance of a class. The classes themselves can be organised into class hierarchies. Such class hierarchies allow similar classes to be related together in such a way that commonalities of one class can be inherited (reused) rather than duplicated by classes lower in the hierarchy, thus simplifying the design and implementation of those lower level classes.

The properties and behaviour of objects, and hence their commonalities, are described in terms of **attributes** and **operations**. An attribute is a named property of an object which holds a value and maintains an abstract state for that object. An operation identifies an action which may be applied to objects of a class. Objects from each class are manipulated by invoking the operations upon the attributes of these objects. **Inheritance** is a mechanism which permits classes to share attributes and operations based on relationships of specialisation and generalisation between them within a hierarchy of classes.

An object-oriented approach encourages abstraction because a class represents a common abstraction of a collection of similar objects. In general, classes are seen as a means of expressing the commonalities between a group of similar objects when building a software system, whereas objects are instantiated from a class in an executing software system and encompass the behaviour of a particular real-world application. In this way, an object-oriented approach allows designers to create abstractions with which it is possible to deal with concepts which are close to real-world applications.

Because of the perceived importance of an object-oriented approach, several methodologies have recently emerged to support object-oriented design. Ideally, an object-oriented design methodology should allow designers to produce software systems mainly in terms of classes, objects and inheritance. Nevertheless, this point of view is not usually emphasised by some current methodologies as will be seen in Chapter Three.

Another important idea brought forward by software engineering is the concept of **software life cycle models**. Software life cycle models have been proposed in order to systematise the several stages which a software system goes through. The software life cycle can be divided into different phases, although in practice some phases may overlap each other. Despite some variations, the main phases for the traditional and best known software life cycle model are: analysis (requirements and specification), design, implementation and maintenance.

Software development involves a set of transformations starting from requirements and ending with implementation. Between these two points, a number of other abstract representations are described. The aim is to divide a complex software system by steps into more manageable pieces, that is, each new abstract representation gives the designers more details about a software system than the previous one and allows them to make additional refinements in order to move towards the next abstract representation. Thus, software development may also be seen as a process of creation, manipulation and refinement of abstractions of real-world applications. The complete software life cycle spans from requirements to implementation, followed by an operational phase of maintenance during which bugs are fixed and enhancements are introduced. Therefore, the software life cycle circumscribes the time from the definition of the need for a software system until the moment when it falls into disuse.

Nevertheless, the most well-known software life cycle models do not take into account the issue of how to reuse existing software components when the design of a new software system is being undertaken. The main issue in software reuse is the creation of components which can be reused in software systems other than that software system for which they were originally created. **Reusability** is seen as a suitable technique for improving software quality and reducing software development costs and time, and it has therefore emerged as an important issue in software engineering.

In the past, reusability was primarily concerned with using subroutines from a library during the implementation phase of the software life cycle. Nowadays, however, a great deal of research has been carried out in order to accomplish reusability during the design phase as well. The idea of reusability within an object-oriented approach is special because it is not just a matter of reusing only the code of a subroutine but any commonality expressed in class hierarchies.

The inheritance mechanism encourages reusability within an object-oriented approach (rather than reinvention) by permitting a class to be used in a modified form by deriving a sub-class from it. Reusability should also be easier to accomplish within an object-oriented approach because classes support encapsulation. Encapsulation hides internal details of attributes and operations concerned with class implementation in such a way that the designer (as a *re-user*) needs to know only how to call an operation. Encapsulation encourages reusability because it rewards the development of modular components. In an object-oriented approach, classes are the potentially reusable modular components.

However, one of the major problems which designers are faced with in trying to reuse software is the difficulty of finding reusable components, once such components have been produced. This is primarily because few mechanisms are available to help identify and relate components. In order to provide more convenient reuse, the question of which kinds of mechanisms might help solve this problem arises. The answer is typically couched in terms of finding components which provide specific functionality, from a library of potentially reusable components linked through relationships which express their semantics and functionality. Clearly, what is needed are techniques to create, classify and relate components, and tools which help to store, select and retrieve potentially reusable components. As will be seen, reusability and its supporting mechanisms form a major part of the research reported in this thesis.

Despite all of the progress so far in the object-oriented paradigm, there is a gap in the knowledge concerning object-oriented design. That is, despite the acknowledged importance of software design methodologies and the increasing popularity of the object-oriented paradigm, there is no generally accepted object-oriented design methodology which essentially addresses object-oriented design and considers reusability as part of the software life cycle model. From the theoretical and practical point of view, the development of a new object-oriented design methodology remains a topic of major research interest.

This research is aimed at the creation of a **Methodology for Object-Oriented Design** (**MOOD**)[1], which takes reusability into account as an important aspect of the software life cycle. The main characteristic obtained through the use of MOOD is the design of a software system following strictly object-oriented concepts, as MOOD concentrates on identifying and representing classes, inheritance and objects. In doing so, the architecture of an object-oriented software system at the design level is built around sets of classes and objects. Moreover, considering reusability as a pragmatic (and desirable!) process within the design phase helps the designer to relate software components to each other through relationships which show where a component is defined and used, and in what context. In this way, reusability is considered within a software life cycle model as part of the object-oriented design methodology.

In the context of this research, the product of an object-oriented design is viewed partly as a collection of classes and objects. Classes model real-world

1. Belatedly, another methodology with the same name was discovered, however, the similarity ends in the acronym, as will be seen in Chapter Three.

applications and can be related to one another through the application of inheritance, where inheritance is seen as a mechanism for organising classes with similar properties and behaviour into class hierarchies. MOOD is based on graphical representations of classes, objects and inheritance, with associated rules, principles and guidelines which facilitate the identification and representation of software systems in terms of these classes, objects and inheritance.

MOOD is aimed at designing software systems by following prescribed steps which allow the designer to represent and describe software systems at different levels of abstraction. A design using MOOD basically starts from a given system analysis and produces a graphical description to be implemented, which comprises an information model and a behaviour model, together called the **design model**. The information model is a static representation of a software system using a set of diagrams which shows a global view of the classes and the class hierarchies, built during a stage named **static design**. The behaviour model shows the dynamic relationships between objects, created during a stage denominated **dynamic design**. The design model enables MOOD to maintain close links between the system analysis, design and implementation phases of the software life cycle model, so that the gap which often exists between these three phases can be bridged. In this way, MOOD supports traceability from system analysis through to implementation, and this is believed to be another important aspect of this research.

MOOD employs steps which help the designer to identify classes, build class hierarchies using inheritance, describe the software system behaviour with objects and represent a design graphically with several kinds of diagrams. The rigorousness of the proposed notation increases as the steps are carried out because the design process starts from a given abstract model of the application, often informal, and ends with an object-oriented graphical representation of the software system.

This graphical representation and description can be built interactively within a software development environment or CASE (Computer-Aided Software Engineering) environment. Such an environment provides computer-aided support for a methodology through a set of tools which form that environment, and brings many improvements in software quality and efficiency for software production. The support encompasses consistent coordinated aid and traceability between abstract representations of the software system throughout different phases of the software life cycle. As a by-product of using such an environment, communication among designers is enhanced because they use the same tools to develop a software system. Indeed, it has been found that CASE environments have a profound effect on software development as well as on the software system produced.

Hence, the last few years have also seen a growing interest not only in new methodologies for software development, but also the way in which these methodologies can be supported by computerised tools. Consequently, the ultimate aim for MOOD is that it should be supported by a CASE environment. By this means, the rules, principles, guidelines and graphical notations set up by MOOD can be enforced and followed by designers, and design inconsistencies can be exposed. A MOOD prototype comprising a set of tools which provides automated support for MOOD was implemented in C++ and InterViews, and this will be described later in this thesis.

The MOOD prototype has tools which facilitate the construction of the MOOD diagrams, allow the consistency and completeness of a software design to be checked, and perform the functions of software configuration management. The tools also handle the problem of feedback information (which is a vital part of a design) and help provide documentation for a software system via a report generator. The tools in the MOOD prototype are integrated with each other through a common interface together with a single database used to store design information such as names of classes, attributes, operations and objects, in a uniform representation model.

1.1 THESIS AIMS

This thesis presents the results of research into the development of MOOD. This research also gives emphasis to a different software life cycle model which explicitly recognises the importance of reusability during the design phase. Thus, the primary goals of this research are:

- to classify existing object-oriented methodologies according to their suitability for a particular phase of the software life cycle and their domains of applicability. This classification can be used to understand which methodology is best applied to specific phases of the software life cycle or certain kinds of software systems;

- to create and evaluate a new methodology for object-oriented design independent of any other methodology or any programming language. This methodology (MOOD) can be applied to the design of software systems which will then conform to object-oriented concepts such as classes, objects and inheritance;
- to propose an alternative software life cycle model within an object-oriented framework, which emphasises the importance of software reuse during the development of object-oriented software systems;
- to implement a prototype of an integrated environment for object-oriented design, called the MOOD environment, consisting of a set of tools which provides automated support for the steps, principles and notational conventions proposed by MOOD.

1.2 THESIS OUTLINE

The remainder of this thesis is organised into seven chapters as follows.

Chapter Two expands upon the background of the object-oriented paradigm and is divided into six sections. Section one presents the notions of application domains and solution domains, and shows how the object-oriented paradigm helps to bridge the gap between these two domains. Section two discusses the profile of present software systems in terms of complexity, friendliness, extensibility and reusability. The road towards an object-oriented approach is described in section three which also presents an overview of the most familiar object-oriented programming languages. Section four reviews the main concepts related to an object-oriented approach, and introduces the terminology associated with the object-oriented paradigm which is used in the remainder of this thesis. The terminology to be presented is independent of any object-oriented programming language or system. The fifth section presents the philosophy upon which object-oriented design is based, and discusses important issues related to object-oriented software development such as domain analysis, reusability and software life cycle models. A summary of this chapter is presented in the sixth section.

Chapter Three discusses a series of classification schemes to assist in the understanding of several existing object-oriented methodologies. In the first section of the chapter a series of different schemes for classifying object-oriented methodologies is discussed. This section also compares and contrasts the main differences between the methodologies and their divergent background. The second section introduces the most well-known methodologies, methods and techniques to tackle the analysis and design phases of object-oriented software development. This section discusses the different terminologies employed by them, evaluates the problems and limitations of those methodologies and hence identifies the necessity of developing a new methodology for object-oriented design which can overcome those problems and limitations. The conclusion which can be drawn from this chapter is that up to the present, there is no widely accepted methodology for developing software systems based solely within an object-oriented framework.

Chapter Four contains a detailed description of a new methodology for object-oriented design (named **MOOD**), which is based on the concepts described in the second chapter. The first section places the methodology in the context of software development. The steps which must be followed in order to design a software system according to MOOD are discussed in the second section. A means of graphically representing the produced design with a series of different diagrams is presented as well. The section also discusses how to structure a large software system into manageable pieces. In the third section of the chapter, the methodology is applied to model an electronic mail system in terms of its classes and inheritance. The results of this first experience with MOOD have been used as feedback to improve the methodology and have shown the need for a software development environment to support MOOD. The requirements for an integrated software development environment consisting of a set of tools which automates MOOD is presented in the fourth section. The chapter concludes with a review of the MOOD steps, outlining some of the issues which should be considered when designing an object-oriented software system.

Chapter Five provides an account of reusability and software life cycle issues which arise during object-oriented design. The first section outlines existing mechanisms for achieving reusability. Besides, it concentrates on the main reasons why software components are not reused and examines the problems associated with reusability during the design phase. Additionally, in this section software reusability is added to the framework of the new methodology presented in Chapter Four. Section two proposes a new software life cycle model which encompasses MOOD and addresses reusability, within an object-oriented software development framework. The section also considers the role (during software development) of the knowledge about the application domain, and discusses how the knowledge that the designer has about the application domain can affect the development of object-oriented software systems in terms of a top-down, bottom-up or middle-out approach to software development. The chapter finishes with comments on the software development process presented in the two previous sections.

Chapter Six describes the **MOOD prototype**, the prototype of an environment for object-oriented design which provides automated support for the methodology introduced in Chapter Four. The first section of the chapter discusses overall issues related to the implementation of the prototype. Section two presents the interface for the environment and discusses how the interface classes are derived from the classes provided by InterViews. Section three describes the MOOD database and shows how integration among tools is accomplished using a unified representation model, a single database and a uniform interface. In the fourth section, the features of the tools are outlined. The chapter ends with observations on the use of the MOOD prototype and the design process.

Chapter Seven discusses the results of using MOOD to design a software development environment comprising a set of tools which automates MOOD itself. The purpose of this activity was twofold: firstly, to help improve earlier versions of that methodology by providing feedback based on the outcome of such an experiment; and secondly, to evaluate that methodology and the potential benefits which can be gained through its use, as presented in the first section of this chapter. Other general aspects related to the fully applicability of MOOD within the proposed software development life cycle model are discussed throughout section two.

Finally, Chapter Eight concludes this thesis by reiterating the aims of the research reported in the thesis and discussing the work which has been carried out together with its main contributions. Section one provides some concluding remarks on the experience gained designing and experimenting with MOOD within the alternative software life cycle model proposed. The second section considers future directions and further ideas for the research which has been presented in previous chapters. The last section presents an overall conclusion of the thesis regarding the future of object-oriented software engineering.

Chapter 2

OBJECT-ORIENTED DESIGN

Over the past twenty years, several software development methodologies have appeared. Such methodologies address some phases of the software life cycle ranging from requirements to maintenance. These methodologies have often been developed in response to new ideas about how to cope with complexity in software systems. More recently, due to the increasing popularity of object-oriented programming, development of object-oriented methodologies has become a growing field of interest.

There has also been an explosive growth in the number of software systems described as object-oriented. Some object-oriented ideas have already been applied to various areas such as software engineering, office information systems, system simulation and artificial intelligence. Moreover, programming languages, programming styles and user interfaces have been defined using the object-oriented paradigm, and the need for software development methodologies which follows this paradigm has become imperative.

This chapter covers the background of the object-oriented paradigm and is divided as follows. Section one presents an introduction to application domains and solution domains. Section two discusses the profile of present software systems and shows that the object-oriented paradigm is suitable to develop such software systems. The third section shows that the object-oriented paradigm is not new. Actually, it is based on ideas which have been evolving since the early 1970s, and it combines, purifies and evolves existing techniques, such as abstract data types and system simulation. Section four reviews the main concepts related to the object-oriented paradigm, and introduces the terminology to be used in the next chapters. The fifth section presents the philosophy of object-oriented design. The chapter ends with a summary of its main topics.

2.1 APPLICATION DOMAINS AND SOLUTION DOMAINS

In order to attempt to characterise software development methodologies, it is necessary to understand the concept of software in terms of application domains and solution domains. An application domain may be defined as a set of real-world applications. Similarly, a solution domain is a set of possible solutions to those real-world applications. An entity which belongs to an application domain may be mapped into a solution domain through an abstract representation in such a way that operations on this abstract representation correspond to operations in the real-world application.

In software terms, mapping requires the main ideas about a real-world application to be represented in abstract terms so that the designer can understand the application. The designer can then use those abstractions to develop a model which simulates that real-world application. The mapping may be viewed as a process of building up a model which, when executed by a computer, provides output which is equivalent to results of the application behaviour. Thus, when the designer thinks about a real-world application in an application domain, there should be a mapping to a solution in a solution domain. The solution might be represented by a software system which models the behaviour of that real-world application. The process by which a software system is built up may be aided by a software development methodology. Figure 2.1 illustrates such concepts.

Software development methodologies can provide the means by which it is possible to map an abstraction of a real-world application (which belongs to an application domain) into a software representation (which belongs to a solution domain). A software system represents one model of a solution, among possible solutions to that real-world application. Therefore, software development may be seen as a process of creation, manipulation and refinement of representations which model real-world applications. When the designer transforms a representation or creates an initial one, different representations of a software system are being dealt with.

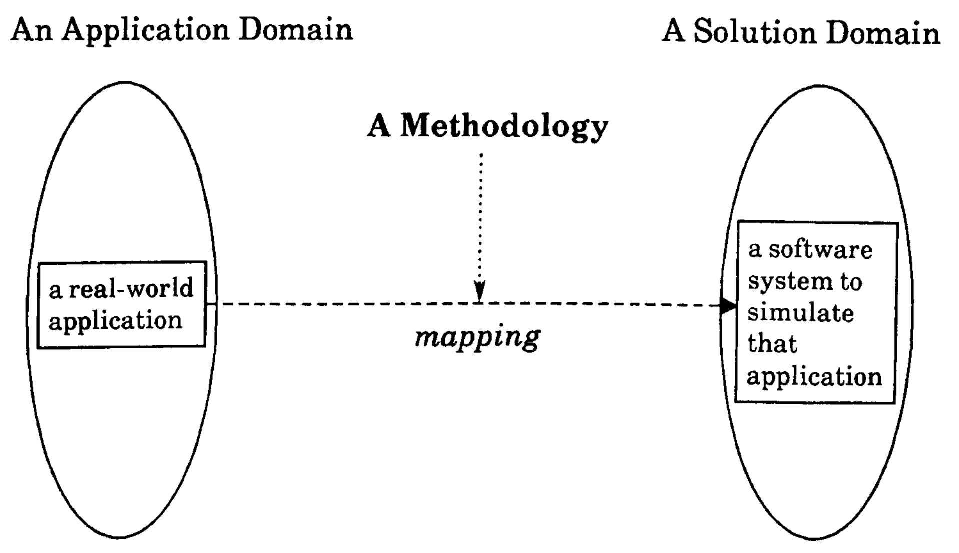


Figure 2.1 Mapping between Application and Solution Domains

The distance between an application domain and a solution domain is called the **semantic gap**. Intuitively, it seems to be evident that the smaller the semantic gap (that is, the closer a software system is to the modelled real-world application) the easier will be the development of that software system, and the greater will be the possibility of guaranteeing understandability, reliability and quality in the achieved solution. One of the objectives which should be pursued during the creation of a new software development methodology, programming language or tool is to narrow the semantic gap as much as possible (Ledgard, 1977).

In spite of some advances, for the last twenty years software development has in the main concentrated on procedures and data separately. Unlike other approaches, an object-oriented approach gathers together procedures and data into a unit, named an object. This point of view for software development comes from the principle that the real-world consists of entities which are composed of data, and operations which manipulate that data. Such entities can be represented by objects and can be abstracted to constitute classes of objects, and each object encompasses data and a set of operations which are meaningful to these data.

In this aspect, an object-oriented approach allows the designer to create abstractions with which it is possible to deal with concepts that are close to the real-world application and the designer is only concerned with those abstractions. Thus, an alternative approach which is based on the identification and manipulation of abstractions represented by classes and objects, and is able to narrow the semantic gap, has appeared.

To summarise, during software development, designers should map a real-world application in an application domain onto an abstract solution (represented by a software system) in a solution domain. Using an object-oriented methodology designers can create abstractions in terms of classes and objects. Like other approaches, an object-oriented approach deals with methodologies and languages, both of which discipline the abstraction process to build software systems. However, in this case, classes and objects are the main items used to model real-world applications.

2.2 THE PROFILE OF PRESENT SOFTWARE SYSTEMS

This section discusses the profile of present software systems and shows that the object-oriented paradigm is suitable to develop software systems that meet that profile. Some important features of new software systems and their requirements include:

- **Complexity**: the internal architecture of current software systems is complex, often including concurrency and parallelism. Abstraction is a technique that helps to deal with complexity. Abstraction involves a selective examination of certain aspects of an application. It has the goal of isolating those aspects which are important for an understanding of the application, and also suppressing those aspects which are unimportant. Abstraction must have a purpose, because the purpose determines what is and is not important for the abstraction. Many different abstractions of the same thing are possible, depending on the purpose for which they are made. Forming abstractions of an application in terms of classes and objects is one of the fundamental elements of the object-oriented paradigm.

- **Friendliness**: this is a relevant requirement of current software systems. Iconic interfaces provide a user-friendly interaction between users and software systems. Icons are graphical representations of objects on the screen and are usually manipulated with the use of a mouse, which has come to be known as *WYSIWYG* (What You See Is What You Get) interaction. In such interfaces, windows, menus and graphical elements are all viewed as objects. The trend to object-oriented graphical interfaces is permeating many areas of software development. This is acknowledged in the most recent generations of software systems for window management systems. Experience would suggest that user interfaces are significantly easier to develop when they are written in an object-oriented fashion. Thus, the object-oriented nature of the WYSIWYG interfaces maps quite naturally into the concepts of the object-oriented paradigm.

- **Extensibility**: this is a property that permits new functionality to be easily added with little modification to existing software systems. With this property, software systems can be easily extended to meet new requirements. New software developments may be carried out entirely by making modifications to what already exists. This incremental development is part of object-oriented thinking.

- **Reusability**: this property facilitates rapid software development by reusing software components already available and also promotes the production of components that could be reused in future software developments. Taking components created by others should be considered more desirable than creating new ones. If there exists a good library of reusable components, reviewing existing components to identify opportunities for reuse could have precedence over writing new software components from scratch. Inheritance is an object-oriented mechanism that increases software reusability.

There are several ways to tackle the problem of software complexity and to achieve friendliness, extensibility and reusability. Investigations of new methodologies, and proposals for new software life cycle models, as well as automated support provided by tools within a software development environment, have all been under research. These trends may be gathered together, and an object-oriented approach seems to be the way to converge them, as can be inferred from the discussion above.

2.3 TOWARDS AN OBJECT-ORIENTED APPROACH

The notion of objects naturally plays a central role in object-oriented software systems and although this concept is much in evidence nowadays, the idea is not a new one. In fact, it could be said that the object-oriented paradigm was not invented, but it actually evolved by refining already existing practices. The confluence of the object-oriented paradigm with other concepts of computer science suggests that the object-oriented paradigm has been biased by other approaches.

The term *object* emerged almost independently in various areas of computer science. Some approaches that have influenced the object-oriented paradigm are: simulation, operating systems, data abstraction and artificial intelligence. Appearing almost simultaneously from the early 1970s, these approaches all cope with the complexity of software in such a way that objects represent abstract components of a software system. For instance, some notions of objects that have emerged in these fields are:

- *Classes* of objects used to simulate real-world application, in Simula-67 (Dahl, 1970). In this language an execution of a computer program is organised as a combined execution of a collection of objects, and objects sharing common behaviour are said to constitute a class.

- Protected resources in operating systems. Hoare (1974) proposed the idea of using an enclosed area as a software unit and introduced the concept of a *monitor*, which is concerned with process synchronisation and contention for resources among processes.

- Data abstraction in programming languages such as CLU (Liskov, 1977), which refers to a programming style in which instances of *abstract data types* (i.e. objects) are manipulated by operations that are exclusively encapsulated within a protected region.

- Units of knowledge called *frames*, used for knowledge representation. Minsky (1975) proposed the notion of frames to capture the idea that behaviour goes with the entity whose behaviour is being described. Thus, a frame can also be represented as an object.

These influences are shown in Figure 2.2. The common characteristics of these concepts are that an object is a logical or a physical entity that is self-contained. Clearly, other items could be added to this list, such as advances in programming languages, as demonstrated in Ada (Buzzard, 1985); and advances in programming methods, including the notion of modularization and encapsulation, as in Modula (Wirth, 1976).

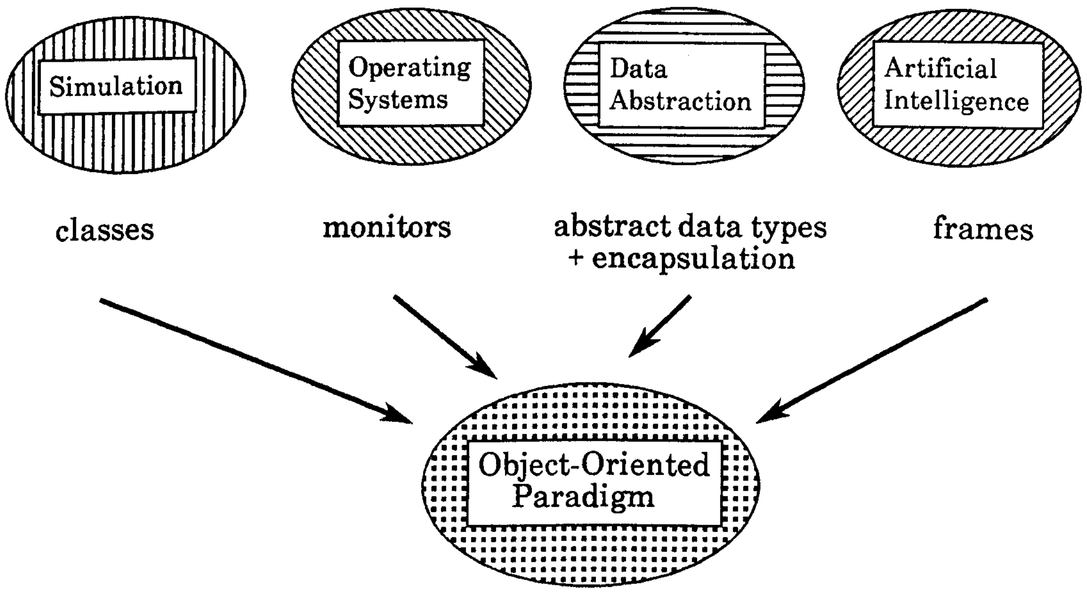


Figure 2.2 The Background of the Object-Oriented Paradigm

The Simula-67 was the first programming language which had objects and classes as central concepts. Simula-67 was initially developed as a language for programming discrete-event simulations, and objects in the language were used to model entities in the real-world application which was being simulated. Despite the early innovation of Simula-67, the term object-oriented became prominent from the Smalltalk language (Goldberg, 1983). The Smalltalk language, first developed in 1972 in the Learning Research Group at Xerox Palo Alto Research Center, was greatly influenced by

Simula-67 and also by Lisp. Smalltalk was the software half of an ambitious project known as the Dynabook which was intended to be a powerful personal computer. Research on Smalltalk has continued and the Smalltalk-80 language and environment are the product of that work.

From Smalltalk, some common concepts and ideas have been identified which have given support, at least informally, to the object-oriented paradigm which has now established itself. On account of the evolution and dissemination of programming languages like Smalltalk, this new paradigm has been evolved by several research groups, and new methodologies, languages and tools have been appearing. The object-oriented paradigm deals with its own concepts, terminology and notation for software development. These issues are discussed further, later in this chapter. The evolution from abstract data types and classes to the object-oriented paradigm is the focus of the next subsections.

2.3.1 Abstract Data Types

It is easiest to learn new ideas in terms of more familiar ones. Because the object-oriented paradigm has been strongly influenced by the notion of **abstract data types**, it is convenient to understand this influence, and the evolution from the concept of abstract data types to the object-oriented paradigm.

The notion of abstract data types is one of the most important ideas that has emerged from research in programming languages. The term abstract data types refers to a concept in which data structures, and related operations which manipulate those data structures, are encapsulated within a protected region. A language is said to support abstract data types when it allows designers to define new abstract data types consisting of declarations that bring together operations which manipulate private data structures.

According to Liskov and Zilles (1975), abstract data types can be used by designers to introduce new data types which are deemed useful in the application domain. The designer is concerned with the behaviour of those data types and what kind of information can be stored into them and obtained from them. Nevertheless, the designer is not concerned with how to implement them. Therefore, designing with abstract data types can be considered as a way of managing complexity since the designer can define and make use of abstract data types without concern for their internal implementation.

The object-oriented style has been influenced by the notion of abstract data types because an object can be viewed as an instance of an abstract data type, which encapsulates a data type and provides a defined set of operations to manipulate and access that data type. Actually, in most object-oriented programming languages, a class definition describes a data type and the operations which can be performed on that data type. Furthermore, the concept of abstract data types assumes an important role within an object-oriented approach because it may be seen as a way of providing abstract and simplified representation for a software system. In addition, abstract data types bring others benefits such as modularization and encapsulation that are also relevant to the object-oriented paradigm, as is discussed next.

Modularization

Modularization is inherent to the successful development of large software systems because it is used to break a large software system into small modules which can be combined to simulate the original application. The production of a large software system presents many challenging problems that do not arise when developed via smaller modules (DeRemer, 1976). Therefore, the same methods and techniques that work well with modules do not necessarily apply to large software systems. Restrictions that merely apply to the size of the modules, however, do not improve the quality of a software system in any real sense, and splitting a large module into a sequence of smaller modules does not necessarily make a software system any better.

Modularity can be achieved by a collection of abstract data types which the designer thinks belong together, and provided with an interface which specifies the data structures and operations which can be used outside a module. A more helpful notion of modularity refers to the factoring of large software systems into units that can be modified independently, which means that each module should be understood and, possibly, implemented independently of any other modules of the software system. Thus, each module should realise a single and a simple conceptual functionality of a software system.

Modularization relates to the idea of cohesion, which measures the degree of connectivity among modules. The most desirable form of cohesion is functional cohesion, in which modules are closely related and provide particular functionality. Sommerville (1989) argues that a high degree of cohesion is a feature of object-oriented software systems because classes making up a software system are naturally cohesive since they should encompass operations to achieve a particular purpose.

Encapsulation

Another important topic which relates to abstract data types is encapsulation, also known as information hiding. Encapsulation suggests that a data structure must be resident within a module. An interface provides the access to that data structure which is needed by other modules. Thus, communication among modules should be done through well-defined interfaces which prevent data structures inside a module to be directly accessed. Encapsulation minimises inter-dependencies among separately written modules by defining strict interfaces.

Parnas (1972) attempted to systematise the modularization process, based on the concept of encapsulation as a criterion for decomposing software systems into modules. Encapsulation as a design principle favours the production of highly independent modules because the state of a module is contained in its private data structures, visible only within the scope of the module. In fact, design decisions internal to a module are hidden and do not affect the cooperation between modules. Once the module interface has been carefully designed, modules can be developed independently of each other, stored in a library, combined later to build a unique software system, and reused in other software developments.

2.3.2 Evolution of Abstraction in Programming Languages

According to Ghezzi and Jazayeri (1982) abstraction helps cope with complexity. By using an abstraction of an application, one is able to concentrate only on the relevant qualities or properties of that application. What is relevant depends on the purpose for which the abstraction is being made. For example, someone learning to drive can represent a car by four properties: an accelerator, a brake, a clutch and a steering wheel. These are abstractions for some visible elements of a car. For a driver, chemical reactions going on inside an engine and the engine itself are irrelevant properties but they would not be for an abstraction made by a mechanic.

In the same way, programs may be seen as an abstraction of the computer central processing unit and memory locations. For example, in a program for calculating the area of a square, the designer can use self-explanatory names such as *area* and *side*, and perform the operation *side* multiplied by *side*, assigning the result to *area*. With this example in mind the designer can divide the abstraction of a program into two different aspects: the data and the control aspects. The data aspect models the operands manipulated by programs and the control aspect models the operations performed by programs.

At the beginning of programming language development, assembly languages only enabled designers to write programs based on machine instructions (operators) which manipulate the contents of memory locations (operands). Therefore, the level of data and control abstraction achieved was very low. A great step forward occurred when the first high level languages such as Fortran, Algol and Pascal appeared. The operators turned into statements and operands into variables and data structures. The traditional view of programs in these languages is that they are composed of a collection of variables which represent some data, and a set of procedures which manipulate these variables. Most traditional programming languages support this data-procedure paradigm. That is, active procedures operate upon passive data that is passed to them. Things happen in a software system by invoking a procedure and passing it some data to manipulate. Early high level programming languages have reasonable support for representing actions through statements and procedures; however, they are deficient in representing abstract data types.

Abstract data types are abstractions that may exist at a higher level than operands and operators, or variables and procedures separately. The starting point for creating a specification of an abstract data type is to identify the operations on that particular data type (Cardelli, 1985). For example, suppose the designer wants to manipulate a *pile* to model a first-in-last-out queue discipline. Two operations that are fundamental to manipulating a *pile* are *push* and *pop*. Basically, *push* adds an element to the top of the *pile* and *pop* gets an element from the top of the *pile*. Then the designer could gather together into an abstract data type, a *pile* data structure and the operations *push* and *pop* which are relevant to them. More advanced programming languages have depended on abstract data types to manage complexity (Shaw, 1984). Some languages such as Simula-67 provide a construct that allows both variables and procedures to be defined in a single unit called a *class*, which enhances the definition of abstract data types. Equivalent ideas can also be found in CLU (Liskov, 1977) through the concept of *cluster* and Ada (Buzzard, 1985) through the *package* construct.

The object-oriented paradigm goes a step further than abstract data types. If two abstract data types are similar but not identical, there is no means of expressing their similarities conveniently in a programming language which supports only abstract data types. However, object-oriented languages allow similarities and differences between abstract data types to be expressed through *inheritance*, which is a key defining feature of the object-oriented paradigm. Therefore, it would be better to characterise the evolution of object-oriented languages based on abstract data types and inheritance; in this case the immediate ancestor of object-oriented languages is Simula-67, which is an Algol-based language and also first introduced the concept of classes. Besides, because object-oriented concepts have also arisen from the artificial intelligence community, it is not surprising that Lisp has influenced a number of object-oriented languages as well. Some of these are Flavors (Moon, 1986), Loops (Stefik, 1986) and CLOS (DeMichiel, 1987), which have used concepts from Lisp and Smalltalk.

The prominence of the object-oriented paradigm has influenced the design of other languages. There has been some work to add object-oriented constructs to the popular C, Pascal and Modula-2 languages, resulting in the hybrid languages Objective-C (Cox, 1986), C++ (Stroustrup, 1986), Object Pascal (Tesler, 1985) and Modula-3 (Cardelli, 1989). The addition of object-oriented ideas into traditional languages has sophisticated them, in that, programmers have the flexibility to use or not use the object-oriented extensions and its benefits. Although these hybrid languages have become more complex, these extensions have given a handle to programmers who have considerable experience with those traditional languages, to explore incrementally the different concepts provided by the object-oriented paradigm. Nevertheless, when using a hybrid language, programmers must exercise more discipline than those using a pure object-oriented language because it is too easy to deviate from sound object-oriented principles. For example, C++ permits the use of global variables, which violates the fundamental principle of encapsulation.

As far as concurrency is concerned, objects can also be defined as concurrent agents which interact by message passing, and emphasise the role of entities such as actors and servers in the structure of the real-world application. The main idea behind object-oriented languages which support concurrency is to provide designers with powerful constructs which allow objects to run concurrently. Concurrency adds the idea of simultaneously executing objects, exploiting parallelism on a large scale. Languages with this purpose include Actor (Agha, 1986), ABCL/1 (Yonezawa, 1987), POOL-T (America, 1987), Orient/84 (Yutaka, 1986), ConcurrentSmalltalk (Yokote, 1987) and Mushroom/Must (Hopkins, 1987; Hopkins, 1989).

Other languages, such as Beta (Kristensen, 1985), Trellis/Owl (Schaffert, 1986) and Eiffel (Meyer, 1988), have also appeared (influenced basically by Simula-67, CLU and Smalltalk) and are believed to give good support for the object-oriented paradigm. Although Smalltalk, Trellis/Owl and Eiffel seem to be the most coherent object-oriented languages and provide an integrated programming environment, it is more likely that C++ will continue to be the object-oriented language most used in the near future because of the influence of UNIX, the portability and efficiency of C++, and the knowledge and popularity of the C language from which C++ has derived. However, C++ still requires a more robust program development environment to manage library of classes. Analysing the evolution of all those languages through time leads to the dependency graph shown in Figure 2.3.

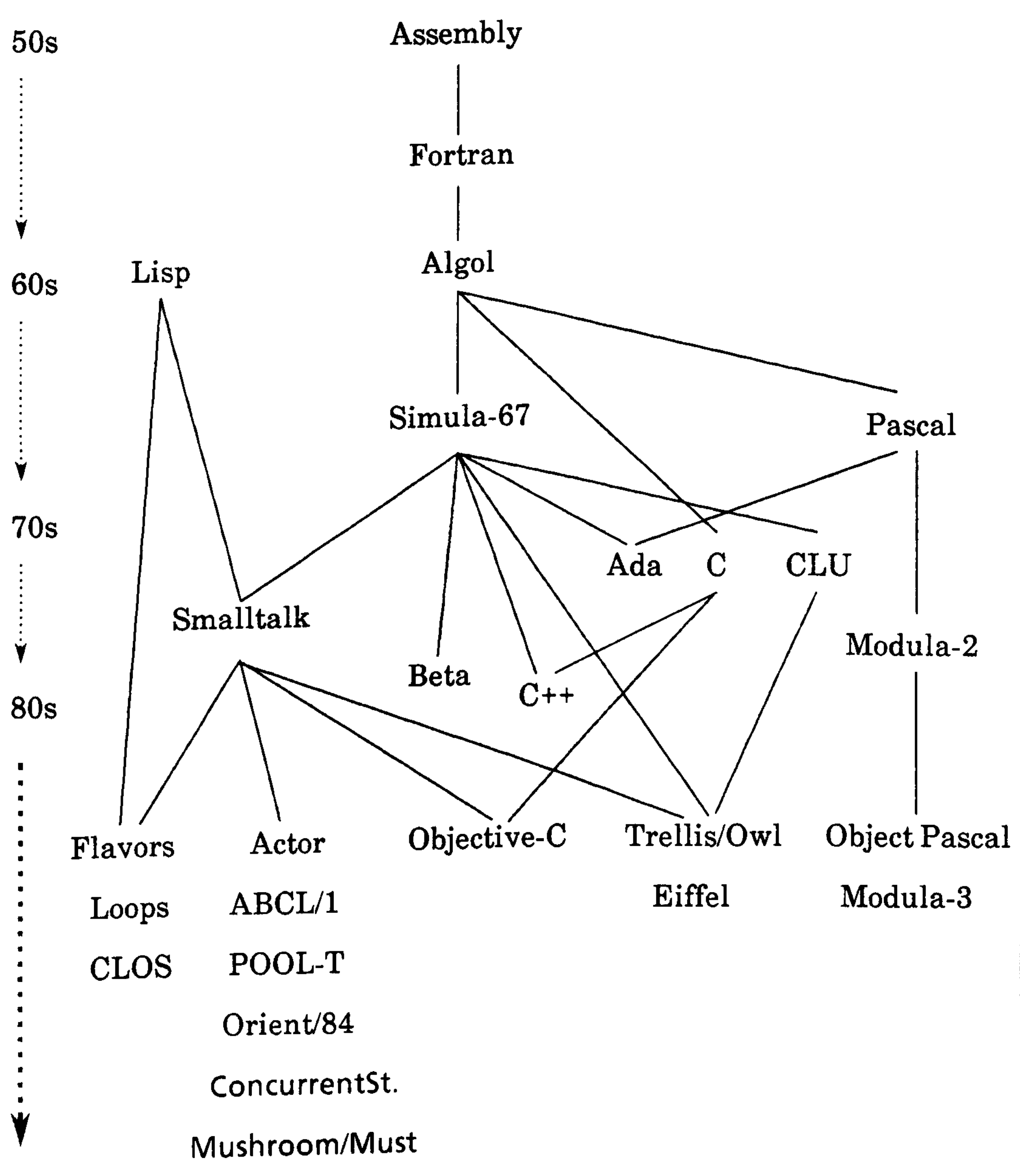


Figure 2.3 Language Evolution

2.3.3 Comparison between "Object-Oriented" Languages

Object-oriented design has naturally evolved from object-oriented programming. Thus, it is helpful to study some object-oriented languages to realise how object-oriented design has emerged. As discussed above, the object-oriented paradigm has been implemented in different languages, and over the last decade a number of object-oriented languages have appeared. In this subsection some so-called "object-oriented" languages are briefly reviewed and compared, and their main features are outlined.

Object-oriented programming has been defined in such a way that any language in which a particular unit has a state, and applicable operations associated with it, is said to be an object-oriented language. Nowadays, there is an attempt to explain object-oriented programming by examining the presence or lack of specific programming features, such as messages and inheritance. But, which features must a programming language have, to be considered an object-oriented language?

Wegner (1987) claims that a programming language is called *object-based* if it is based on objects. A language is called object-oriented if it provides linguistic support for objects and additionally requires that objects are instances of classes. Furthermore, an inheritance mechanism must be supported. Thus: object-oriented = objects + classes + inheritance. According to this classification, the set of object-based languages includes Ada and CLU because objects in Ada are realised by packages and objects in CLU are instances of clusters. The set of object-oriented languages is narrower than the set of object-based languages, and excludes languages like Ada and CLU but includes languages like Smalltalk and C++ because the latter two provide inheritance.

The characterisation of object-oriented languages to be presented here is an informal one which appears appropriate in the context of the object-oriented paradigm in general. It is claimed that an object-oriented language should support abstract data types, inheritance, dynamic binding and it is relevant that all items be objects. Dynamic binding provides great flexibility in manipulating objects at run-time and is one of the major reasons for the flexibility of object-oriented languages. In addition, support for library facilities, which is not part of the language definition, but useful for object-oriented software development is considered.

Clearly, some languages are better suited than others for supporting an object-oriented approach. For example, Smalltalk provides clear ways to work within an object-oriented approach but other languages, such as Ada, may be used in an object-oriented fashion through the use of some language tricks. The main point is how far a particular programming language can naturally embody and enforce the properties of classes, objects and inheritance.

Simula-67

Many of the ideas behind object-oriented languages have roots going back to Simula-67, which introduced the notion of class as a mechanism for encapsulating variables and procedures. In Simula-67, one class X can be a specialisation of another class Y. That is, X inherits all local variables and procedures of Y. In addition, X can add variables and procedures of its own. Simula-67 is a general purpose language which supports abstract data types, single inheritance and dynamic binding. It is a strongly-typed language but does not provide library facilities nor multiple inheritance.

CLU

CLU was motivated by a desire to support general mechanisms for the definition of abstract data types which make their representation completely encapsulated, in such a way that user-defined abstract data types are treated as similarly as possible to built-in types. An abstract data type in CLU is implemented by a language construct called the *cluster* which identifies a set of data structures and a set of operations for manipulating those data structures. In this aspect, CLU also made a serious effort toward an object-oriented approach.

Ada

Ada is a general purpose language which has been designed primarily to embody and enforce software engineering principles of abstraction, encapsulation and modularity. Ada is not really an object-oriented language. It has a construct, namely the *package*, which can be used to represent a class and helps to support an object-oriented approach. However, in Ada there are serious limitations on defining new abstract data types by specialising some existing ones. This weakness results in the impracticability of using inheritance. Ada is a strongly-typed language that performs all binding at compile-time, but there is a reasonable support for library facilities.

Smalltalk

Following the introduction of Simula-67 and CLU, a number of languages that support abstract data types have been introduced. However, in spite of earlier languages which also contain some object-oriented ideas, the term object-oriented itself is generally associated with Smalltalk. More than a programming language, Smalltalk is a complete programming environment composed of an object-oriented language kernel, a persistent programming system and a user-friendly interface (Goldberg, 1984). Although Lisp influenced Smalltalk and the notion of class came from Simula-67, many concepts were born with it, such as message-selectors and methods. The radical difference between Smalltalk and previous languages is that in Smalltalk everything is an object, from the primitive language types like integers and characters to user-defined types such as graphics and windows.

In Smalltalk, there is one basic unit, called an object, which contains instance variable declarations and method definitions. Every object is an instance of some class. Smalltalk also supports inheritance which permits a hierarchy of classes to be built. All instance variables and methods of an object are defined in the class of that object, or in its super-classes. The top level super-class is called *Object*. All classes are refinements of the *Object* super-class in that they add new or different methods or allow more variables in their instances. Classes themselves are objects and are instances of other classes, named the *meta-classes*.

Smalltalk objects interact by exchanging messages. In addition to message passing, different objects of a class can share instance variables, called *class variables*. Class variables are defined in a meta-class and are accessible to any method defined in that class. Smalltalk supports automatic garbage collection which means that the lifetime of all objects is determined not by the programmers but by the system. Smalltalk provides excellent information hiding, dynamic binding and an extensive library which encourages prototyping and reuse of existing classes.

Objective-C

Two separate efforts attempt to make some of the benefits of the object-oriented paradigm available to programmers trained in the C language. Objective-C and C++ are both hybrid languages, designed as extensions of the C language and contain the facilities offered by C, such as efficiency and portability. Although the aims of both languages are similar, the two languages differ significantly in that Objective-C adds Smalltalk constructs to C, whereas C++ shows a clear Simula-67 influence. In fact, Objective-C is a super-set of C, that is, it kept some C features and included some object-oriented basic concepts. As a result, Objective-C provides encapsulation and adds the notion of a class definition mechanism to C as well as inheritance and dynamic binding. With Objective-C the term *Software-IC* also was introduced to identify the possibility for reusable software components.

C++

As far as C++ is concerned, it is a general purpose language and a super-set of C. Objects in C++ are instances of some class. Variables and functions are referred to as the *members* of the class. A member can be *public*, *private* or *protected*. A public member can be accessed by both members and non-members of the class. A private member is accessible only to other members of the class and protected members are accessible to other members of the class and sub-classes. A *constructor* is a special operation used to create and initialise objects. On the other hand, another special operation called the *destructor* can be invoked to destroy an object. Inheritance can be used to implement *derived* classes which have the refined commonalities of some other *base* classes.

Assuming that the learning curve from C to C++ is not very significant, the real hurdle of using C++ has to do more with programming style than with language syntax. For those who know C, the real advantage of C++, is that new object-oriented programming features can be learned incrementally (or corrupted gradually!). This is advantageous for some and disadvantageous for others who claim that there is a danger of "cheating" and reverting to procedural programming using an object-oriented language.

CLOS

In the early eighties a number of programming languages merged concepts of Lisp and Smalltalk. Flavors and Loops were derived from Lisp with object-oriented concepts built on top, and follow the same concepts adopted by Smalltalk; both provide multiple inheritance, but do not provide library facilities. Through the experience derived from both these languages, another object-oriented language called CLOS has appeared. CLOS is also a Lisp-based language based on the concepts of class, meta-class, method and multiple inheritance. A common criticism that can be made for these languages is that, due to the Lisp influence, all of them provide features useful to deal basically with artificial intelligence applications.

Object Pascal

Object Pascal is an object-oriented extension of Pascal developed by Apple Computer for the Macintosh personal computer. Object Pascal implements classes as an extension of the Pascal *record* structures. In this language a module is called a unit. A unit is divided into an interface component and an implementation component, which can be compiled separately. The greatest strength of Object Pascal is its simplicity, which was an attempt to streamline the object-oriented language learning curve.

Beta

Beta is a language in the Simula-67 tradition. A program in Beta is regarded as a simulation of an application modelled by interacting objects. Beta replaces *class* and *type* notions by another very general abstraction concept called the *pattern* which also may be organised in a classification hierarchy by means of *sub-patterns*. However, Beta does not support multiple inheritance. An object in Beta is described by a set of attributes, which portrays the properties of the object, and a sequence of actions. Finally, patterns may be checked at compile time, even though the binding is done at run-time.

Eiffel

Eiffel is a true object-oriented language and shares the basic properties of the languages shown above by offering single and multiple inheritance, generic classes and dynamic binding. Furthermore, Eiffel combines the object-oriented paradigm with expressions of formal program properties such as assertions and invariants. The language also comes with an environment geared towards the development of sizable software systems in a production environment.

Trellis/Owl

Trellis/Owl consists of an object-oriented programming environment called Trellis, with a programming language called Owl. Trellis consists of several user-friendly tools which support editing, compiling and debugging. Owl is a general purpose object-oriented language which is strongly-typed, therefore, it is necessary to specify the type of all references to all objects. Owl enforces abstraction, provides multiple inheritance and an extensive library. Like Smalltalk and Eiffel, Trellis/Owl provides a good support for the object-oriented paradigm.

Table 2.1 shows a comparison among the so-called "object-oriented" programming languages discussed.

Features / Languages ↓ →	Abs. Dt. Ty.	Inherit.	Mult. Inher.	Dyna. Bind.	All are Object	Library Support
Simula-67	y	y	n	y	n	n
CLU	y	n	n	y	n	n
Ada	y	n	n	n	n	y
Smalltalk	y	y	y	y	y	y
Objective-C	y	y	n	y	n	y
C++	y	y	y	y	n	n
Flavors, Loops, CLOS	y	y	y	y	y	n
Object Pascal	y	y	n	y	n	n
Beta	y	y	n	y	n	n
Eiffel	y	y	y	y	y	y
Trellis/Owl	y	y	y	y	y	y

Table 2.1 Comparison among "Object-Oriented" Languages

('y' means yes, the feature is present;
'n' means no, the feature is not present)

Some of the current languages support the object-oriented paradigm better than others, however, the perspective on the paradigm is as important as the language statements. It is possible to think in object-oriented terms without a language that supports the paradigm. But, the main point is not to force languages to deal with concepts which are not naturally supported. Although it is possible to conform to an object-oriented approach using standard Pascal (Jacky, 1987), it is not as suitable for the object-oriented paradigm as Smalltalk, Eiffel, Trellis/Owl or even C++ are.

Stroustrup (1987) states that a language is said to support a paradigm of programming if it provides facilities which make it easy, safe and efficient to use that paradigm. On the other hand, a language does not support a paradigm if it takes exceptional effort or exceptional skill to follow such a paradigm. This means that support for a paradigm must come not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile-time and run-time checks against unintentional deviation from the paradigm.

It can also be concluded that, despite the possibility of following an object-oriented approach using these languages (shown in Table 2.1) with less or more difficulty, direct language support is beneficial in facilitating and encouraging the use of the object-oriented paradigm, such as in Smalltalk, Eiffel or Trellis/Owl. Not only do these languages support the object-oriented paradigm, but they also enforce it because the main concepts dealt with are classes and objects. The danger in trying to force object-oriented concepts into a language, such as in Ada, which does not enforce object-oriented concepts is that inconsistent constructions may be produced, impairing software development and jeopardising the quality of the resulting software systems.

2.4 CHARACTERISATION OF AN OBJECT-ORIENTED MODEL

Although object-oriented programming has its roots in the 1960s, there are many definitions about precisely what the term *object-oriented* means, but at the moment, none generally accepted. The term means different things to different people because it has become very fashionable to describe any software system in terms of object-oriented concepts.

To some, the concept of *object* is merely a new name for abstract data types; each *object* has its own private variables and local procedures, resulting in modularity and encapsulation. To others, classes and objects are a concrete form of type theory. In this view, each object is considered to be an element of a class which itself can be related through sub-type and super-type relationships. To others still, object-oriented software systems are a way of organising and sharing code in large software systems. Individual procedures and the data they manipulate are organised into a tree structure. Objects at any level of this tree structure inherit behaviour of higher level objects; such inheritance is the main structuring mechanism which makes it possible for similar objects to share a program code.

Despite many authors being concerned with providing precise definitions for the object-oriented paradigm, it is difficult to come up with a single definition. The object-oriented paradigm is not something that can be simply defined. Therefore, it would be fairer to characterise an object-oriented approach for software development, as will be seen later in this section.

Rentsch (1982) defines object-oriented programming in terms of inheritance, encapsulation, methods and messages, as found in Smalltalk. Objects are uniform in that all items are objects and no object properties are visible to an outside observer. All objects communicate using the same mechanism of message passing, and processing activity takes place inside objects. Inheritance allows classifying, sub-classifying and super-classifying of objects, which permits their properties to be shared.

Pascoe (1986) also presents object-oriented concepts, terminology and characteristics from the Smalltalk perspective. Pascoe defines an object-oriented approach in terms of encapsulation, data abstraction, methods, messages, inheritance, and dynamic binding for object-oriented languages. It can be seen that these features are slightly different from those identified by Rentsch because the former does not consider the importance of dynamic binding. Pascoe also affirms that some languages that have one or two of these features have been improperly called object-oriented languages. For instance, Ada could not be considered an object-oriented language because Ada does not provide inheritance.

On one hand, according to Pascoe and Rentsch, messages and inheritance are fundamentals in object-oriented programming, so any language which does not support messages and inheritance can not be called an object-oriented language. Other authors, such as Robson (1981), Stefik and Bobrow (1986) and Thomas (1989) also emphasise the idea of message passing between objects and dynamic binding as fundamental to object-oriented programming. In fact, all of these authors have been influenced by Smalltalk, where the message passing mechanism plays a fundamental role as the way of communication among objects. However, message passing is just an implementation technique and not at all an inherent part of the object-oriented paradigm (e.g. Trellis/Owl and Eiffel use procedure calls). On the other hand, Stroustrup (1987) claims that object-oriented programming can be seen as programming using inheritance. Thus, object-oriented programming would be: decide which classes designers want; provide a full set of operations for each class and make commonality explicit by using inheritance.

Nygaard (1986) discusses object-oriented programming in terms of the concept of objects in Simula-67. In that language an execution of a computer program is organised as the joint execution of a collection of objects. The collection as a whole simulates a real-world application, and objects sharing common properties are said to constitute a class. Madsen and Moller-Pedersen (1988), like Nygaard, regard object-oriented programming as a model which simulates the behaviour of either a real or imaginary part of the real-world. The model consists of objects defined by attributes and actions, and the objects simulate phenomena. Any transformation of a phenomenon is reflected by actions on the attributes. The state of an object is expressed by its attributes and the state of the whole model is the state of the objects in that model.

Nevertheless, there is more to object-oriented programming than message passing and simulation. The object-oriented paradigm is still lacking a well-known and profound theoretical understanding, but some research has been appearing in this area. For instance, Wolczko (1988) attempts to define object-oriented programming, by defining what an object-oriented language is. Wolczko goes on to describe the essential features of the object-oriented paradigm, such as objects, classes, dynamic binding and inheritance with formal methods using denotational semantics based on VDM (Jones, 1986).

Lastly, Wegner (1987) defines an object-oriented approach in terms of objects, classes, inheritance and abstract data types. Objects are autonomous entities that respond to messages or operations and have a state; classes classify objects by their common operations; inheritance serves to classify classes by their shared behaviour; and abstract data types hide the representation of data and the implementation of operations. The characterisation of an object-oriented approach by Wegner is closest to the one to be presented in this thesis.

As it has been shown, there are many different interpretations of the object-oriented paradigm. Nevertheless, one thing that all definitions have in common, not surprisingly, is the recognition that *object* is the primitive concept for the object-oriented paradigm. Therefore, it is better to characterise what the term *object* means before starting to use it. Curiously though, it is notoriously difficult to capture precisely what is meant by an object. In fact, there are two aspects with which the object-oriented paradigm deals: the first is an object-oriented model composed of objects, classes and inheritance mechanism (discussed in the next subsections) and the second is the philosophy of object-oriented design (discussed in the next section).

The object-oriented model comprises a collection of principles which forms the foundation of the object-oriented paradigm. The next subsections cover the concepts, features and mechanisms which are common to the object-oriented paradigm and set the terminology to be used in the remainder of this thesis.

2.4.1 Objects

An **object** is an encapsulation of some state together with a defined set of operations on that state. An object embodies an abstraction characterised by an entity in the real-world. Hence, it exists in time, it may have a changeable state and can be created and destroyed. An object has an identity (which is a distinguishing characteristic of an object) that denotes a separate existence from other objects. The object's behaviour characterises how an object acts and reacts in terms of changes in its state. In fact, each object could be viewed as a computer endowed with a memory and a central processing unit, and able to provide a service.

2.4.2 Classes

A **class** (or *type*) is a template description which specifies properties and behaviour for a set of similar objects. Every object is an instance of only one class. A class may have no instances (usually termed an *abstract class*). Every class has a name and a body that defines the set of attributes and operations possessed by its instances. Note that the term *object* is sometimes used to refer to both class and instance (especially with languages like Smalltalk where a class is itself an object). However, it is important to distinguish between an object and its class; here the term class is used to identify a group of objects and the term object to mean an instance of a class.

Attributes and **operations** are usually part of the definition of classes. Attributes are named properties of an object and hold abstract states of each object. Operations characterise the behaviour of an object, which is expressible in terms of the operations meaningful to that object. The operations are the only means for accessing, manipulating and modifying the attributes of an object. An object communicates with another through a request, which identifies the operation to be performed on the second object. The object responds to a request by possibly changing its attributes or by returning a result. The *interface* comprises of the set of operations which can be requested by other objects; the external view of an object is nothing more than its interface. Figure 2.4 represents a real-world entity called dictionary mapped in terms of object-oriented concepts.

By using the concepts involving classes and objects as stated above some important characteristics, such as abstraction, encapsulation and modularity are achieved. These characteristics are recognised as being features of good quality software, therefore the object-oriented paradigm, in theory, encourages high quality software development.

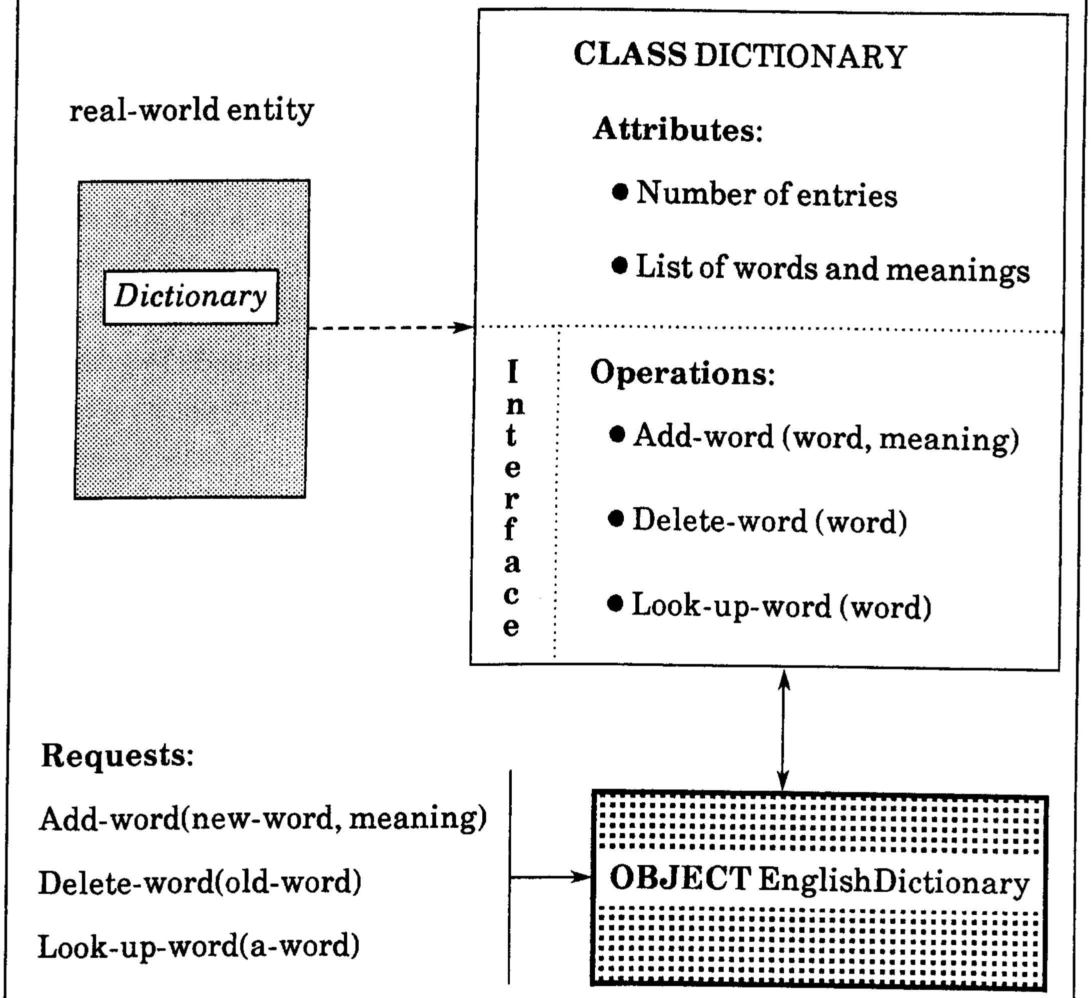


Figure 2.4 Concepts Related to the Object-Oriented Paradigm

2.4.3 Inheritance

The **Inheritance** mechanism can be used to represent a relationship between classes. It is a mechanism for sharing commonalities (in terms of attributes and operations) between classes. Every inheritance relationship has parents called the *super-classes* and children called the *sub-classes*, and attributes and operations inherited. When a sub-class inherits commonalities from one super-class, this is called *single inheritance.* When a sub-class inherits commonalities from two or more super-classes, this is called *multiple inheritance.* Thus, single inheritance is a particular case of multiple inheritance.

For example, quadrangles and triangles are special kinds of polygons. In the same way, squares and rectangles are special kinds of quadrangles. These relationships are easily captured by inheritance, as illustrated as a hierarchy of classes in Figure 2.5. When the *quadrangle* class inherits from the *polygon* class, *quadrangle* class is referred to as sub-class and *polygon* class as super-class. At the highest level, all *polygon* objects may have the *number-of-sides* attribute, and are able to be *drawn* and *scaled*, which can be seen as operations on *polygon* objects. These attributes and operations may be defined in the root *polygon* class and inherited as they are, or even be modified in the *quadrangle* class. The *quadrangle* class might also define the *rotate* operation for itself. In this case, the *quadrangle* class has two parts, an inherited part and an incremental part. The inherited part is derived from the *polygon* class and the incremental part is the new part defined in the *quadrangle* class.

Problems with Inheritance

Inheritance is a powerful concept but gives rise to some complex issues including overriding of inherited commonalities. The evaluation of inheritance also reveals that it may violate encapsulation because by using inheritance a sub-class might access or refer to an attribute in its super-class and an operation in a sub-class can call a private operation of its super-class.

Furthermore, inheritance could also be misused in the situation where a super-class is created as an aggregation of sub-classes, not as a

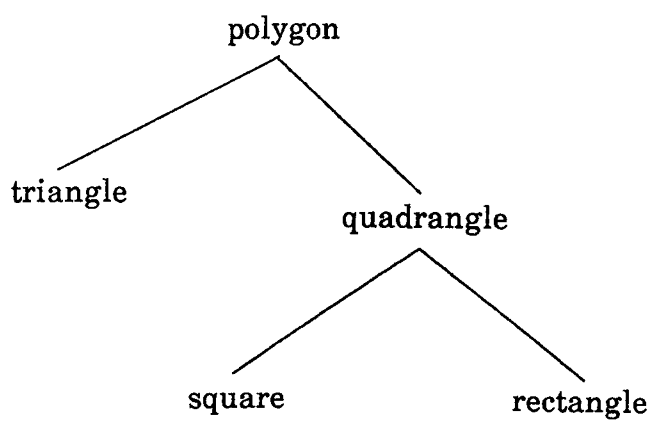


Figure 2.5 Hierarchy Representing Inheritance

specialisation. For instance, a *course* class might be created as an aggregation of a *student* class and a *lecturer* class, but in the real-world, students and lecturers are not specialisations of courses, and therefore they should not be sub-classes of a *course* class.

The use of inheritance is sometimes acceptable but not recommendable when it does not reflect concepts in the real-world. For instance, an ellipse defined by two focii and one radius is not a specialisation of a circle defined by one focus and one radius. Actually, a circle is conceptually a specialisation of an ellipse because a circle has two coincident focii.

In the context of this thesis, inheritance is defined as a mechanism for a hierarchical classification of attributes and operations at design level, and resource sharing at implementation level based on class hierarchies. Since commonalities can be shared by means of inheritance, a library of reusable components is typical during an object-oriented software development as can be seen in Chapter Five.

2.5 PHILOSOPHY OF OBJECT-ORIENTED DESIGN

Object-oriented design is an approach to software development in which the design of a software system is based on the creation of a collection of classes which map entities of a real-world application into a software solution in a solution domain. The real-world entities are subsequently simulated in a computer by corresponding software objects which mirror the behaviour of their real-world counterparts. The design process, in itself, is independent of any particular programming language, and similarly, a good design methodology should be independent of any programming language.

Some designers are used to thinking in terms of functional decomposition, emphasising functions and processes, rather than in terms of classes and objects which characterise object-oriented design. As a result, when those designers first try to use the object-oriented paradigm, they map the functions they would have created directly onto objects. They also have other difficulties such as mapping system behaviour with wrong objects, or creating class hierarchies which poorly correspond to the real-world applications. The problem is that such designers are not skilled in how to apply the object-oriented paradigm.

Other designers suggest that a good way to find classes for an object-oriented software system is to start from a natural language description of the software requirements and to underline nouns which will represent objects. For example, a sentence from a requirements description of the form "the radar must track the position and speed of an incoming aeroplane" would lead an object-oriented designer to detect the need for two objects, *radar* and *aeroplane*.

According to Meyer (1988), this is only a simple minded technique, and it can only give rough first results. For example, it is not clear whether two of the nouns in the sentence, *position* and *speed* should also be identified as objects rather than attributes of an *aeroplane* object. A better approach is to use the idea behind abstract data types: a concept should only be made into a class if it describes a group of objects marked by interesting operations with meaningful properties. So, should the *position* of airplanes yield a class? If there are no specific operations on *position*, then it should be an attribute of a class *aeroplane*. Alternatively, if *position* is a meaningful entity with associated operations (for instance, distance to another point, measurement error or conversion to another coordinate system) then it is worthwhile defining a *position* class.

Despite the importance of creating a model of an application during software design, current object-oriented design methodologies have not highlighted how to construct a design model based only on classes, objects and inheritance. Designers using these methodologies are often faced with the following difficulties:

- where do classes and objects come from?
- what concept in the application is to be a class of objects, and what is not to be?
- what is the best way to decompose a software system into a set of classes and objects?
- which operation realises a required functionality?
- which is the best set of operations to perform a particular functionality?

These were the central questions posed at the Workshop on Specification and Design of Objects (Power, 1988). Unfortunately, the consensus among the participants was that it is difficult to find the right classes and objects. Nevertheless, a design methodology might help to answer these questions because it provides rules and guidelines which aim to build a design model composed of classes and objects, and therefore this would systematise the whole design process.

Building a design model is an activity of fundamental importance because the efficiency of the design process and the quality of the final software system greatly depend on the clarity, completeness and consistency of that design model. The representation of a model may consist of organisational and graphical notations suitable for describing the application diagramatically. Certain diagrams are believed to be useful for describing the design model. For instance, class diagrams might show the hierarchies among the classes, and object diagrams could show communications between objects.

Therefore, the best way to characterise the philosophy behind object-oriented design is to centralise the design process on the concepts of classes and objects. That is, similar objects in an application domain are identified and abstracted as classes in an solution domain; properties of the objects are abstracted as attributes and operations, and communications between objects are abstracted as requests. Relationships between classes define class hierarchies; classes with common properties may be generalised into a super-class; classes may also be specialised into sub-classes. This process can be repeated and, as a result, several class hierarchies might emerge.

The philosophy of object-oriented design is inspired by three different ways in which designers can structure their knowledge about an application:

1) Classification and instantiation.

2) Generalisation and specialisation.

3) Decomposition and composition.

Classification and **instantiation** help organise objects into classes because a class is a set of objects which share common properties; classification of objects leads to classes, whereas instantiation of classes leads to objects. Typically, classification starts by identifying properties shared by more than one object in a given design. Identification of common properties depends on the intended use of the objects. The objects which share the same properties are grouped together into classes and regrouping may occur several times until the organisation of a set of classes becomes stable.

Generalisation and **specialisation** introduce the concepts of super-classes and sub-classes. These closely related concepts provide much of the power of the object-oriented paradigm. Generalisation supports the exploitation of commonalities between classes. When two or more classes represent overlapping sets of attributes and operations, the commonalities may be factored out of both classes and used to create a new super-class with those previously overlapping commonalities. Specialisation guides designers in the reuse of existing abstractions by defining new sub-classes which are more specific than existing classes. The sequence of creating class hierarchies may vary. A super-class may be created first and then the sub-classes, or vice versa.

Decomposition and **composition** are two important techniques which can be used during the design process. Decomposition divides a large component into smaller and simpler components which may then be refined independently. Using the object-oriented paradigm, decomposition is based on classes and objects. In contrast, small and simple components can be aggregated and evolve incrementally into larger components through composition.

During object-oriented design a hierarchy of classes may be built from general to particular, from particular to general or through a mixture of both approaches. The first approach starts from a general class and through specialisation reaches sub-classes, and eventually objects. The second approach starts with objects or sub-classes, from which properties are generalised into super-classes. Classification, instantiation, generalisation and specialisation may be used to structure and rearrange existing classes into new ones or into class hierarchies. In fact, a combination of both approaches is more likely to be used, depending on what designers know best; whether classes as a whole or some objects. These issues are further investigated in Chapter Five.

2.5.1 Domain Analysis

Domain analysis involves the investigation of a specific application domain and seeks to identify and classify the components which commonly occur in software systems within that application domain in order to formulate concepts about that application domain (Prieto-Diaz, 1988). Thus, domain analysis is an activity which can be carried out at the beginning of software development.

The idea of domain analysis is to attempt to identify and classify components and relationships which are perceived to be important within an application domain. Such identification and classification of components may arise from the vocabulary used in that application domain, usually in the form of key words and semantic relationships between the components. The identification and classification of components help to establish important relationships between them, which can be connected and organised according to their semantic meaning in that application domain.

Domain analysis and the object-oriented paradigm are closely related. The domain analysis process can be employed to produce an initial set of classes which are reflections of the main conceptual entities within an application domain. Essential features of that application domain can be captured and initial candidates for classes can be identified. Consider for example the application domain of airline reservation systems. Typical components of these systems are: seats, flights, crews and passengers; and relationships might be: schedule a flight, reserve a seat to a passenger, assign a crew to a flight, and so on. During object-oriented software development, designers also have to identify entities in the real-world application which will become classes and objects in the software system. The identification of classes and objects can also be pertinent to a domain analysis process.

Relatively little work has been done in the area of object-oriented domain analysis. Perhaps the most significant work is being carried out within the Ithaca environment (Tsichritzis, 1989), where a range of object-oriented tools for early software development has been created. Ithaca aims at building an environment to support the development of object-oriented software systems in a variety of application domains. The environment includes:

- An object-oriented language with database support.
- A software information base which stores and manages information concerning reusable software and its intended use.
- A selection tool for browsing and querying the software information base.
- A variety of application development tools built around the software information base.

The main idea behind Ithaca is to provide an environment which will support software engineers in developing and maintaining software systems in a number of different selected application domains. Instead of focusing on the individual application, the goal is to produce workbenches containing software components and generic application frameworks which characterise the software systems of a particular application domain.

Gossin and Anderson (1990) propose a method for domain analysis which produces a collection of reusable components specific to an application domain. They claim that the components which result from domain analysis are better suited for reuse because they capture the essential functionality required in that application domain. Thus, designers find these components easier to include in new software systems in that application domain. The key issue is a careful domain analysis in order to identify the basic components within that application domain and, if possible, reuse them from a library of reusable components.

2.5.2 Reuse of Software

Reusability is the practice of incorporating existing software components into software systems for which they were not originally intended. Reusability is an important area in software engineering and holds the promise of improving software quality and reducing software development costs and time; as a result, reusability can bring about great improvements in productivity. It is likely to be more cost effective spending some time searching for a reusable component rather than defining, implementing and testing a new component.

In the past, the idea of reusability was linked with source code reuse or invoking subroutines from a library. Therefore, software reuse was usually performed at the implementation phase (Freeman, 1984). However, reuse of source code during the implementation phase is a very limited kind of reusability. Moreover, it is too late to consider reusability only at the implementation phase. Greater benefits are obtained when reusability occurs at more conceptual levels. From the beginning of the software life cycle, designers should be aware of the potentially reusable software components because reusability at design time will certainly influence the implementation phase.

Some researchers have been investigating means to reuse parts of a software specification and design (Lanergan, 1984; Horowitz, 1984; Neighbours, 1984). At a higher level than implementation, reusability involves a classification of software components which gives the information on what each component does, and accessibility which allows a component to be searched for, retrieved and hence reused (Bifferstaff, 1987).

There are many reasons for disappointments regarding design with reusability. Most of the difficulties have centred around the problems of classifying the components, searching for potentially reusable components and accessing libraries of reusable components. Furthermore, reusability is inhibited by a high initial amount of time required to explore libraries of reusable components, which may also involve accessing components, and adapting them if necessary. Components may have a lot of characteristics which need to be understood, and this is, of course, time consuming. Another relevant factor that hinders reusability is that many software development environments do not have an automated support for libraries of reusable components, and those which have, suffer from a steep learning curve because the components were not explicitly designed for reuse.

In addition, there are problems involving the design of components for reusability. For instance, there is a conflict between the need to develop components on schedule for use in a specific software system or to take additional time to make them generic for possible reuse in future software developments. Therefore, reusable components need more development effort because more time is spent to make them generic and robust.

Reusability throughout the entire software life cycle is an idea that has appealed to software engineers for a long time. Unfortunately, the infamous not-invented-here argument is particularly apparent when dealing with reuse of software. Additionally, Tracz (1988) exposes some myths about software reuse and states that:

- Software reuse is a technical and non-technical problem involving psychological and economic barriers.

- Available database technology can be applied to store and retrieve reusable components.

- Reuse at most doubles productivity during software development.

- Domain analysis can play a role in solving the reuse problem.
- Designing software from reusable parts is not like designing hardware using available integrated circuits.
- Reusing software that was not planned for reuse is harder than reusing software that was designed for reuse.
- Software reuse will not just happen.

This thesis is only concerned with the mechanisms used to put reusability into the context of the methodology for object-oriented design which has been developed (see Chapters Four and Five); it does not consider how software components are stored in a reusable library nor how they are retrieved. It is simply assumed that software components are available from a reusable library. Some solutions to the problem of how object-oriented software components are managed within a reusable library can be found in Tarumi *et al.* (1988), Oosthuizen *et al.* (1990), Embley and Woodfield (1987) using expert systems, and Sixtensson and Wenchuan (1990) for telecommunication systems.

Reusability during Object-Oriented Design

The ability to support software reuse is an important aspect of the object-oriented paradigm. This paradigm encourages reusability rather than reinvention because it offers some advantages such as:

- Classes and objects are good abstractions of concepts present in real-world applications.
- Classes and objects support modularity and encapsulation.
- Reusable classes can be easily stored in and retrieved from libraries.
- Classes can be specialised by sub-classifying, and both attributes and operations can be reused in sub-classes.
- Classes can be organised into frameworks to serve as templates to a particular application domain.

Micallef (1988) suggests that when an existing class is not exactly what is required for a new software system, designers should customise that existing class in some way to fit its new purposes. There are three ways of performing such customisation:

a) modify the original class definition;

b) make a copy of the original class definition and modify the copy;

c) modify the original by augmentation.

The problem with the first approach is that the original class becomes more complicated as it is tailored for use in several applications. The modification usually consists of a *case* statement which executes a different code depending on which application is currently using that class. This creates classes which are difficult to understand and to extend, and this really is against the object-oriented paradigm. Modifying a copy of a class has an updating problem, because replicated changes are not usually made automatically to all copies. The third alternative is achieved with inheritance where a new abstraction is defined by specifying in a new sub-class the difference between that new class and a preexisting super-class, and appending that new class to the old one by making the former a specialisation of the latter.

A good example of reusability can be found within the Smalltalk system community who has no aversion to reusing the system components. Users of the Smalltalk system often spend as much time browsing the system classes to see whether there are classes which can be reused as they spend writing new classes. Software reuse in Smalltalk is prevalent because the Smalltalk language and environment are so special that it is easier to modify and reuse existing classes than to create new ones.

Nevertheless, reusability is not straightforward in practice. Raj and Levy (1989) observe that in order to make reusability a reality under an object-oriented approach, designers must create new reusable classes and easily find potentially reusable ones, already developed. Creating new reusable classes is a difficult task because the class correctness becomes more critical since errors are replicated whenever and wherever a carelessly designed class is reused.

Finding existing classes means organising them in such a way that they can be rapidly found when needed by designers. One real disadvantage of reusing classes from reusable libraries is the time that it takes to master large, and not always well organised, libraries of reusable classes. Traditionally, ways to overcome these difficulties have included attempts to provide written documentation for classes and to develop browsers which facilitate the selection and reuse of a class. Another major problem in designing a large reusable library is organising the complex semantic relationships that exist between classes in an application domain. Therefore, there will be a slow learning curve due to the inherent difficulties in understanding and relating the classes in a reusable library.

These problems often turn reusability into a superficial and haphazard process, and the usefulness of a reusable component depends more on the similarity of two applications, luck, and foresight of designers rather than on engineering purpose. To achieve anything better than this *ad hoc* process, semantic relationships between the reusable components need to be identified and kept in reusable libraries. A set of fixed relations which can be derived from the methodology for object-oriented design proposed in this thesis, and which can express links between software components is proposed in Chapter Five.

Johnson and Foote (1988) emphasise that the object-oriented paradigm is not a panacea for reusability. They argue that software reusability does not happen by chance and designers must plan to reuse old classes and new classes must be designed for reusability. Tools which facilitate the selection of potentially reusable components and methodologies which enforce reusability are the keys to successful reuse of software components. Reusability should be enforced as part of a methodology which gives support for it through pragmatic steps which help identify reusable components. As will also be presented in Chapter Five, by using the object-oriented methodology proposed in this thesis, reusability is taken into account during the design of a software system.

2.5.3 Software Life Cycle Models

In this subsection, some important issues concerning software life cycle models are discussed. It is appropriate to examine different software life cycle models in general and to point out their strengths and weaknesses before a new one is proposed. This discussion provides background understanding the object-oriented software life cycle model proposed in Chapter Five.

Software life cycle models for software engineering have long been used by the software development community. The primary utility of software life cycle models is to determine the order of the phases or stages involved in software development and evolution. Software life cycle models are also important because they provide guidance for the order in which the major tasks to construct a software system should be carried out.

The Waterfall Software Life Cycle Model

The classic description of the software life cycle is based on a model commonly referred to as the iterative **waterfall model** (Royce, 1987), which has become the most prevalent software life cycle model. This model initially attempts to identify phases within software development as a linear series of actions, each of which must be completed before the next is commenced. Although there are a variety of different names for each of the phases, they are basically: requirements, specification, design, implementation and maintenance.

At a gross scale, three phases of the waterfall model are generally agreed upon: specification, design and implementation. Often the requirements and specification phases are called analysis, and therefore the analysis phase covers the time from the initiation of the software system, through the user needs and feasibility study, to the high level specification for the software system. Design can be divided into early design and detailed design; following from the design, implementation is carried out. During the maintenance phase, software engineers are asked to add new functionality, fix faults or modify some existing behaviour.

The waterfall model is marked by the apparently neat, concise and logical ordering of the series of obvious phases which must be followed in order to obtain a software system. Such a model assumes that the specification phase should be completed and verified before design could begin, and that the design phase should be completed and verified before implementation could begin, and so on. Therefore, the waterfall model supposes that designers complete an entire step, before going to the next.

Further refinements to this software life cycle model consider that completion is seldom absolute and that iteration back to a previous stage is likely to happen. In an iterative model, if there is sufficient reason to do so, designers may return to a previously completed step, introduce a change, and then propagate the effects of that change forward in the software life cycle, as represented in Figure 2.6.

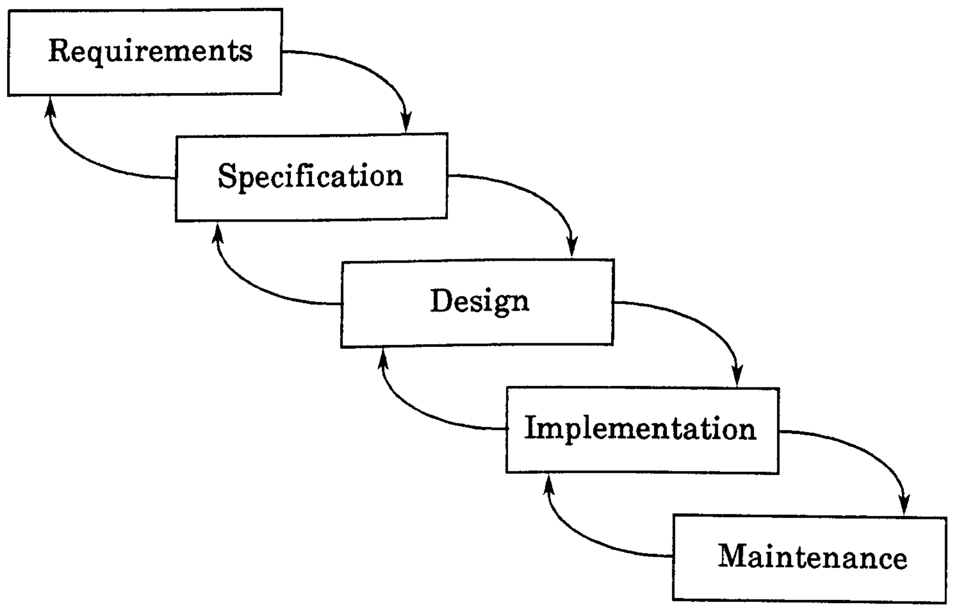


Figure 2.6 The Classic Waterfall Software Life Cycle Model

The waterfall model, as described above, is frequently based on a view of the real-world application interpreted in terms of a functional decomposition. The idea is simple enough: select a piece of the application (initially the whole application) and determine its main parts, generally based on the required functionality for the software system. Repeat the previous steps on each of the subparts until the functionality is well detailed. Functional decomposition is typically a top-down process and tools used to support this approach, which is usually based on functions and data flow, include data flow diagram editors, data dictionary generators and structure chart editors.

It is possible to describe the functional decomposition approach as fundamentally top-down. A top-down approach has the following characteristics:

- it progresses from the general to the specific;
- it decomposes software systems into layers and in each layer there is a uniform level of abstraction;
- components at higher level of abstraction treat components at lower level of abstraction as black boxes;
- components at low level of abstraction are unaware of components at higher levels of abstraction.

The top-down approach imposes some discipline during software development, but it has been criticised as not being totally appropriate to support contemporary software development paradigms, such as prototyping and object-oriented. However, despite the frequent criticism of the waterfall model, no satisfactory replacement has gained widespread acceptance. The main flaws in the waterfall model can be summarised as follows:

- it takes no account of evolutionary development and prototyping;
- it often characterises a software system as a single and large high level function;
- it is based on a functional decomposition approach, and the data aspects are often neglected;
- it does not encourage reusability within its phases;
- it does not address the concern of developing similar software systems;

- it does not consider the previous knowledge that designers may have about the application domain;
- it assumes a relatively uniform progression through software development;
- it attempts to separate software development into distinct phases, though it is quite common to carry out some of them in parallel.

The successive stages used in the waterfall model have helped eliminate many of the difficulties previously encountered during software development and it has gained great acceptability. But even with the extensive revisions and refinements of the waterfall model, its basic scheme has encountered significant difficulties, and these have led to the formulation of substitute software life cycle models.

A Spiral Software Life Cycle Model

An alternative software life cycle model, named a **spiral model** (Boehm, 1988), has been proposed mainly in order to speed up software development. A spiral model, as simply depicted in Figure 2.7, makes software development more flexible but it is strongly linked with prototyping. Prototyping is the process of building software system models which exhibit some of the behaviour of the final software system. Prototyping provides constructive feedback to the potential users and designers so that requirements can be refined and clarified early during software development.

A typical spiral model usually observes the following stages:

1) identify the basic requirements and objectives of the software system;
2) study alternatives to implement a software system which meets these requirements;
3) select one alternative which satisfies partial requirements of the software system;

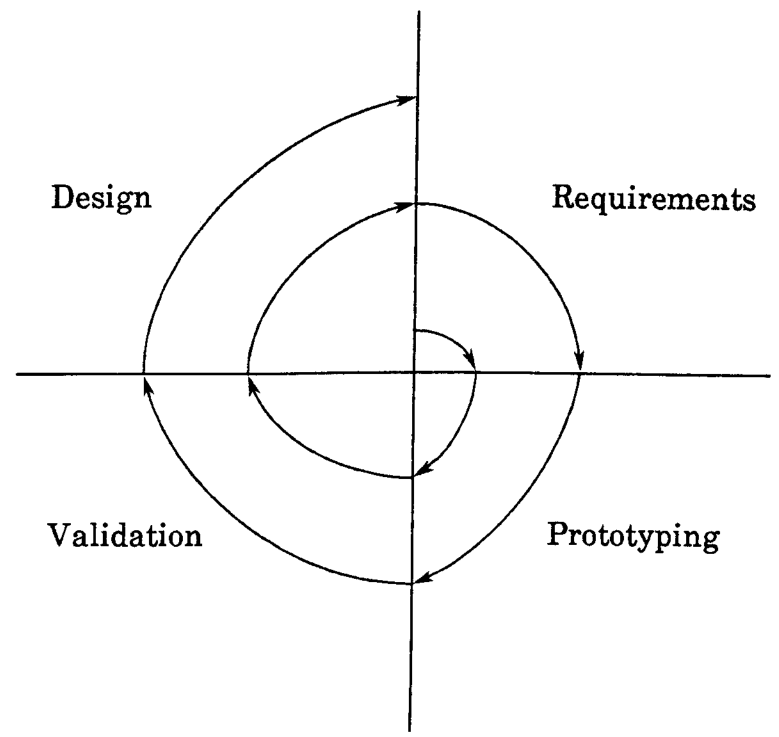


Figure 2.7 A Spiral Software Life Cycle Model

4) implement a prototype with a minimum effort in order to understand the overall nature of the software system;

5) exercise and validate the prototype against the requirements and objectives, based on the experience from its use;

6) use the feedback to understand better the design and requirements to reflect the user needs;

7) go back to the first step.

The idea of producing, as early as possible, a working prototype is widely accepted in engineering. The main idea behind this software life cycle model is to build, cheaply and quickly, a prototype which partially meets known requirements for a software system, with a small team of designers. The purpose of a software prototype is to produce an experimental model which is essentially a learning device and provides feedback to designers so that the final implemented version has a better chance of meeting the requirements.

A big advantage that a prototype can bring to the requirements definition process is the capability of bridging the communications gap which often exists between final users and designers because of their different background. The potential users utilise the prototype for a period of time and supply feedback to designers concerning its strengths and weaknesses. Each cycle is completed with a validation and review of the prototype, and the improvements to be added to the prototype, until a complete software system is built.

There are some advantages to this kind of incremental software development:

- important feedback to designers is provided at the beginning of software development, when it is most needed and most useful;
- designers could use several prototypes, which allow them to evaluate and take a decision among several alternatives;
- many errors, non-viable and unattractive alternatives can be eliminated early;
- iterations and feedback are accommodated in the software life cycle model;
- the system interface is defined and tested early;
- designers and users can see results much earlier in the development process, which provide a good psychological boost to them;
- and most important, costs are reduced because users do not change their requirements late in the development process.

A spiral model presents an organised approach to prototyping and eventually to software development. Furthermore, a spiral model is useful in software developments where many options are possible, and requirements and constraints are unknown at the beginning of software development. Evolutionary prototypes provide incremental software development, so that software systems may be gradually developed and tested, allowing errors to be revealed and corrected earlier than in the waterfall model, which means that they are often cheaper to fix.

A prototype is not intended to be complete nor is it supposed to be robust in the sense of a final piece of implemented software system. Therefore, not all aspects of the software system are prototyped. Only the most important functionality is emphasised, not the exceptional conditions nor particular special cases. Data validation and error handling are not as comprehensive in the prototype as they are in the final software system. Performance considerations are frequently ignored in the prototype. Thus, a prototype is usually inefficient and clumsy because it has very limited error trapping and recovery procedures. Basically, the prototype is there to simply show, develop and test an idea.

Because of those limitations, prototyping can be effective but only as part of a disciplined software development. Without clear and explicit goals and a commitment to keep design up to date, this style can degenerate into uncontrolled hacking. Consequently, employing rapid prototyping can be a risk if it is decided to deliver a prototype as a product instead of discarding it.

The construction of a prototype follows essentially a bottom-up approach. A bottom-up approach has the following characteristics:

- it surveys an application and attempts to identify necessary components;
- it gives priority to the discovery or modification of components over creation of new ones;
- it usually considers components as black-boxes;
- it assembles simple components to form larger and more complex ones;
- it progresses from the specific to the general.

Nevertheless, a bottom-up approach is sometimes chaotic, and some designers might say that it is marked by a code-first-think-later mentality. Moreover, a purely bottom-up approach is not appropriate for developing large software systems, thus, the smaller a software system is, the greater the likelihood that a bottom-up approach will be used.

Other Software Life Cycle Model Proposals

Blair *et al.* (1991) state that there is a significant change to software life cycle models as a result of the new way of using an object-oriented approach to software development. Many object-oriented designers have identified the need for software life cycle models which permit the kind of iterative software development and gradual change which occurs in the development of large object-oriented software systems. However, only recently have object-oriented software life cycle models been studied.

Henderson-Sellers and Edwards (1990) propose a top-down analysis and a bottom-up implementation of object-oriented software systems. This is based on the recognition that the designer's view of a software system changes continuously, and software development is rarely well-behaved. The **fountain model** proposed by them defines the software life cycle in terms of merging and overlapping the following activities: requirements analysis, user requirements specification, software requirements specification, system design, program design, coding, unit testing, system testing, program use and maintenance or further software developments. Nevertheless, the fountain model does not state clearly, for example, when one stage finishes and another starts or what the product of each stage should be. Actually, this model is different from the waterfall model, in that, it acknowledges natural overlapping between two adjacent phases.

Booch (1991) argues that it is not possible to categorise object-oriented design as fundamentally a top-down or bottom-up approach. Booch proposes what is called "analyse a little, design a little and implement a little" approach to software development. However, there are no systematic stages in this approach and it seems to be a kind of design by trial and error offering an excuse for hacking. This style might be viewed as similar to a middle-out approach. This line of thought does not make much sense because it is difficult to trace software development accurately, and depending on the predominant directions, it could be classified into a top-down or a bottom-up approach.

Software life cycle models should provide a systematic framework for software development in such a way that progress can be effectively monitored with a provision of checkpoints and well-defined stages. Planning techniques should be effectively applied and a library of reusable components could be extensively used. An alternative software life cycle model suitable for object-oriented software development is presented in Chapter Five.

2.6 SUMMARY

This chapter has expanded on the background of the object-oriented paradigm described in this thesis. At first, it has introduced the notions of application domains and solution domains, and has showed how the object-oriented paradigm helps to bring these domains together. After, the profile of current software systems has been characterised in terms of complexity, friendliness, extensibility and reusability. Then, the chapter has described the road towards object-oriented design, has presented an overview of the most well-known object-oriented languages and has showed how an object-oriented approach helps tackle the issues presented earlier.

Furthermore, this chapter has investigated the main trends in the object-oriented arena and introduced some definitions for terms and concepts related to an object-oriented model (in particular for classes, objects and inheritance) in order to establish the terminology employed in this thesis. This has been necessary because there have been no generally accepted definitions of what these various terms mean even though the object-oriented paradigm has its roots back in the late 1960s. The terminology presented is programming language independent and is not linked with any object-oriented system. In fact, the proposed terminology has been used as way to characterise an object-oriented model, rather than a precise definition of the object-oriented paradigm.

Additionally, the chapter has presented essential background knowledge, and the philosophy upon which object-oriented design is based, by addressing questions such as: How can one design software systems entirely within an object-oriented framework? How can one represent and describe software systems designed in this fashion? How can one relate application domain, reusability and software life cycle models with an object-oriented approach? By addressing these issues, the most relevant features of object-oriented design have been characterised for use in subsequent chapters. It is important that the object-oriented design methodology to be reported in this thesis be general enough to cover these issues.

In parallel with advances in object-oriented programming languages, several software development methodologies have been emerging, as well as tools which totally or partially automate such methodologies. In particular, various object-oriented methodologies have recently arisen to support software development based on the object-oriented paradigm, as will be seen in the next chapter.

Chapter 3

CLASSIFICATION OF OBJECT-ORIENTED METHODOLOGIES

This chapter presents the current state-of-the-art in object-oriented methodologies. The purpose of the first section is to identify the major similarities and differences between the methodologies, and hence to compare and classify them. To a large extent, the classification and comparison synthesise different directions of thoughts, for instance, the phase of the software life cycle for which a methodology is suitable; whether it is language-dependent; and whether it mixes with other approaches.

The second section presents a flavour of the most well-known object-oriented methodologies, in order to evaluate their main strengths and weaknesses. The coverage of these methodologies is concise and any graphical notation associated with each methodology is shown only briefly; the references can be used to provide additional information. From this outlook, the chapter points out gaps which could be filled by new methodologies.

It is useful at this point to define concepts such as method, technique, methodology, tool and environment. A *method* is defined as a set of systematic activities to carry out a task. A *technique* is the way to execute activities recommended by methods, and a *methodology* is a set of methods and techniques with which an objective may be reached. A *tool* is a resource, automated or manual, that aids the application of a methodology; and finally, an *environment* is a set of tools.

3.1 CLASSIFICATION OF METHODOLOGIES

Many methodologies have been proposed over the last few years. Such methodologies provide some discipline in handling the problem of software complexity because they usually offer a set of rules and guidelines to help software engineers understand, organise, decompose and represent software systems. Such methodologies may be classified into three approaches. Firstly, some methodologies deal with functions; they emphasise refinement through functional decomposition. Typically, software development follows a top-down fashion by successively refining functions, for example, Structured Design (Yourdon, 1979), HIPO (Stay, 1976) and Stepwise Refinement (Wirth, 1971).

In a second trend, there are methodologies which recommend that software systems should be developed with emphasis on data rather than functions. That is, the system architecture is based on the structure of the data to be processed by a software system. The software system should be structured mainly through the identification of data components and their meaning. This sort of style can be noted in an early Jackson Structured Programming methodology (Jackson, 1975), SLAN-4 (Beichter, 1984) and the Entity-Relationship Model (Chen, 1976). The Entity-Relationship Model (ERM) is the most common approach to data modelling. ERM is a graphical technique which is easy to understand, yet powerful enough to model real-world applications, and entity-relationship diagrams are readily translated into a database implementation.

A third fashion consists of methodologies which aim to develop software systems from both functional and data points of view but separately. Examples of such methodologies are SADT (Ross, 1977), Structured Analysis and System Specification (DeMarco, 1979) and Structured System Analysis (Gane, 1979). SADT provides different kinds of diagrams to represent functions and data. As far as Structured Analysis and Structured System Analysis are concerned, designers can represent and refine functions through data flow diagrams and use a data dictionary to describe data. These methodologies organise a specification and design around hierarchies of functions. They begin by identifying one or more high level functions which describe the overall purpose of a software system. Then, each high level function is decomposed into smaller, less complex functions, until they can be implemented.

There are also some software development environments which have automated some of those methodologies. The chief purposes of these environments are to increase productivity and enhance the quality of the developed software system. PSL/PSA (Teichroew, 1977) and EPOS (Lauber, 1982) are good examples of such environments.

A combination of approaches which follow a structured analysis, structured design and structured programming is collectively known as the **structured development** approach. Structured development iteratively divides complex functions into subfunctions. When the resulting subfunctions are simple enough, decomposition stops. This process of decomposition is known as the *functional decomposition* approach. Structured development also includes a variety of notations for representing software systems. During the specification and analysis phases, data flow diagrams, entity-relationship diagrams and a data dictionary are used to logically describe a software system. In the design phase, details are added to the specification model and the data flow diagrams are converted into structure chart diagrams ready to be implemented in a procedural language.

Preferably, there should be specific methodologies suitable to object-oriented software development because there are specific object-oriented concepts involved. The unsuitability of those mentioned methodologies for tackling the object-oriented software development problem suggests the use of different methodologies followed by an informal change of approach, from a functional decomposition to an object-oriented, during software development. For instance, the designer starts analysis following a functional decomposition point of view and afterward, during the implementation phase, changes to an object-oriented point of view. This change in approach leads the thought process to follow an object-oriented approach in the middle of software development instead of starting software development based on classes, objects and inheritance from the outset.

Structured analysis has been suggested as an attractive front-end to object-oriented design primarily because it is well-known, many designers are trained in its techniques, and many tools support its notation. However, structured analysis is not the optimal front-end to object-oriented design, mainly because it can perpetuate a functional decomposition view of the real-world application. Applying a functional decomposition approach first and an object-oriented approach later on the same software system is likely to lead to trouble because functional decomposition can not be properly mapped into object-oriented decomposition. A better trend in analysis is to use an object-oriented analysis method in which there are attempts to identify and model the essential classes and objects of a software system.

The object-oriented paradigm organises software systems into classes and establishes relationships between them. Objects model real-world entities and combine both attributes and operations. Each object is an instance of a class which is the building block of a system architecture. A positive benefit of following an object-oriented approach is traceability between software abstractions and reality because organising a software system around classes maps real-world entities into software components, particularly classes and objects. Thus, any methodology which deals with the object-oriented paradigm should have a means of representing classes, objects and inheritance. Ideally, object-oriented design and implementation should be part of a software development process in which an object-oriented philosophy was used throughout software development, as illustrated in Figure 3.1. Such a figure was derived from discussions on similar issues presented by Loy (1990) and Henderson-Sellers and Constantine (1991). In Figure 3.1, the dashed arrows represent an unnatural mapping between concepts of different approaches as opposed to bold arrows.

Experience has shown that simply attempting to combine an object-oriented approach with a structured development approach is likely to give rise to some problems. It jeopardises traceability from requirements to implementation because, in early phases, a software system is described in terms of functions and later on the description is changed in terms of object-oriented concepts (see Figure 3.1). Furthermore, structured development methodologies do not localise information around objects but on data flow between functions, and a software system is composed of data flow and functions. In contrast, the object-oriented paradigm organises a software system around classes and objects which exist in the designer's view of the real-world application.

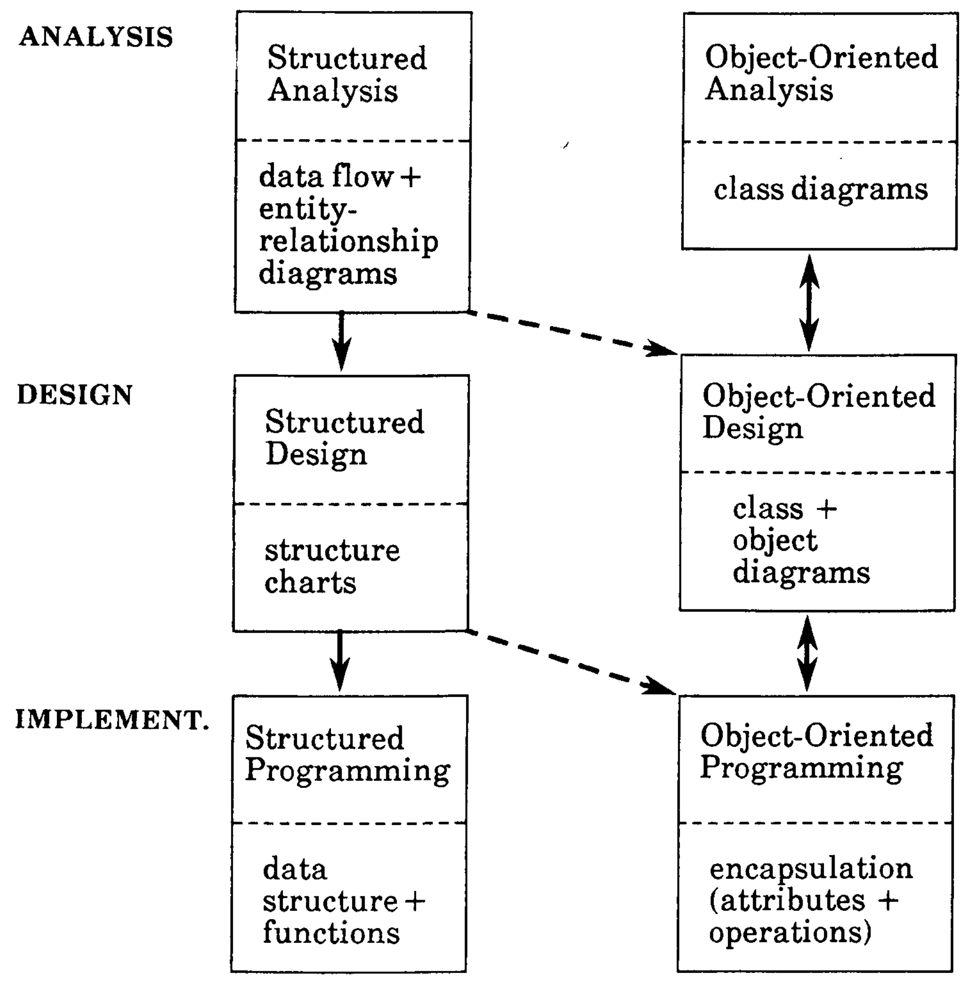


Figure 3.1 Some Combinations of Approaches

Moreover, since the concepts of class, object and inheritance are fundamental to the object-oriented paradigm, if a software system has been designed using object-oriented concepts, an object-oriented programming language should be used for the implementation phase. Of course, there are attempts to implement concepts without a language that directly supports them, but the results are not likely to be as good as using a proper object-oriented language. Thus, to implement an object-oriented design, an object-oriented language is also recommended.

From the Designer's Point of View

Designers who have a strong background in object-oriented programming naturally have a good grasp of object-oriented concepts and have almost no problem in identifying classes and objects, but some of them tend to consider software engineering methodologies unnecessary. These designers have written code using an object-oriented programming language but are often unable to separate design from implementation. Some of them sometimes even suggest that methodologies are unnecessary, and that the object-oriented paradigm obviates the need for methodologies.

A second trend is defined by designers who have good experience with structured development methodologies. These designers have a relevant background based on data flow diagrams, entity-relationship diagrams and structure charts, and argue that structured development and an object-oriented approach are correlative. These designers have a grasp of more traditional software engineering approaches and tend to freely mix concepts from a number of different approaches with an object-oriented approach. They believe that structured development and an object-oriented approach are complementary rather than opposing techniques and each might be applicable at different stages of software development. However, these designers have come to realise that those traditional structured methodologies based on a functional decomposition approach do not take the advantage of the inheritance mechanism.

Another trend is characterised by those designers with a good understanding of both object-oriented concepts and software engineering principles. Such designers have attempted to introduce new software engineering concepts and bring them into synchronisation with the object-oriented paradigm. They believe that the object-oriented paradigm is so unique that designers basically have to throw away traditional ideas of the past and tackle object-oriented software development with a clean slate. This author and the work presented in this thesis are associated with this trend. These tendencies are further discussed in the next subsection.

3.1.1 Classification of Existing Object-Oriented Methodologies

Recently, there has been a profusion of object-oriented methodologies for analysis and design influenced by a variety of different backgrounds (Arnold, 1991). Nevertheless, it can be noticed that there are two major directions concerning object-oriented methodologies:

a) **adaptation**: this is concerned with the mixing of an object-oriented approach with well-known structured development methodologies;

b) **assimilation**: this emphasises the use of an object-oriented approach for developing software systems, but following the traditional waterfall software life cycle model.

Adaptation

Adaptation proposes a framework for mixing an object-oriented approach with existing methodologies. It has been suggested that a combination of structured development and an object-oriented approach helps tackle the problem of software development. Designers use their experience and intuition to derive a specification from an informal description in order to get a high level abstraction for a software system, based on functional decomposition. The adaptation of structured development to an object-oriented approach preserves the specification and analysis phases using data flow diagrams and proposes heuristics to convert these diagrams into an object model in such way that subsequent phases can then follow an object-oriented approach.

The advantages of this adaptive approach are:

- a complementary (though unnatural) coupling between structured development and an object-oriented approach;
- structured development methodologies are widely known and used, and support top-down functional decomposition, the most common fashion for software development;

- a smoother migration from an old, practised and well-known approach to a new one including classes, objects and inheritance;
- a gradual change of tools and environments to an object-oriented approach.

Currently, the most widely used software engineering methodologies are those for structured development. Those methodologies are popular because they are applicable to many types of application domains. On account of this popularity, structured development has been mixed up with an object-oriented approach. Therefore, those designers who come from a traditional software engineering background, such as functional decomposition and data modelling techniques, will probably find the methodologies of Shlaer and Mellor (1988), Coad and Yourdon (1990) and Rumbaugh *et al.* (1991) familiar because these methodologies are clearly adaptations of traditional structured development methodologies and data modelling techniques. These methodologies may be used during a period of transition from structured development to an object-oriented approach as a compromise. However, they cannot permit the full advantages of an object-oriented approach to be gained.

This tendency can also be clearly seen in the early version of Booch (1986) methodology and its successors, such as Seidewitz (1989), Heitz (1989) and Jalote (1989). These methodologies do not make an adequate distinction between definition of class and use of objects, which is essential for the exploitation of the object-oriented paradigm. Similarly, they have offered limited support for inheritance of commonalities in a hierarchy of classes; they tend to be oriented to Ada notations of *package* and *task*, rather than to more general notions of object-oriented design.

Other less known proposals where object-oriented concepts are by-products of structured development could also be considered. Some of these methods are merely extensions of structured development methodologies. Masiero and Germano (1988) and Hull *et al.* (1989) put together an object-oriented approach with Jackson (1983) methodology, and the product of a design is implemented in Ada. Kerth (1988) extends Ward and Mellor (1985) structured development methodology for real-time systems. Bailin (1989) and Bulman (1989) mix up an object-oriented approach with Structured

System Analysis (Gane, 1979) and the Entity-Relationship Model (Chen, 1976) for a object-oriented requirements specification model. Lastly, Alabiso (1988) and Ward (1989) combine the object-oriented style with Structured Analysis (DeMarco, 1979), Structured Design (Yourdon, 1979) and the Entity-Relationship Model (Chen, 1976).

Adaptation approaches are trying to evolve object-oriented methods from existing ones and as a result bringing their limitations with them. More importantly, they do not fit object-oriented software development because inheritance is not fully exploited.

Assimilation

Assimilation is a trend that puts the object-oriented paradigm within the traditional waterfall software life cycle model. In recent years several object-oriented methodologies have appeared but they cover only partially that software life cycle model. Several authors have tried to fit the object-oriented paradigm into this framework: Booch (1991), Wirfs-Brock *et al.* (1990), Wasserman *et al.* (1990), Pun and Winder (1989), and Lorensen (1986) can be considered as good examples.

So far these methodologies are not well-known and not generally accepted, but their main ideas encompass object-oriented concepts because they are based on at least classes, objects and inheritance. These methodologies still need to be used in practical contexts to develop large scale software systems in order to be evaluated and improved. A discussion of why a new methodology is required is presented in the third section of this chapter, after a survey of existing object-oriented methodologies.

3.2 SURVEY OF EXISTING METHODOLOGIES

When unimportant or trivial software systems must be developed, there is no need to consider software development methodologies. Designers go directly to the computer and start writing their programs immediately; a lot of programs have been written in this way. This form of development is feasible if a few designers work on a small software system, but it is not suitable when large software systems have to be built. In this case, the use of software development methodologies is highly recommended because they help to structure software systems as a whole and standardise software development among designers.

Because software systems are essentially abstract, they themselves are difficult to represent. In fact, there are many kinds of software systems, for instance, real-time, process-control, scientific and commercial software systems, which all require different supporting methodologies. Since the application domains are different, methodologies with particular characteristics should be employed. Moreover, there are some specific methodologies which are applied to specific phases of the software life cycle. Such restrictions are manifested in the absence of one methodology for software development which is widely accepted and used. As a result there are a number of methodologies, methods and techniques, different notations and conflicting rules, each with its own advantages and disadvantages, as can be seen in the next subsections.

3.2.1 The Booch Methodologies and Their Influences

The history of object-oriented technology dates from the 1970s, but up to the mid-1980s, much of the work in the object-oriented arena focused on object-oriented programming. The application of an object-oriented approach to software design has occurred since that time, mainly for those familiar with Simula-67. From the beginning of the 1980s, some attempts to develop software systems using an object-oriented approach have emerged. The first significant step towards an object-oriented design methodology, started within the Ada community. Many ideas about object-oriented design came out with the research of Abbott (1983) and Booch (1983a), (1983b).

Booch set out to find some mechanism for introducing software engineering into the Ada training effort and identified the work of Abbott as relevant. Abbott had described a simple methodology to design using nouns and verbs. Booch rationalised that methodology, and referred to it as *Object-Oriented Design* (Booch, 1983b). Both Abbott and Booch have recommended that a design should start with an informal description of the real-world application and from this description designers could identify objects. The work of Booch is significant because it was one of the earliest object-oriented design methodologies to be described in the literature. Booch is also one of the most influential advocates of object-oriented design within the Ada community.

As far as Booch's influences are concerned, they can be summarised a follows: what has come to be known as object-oriented design in the context of Ada was first proposed by Booch (1983b), later extended by Booch (1986) and after refined by Seidewitz (1989), Heitz (1989) and Jalote (1989). Berard (1986) and Sincovec and Wierner (1987) also present principles and methods biased by Booch (1983a) with implementation also totally driven towards Ada. These design methodologies concentrate on identifying objects and operations, and are object-oriented in the sense that they view a software system as a set of objects. Most of these methodologies are based on an informal description or representation of the software requirements, from which objects, attributes and operations can be identified. Moreover, all of these methodologies apply hierarchical decomposition, a trend to decompose a software system by breaking it into its components through a series of top-down refinements.

The Abbott Methodology

The Abbott methodology is based on the definition of a real-world application using a natural language description, and deriving a design by considering mainly nouns and verbs in that description. Abbott believes that the main ideas of the application should be stated in sentences with which designers work in order to understand and refine the features of the application.

The Abbott methodology consists of three steps:

1) Develop informal and general statements for the application. This informal description, which states the application domain and the application itself, should be expressed in application domain terms. In other words, write straightforward natural language paragraphs which describe the application within its application domain.

2) Formalise the informal description. The formalisation consists of identifying data types, objects, operations and control structures by

looking at the natural language words and phrases in the informal description. The formalisation sub-steps are:

a) Identify the data types. Common nouns and noun phrases are good candidates.

b) Identify the objects (program variables) for those data types. A proper noun or direct reference suggests an object.

c) Identify the operations to be applied to those objects. This can be done by examining verbs, predicates and descriptive expressions.

d) Organise the operations into the control structures implied in a straightforward way by the informal description.

3) Segregate the solution into two parts: packages and subprograms. The packages will contain the formalisation of the application domain, that is, the data types (objects) and their operations. The subprograms will contain the specific steps (expressed in terms of the data types and operations defined in the packages) for simulating that particular application.

The steps focus on what must be done to solve a problem and then how a solution can be accomplished. The level of abstraction is very high before step 3 is reached. In simple terms, the main idea is to identify all nouns and verbs in a specification. Objects in a design will derive from nouns; object operations will derive from verbs. Obviously, some judgment must be used to disregard irrelevant nouns and verbs and to translate the remaining concepts into a design of objects. Adjectives and adverbs become attributes of objects and operations, respectively. An attribute helps discern specific characteristics of objects, and can be used to establish the constraints of a software system.

It must be noticed that although these steps may appear mechanical, they are not an automatic procedure. It requires intuitive understanding of the application by designers. The process of identifying the data types, objects, operations and control structures from a given natural language description requires a great deal of knowledge about the application and an intuitive understanding of the application domain. It is not just a matter of examining the syntax of a natural language description.

Several observations can be made at this point. Firstly, Abbott is not concerned with the concept of inheritance as the steps are being carried out. Secondly, the viability of the technique of creating an informal narrative description of a problem, and then selecting data types, objects, operations and attributes of a software system from nouns, verbs, adjectives and adverbs of this narrative description is questionable. It inherently lacks rigour due to the impreciseness of natural languages. Thirdly, methodologies which rely on natural languages are useful to describe very small software systems where conceptual links are minimal, but not to design large and complex software systems because of the ambiguity of natural languages and the lack of standard structures. Finally, when a software system is large and complex with many interacting components, it is very difficult to produce a precise, concise and complete narrative description. Therefore, the suitability of the Abbott methodology for developing large and complex software systems may be questioned.

The Booch Methodologies

Booch (1983b) has also viewed the technique of identifying nouns and verbs as one out of many which could help identify objects. However, realising the drawbacks of this technique in early works and recognising that this technique was never actually intended for developing large software systems, Booch (1986), (1987) methodologies have not advocated the use of a narrative description anymore. Instead, Booch (1986) has combined object-oriented design with existing methodologies and called it *Object-Oriented Development.* Booch suggested that existing methodologies such as SREM (Alford, 1977), Structured System Analysis (Gane, 1979) or Jackson Structured Development (Jackson, 1983) could be used during the requirements and specification phases as a step before object-oriented design.

Booch describes a methodology for object-oriented design which begins with a data-flow-diagram-based specification, decomposes the specification into objects, finds dependencies between objects, and maps the design into Ada statements. Booch argues that inheritance is an important, but not a necessary, object-oriented concept and that software development without inheritance still constitutes object-oriented software development.

According to Booch (1986) methodology, object-oriented software development can be divided into the following steps:

1) Define the problem, even informally.
2) Devise a specification.
3) Identify the objects and their attributes from the specification.
4) Identify the operations provided by and required of each object.
5) Establish the visibility to each object in relation to other objects.
6) Establish the interface of each object.
7) Implement each object.

Booch defines an object as an entity that:

- has state;
- is denoted by a name;
- is an instance of some class;
- has restricted visibility of and by other objects;
- is characterised by actions that it provides and that it requires of other objects.

While the identification of objects is being carried out, objects with similar properties should establish a class, so the concept of class appears early in a design. Booch also uses a graphical representation for a class which is helpful in visualising the architecture of a software system (see Figure 3.2), based on a box which consists of an externally visible part which represents a type and its operations, and a private part used to describe implementation details. Such boxes compose diagrams which show the inter-dependencies among the objects through arrows.

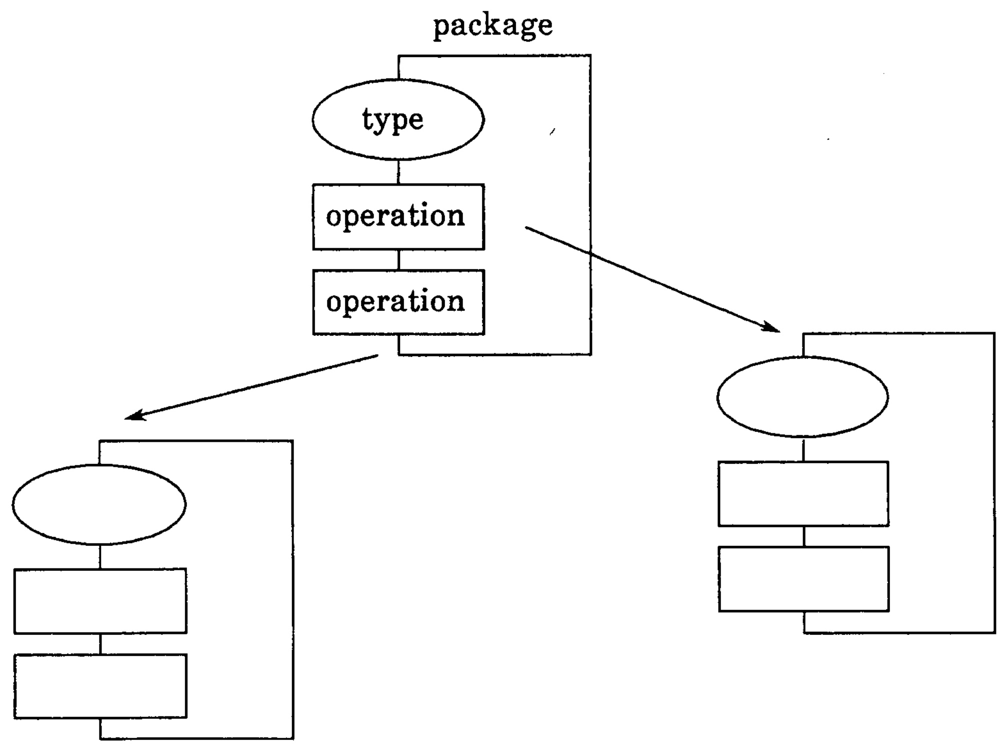


Figure 3.2 Classes as Packages

The implementation of a class is to be performed using Ada. The transformation of a design described by a graphical notation is straightforward; each class represented by a box is denoted as a package in Ada. A package is made up of two parts: a specification that defines the data types which can be used externally, and a package body which describes the implementation of the specified operations.

Booch (1991) has proposed a new version of the methodology derived from the previous one, that is Booch (1986). But at this time Booch has tried to convey the opinion that the new version of the methodology is not Ada-oriented anymore. The first step involves the identification of classes and objects at a high level of abstraction; at that level, the important activity is the discovery of key abstractions in the application. The second step involves the identification of the semantics of these classes and objects; here, the important activity is for designers to act as detached outsiders, viewing each class from the perspective of its interface.

The third step involves the identification of relationships among these classes and objects; this step establishes how objects interact within the software system, with regard to the semantics of the key abstractions found in the former step. The fourth step involves the implementation of these classes and objects; the important activity is choosing a representation for each class and object, and allocating them to modules. The fourth step is not necessarily the last step because its completion usually requires that the whole design process needs to be done all over again, but this time more details about the application are known.

The methodology also proposes a graphical notation to represent a design, which can form the basis for automated tools. Nevertheless, most of the notational conventions are directed towards the representation of Ada concepts, such as packages (see Figure 3.2), task communications and module visibility. As previously, Booch has not strongly considered inheritance because Ada is in the background of this methodology, and Ada does not provide inheritance. Besides, there is no consideration of reusability during the design process.

GOOD

Seidewitz (1989), influenced by Booch (1986) and Yourdon and Constantine (1979) works, has created a General Object-Oriented Development (GOOD) methodology. GOOD also begins a design with data flow diagrams, at the specification level, then objects are identified and refined into object diagrams. Once the design level is completed, each object may be decomposed into sub-objects represented by lower level object diagrams designed to meet the specification for the object in the upper level. The result is a layered collection of object diagrams which completely describes the hierarchical structure of a software system (see Figure 3.3). The object diagrams show control flow and module dependencies between objects. At the lowest level, objects are completely decomposed into procedures and their internal data structures.

A seniority hierarchy, as illustrated in Figure 3.3, is expressed by the topology of connections on object diagrams. Each object is a node of a graph, and if object A somehow invokes any operation of object B, then there is an

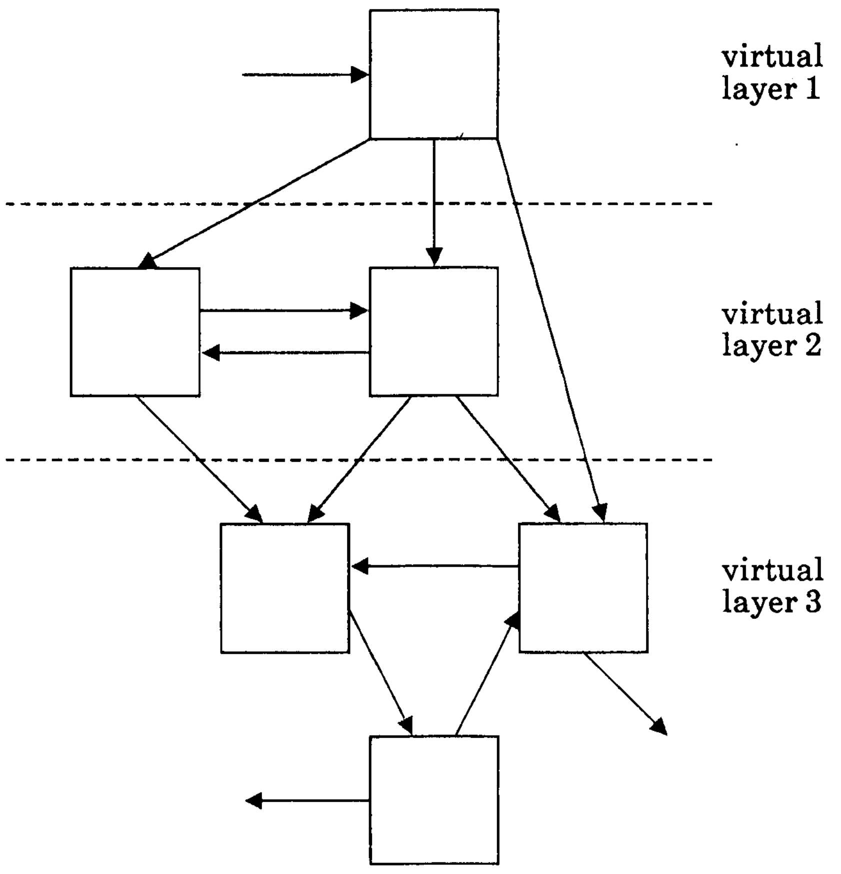


Figure 3.3 Seniority Hierarchy in Object Diagrams

arrow from A to B. Any layer in a seniority hierarchy can call on any operation in an equivalent or lower level layer, but never any operation in a higher level layer. Thus, all cyclic links between objects must be contained within the same layer in the seniority hierarchy. GOOD also provides an object description which includes a list of all operations provided by an object, and for each outgoing arrow, the operations required by the object.

The GOOD methodology is similar in many ways to the traditional structured development through functional decomposition, and does not even seem to consider class hierarchies. At the lowest level, objects are completely decomposed into primitive objects and then each primitive object is implemented in Ada.

HOOD

Hierarchical Object-Oriented Design (HOOD) (Heitz, 1989) is also an Ada-based methodology, that is, with Ada as the target programming language. HOOD was developed taking mainly as its starting point the ideas from Booch (1983a) methodology. HOOD proposes four stages for software development:

1) Definition and analysis of the real-world application.

2) Specification of a design solution using natural language descriptions.

3) Selection of nouns to make a list of objects and verbs to form a list of operations. This is called the Informal Solution Strategy.

4) Production of a formal Object Description Skeleton which will be used to generate code in Ada.

The HOOD tool set provides support for the third stage by building word lists for object and operations. The objects and operations lists are then combined into an object operation table from which object diagrams can be produced. These diagrams do not show a software system as a collection of classes but show the behaviour of objects. There is also a tool to deal with the generation of code in Ada from the Object Description Skeleton which contains formal descriptions of the operations. Although the design process is based on the encapsulation of data and procedures by means of objects, inheritance is not considered.

The Jalote Methodology

Jalote (1989) has proposed an object-oriented design methodology which consists of three main steps:

1) Define the problem.

2) Develop an informal strategy.

3) Formalise the strategy. This step has four sub-steps:

a) identify the objects and their attributes;

b) identify the operations on the objects;

c) establish the interfaces of the objects;

d) implement the operations.

The aim of the first step is to properly understand the nature of the problem. In the second step an informal strategy to solve that problem is stated using natural language descriptions. In the third step, the objects are first identified from an informal strategy. Then the operations on the objects are identified. The interfaces of the objects are then established by describing the visibility of each object. The informal strategy is then converted into a formal description of the problem by associating the operations in the informal strategy with the identified objects.

Jalote has introduced two concepts called the functional refinement stages and nested objects. In the functional refinement stages, new objects and operations are identified. Furthermore, operations may also be identified upon the objects which were identified in earlier functional refinement stages. The methodology refers to the group of all identified objects, when the functional refinement stages terminate, as the Problem Space Object Set (PSOS). When this refinement process finishes, the algorithms needed to solve the original problem would have been decomposed to some required level, with each operation in the algorithms being an operation on some object.

To refine an object, informal descriptions of all operations defined on the object are written. Then new objects (called nested objects) which were required to implement these operations are identified. This process is repeated for each object in the PSOS. Once all the operations on the objects are identified, the refinement of nested objects can begin. This step ends when objects are simple enough to be implemented directly, which is carried out in Ada.

By introducing the idea of nested objects, the methodology incorporates a top-down refinement technique with object-oriented design, which is more general and more suitable for designing large software systems.

Nevertheless, the methodology focuses only on a systematic process to carried out design towards implementation in Ada, and does not introduce any graphical notation for representing a design, nor considers inheritance and software reuse.

Comments on Booch Methodologies and Their Influences

Biased by Booch methodologies, several object-oriented design methodologies which combine object-oriented concepts with other well-known methodologies have emerged, as has been described briefly above. Nevertheless, methodologies based on natural language descriptions have many problems because of the informality and ambiguity inherent in the use of natural languages. For example, what exactly does the phrase "get a course from a course list and assign the course a room number in a building" mean? It is not enough to rely on normal usage of these terms; a more complete explanation is required to derive a design. In fact, the production of a complete application description is neither straightforward nor concise, and usually needs much more detailed information about the application and its constraints. Natural language descriptions work well when the narrative description of the application can be written concisely. However, when the narrative description is large, that description can be very difficult to be written and understood.

In addition, all methodologies influenced by Booch methodologies are based on hierarchical decomposition of software systems into layers, supporting a top-down fashion which allows designers to start with a high level abstraction and work top-down towards an implementation in Ada. Therefore, those methodologies do not strongly address an important object-oriented concept (that is, inheritance) which makes the object-oriented paradigm more powerful. Booch (1991) extends previous Ada-oriented work to object-oriented software development in general. The latest methodology from Booch includes a variety of models which address functional and dynamic aspects of software systems. However, the phases at the beginning of software development are not addressed in depth.

3.2.2 OOSD

Wasserman *et al.* (1990) have proposed OOSD, a graphical representation for Object-Oriented Structured Design. OOSD provides a standard design notation by supporting concepts of both structured and object-oriented design since the main ideas behind OOSD come from Structured Design (Yourdon, 1979) and Booch (1986) notation for Ada packages. Therefore, OOSD allows designers to gradually shift from structured development to an object-oriented approach. OOSD also incorporates the concept of *monitors* for concurrent programming and has other important features:

- it supports class hierarchy and inheritance;
- it is independent of any programming language;
- it provides a BNF grammar for a textual description of a design;
- it has a mechanism to represent exceptional conditions;
- it supports a wide variety of software systems, including both sequential and concurrent models of execution.

In general terms, OOSD represents a class with a rectangle and uses overlapping small boxes to show operations. An operation overlapping a class rectangle suggests a visible part of that class. Moreover, an operation can also be placed completely inside a class rectangle to indicate that the operation is hidden from other classes. OOSD uses the parameter passing notation of Structured Design, where arrows indicate the kind of parameters (e.g. data or control). Exceptions are represented by diamonds overlapping a class rectangle anywhere around its perimeter (see Figure 3.4). After defining a class, designers can instantiate its objects. However, OOSD regards this step as a matter of implementation because the instantiation is related to the use of a class not to its definition. Inheritance is represented by a hierarchy of class rectangles linked by dashed arrows.

It is evident that OOSD mixes up object-oriented concepts with another approach because the convention for representing parameter passing comes from Structured Design. OOSD gathers together Booch ideas with Structured Design principles following a top-down functional decomposition into modules and ends up with Ada as an implicitly suggested

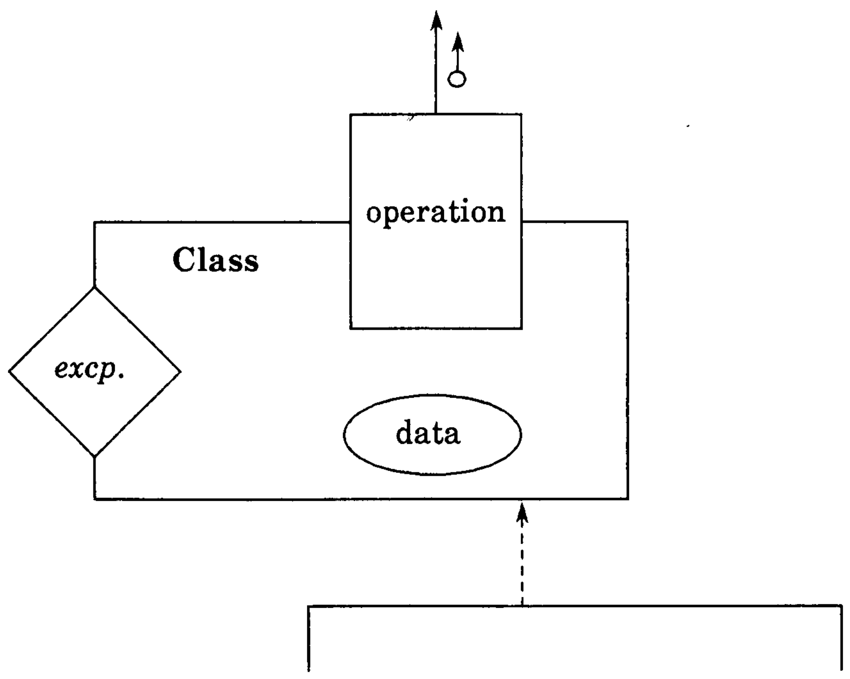


Figure 3.4 Class Representation in OOSD

implementation language, and thereby accommodates two important design notations.

OOSD focuses mainly on a graphical representation without addressing the method by which a design could be created and gives no explicit technique for diagramming software system decomposition, needed for large software systems. Furthermore, it does not set up guidelines to identify classes and operations, so it is supposed that designers should follow steps recommended by other methodologies. Although OOSD clearly shows relationships between classes, it does not represent detailed interactions between objects. It is expected that designers will use their own methods together with OOSD, which simply provides a notation for object-oriented design, not a step-by-step methodology.

3.2.3 Responsibility-Driven Design

Wirfs-Brock *et al.* (1990) have focussed on the identification of *responsibilities* and *contracts* to build a *responsibility-driven* design. Responsibilities are a way to apportion work among a group of objects which comprise an application. A contract is a set of related responsibilities defined by object interactions, and describes the ways in which a given client object can interact with a server object.

This methodology emphasises the actions that must be accomplished and which objects will perform these actions. The responsibilities of an object are the services it provides for all objects which communicate with it. Both objects must fulfil a contract: the client object by making the requests that the contract specifies, and the server object by responding appropriately to these requests. Objects fulfil their responsibilities either by performing the necessary computation themselves or by collaborating with other objects. The responsibility-driven design recommends the following steps to design a software system:

1) Select noun phrases from the specification and build a list of nouns. Identify candidate classes from noun phrases by modelling physical and conceptual entities in the application. Then, identify candidates for high level classes by grouping entities which share common properties, and write a short statement of the purpose of each class on a *class card* which is used to capture information about a class.

2) The design continues by defining the purpose of each class and the role it will play in the software system in terms of responsibilities. Responsibilities include the state that a class maintains and actions that a class provides. When responsibilities are assigned to a class, every instance of that class will naturally have those responsibilities. Therefore, responsibilities are meant to convey the services which an object provides and its place in the software system. The responsibilities are found by recalling the purpose of each class in terms of the actions that its objects offer. This step considers the use of inheritance as a means of grouping common responsibilities as high as possible in class hierarchies so that the relationship *is-kind-of* between classes is valid. Services defined by a class include those

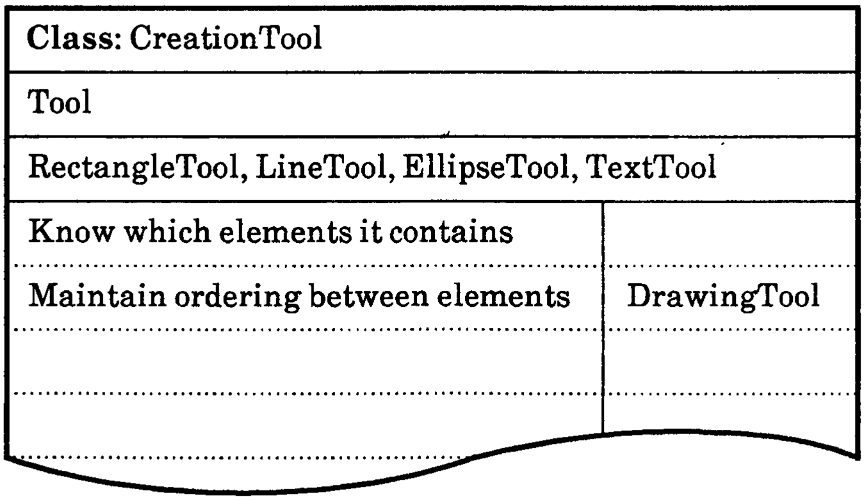
listed on the class card definition, plus the responsibilities inherited from its super-classes.

3) During the third step, subsystems are identified. A subsystem is a collection of classes (and possibly other subsystems) collaborating to fulfil a common set of responsibilities (contracts). From the outside, a subsystem can be viewed as a collection of classes which provide clearly delimited services. In order to identify possible subsystems, designers should look for collaborations which happen frequently. Collaborations represent an exchange of messages from one object to another in fulfilment of a contract. Classes in a subsystem should collaborate to support a cohesive set of responsibilities; should be strongly interdependent and the subsystem division should minimise the number of collaborations a class has with other classes or subsystems. The methodology also recommends going back to earlier stages to refine the responsibilities, classes and subsystems.

As the steps are followed, the responsibility-driven design methodology considers walk-throughs to explore design possibilities and to record the result of a design on class cards. The technique of recording design on cards was introduced by Beck and Cunningham (1989) who have proposed the Class, Responsibility, and Collaboration (CRC) cards. They have found that index cards are a simple technique for teaching object-oriented thinking to newcomers of the object-oriented paradigm.

Each class card contains the name of a class, a description of the responsibilities associated with that class and its collaborators. They also record sub-class and super-class relationships. Each candidate class is written on a class card, as illustrated in Figure 3.5, its super-classes and sub-classes are described in the lines below the class name. Each identified responsibility is succinctly written on the left side of the card. If collaborations are required to fulfil a responsibility, the name of each class which provides the necessary services is recorded to the right of the responsibility.

Some observations should be made on responsibility-driven design. The methodology is based on the identification of classes by looking at nouns in a natural language description of the system specification. This technique has the same problems as the first versions of Abbott and Booch methodologies.



Class: CreationTool	
Tool	
RectangleTool, LineTool, EllipseTool, TextTool	
Know which elements it contains	
Maintain ordering between elements	DrawingTool

Figure 3.5 A Class Card

The methodology also suggests the use of class cards to describe classes and subsystems. This technique may work well with simple software systems but its use in the design of large and complex software systems is doubtful because as the number of classes and subsystems grows sharply, the number and arrangements of class cards may become cumbersome and difficult to manage.

Additionally, responsibility-driven design bases inheritance only on responsibilities, and ignores inheritance of attributes. Moreover, the methodology divides a large software system into subsystem only after identifying some classes and their responsibilities. However, the partitioning a software system into subsystems should be considered at the beginning of software development as a technique to decompose large software systems.

3.2.4 OORA

The main steps established by Coad and Yourdon (1990) to an Object-Oriented Requirements Analysis (OORA) methodology can be briefly described as:

- identify objects;
- identify structures;
- identify subjects;
- define attributes;
- define services.

Identification of objects encompasses the understanding of the application domain and finding abstractions of data and processing on that data. Designers have to concentrate on the application domain and look for entities, devices, events, roles played and organisational units in order to consider needed behaviour, requested services and essential requirements, which at the end of this stage are identified as objects.

Identification of structures aims at managing the complexity in the application domain by constructing a hierarchy based on specialisation and generalisation of objects; and by assembling objects by aggregation and decomposition, reflecting whole and parts of a software system. Subjects provide a mechanism for controlling how much of a software system a designer is able to consider and comprehend at one time. They are associated with the structures identified in the previous step. Subjects divide a software system into partitions composed of structures, and grouping together similar subjects defines subsystems.

Attributes are seen as data elements used to describe objects. Defining the attributes involves examining the application domain, and attaching an individual data element or a collection of related data elements to the entity which it actually describes and then to the object associated with that entity. A service is the processing to be performed upon receipt of a message. The central issue in defining services is to identify the required behaviour for each object. A second issue in defining services is to arrange the necessary communication between objects. The methodology starts by considering fundamental services such as *store* and *calculate* based on external events and required responses to an object during its lifetime.

OORA also presents a graphical notation for objects, as illustrated in Figure 3.6. Basically, an object is represented individually by a rectangle containing its name and its attributes and services. Specialisation is denoted by lines with semicircles at the end of the sub-objects. Subjects are separated by dotted lines. The relationships between objects can be 1:1 or N:M. Finally, messages are indicated by dashed lines linking the involved objects.

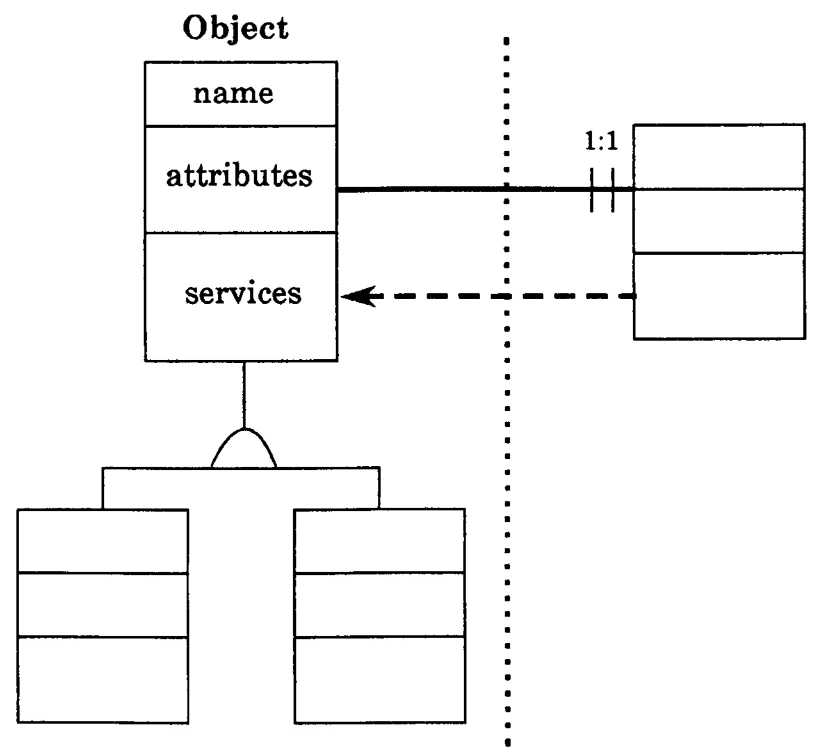


Figure 3.6 Object Representation in OORA

Coad and Yourdon have oversimplified the object-oriented paradigm by misusing the concepts of classes and objects during the analysis phase as objects only. Throughout the methodology there is confusion about the concepts of classes and objects. Basically, OORA concentrates on modelling real-world entities as objects, and it can be considered as an extension of the Entity-Relationship Model (Chen, 1976), suggesting that OORA is an incremental improvement over an existing approach to data modelling. Unfortunately, Coad and Yourdon have proposed the use of ordinary 3" by 5" index cards as a substitute for software tools, which is a trivialisation of the benefits of using computer-aided tools. Moreover, they have not discussed the impact of their methodology on other phases of the software life cycle.

3.2.5 OMT

Rumbaugh *et al.* (1991) have developed the Object Modelling Technique (OMT) which focuses on object modelling as a software development technique. Basically, OMT proposes three models for software development:

- The *object model* describes the aspects of a software system in terms of classes, in an object model diagram. The object model is represented by graphs whose nodes are classes and arcs denote relationships of specialisation, composition, or any association between classes (see Figure 3.7).
- The *dynamic model* describes the aspects of a software system which may change over time due to events, and it is used to understand the control flow of a software system. The dynamic model is represented by familiar state transition diagrams whose nodes are states and arcs are transitions (caused by events) between states.
- The *functional model* describes the data transformations within a software system and employs the well-known data flow diagrams to represent computations of output values from input values.

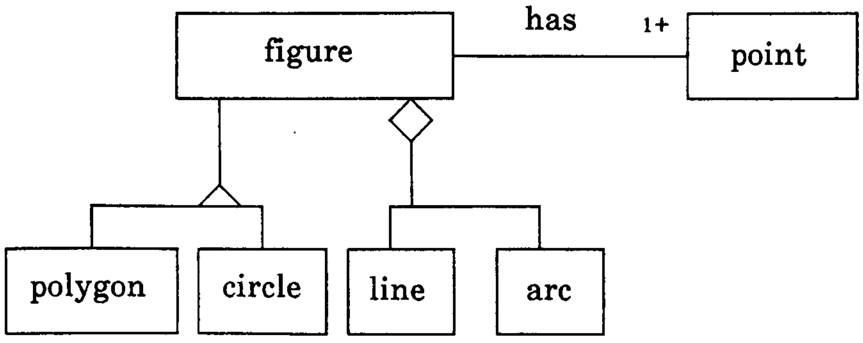


Figure 3.7 An Object Model Diagram

OMT proposes three stages which should be carried out to produce a design:

1) The *analysis stage* captures important properties which are meaningful to a real-world application and uses the object, dynamic

and functional models to represent those properties. The purpose of this stage is to understand the application in terms of classes.

2) The *system design stage* focuses on decisions about the high level structure of a software system. In this stage the software system is divided into subsystems, and no concept related to the object-oriented paradigm is employed.

3) The *object design stage* is targeted at the data structures and algorithms need to implement each class in the object model. During this stage, dynamic and functional aspects are combined and refined, and more details about the control flow of a software system are defined.

Some considerations on OMT can be made at this point. The first stage of OMT is equivalent to OORA, in that it basically models the application in terms of classes and special relationships between them. However, it is confusing that the object model actually deals with classes. Later, during the system design and object design stages, OMT incorporates structured development based on a functional decomposition approach following the traditional waterfall software life cycle model.

3.2.6 OOSA

Shlaer and Mellor (1988) have proposed a system analysis methodology and an associated graphical notation called the Object-Oriented System Analysis (OOSA) which is based on a variation of the Entity-Relationship Model (Chen, 1976) combined with Structured System Analysis (Gane, 1979). The notation can be used to describe objects, attributes and relationships, where relationships are any pattern of association between objects.

OOSA suggests that designers should construct three models:

- the *information model*;
- the *state model*;
- the *process model.*

The information model represents objects, attributes and relationships in an information structure diagram, as illustrated in Figure 3.8. The state model expands the information model by representing the behaviour of each object using a standard state transition diagram. The process model comprises data flow diagrams representing states and transitions in the state model.

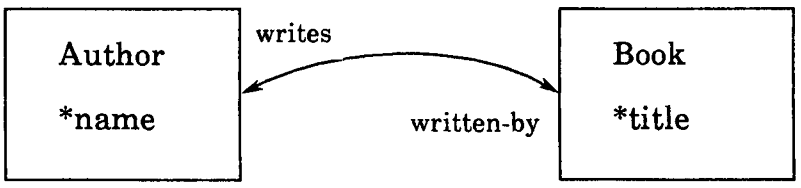


Figure 3.8 An Information Structure Diagram

OOSA provides a data modelling technique and embraces the concept of an object as a record in a relational database. Nevertheless, OOSA fails to account for the vast majority of object-oriented concepts and no new graphical notation is provided for object-oriented system analysis. Instead, the graphical notation is taken primarily from entity-relationship diagrams and data flow diagrams found in others structured methods. Moreover, there is no way to express concepts such as classes and operations, and the notion of inheritance is entirely missing.

Shlaer and Mellor have described a methodology for object-oriented system analysis which is, in many aspects, similar to OMT and OORA. All these methodologies have in common the emphasis on data modelling as the first task to be performed during software development. However, the excessive preoccupation in OOSA with relational databases has little to do with the object-oriented paradigm.

3.2.7 Other Methodologies

The state-of-the-art of object-oriented methodologies is evolving rapidly. As object-oriented methodologies mature, they are likely to borrow ideas from one another. This subsection describes more restrictive and less known object-oriented methodologies.

The Pun and Winder Methodology

Pun and Winder (1989) have proposed a methodology targeted at object-oriented design and programming. The methodology is divided into three levels: *conceptual*, *system* and *specification* levels. The first level assists designers in analysing and examining the application in an object-oriented fashion. The second level concentrates on some important issues of the design process, such as objects and inheritance. Finally, the third level concerns the production of a class structure chart which will be passed to the implementation phase.

The main objective of the conceptual level is to identify the objects and their interactions involved in a particular software system. Designers take the output of a conceptual analysis phase and generate a set of object interaction diagrams. Object interaction diagrams are divided into two layers: the user-interface layer and the user-transparent layer. The user-interface layer contains objects which communicate and interact directly with the end-users of the software system. This layer provides a visual presentation of how a software system appears to its end-users, and menus and forms are depicted as typical interfacing objects.

The user-transparent layer contains specific software system objects which are transparent to end-users who need not be aware of the existence of such objects. Only designers know exactly what these objects are. The separation of the conceptual level into two layers highlights the importance of user interface design during software development. Object interaction diagrams are the documentation of the objects and interactions found in the software system. Thus, at the end of the conceptual level, a list of objects and actions have been identified.

After identifying the objects and operations which are involved in the software system, designers have to build such objects. The system level is where the construction of each individual object takes place. The construction emphasises object instantiation and inheritance. During the process of object creation, every object has to be an instance of some class. If there is no suitable class for an object, designers have to make up a new one. So, at this point, designers have to search and get to know which classes are available. The new class may be a completely new one or it can be either a sub-class or super-class of existing classes. Thus, the creation of new classes may involve inheritance.

The specification level mainly helps designers to set up an implementation for a software system. With the information obtained from the conceptual and the system levels, designers can identify the main objects and their interactions. These interactions are presented in class structure charts. The objectives of class structure charts are to explicitly express class hierarchical structures and lay out operations and messages related to classes.

Some observations concerning this design methodology can be made. Firstly, Pun and Winder have claimed that very often an object can be an abstraction of several objects. This argument leads to the misleading concept of sub-objects rather than sub-classes. Secondly, this methodology always starts software development by identifying objects rather than abstracting the main classes in a software system.

The Lorensen Methodology

Lorensen (1986) has described the rudiments of object-oriented software development by explaining that it is fundamentally different from traditional structured development, such as those based on data flow diagrams and a functional decomposition approach. The methodology suggests the following steps to design a software system:

1) Identify the abstractions from the real-world application. Working from the requirements document, the abstraction process should be performed top-down when possible. These abstractions will be the

classes of the software system. Often the classes correspond to a group of physical entities within the application being modelled.

2) Identify attributes for each abstraction. The attributes will become the instance variables for each class. If classes correspond to physical entities, the required instance variables should be obvious.

3) Identify operations for each abstraction. The operations are specific to each class. Some operations access and update instance variables, while others execute actions unique to a class.

4) Apply inheritance where appropriate. If the abstraction process in step 1 is performed from abstract classes, introduce inheritance there. However, if abstractions are created bottom-up, apply inheritance before going to a higher level of abstraction.

5) Create objects and identify interactions between them. This step defines the messages that objects send to each other. New objects may be required to respond to requests from other objects in the software system.

6) Assess the ability of the design to match the system requirements and work out interactions to satisfy each requirement.

Designers should repeat these steps in order to refine a design. Lorensen has described an alternative methodology to object-oriented design which has evolved from software development using Smalltalk, which directly supports classes and inheritance. The methodology aims to identify and characterise abstractions in a manner that results in a definition of all important classes, attributes, operations, objects and messages during the design phase.

ObjectOry

Jacobson (1987) has claimed to have a full object-oriented development methodology called the ObjectOry. It combines a technique to develop large software systems, called the **block design** (Jacobson, 1986), with Conceptual Modelling (Borgida, 1985) and object-oriented concepts. Jacobson has stated that it is quite natural to unite these three approaches since they rely on similar ideas aiming at, among other things, the production of reusable software components. Conceptual modelling emphasises finding concepts suitable to model a real-world application, and it is appropriate for representing the entities of the application as well as the relationships between such entities. ObjectOry concerns the design of some large scale software systems which have been developed today using techniques such as SADT (Ross, 1977) and Structured System Analysis (Gane, 1979).

The base of the methodology is a design technique named the block design, originating from Ericsson Telecom, which is now widespread within the telecommunication industry. Block design can be seen as assembling a collection of properly interconnected blocks and components, each one representing a packaged service of a software system (see Figure 3.9). The software system is viewed as a group of properly interconnected blocks and components which are selected in a top-down fashion. In a simplified form: the software system is built from a group of reusable blocks. A block may itself be made up of lower level reusable components. Components are standard modules which can be used in many different software systems. Reusable blocks and components are implemented as classes using an object-oriented programming language.

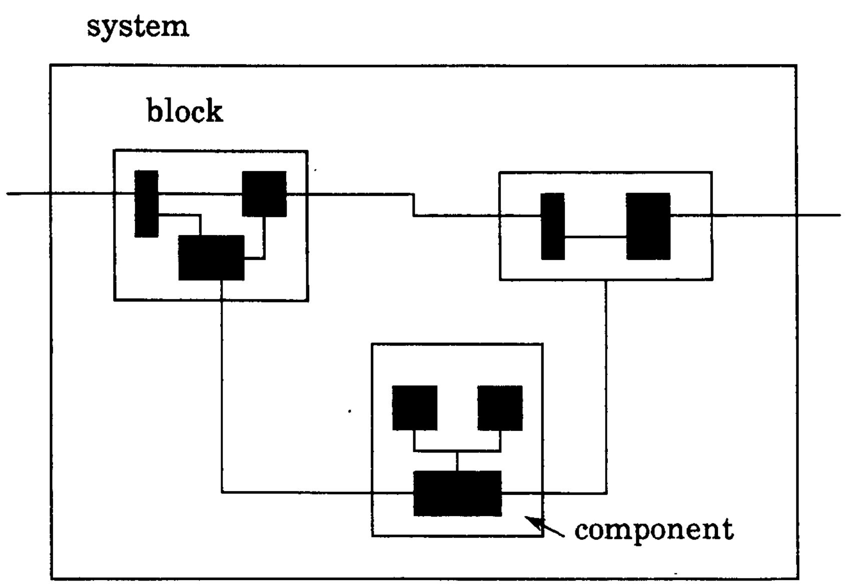


Figure 3.9 An ObjectOry System

The scenario that ObjectOry assumes for building a software system is similar to the manner in which production is carried out in many other engineering disciplines such as house construction or the design of electronic systems. The main criticism that can be made of ObjectOry concerns the fact that this methodology is not purely object-oriented because it is mixed up with other different approaches. Furthermore, the niche where this methodology excels is telecommunications applications because block design was created to address software development in that application domain.

JSD

Jackson has proposed a methodology called the Jackson System Development (JSD) (Jackson, 1983). JSD does not distinguish between analysis and design and instead lumps both phases together as *specification.* The methodology begins by considering the real-world application and first determines the *what-to-do* and then the *how-to-do*. This methodology is intended especially for real-world applications in which timing is important. A specification model describes the application in terms of entities, actions and ordering of actions. JSD also accepts that entities usually come out as nouns in requirements statements and actions emerge from verbs in that description.

JSD has some features which may appear on the surface to be similar to object-oriented design. The main task is to model the real-world application and to identify *entities* (which could be viewed as objects), *actions* (i.e. operations) and their interactions. However, JSD is not fully suitable for object-oriented design because there is very little to support the object-oriented paradigm. For instance, there is no consideration of encapsulation and inheritance.

Multiple-view Object Oriented Design

Multiple-view Object Oriented Design has been proposed by Kerth (1988) as a methodology for structured object-oriented design, and supports the construction of programs from an abstract model with real-time extensions developed by Ward and Mellor (1985). The methodology addresses different issues, including identification of objects and concurrency, and allows concurrent processes to be expressed as tasks rather than objects. The methodology supports an object-oriented approach. However, from its root, it can be concluded that this methodology has its fundamentals based on traditional structured methods for software development. (The MOOD acronym has also been used in this thesis before Kerth's was uncovered).

Graphical Notations for Implementation

Cunningham and Beck (1986) have proposed diagrams for representing Smalltalk programs. In the diagrams, objects are represented by boxes, as shown in Figure 3.10. Each box is labelled by a class name and possibly its super-class. The diagrams also emphasise the representation of the message passing which takes place between objects. When one object invokes a method upon another object through a message, that message is shown as an arc originating in the sending object and landing in the receiving object. The various computations are implemented by distinct methods, each labelled with the method selector.

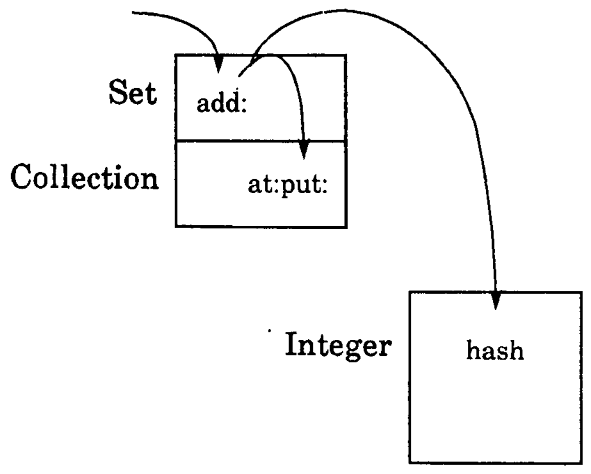


Figure 3.10 Representing Classes and Methods

The diagrams as proposed are only suitable to small programs because the power of representation is weak. Classes are listed with the most specialised class at the top leading some designers to complain that it is not intuitive to place sub-classes above super-classes. Another limitation concerns the restrictiveness of the graphical elements in the diagrams, which are more suitable for representing Smalltalk programs and are more likely to be used during the implementation phase.

As far as C++ is concerned, Ackroyd and Daum (1991) have suggested a basic graphical notation for classes, objects, methods and inheritance (see Figure 3.11). The graphical notation also adds numerous specialised representation for polymorphism, overloading, delegation, static variables and many other properties, which can provide graphical representation for object-oriented programs. Nevertheless, the proposed notation is tied closely to C++ features, particularly those for private, protected and public classes. Besides, it is not apparent how to use the proposed notation for other programming languages nor how to use it as a language-independent graphical notation for object-oriented design. Additionally, the diagrams may depict an extensive representation of implementation details which, makes them hard to use and comprehend for large and complex programs.

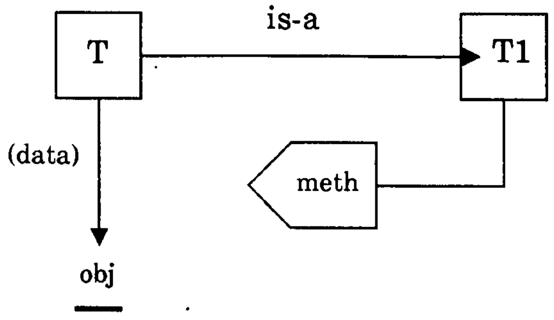


Figure 3.11 Notation for Classes, Objects, Methods and Inheritance

3.3 FINAL REMARKS ON CLASSIFICATION

This chapter has briefly reviewed many object-oriented methodologies and outlined their oversights and weaknesses. There is some dissatisfaction with current methodologies which seem to place too much emphasis on designing for the task in hand and not enough on designing reusable components nor designing with reusable components. Based on this survey, it is evident that further research on object-oriented methodologies is required in order to overcome the deficiencies and limitations of existing software development methodologies.

Most of the early object-oriented methodologies which have appeared either focus on an implementation in Ada (which does not provide inheritance) or disregard abstraction in terms of classes, and instead focus on object instantiation. Other methodologies with the intention of combining existing structured methodologies together with object-oriented concepts have led to the misuse of objects only as data without regarding the operations on that data; the operations are treated separately as functions. Another problem with such combinations is the mapping of concepts from one approach to another. The adoption of new concepts, the change of vocabulary and notation can confuse designers about which one should be used in which phase of the software life cycle.

Therefore, it can be concluded that so far there has been no generally accepted object-oriented design methodology. Only limited object-oriented methodologies have been found. As a result it is the intention of this research to create an object-oriented design methodology which allows designers to apply powerful object-oriented principles to the design of a wide range of applications from the beginning of software development. To put it in other words, **revolution** is what is needed to tackle the problems of software development. New methodologies which exploit the benefits of the object-oriented paradigm within a substitute software life cycle model have to be pursued.

This research is interested in an approach which yields a single coherent object-oriented design methodology, rather than separate methods to solve specific parts of a design. Such a methodology must pay attention to object-oriented concepts already discussed, for instance, classes, objects and inheritance. The proper use of these concepts can lead to a truly general object-oriented design methodology as independent as possible of any programming language. Moreover, reusability should be emphasised as part of the methodology within an alternative software life cycle model. Another objective of this research is to provide a CASE environment to design object-oriented software systems. The environment should offer diagram editors and checkers, and encourage the use of reusable components to build software systems. These issues are covered in the next chapters.

Chapter 4

A METHODOLOGY FOR OBJECT-ORIENTED DESIGN

It has been claimed in the literature that software systems developed using an object-oriented approach can be significantly more elegant than those developed using traditional structured development approaches, and that more software components can be reused during software development. However, using an object-oriented programming language does not, by itself, guarantee miraculous results. Like any other engineering activity, methodologies play an important role during the design of object-oriented software systems.

The early chapters have presented some aspects related to the object-oriented paradigm and the advantages achieved when it is used. A variety of existing methodologies have been discussed and their strengths and weaknesses have been pointed out. As can be seen from previous chapters, although the general principles of abstract data types, modularization and encapsulation are generally accepted as good mechanisms for software design, there is little agreement on either a design methodology or a design notation for representing an object-oriented design. Indeed, so far there has been a proliferation of such notations rather than a single well-known and widely used representation.

This chapter presents a new methodology for object-oriented design called **MOOD**, which is based on, and obtains the benefits of the object-oriented paradigm presented in the second chapter. MOOD supports the design of a software system following an object-oriented approach and it is independent of the idiosyncrasies of any particular programming language. The chapter contains five sections. The first section places MOOD into the context of software development. In broad terms, MOOD starts with an abstract model of the application and ends with a design model to be implemented. Between these two representations there are steps which guide the designer throughout software design.

The steps which must be followed in order to design a software system using MOOD are discussed in the second section. Such steps help the designer identify classes, build class hierarchies using inheritance, identify objects and understand the behaviour of the software system, as well as represent the design model graphically with four types of diagrams. This section also describes a means for partitioning a large software system into manageable pieces. This partitioning is based on the functionality of the software system, which in turn is related to classes which provide that functionality.

The third section presents an example of a partial software design in which MOOD has been applied and discusses some benefits and drawbacks of the methodology. The results of the experience of applying MOOD provided valuable information about its applicability and have been used as feedback to improve the methodology and to support it within a software development environment. The fourth section presents the main requirements for a CASE environment to support an object-oriented design methodology. A prototype of such a software development environment is presented in Chapter Six. This chapter concludes with final remarks on MOOD, outlining some of the issues which should be considered when designing and representing an object-oriented software system. In addition, the last section presents a summary of the MOOD steps.

4.1 THE CONTEXT OF MOOD

The designer of a software system should start the design from an abstract model of the application, which is accomplished through a system analysis. That abstract model is the first representation of the software system. But what should the system analysis be? System analysis is characterised by obtaining information about the application, and hence this information is non-structured, often incomplete and sometimes contradictory. The system analysis then produces a description in terms of the requirements and objectives of the software system which can be refined through the addition of details to that abstract model.

Methodologies which aim to give support for system analysis should consider the possibility of dealing with an incomplete abstract model and describing partial aspects of the application, then refining and complementing that abstract model. The result of system analysis comes as a graphical or textual, informal or formal, abstract model of the application. The more complete and consistent it is, the better. Therefore, system analysis is a means for understanding the application. The purpose of an abstract model is to provide a description (graphical or textual, informal or formal) of the application.

MOOD commences with an abstract model of the application as input and supports the production of a design model as output. The design model comprises an information model and a behaviour model. The information model is a static representation of the software system using a set of diagrams which represents a global view of the software system classes and components. The behaviour model depicts the dynamic of the software system. Figure 4.1 shows, within the dashed rectangle, the context of MOOD. Although the boundary between each representation is usually fuzzy, the abstract model of the application, the design model and the program are three different, yet related abstractions of the software system.

MOOD consists of a sequence of steps which help the designer to refine an abstract model of the application. By using MOOD, the designer could begin by identifying classes and considering object interactions. It has to be emphasised that the use of MOOD is aimed at maintaining as close a relationship as possible between the application and the object-oriented software system by representing real-world entities only through classes, objects and inheritance.

It is sometimes affirmed that the distinction between system analysis and design is that system analysis states "What?" is to be built and design represents "How?" it is to be built. Therefore, during the system analysis, the question "What is the software system supposed to do?" should be answered, while during the design the question "How should the software system do what is stated during the system analysis?" would be addressed. This

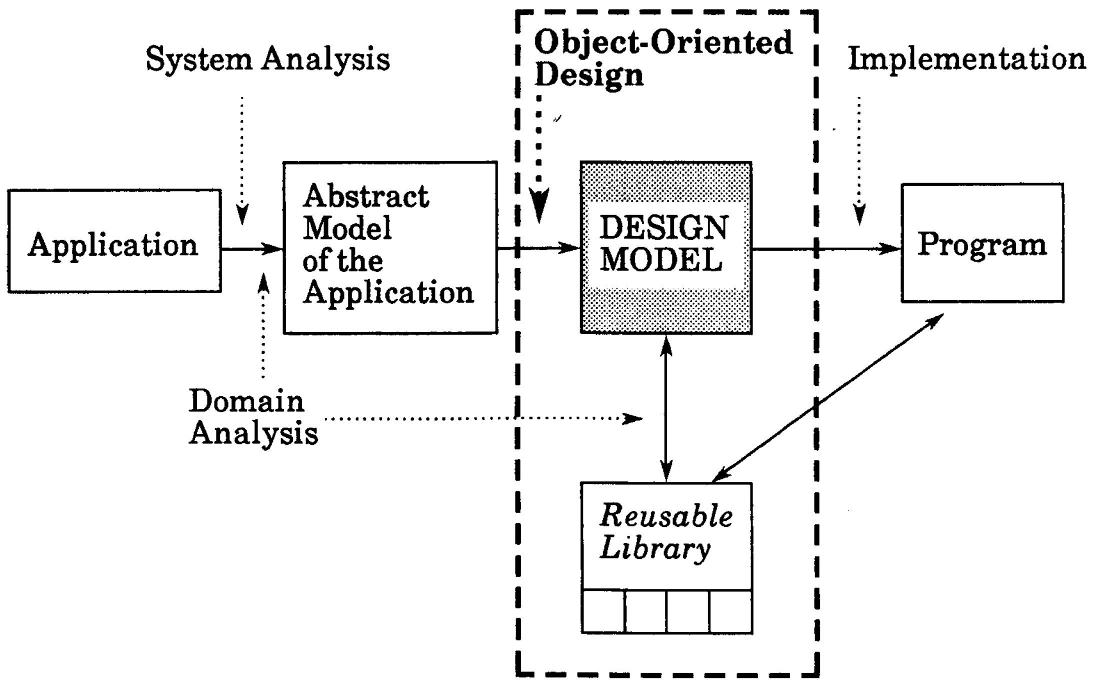


Figure 4.1 The Context of MOOD

distinction contains some important truths but ignores other subtle issues. For instance, is there any definite boundary between system analysis and the design process within an object-oriented framework?

This is difficult to answer because software development may be seen as a process of creation, manipulation and refinement of abstractions. When creating or transforming an abstraction, the designer actually deals with different representations of these abstractions. In order to make the refinement from one abstract representation to another as simple and as free from error as possible, it must be easy to relate one abstract representation to the next. This requires abstractions to be related to one another and concepts which were introduced in one of the abstractions should be found in the other abstraction. The object-oriented paradigm allows the designer to create abstractions that are close to the application, and to manipulate these abstractions throughout software development. Therefore, within an object-oriented framework, it is even more difficult to draw a distinct line between system analysis and design.

As far as the reusable library is concerned (see Figure 4.1), this contains a collection of reusable components, from both application and solution domains, put into and taken from there during software development. The use of a reusable library aims to identify components which can be reused in the developing software system. A reusable library requires a good way of classifying, storing and recovering components. The reuse of such components can take place in two ways: either directly when the component corresponds exactly to that required, or through inheritance when there are some differences, and specialisation and/or generalisation are necessary.

Most software systems are concerned with an application domain, and the identification of the vocabulary for the application domain is the main role of domain analysis. Domain analysis also plays a fundamental role in identifying potentially reusable components within an application domain because it helps the designer to establish a vocabulary for a given application domain and therefore components which are related to software systems within that application domain. A comprehensive hierarchy of classes for an application domain provides the designer not only with application domain reusable components, but also potential parts for a software system. The essence of a good object-oriented design is: design with reuse as well as design for reuse. That means finding application domain reusable components which can be taken from reusable libraries as well as producing solution domain reusable components which may be stored in a library.

The designer cannot be expected to have a perfect understanding of the software system at the beginning of the design. Rather, that understanding evolves through iterations and refinements with constant feedback, but each iteration makes the design model increasingly clear. A top-down fashion for software development generally creates the software system by successive refinements of software system components. If the software system is not trivially simple, it should be decomposed into large components which may be further decomposed. Nevertheless, it is not easy for the designer to divide a software system into components if the software system is almost unknown. It could be argued that the top-down decomposition only works when the designer already knows the application domain. In contrast, a bottom-up approach would be more appropriate. These issues will be further discussed in Chapter Five, but for now the MOOD steps are introduced.

4.2 THE STEPS FOR OBJECT-ORIENTED DESIGN

In order to produce the design model following an object-oriented approach, it is necessary to identify and represent:

- software system classes;
- inheritance between classes;
- software system behaviour in terms of object interactions.

An important aspect of MOOD is how the designer tackles the problem of designing in successive steps in order to produce the design model to be implemented subsequently. Briefly, the fundamental steps proposed by MOOD are:

- divide the software system into manageable components;
- identify classes and/or objects which model the application;
- identify inheritance between classes;
- represent classes and inheritance;
- identify software system behaviour in terms of objects and operations;
- represent software system behaviour in terms of object interactions.

The MOOD approach to tackling these steps is discussed in the next subsections. It is important to note that the sequence in which these steps are carried out depends on the knowledge that the designer has about the application domain, but this issue is discussed in the next chapter.

4.2.1 Representation of the Design Model

Graphical notations have been an integral part of most software engineering techniques. In fact, there are very few software engineering activities which do not benefit from some forms of graphical notation. The use of graphical notations has proved to be an effective mechanism for expressing a design because a clean graphical notation can show the architecture of a software system clearer than a textually-based notation.

A systematic approach to software design can be offered by a set of guidelines supported by a graphical notation used to represent a software system. A good graphical notation should be simple, straightforward and a reflection of the paradigm used to build the software system. Thus, a graphical notation chosen for object-oriented design should directly support the object-oriented paradigm, by providing representation for classes, objects and inheritance.

An important aspect of MOOD is the graphical notation which allows a straightforward visual representation of an object-oriented design. The graphical notation to be presented later in this chapter comprises different types of diagrams which depict the features of a generic object-oriented representation that goes hand in hand with MOOD. The diagrams are independent of any programming language, provide a high level representation of a software system design and span a broad range of object-oriented concepts without making any assumptions about implementation. The guiding principle behind the proposal of these diagrams has been to keep the diagrams as simple as possible.

There are four main types of diagrams:

- **Composition Diagrams**: these represent the composition and decomposition of a software system in terms of its components.
- **Class Hierarchy Diagrams**: these are a simple but effective way to display classes, their attributes and operations, and inheritance relationships.
- **Object Diagrams**: these show relationships among objects based on requests for operations between them.
- **Operation Diagrams**: these depict how operations are combined to provide particular software system functionality.

Each diagram proposed by MOOD and used during the design should be associated with a unique identifier. It is recommended that the identification should give a clue to the type, the number and the name of each diagram. The identification for a composition diagram could start with "CO", class hierarchy diagrams with "CL", object diagrams with "OB" and operation diagrams with "OP"; followed by the number and name of the diagram.

These diagrams are the principal means for representing the design using MOOD. They constitute the graphical representation of the software system design as a whole and can be viewed as communicating meaning between designers. The diagrams are composed of simple symbols, and their automated support in a computer-aided software development environment will be presented in Chapter Six. The number of different basic symbols is small, the symbols are unambiguous and the visual impact of the arrangements of the basic symbols connotes the semantics of object-oriented concepts.

These diagrams are integrated with each other, and the information present in a particular diagram must be consistent with that information used in another diagram. For instance, the operations used in a certain operation diagram must be defined in any class of another class hierarchy diagram. Such interrelationships between diagrams makes them a powerful collection of graphical notations capable of representing a whole design. The next subsections show how to build up these types of diagrams which are the means by which the designer represents the software system at design level when MOOD is used.

4.2.2 Identification of Components

The development of large software systems imposes some characteristics on the design process. Often, the design of a software system of any significant size must be divided among different designers or several designers grouped into small teams. A large software system will probably require numerous components which make such a software system difficult to comprehend. The complexity of a large software system can be mastered by first identifying large software components based on the functionality required from the software system.

When should identification of components be introduced? This depends on the functionality of the software system. Large components are more likely to be identified during the early stages of a design and can be further divided into sub-components. In a small software system, where just a few services are provided, identification of components may not be necessary. However, a large and/or complex software system, providing many different and complex services, needs decomposition to be applied from the beginning of the design. Nevertheless, finding a good division into components for an object-oriented software system still seems to be subjective.

Composition Diagrams

Composition diagrams can be used to represent all or part of a software system in terms of its components. Composition diagrams show aggregations of components, where components can be any software item such as subsystems, modules, classes or objects which make up the software system. A component may also be subdivided into other smaller sub-components and vice versa, for instance, interdependent and cooperating components can be composed to offer a particular service.

Each component in a composition diagram fulfils the role of a software item in terms of the services provided by that component. Figure 4.2 illustrates a composition diagram in which a CASE environment software system can be composed of three different components representing the tools (*graphiceditor*, *texteditor* and *checkers*) of the environment. That figure also shows how the *checkers* can be composed of its sub-components (*consistency* and *completeness*). Each component is represented in the diagram by an ellipse. It is not the intention of composition diagrams to capture all the details of a software system, rather the purpose is simply to represent broadly the components of a software system. The relationship between large components and classes will be discussed later in this section.

4.2.3 Identification of Classes

Classes are the most important concept for object-oriented design. Identification of classes in MOOD involves the recognition of important classes in an abstract model of the application. Nevertheless, there is no easy and fast way of defining what is and what is not a good class. Part of the identification process is to assess the consequences of including or excluding

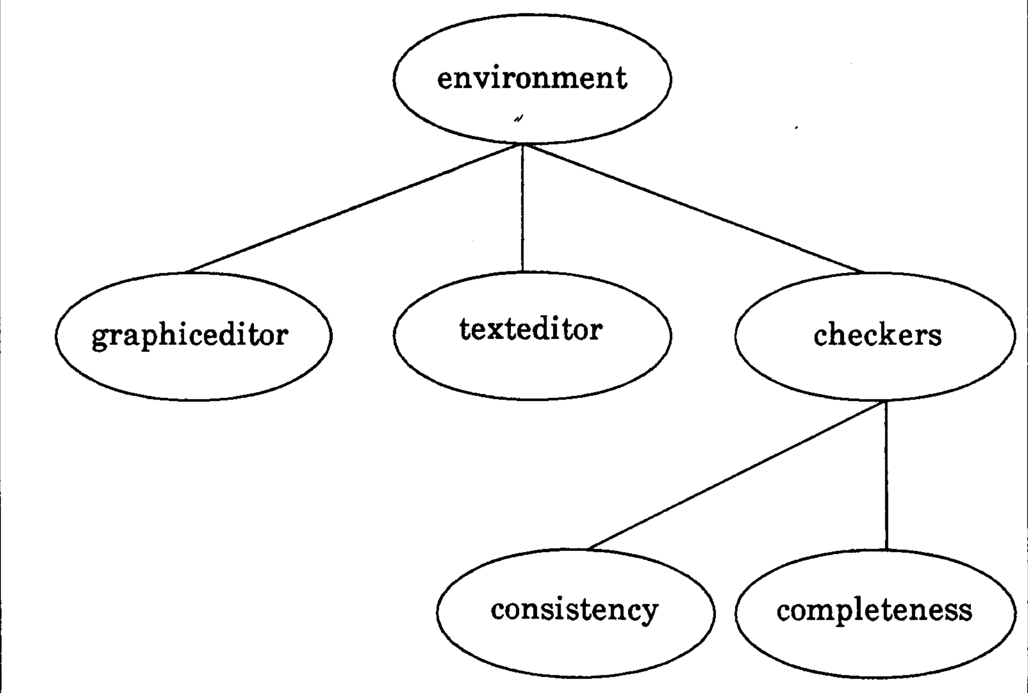


Figure 4.2 A Composition Diagram for an Environment

potential classes. An abstraction in terms of classes depends on the purpose the classes play in the software system as a whole.

Finding classes requires several iterations before a suitable collection of classes can be determined. This is part of an iterative process for object-oriented software development to be presented in the next chapter. Iterations are not a sign of bad design and should be regarded as a healthy process by which learning takes place. The number of iterations depends on the knowledge that the designer has about the application domain, and the intuition and skills of the designer, which are gained through insight and experience. As a result, it is only possible to propose guidelines which assist the designer in determining the classes for a particular design.

Firstly, the designer needs to understand the concept of a class and what may be a class in the software system. Candidate classes can be:

- abstractions of things: e.g. books, planes, sensors;
- abstractions of people: e.g. students, engineers;
- abstractions of concepts: e.g. departments, graphics, flights, accidents;
- roles played by things, people or concepts: e.g. people are *voters* for politicians or *taxpayers* for the government;
- relationships between abstractions of things, people and concepts: e.g. people *purchase* things, people *marry* people.

As described in the second chapter, other important concepts concerned with a class are attributes and operations. As far as this step of MOOD is concerned, the designer should also consider the abstract model of the application and take into account services provided by a class, which can be identified as its operations. In order to identify the services, for each class, the designer should answer these questions:

- Which operations can be performed on instances of this class?
- Which actions do the attributes of this class undergo?

At the end of this stage, a set of classes (which provide abstractions for conceptual entities of the real-world application) and their attributes and operations should be identified. Thus, the designer can view the software system design represented as a collection of classes. The graphical notation which has been developed to represent classes is discussed in the next subsection.

4.2.4 Identification of Inheritance

Identifying classes is only the first step in designing an object-oriented software system. After some classes have been identified, they can be related to one another to form class hierarchies. The notion of inheritance plays a key role during object-oriented design because it helps the designer to derive new classes from primitive ones by exploiting their commonalities in terms of attributes and operations, and build up class hierarchies.

Inheritance enables the designer to create a new class simply by specifying the differences between a new class and an existing class instead of starting from scratch each time. As a design technique, the use of inheritance is similar to a *stepwise* refinement approach where inheritance divides the classes which model an application into hierarchies of classes. Thus, inheritance can be used by defining a new class to be a specialisation or generalisation of existing ones.

Although formal or automated techniques for refining class hierarchies do not exist, there are a few rules which might help reorganise a collection of classes into hierarchies. The main rule for building a class hierarchy is to identify common attributes and operations and migrate them to a super-class, then eliminate from the super-class those operations that are frequently overridden in its sub-classes rather than inherited by these sub-classes. This rule makes the super-class more abstract and hence more generally useful.

In order to identify inheritance, the designer must settle upon a collection of primitive classes from which all others can be derived by using the generalisation and specialisation mechanisms. A common example of inheritance by generalisation is a *basic-screen-window* class which has been created as a super-class from the *coloured-window* and *framed-window* sub-classes. Such a super-class embodies the minimal characteristic of all screen windows. The two sub-classes could add greater functionality to the *basic-screen-window* class by defining such attributes as *foreground* and *background* colour, and *border-width.*

As far as inheritance by specialisation is concerned, the designer should try to identify classes that cannot properly describe all their instances. In this case two or more specialised sub-classes can be derived. It is possible to specialise in the sub-class the general properties defined in the super-class. A good example of inheritance by specialisation is the following: consider a *vehicle* class, with attributes *licence-number*, *maker* and *model.* This class can be further specialised into *freighter*, *bus* and *car* sub-classes. This hierarchy of classes could be depicted textually using the following indentation scheme:

- *vehicle* class: *licence-number*, *maker* and *model* attributes;
  - *freighter* sub-class: *licence-number*, *maker*, *model* and *permitted-load* attributes;
  - *bus* sub-class: *licence-number*, *maker*, *model* and *number-of-passengers* attributes;
  - *car* sub-class: *licence-number*, *maker*, *model* and *nationality* attributes.

The *freighter* class could be specialised into *van* and *lorry* sub-classes. The attributes of the *vehicle* class are inherited by its sub-classes so that, for instance, a *van* object has the *licence-number* attribute (from the *vehicle* class), as well as the *permitted-load* attribute (from the *freighter* class) besides its private attributes. A *van* object could now be handled either as a *vehicle* object, as a *freighter* object, or as a *van* object; the different viewpoints imply restricted visibility of the attributes, and make it possible to handle all sub-classes simply as a *vehicle* class, whenever that is desirable. Attributes and operations of the super-class are available to their sub-classes and new attributes and operations can be defined within the sub-classes. Overriding and renaming of attributes and operations are permitted to adjust the sub-classes to a particular context.

After having identified the primitives classes which may form the basis for the software system, the designer can also use *set theory* in such a way that intersections between sets of attributes and operations of different classes can be singled out and inheritance identified. **Venn diagrams** are a convenient means of representing *sets* and can be used as a technique to help in the identification of inheritance as well. Such diagrams increase the understanding of inheritance by treating classes as sets of attributes and operations. Venn diagrams can give a hint of possible super-classes by showing which attributes and operations are held in common among classes. If two or more classes have some attributes or operations in common, they could inherit them from a common super-class. If that common super-class does not already exist, the designer should create one and move the common attributes and operations to it. By using Venn diagrams, inheritance is founded on the underlying mathematical basis of *set theory*.

The procedure for identification of inheritance from Venn diagrams is as follows: for each class define a set comprising the attributes and operations associated with that class and look for intersection between these sets. The intersection may define a new super-class. Figure 4.3 shows some examples of identification of inheritance from Venn diagrams. The class hierarchy is created by pulling up the commonalities and by pushing down the differences. The main goal of this procedure is to place common attributes and operations as high as possible in the class hierarchy so that more sub-classes can share them.

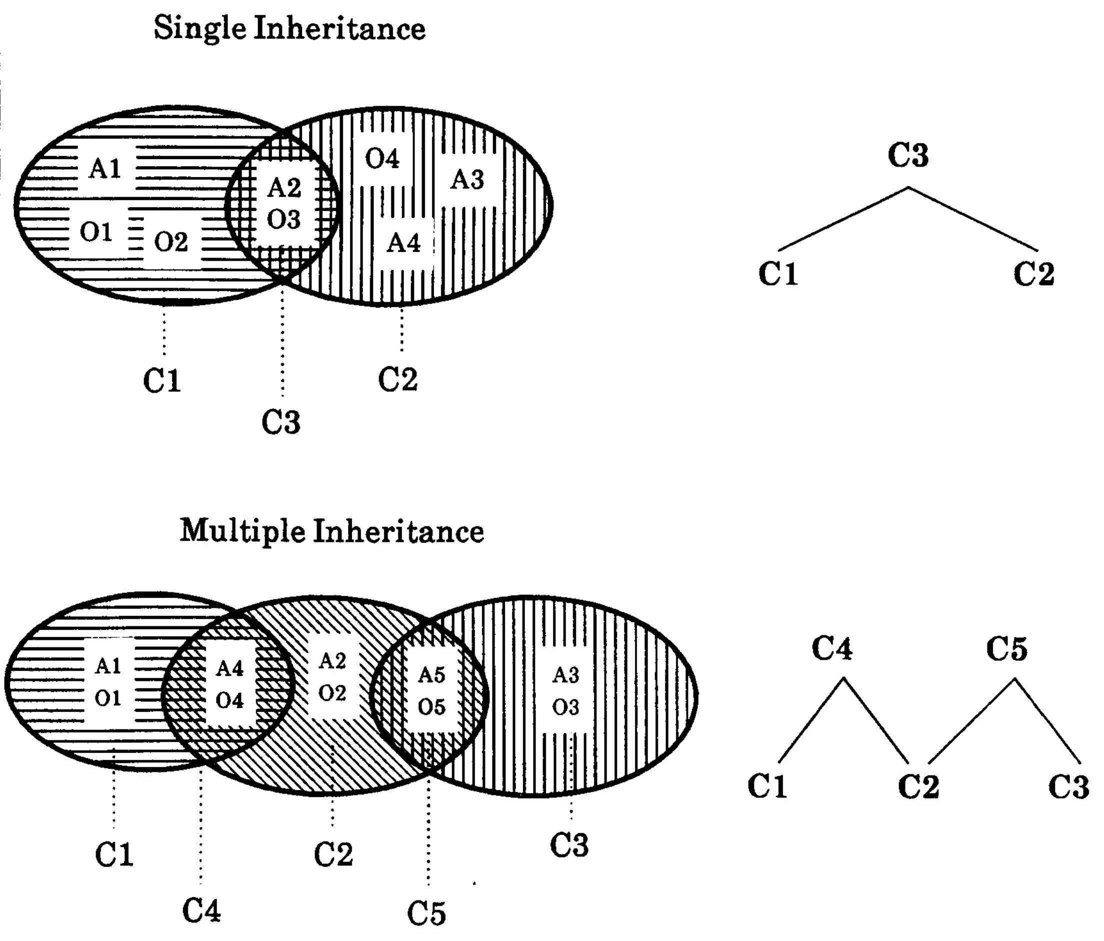


Figure 4.3 Identification of Inheritance from Venn Diagrams
(A: Attributes, O: Operations, C: Classes)

The following kinds of changes to the collection of classes are to be expected during the evolution of a design, while the class hierarchies are being built:

- add new classes to the collection;
- change the attributes and operations of a class;
- reorganise the class hierarchy using the specialisation and generalisation mechanisms.

The designer may change a class either to add or delete some attributes or operations. The designer can also define new classes when new key abstractions are discovered. The reorganisation of the class hierarchies takes the form of changing inheritance relationships, adding new generic classes and shifting attributes and operations in the class hierarchies. Reorganisation happens frequently at the beginning of the design and then stabilises over time as the designer better understands the key abstractions.

Class Hierarchy Diagrams

Class hierarchy diagrams in MOOD show the existence of classes and their relationships in a design of a software system. Such diagrams depict how classes are arranged hierarchically. The designer can use class hierarchy diagrams to represent static aspects of the software system, by using only the fixed set of symbols as illustrated in Figure 4.4. A rectangle represents a class and is annotated with its mnemonic name. A class name is indicated in a class hierarchy diagram followed by its attributes and operations. A class may restrict its attributes and operations from other classes by specifying them as *public* or *protected.* Any other attributes and operations in a class will be regarded as private, that is, part of the implementation details of that class and not visible in the class hierarchy diagrams.

A rectangle representing a class displays the externally visible (*public*) attributes and operations that instances of that class will possess. Sub-classes derived from a super-class can, through the application of inheritance, make direct use of the public attributes and operations of the super-class. As well as public attributes and operations, a class designer may wish to export additional attributes and operations which can only be used by sub-classes. These attributes and operations of a class may be labelled *protected*, and will not be part of the normal interface of an object of that class. In the class hierarchy diagrams, protected attributes and operations are preceded by an asterisk ('*').

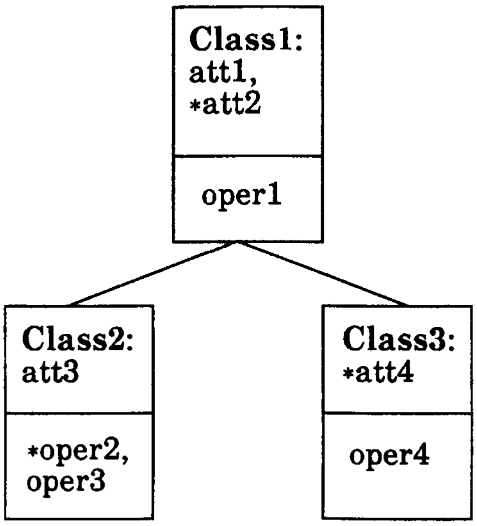


Figure 4.4 A Generic Class Hierarchy Diagram

Relationships between Class Hierarchies and Components

It is possible to divide a software system following an object-oriented approach by partitioning it into independent components and making a correspondence between the services provided by each component and the operations of different classes in the class hierarchies. Large components identify functionality at a high level of abstraction, to be fulfilled by operations of different classes, and a class is associated with a component only if it contributes to the functionality provided by that component.

Large components will represent a collection of logically related classes, each providing different services which, when put together, provide the functionality required from the software system. A particular component has nothing to do with a specific class hierarchy, but with the functionality of the software system. In fact, composition diagrams are orthogonal to class hierarchy diagrams. Large components may use the services of several classes placed in different class hierarchies to contribute to the overall functionality of the software system.

The correspondence between large components and classes defines a **context** where functionality is provided by a set of operations of different classes, as depicted, within the dashed lines, in Figure 4.5 (in that figure, the dotted lines represent links between operations on objects of different classes). A context consists typically of a set of classes, connected together by inheritance relationship and a well-defined pattern of interaction between the objects of these classes. A context groups classes together so that a portion of the overall system functionality can be provided. Contexts will maintain traceability between classes and functionality; indirectly, a context shows the classes related to a component.

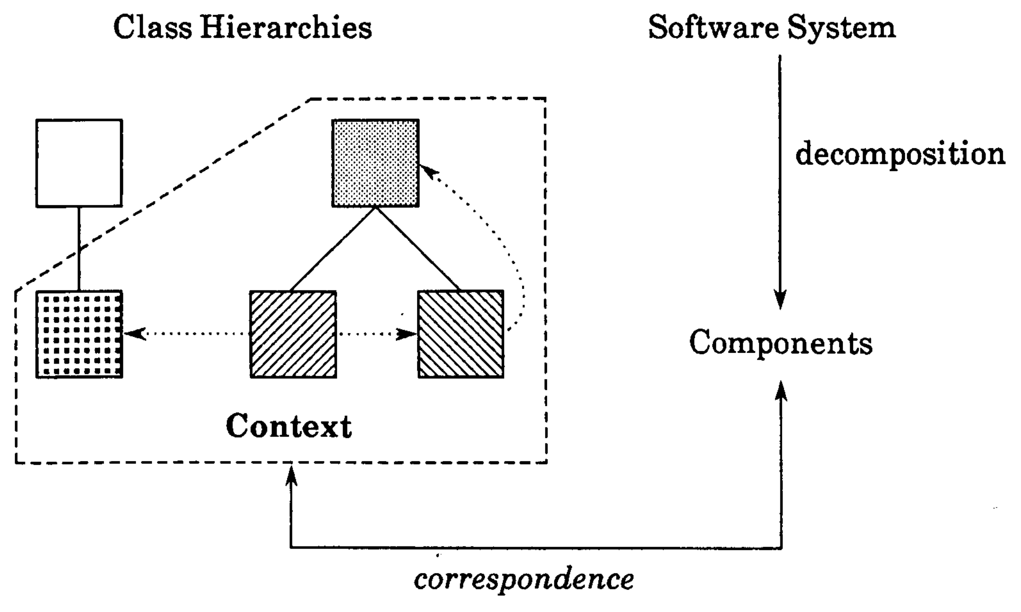


Figure 4.5 Correspondence between Components and Classes

4.2.5 Identification of Objects

This step is concerned with objects by focusing on when and how they are created, destroyed, accessed and changed. An object in the sense of an object-oriented design is a set of operations gathered together with attributes, and instantiated from a class. Some characteristics of object-oriented design regarding objects are:

- objects are instances of some class;
- objects can be created and destroyed;
- operations are related to objects;
- attributes are encapsulated into objects so that other objects can only access them via operations;
- requests are sent between objects to invoke operations or return results;
- functionality can be provided by operations on several objects.

Concrete entities in an application domain, such as people and things, are very likely to be objects in the software system. Therefore, they should be the initial targets to be identified as objects. Identification of objects involves:

- understanding the application;
- depicting its main entities as objects;
- identifying the attributes of an object;
- identifying the operations on an object;
- focussing on interaction between the objects.

Identification of objects is also important because it helps define the dynamic behaviour of the software system in terms of object instantiation. The control flow of the operations, which defines the sequence in which the operations are requested can also be identified. The representation of software system behaviour consists mainly of a network of objects manipulated by operations.

Each object has its own internal states and is linked to the network through requests that establish the sequence of the operations.

In order to understand the interaction among objects, it is necessary to:

- identify the operations requested by an object;
- identify the operations provided by an object;
- identify the relationships between the objects;
- identify the exceptional conditions within the operations;
- identify the operations that can change the state of an object.

All objects go through various states during their lifetime, which means that they assume different states, defined by the values of their attributes. An object is in exactly one state at a time and its current state is recorded in its attributes. Only an operation can change the state of an object.

The states that an object can go through can be represented in a **state transition diagram**. Such a diagram shows the states of an object and the events which cause a transition from one state to another. A state transition diagram can depict the behaviour of an object over a period of time through the states which an object assumes and the events which cause that change of state. The designer should look at the objects, viewing them as state machines and set the events applicable to each state. State transition diagrams may be used as a technique which can help the designer to identify the operations on objects because the events that provoke the change of state of an object can be related to operations on that object. Each event may have conditions associated with it, and these conditions must be satisfied in order to have an operation triggered.

Figure 4.6 represents a state transition diagram for a *screen-window* object. In that figure, the labels in bold represent the various states which a *screen-window* object can go through. The labelled arrows represent events and possible operations on that *screen-window* object, and the effect that these events have on the state of that object. Some operations cause a change of state only when the object is in a certain state, for example the *edit* operation has no effect when a *screen-window* object is closed. These constraints helps the designer to understand the behaviour of the software system.

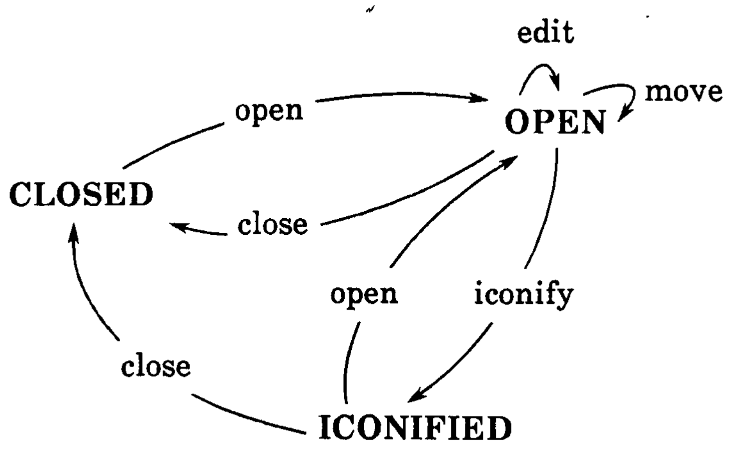


Figure 4.6 A State Transition Diagram for a Screen-Window Object

Object Diagrams

The graphical notation to be introduced in this subsection helps the designer to capture the behaviour of a software system by using object diagrams. Such diagrams show objects and the requests for operations on other objects. A request relates one object to another when one object requires another object to perform an operation. A request also depicts the dependency between a client object (the one which requests an operation) and a server object (the one which performs that operation). An object diagram is simply a network in which the nodes represent objects, and arcs connecting them represent operations. The interactions between objects happen one at a time; concurrency and parallelism are not considered in the object diagrams.

Figure 4.7 shows an example of an object diagram. An object is represented by a circle containing its identification and the class name from which that object has been instantiated. Each object has a unique identifier that allows it to be referenced unambiguously within its class. A line entering an object represents a provided operation and a line leaving an object represents a requested operation. Object diagrams partially define the overall pattern of communication in the software system because they show which objects request which operations from other objects.

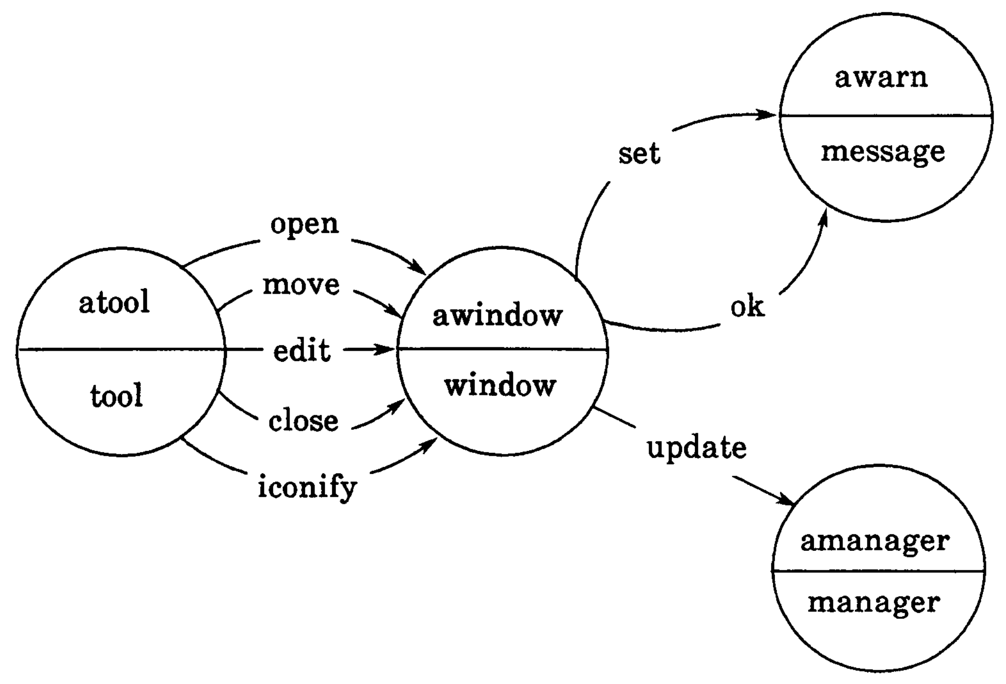


Figure 4.7 An Object Diagram for Objects in an Environment

The object diagram represented in Figure 4.7 conveys the behaviour of some objects when a hypothetical tool is used in a CASE environment. When an *atool* object manipulates an *awindow* object, the *window* object in turn can display a warning message *awarn* object and gets its state updated in a window manager *amanager* object. When an *atool* object is used, it can open an *awindow* object, which can also be moved, edited, closed and iconified. An *awindow* object can also set a warning message *awarn* object to appear on the screen and this disappears after an *ok* operation occurs. A window manager *amanager* object can receive a request to update the position of an *awindow* object. It can also be seen that a client object (an *atool* object) controls the requests to another server object (an *awindow* object).

4.2.6 Identification of Software System Behaviour

An object-oriented software system can be regarded as having a group of objects together with a design which establishes a detailed order of interaction on and between objects. These interactions define the behaviour of a software system. It is helpful to define the dynamic behaviour of a software system in terms of the control flow which defines the pattern of interaction between objects. Therefore, software system behaviour focuses on when objects interact with other objects and when and how objects are created, destroyed, accessed and changed. Thus required functionality of the software system is realised as patterns of interactions between objects.

The designer should consider the sequence in which the operations must be performed. In order to identify the behaviour of a software system, three points are important:

- which operations are performed;
- which attributes are manipulated;
- when the operations are carried out.

Answering the following questions is a useful way to characterise the behaviour of a software system:

- Which objects adequately provide particular functionality?
- When are objects created and destroyed?
- When are objects accessed and changed?
- Which request is needed in order to provide a required service?
- Which services do the objects participate in?
- When does an interaction between two objects take place?

As the design details are expanded, it is normal to realise that in order to define the behaviour of a software system and accomplish a particular service, several operations could be involved, which means that one operation of one class should invoke other operations on objects of other classes and may get some results. The pattern of these invocations establishes the control flow of the software system and shows how a particular service can be accomplished. There are basically two different trends which can be used to express the control flow of an object-oriented software system, namely centralised and decentralised control flow:

- For a centralised control flow, there is always a client object which acts as a scheduler. When a server object finishes executing an operation, the control returns to the client object before the next operation is requested.

- For a decentralised control flow, there is no unique object which can give an overall view of the activity of the software system, but the control flow is spread out among several objects. To invoke an operation, a request is sent to an object. Such an object can send other requests to other objects and the control flow passes between objects until the execution of the software system is over.

At the end of this step software system behaviour is identified in terms of interactions between objects. The definition of the behaviour of a software system is completed when the main objects in the software system have been identified and the pattern of interaction upon these objects has been defined. Operation diagrams, as shown below, can depict software system behaviour.

Operation Diagrams

The functionality of a software system is defined by the services offered by that software system, and can be provided by an operation or a combination of operations. A class alone often cannot provide enough operations to meet the required functionality. Therefore, a set of operations need to be formed so that a service can be offered. For instance, to provide the "landing" functionality for an object of an *aeroplane* class, it is necessary to request operations to manipulate the ailerons in its right and left wings. These wings might be objects of two *leftwing* and *rightwing* sub-classes of a *wing* super-class, in which the operations to control the movements of the ailerons are defined. Thus, the "landing" functionality is provided by requesting operations of two other classes (*leftwing* and *rightwing* classes).

An operation diagram is a graphical representation showing how operations can be combined. Operation diagrams are important because they show:

- which operations are used, and when and where they are used;
- relationships between operations of different classes;
- which operations are needed to offer particular functionality.

An example of an operation diagram is represented in Figure 4.8. A square represents an operation and is annotated with the operation name together with the class name where that operation is defined. The example shows the use of an operation diagram representing the drawing of a polygon on a screen window with its possible associated label, and with the polygon then being stored in a database.

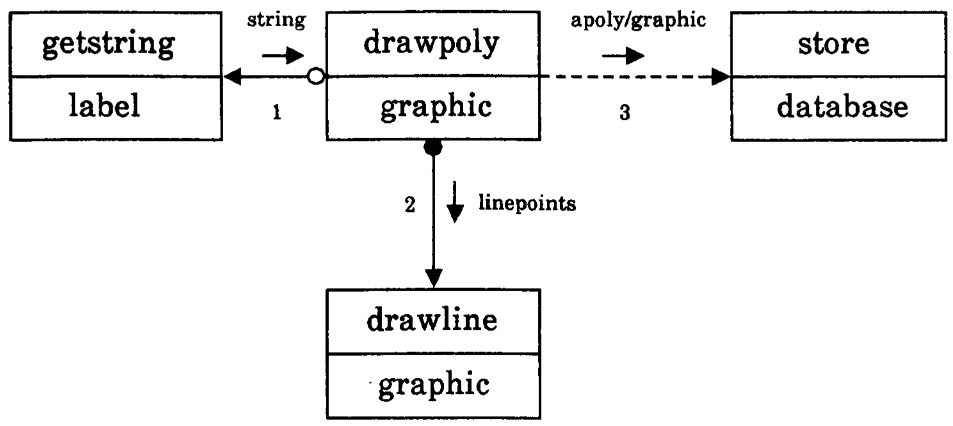


Figure 4.8 An Operation Diagram for Drawing a Polygon

In an operation diagram, a solid arrow means that a caller operation requests a called operation and gets the control flow back after the called operation has been completed. A dashed arrow is used when a caller operation requests a called operation which retains the control flow or passes the control flow to yet another operation while the operation is being carried out. An open circle at the beginning of an arrow means a conditional passage of control flow whereas a filled circle represents an iterative passage of control flow. The numbers dictate the sequence in which the operations are requested in that operation diagram. These conventions help represent the control flow of the software system.

Because operation diagrams focus on the control flow of the software system, they also have to show information and results that might be passed between operations. Each operation has a signature which determines the parameters involved in the requesting of that operation. Parameters are split into input and output parameters. Input parameters are provided by the caller operation and can just be used by the called operation. Output parameters can only be updated by the called operation and are available to the caller operation. Parameters are represented by small arrows beside the operation which they are related to (see Figure 4.8). Besides, objects can also be parameters and in this case the class to which the object belongs must appear after the identification of the object, separated by a slash. Thus, operation diagrams also enable the designer to trace the information flow.

4.3 FIRST EXPERIENCE WITH MOOD

This section presents an example showing how the designer can identify and represent classes and inheritance during the design of an Electronic Mail System following the MOOD steps. This software system is sufficiently complex to show some of the design principles behind MOOD.

4.3.1 An Electronic Mail System

An Electronic Mail System (EMS) should be a unified environment for dealing with electronic correspondence and could contain a number of other facilities to make the processing of electronic messages easy. A simplified list of requirements for the EMS may be as follows.

Each user of the EMS has a unique identifier, called the electronic mail address. An electronic mail address consists of two parts separated by "@": the recipient name and the domain name. An example of a valid electronic mail address might be L.F.Capretz@newcastle.ac.uk. Basically, users of the EMS may send messages to other users and read messages that other users have sent to them. Information sent through a message can be read by the recipient user at any terminal with access to the EMS. As well as being able to send and receive a message, the EMS also includes facilities which allow, for instance, users to get help on how to use the system.

After receiving a message in a special box for incoming messages called the mailbox, the user may read that particular message, redirect that message to another user or save that message into a box of messages for later reading. Several messages may be kept in standard structures called the messagefiles which are similar to separate folders of messages. When the user quits the system, unread messages go back to the mailbox. A message may be deleted by redirecting it to a special wastebin owned by the user, and the messages remain there for some time before being permanently discarded. Thus, messages can be recovered from the wastebin before the wastebin is emptied. The user is presented with a menu from where all of these services or facilities can be chosen.

It is also important to be able to set up a personal mail profile which specifies mail filtering actions to be carried out on incoming messages. The user may define in a mail profile that the messages from a particular user or the messages with a particular subject must be processed by a filter before being put in the user mailbox. This notion of mail filtering is a powerful one which is also intended to support automatic mail redirection and classification according to the preference of the receiver.

These sketchy requirements do not provide enough information about all the attributes and operations for the possible set of classes for the EMS. During the design process it is expected that more knowledge about this application is gained and hence more details added, such as the subject and date of a message. In the next subsections, early steps recommended by MOOD are applied in order to obtain some class hierarchies for the EMS.

4.3.2 Identifying and Representing the EMS Classes

At this step, the requirements presented above should be taken by the designer as a guide to produce an abstract model of the application which can be used by the designer in the identification of classes. The abstract model of the application sets up the functionality and facilities that the software system must provide and it can also be seen as an abstract solution for the software system which offers electronic mail facilities. The designer could select, in principle, the following classes:

- *message* class because the basic entities in the EMS are messages;
- *messagefile* class to keep different types of messages;
- *menu* class for the user interface.

Following this first identification of the main classes, the designer can realise that the EMS has some independent classes, and then identify some general operations which can be applied to objects of these classes. Recalling from the requirements that the user views the EMS as a software system which provides some facilities such as:

- send a message;
- receive a message;
- filter a message;
- save a message;
- get help.

These facilities could be seen at a first glance as operations on the objects of those previously identified classes.

The *message* class is one of the most important in the EMS because the main services are related to messages. At this point, the designer has a rough idea about attributes of a message; and, in conjunction with the requirements, it is possible to infer that a message contains at least the following attributes:

- the addresses of the sender and the receiver;
- the subject of the message;
- the date it was sent;
- the text of the message itself.

The next stage is to attempt to identify the operations which are associated with the classes. For example, for the *message* class a possible list of operations could be as follows (see Figure 4.9 also):

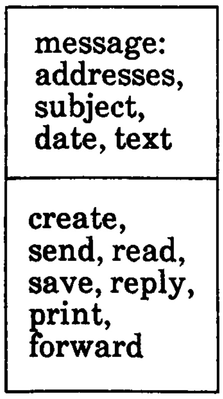


Figure 4.9 The Message Class for the EMS

- create a message, which means filling in the addresses of the sender and the receiver, the subject of the message and so on;
- send a message to another user;
- read a message on the screen;
- save a message in a messagefile;
- reply to a message;
- print a message;
- forward a message to another user.

The second most important class is the one called the *messagefile* whose instances are lists of messages. Every incoming message is accumulated into a single mailbox associated with the electronic mail address of the user. The EMS takes the messages to be read by the user from that mailbox. There are certain messages which the user may wish to place into a particular *messagefile*, identified by a *label*, to be read later. It allows the user to preserve or hold any message in separate folders or lists of messages which are instances of the *messagefile* class. Some basic operations that can be applied upon instances of the *messagefile* class are:

- put a message into a messagefile;
- list the contents of a messagefile;
- get a message from a messagefile;
- copy a message from one messagefile to another;
- move a message from one messagefile to another;
- delete a message from a messagefile.

The filter facilities allow the user to define a set of rules by which all incoming messages should be screened, and a subsequent operation to be performed based on whether conditions described in the mail profile have been met or not. Each rule is represented by a boolean expression followed by an action. The expression can be based on the subject of the message, and from whom the message was sent.

When the user invokes the EMS, the system accesses the mailbox of that user and performs the filtering operation on those newly received messages. The following actions can be performed upon messages of a *mailbox* class:

- list the contents of a mailbox;
- put a message in a mailbox;
- get a message from a mailbox;
- delete a message from a mailbox;
- filter a message from a mailbox.

Deleting a message causes the EMS to put it into a wastebin which is a list of discarded messages. The user may look at which messages have been put in the wastebin and may retrieve any of them before the wastebin is cleaned up. The operations associated with the *wastebin* class might be set as follows:

- put a message into a wastebin;
- list the contents of a wastebin;
- retrieve a particular message from a wastebin;
- clean up a wastebin.

So far, several classes with some attributes and operations have been identified. These classes model the main entities in the EMS and provide the basic functionality required from the software system. However, to achieve the best use of an object-oriented approach, inheritance must be used effectively as the major class organising principle. Although the proper use of inheritance is a difficult skill to master, the existence of guidelines to assist the designer can help work out this problem.

4.3.3 Identifying and Representing Inheritance

Having identified classes, the designer needs to identify inheritance. MOOD presents some guidelines and techniques which help both the novice and experienced designer to use the inheritance mechanism during the design phase. Inheritance occurs between classes and provides a way of sharing commonalities between them. Such a useful mechanism plays an important role in the design and construction of class hierarchies. As far as identification of inheritance by generalisation is concerned, the designer should represent the set of attributes and operations of each class in a Venn diagram and look at intersections between sets in order to determine whether an appropriate super-class may be created.

In the case of the EMS, the designer might realise, for instance, that the *mailbox*, *messagefile* and *wastebin* classes share similar attributes and operations (see the Venn diagram shown in Figure 4.10). The objects of each of these classes are lists of messages (folders) with slightly different operations on these lists. A new abstract *mailfile* class which encompasses all message storage can therefore be created as the super-class of the *mailbox*, *messagefile* and *wastebin* classes.

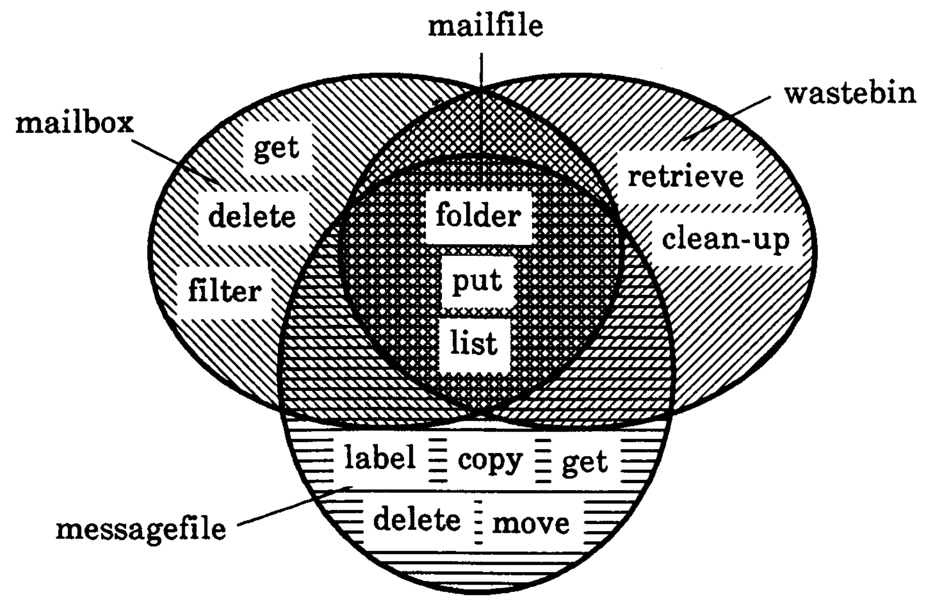


Figure 4.10 A Venn Diagram for the EMS

The designer could put these three conceptually related classes in the same hierarchy, sharing some common attributes and operations placed in a *mailfile* super-class. Each sub-class with some special operations, such as a *filter* operation for the *mailbox* class, a *clean-up* operation for the *wastebin* class and a *label* attribute for the *messagefile* class, can then be derived from that *mailfile* super-class. A more general picture of the *mailfile* super-class and its sub-classes can be depicted in the class hierarchy diagram represented in Figure 4.11.

As far as inheritance by specialisation is concerned, it is common to view a class as a specialisation of another more generic super-class. The specialised class inherits general properties from the generic super-class, defining only its specific differences. Thus, the mechanism of specialisation allows specific abstractions to be designed using more general ones. For the EMS, all services are to be provided via a menu-driven interface, with several kinds of menus, each with a different purpose. Actually, a menu deals basically with groups of lines which are showed on the screen. The main operations provided by a *menu* class are related to displaying and hiding a particular

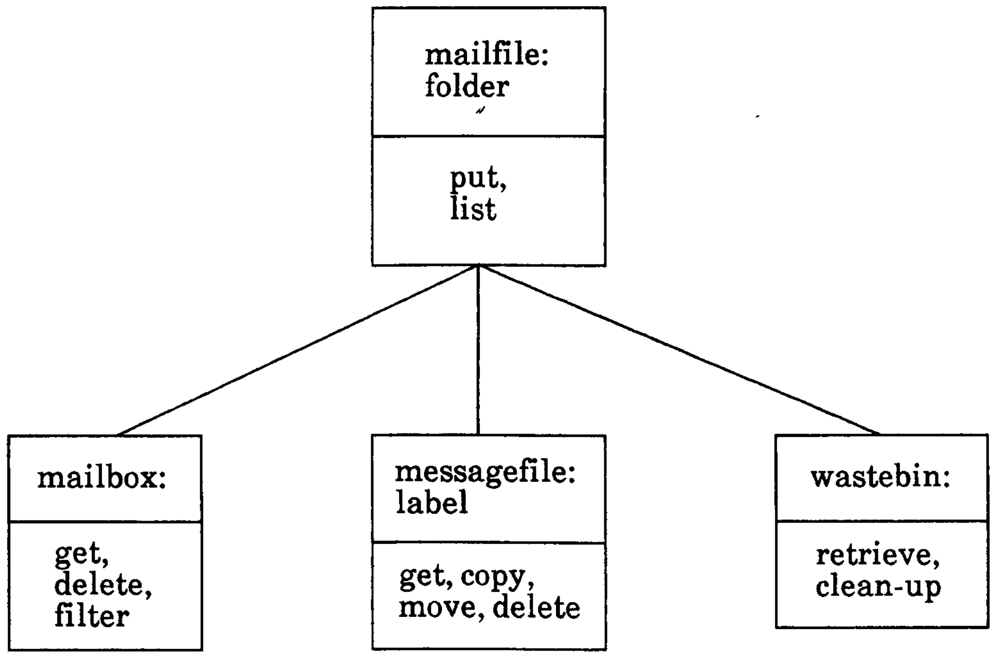


Figure 4.11 The Class Hierarchy for the EMS Mailfile

menu. There may be three different sub-classes of menus which might be specialised from a generic *menu* class. These specialisations are important because different operations may be defined for the sub-classes of menus:

- *Menu-option*: this sub-class represents only menus of lines showing to the user possible choices in a particular situation. This sub-class also needs an operation to get the chosen option. For instance, an object of this sub-class could be the following line of strings: "put, get or list?".

- *Menu-error*: this sub-class defines only a line containing an error message. For example, an instance of this sub-class might be "mail impossible to deliver - domain does not exist".

- *Menu-help*: this sub-class depicts some instructions to help the user. An example of an instance of this sub-class might be "press <ESC> at the end of the text editor".

These three sub-classes can inherit the *display* and *hide* operations from the *menu* super-class, as illustrated in Figure 4.12. The *menu-option* sub-class inherits the operations *display* and *hide* from the *menu* class but it also defines a particular operation to get an option from the user. The inherited operations for the *menu-error* sub-class are just *display* and *hide* an error message. The inherited operations for the *menu-help* sub-class are *display* and *hide* a help message.

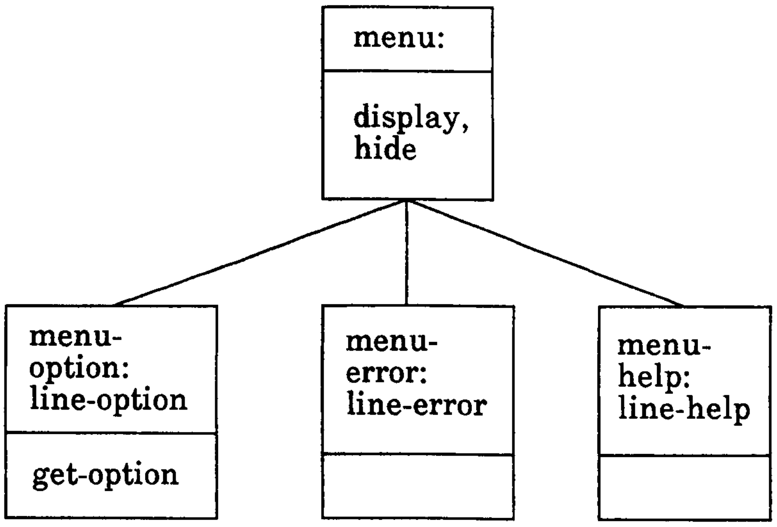


Figure 4.12 The Class Hierarchy for the EMS Menus

4.3.4 Comments on Inheritance within MOOD

Some comments on the use of.inheritance can be made at this point. Inheritance could be misused as a composition of attributes. It is a frequent mistake to consider a super-class and then try to derive sub-classes based on the attributes which make up that super-class. For example, consider a *car* class as the main class in a software system. A car can be divided into engine, transmission, brakes, suspension and so on. An engine can be further partitioned into components like ignition, fuel-injection, starter and so forth. Nevertheless, an *engine* class must not be considered as a sub-class of a *car* class because an engine is just part of a car and it is not a specialisation of a car. Certainly, the attributes and operations applied to a *car* class are completely different from those applied to an *engine* class.

The most important concept relating to inheritance should be specialisation and generalisation, never composition. The philosophy of using inheritance should be as follows: only use inheritance as a specialisation or generalisation mechanism. Successful use of inheritance depends on using a methodology which enforces or guides the application of this mechanism. The use of Venn diagrams as recommended by MOOD helps prevent this misuse of inheritance because the intersections between the set of attributes and operations of those *car* and *engine* classes would be empty, thus, one would not be a super-class of the other.

Another common pitfall is to overuse inheritance. In this case class hierarchies comprise too many sub-classes, with not enough difference in functionality between such classes, leading to a fine granularity. That is, many classes, each one with few attributes and operations. Although this high granularity may offer more potential for reuse, it makes the class hierarchies more difficult to understand. Classes that offer the same sort of services should be merged. Two courses of actions are open: either merge similar classes into one new class, or make them sub-classes of a common super-class providing the shared functionality.

Inheritance perhaps works best in the development of small software systems, which are likely to require a small number of designers to be developed. Where there are many designers updating and extending the class hierarchies, it is very likely that communication problems will cause unawareness of a particular change in the class hierarchies made by some designer. The following recommendations can be made to avoid this mismanagement of class hierarchies:

- All class hierarchies should be represented using the MOOD diagrams.
- Software configuration management techniques should be applied to control versions of classes and to keep track of the history of a design.

- An experienced software engineer to manage the several class hierarchies seems to be necessary.

This section has presented some issues related to the design of an Electronic Mail System. Although other designs (even better ones) are possible, the EMS design has shown how the early steps and the graphical notation introduced by MOOD can be used to produce a partial static design of that software system in terms of its classes and inheritance among such classes. Note that it was not intended to produce a complete design of a software system, but rather to acquire some experience of early MOOD steps, done by hand, not tools.

Many problems and constraints, such as how to manipulate large number of classes and the difficulty of doing a complete design entirely by hand, were enough to show that the use of tools is imperative during the design phase. The limitations presented in that design might be solved if the methodology were supported by a set of tools integrated within a software development environment (this issue is further discussed in the next section). The real evaluation of MOOD, to be presented in Chapter Seven, will be based on the experience of designing a CASE environment which supports MOOD, by using MOOD itself.

4.4 A CASE ENVIRONMENT FOR MOOD

The high cost of software development and maintenance, and the need for reusable software components are some of the factors stimulating the research for better CASE environments that support software development methodologies. Actually, putting a methodology into practice is almost impossible without an environment that provides a set of tools which gives automated support for that methodology.

Ideally, an environment should support a software development methodology by sustaining the steps, rules, principles and guidelines dictated by that methodology. The environment should also support consistency and completeness checking to guarantee that the principles of the methodology are obeyed. The tools should check rules that govern the syntax and semantics of the underlying methodology. These types of checks can eliminate a large proportion of design errors prior to the implementation.

A software development environment is often used in the construction of large software systems because in general those software systems are complex and require a large amount of interrelated information. In such circumstances, an environment is also useful because it ideally:

- simplifies the traceability between different phases of the software life cycle;
- enhances communication among designers;
- provides a uniform means of software representation;
- assists in keeping documentation through report generators;
- provides a means of producing graphical notation from information stored in its database;
- supports team work across many different workstations;
- manages coordination among designers in terms of assignment, deadlines and integration of tasks.

The object-oriented paradigm has been influencing the way that software development environments have been built because some tools are quite related to that paradigm. For instance:

- Library management tools to allow the searching for potentially reusable components such as classes.
- Browsers to facilitate navigation through class hierarchies, storage and recovery of classes, and definition and visualisation of classes in terms of their interfaces, attributes and operations.
- Object-oriented diagram editors to support an object-oriented graphical notation in terms of classes, inheritance and object interaction.

Based on the diagrams presented earlier in this chapter and the aspects of an object-oriented environment discussed above, the following tools can be identified in order to build an environment that supports MOOD:

- a class hierarchy diagram editor;
- an object diagram editor;
- an operation diagram editor;
- a composition diagram editor.

These diagram editors not only support the drawing of the MOOD diagrams but also ensure that the design which is being represented conforms to the rules and principles of the underlying methodology. There should also be other useful tools such as:

- a class browser;
- an object inspector;
- a library management tool;
- a consistency checker;
- a completeness checker;
- a report generator;
- a configuration management tool.

These tools should give some freedom to designers, letting them make some temporary omissions and then checking for such mistakes later in the design process. Moreover, the tools must also relate concepts on one diagram to other diagrams and point out the links and problems. A prototype of an environment that supports MOOD and satisfies the requirements discussed in this section is presented in Chapter Six.

4.5 FINAL REMARKS ON MOOD

An object-oriented software system can be seen at two different layers of abstraction as shown in Figure 4.13. There is an upper generic layer which shows the static description of a software system and a lower level layer which consists of objects and shows the behaviour of a software system. The upper layer can represent the set of classes from which any software system may be constructed, while the lower layer depicts the actual objects and potential object interactions required in a particular software system. The design of each layer may be carried out simultaneously.

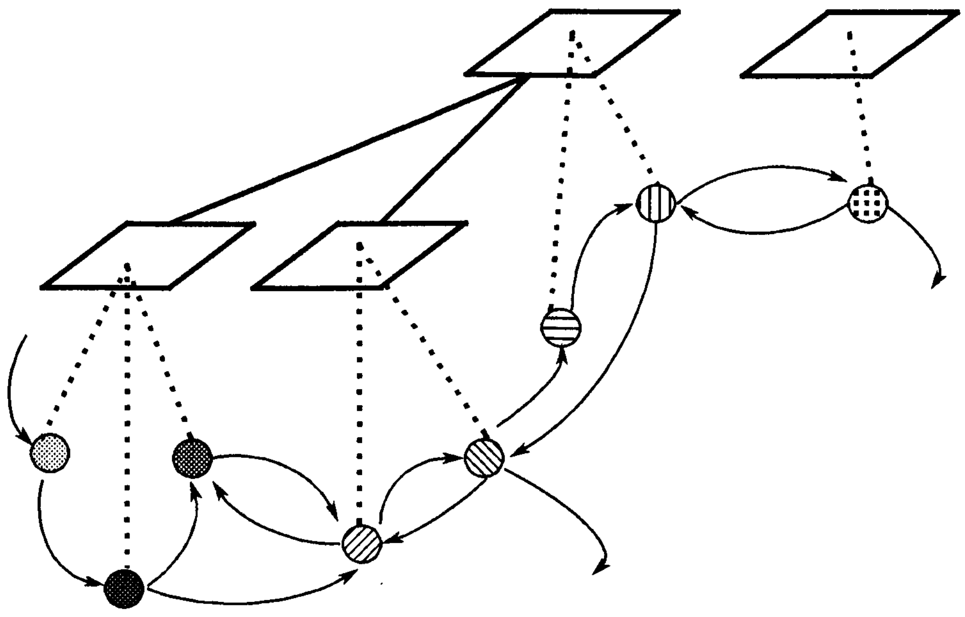


Figure 4.13 The Static and Dynamic Layers of a Software System

It is important to remember that identifying classes is not the same as identifying objects. Classes are a means of expressing static commonalities between objects and templates to create them, whereas objects will have a dynamic life in a running software system. The MOOD diagrams show both the static and dynamic aspects of a design. On one hand, the static design consists of class hierarchy and composition diagrams. On the other hand, the dynamic design is expanded with both object and operation diagrams which represent the overall dynamic behaviour of a software system. At first, the operations on individual objects are presented in object diagrams and then these operations are combined to perform the functionality required for the software system. The steps proposed by MOOD, as discussed earlier in this chapter, are shown in Figure 4.14.

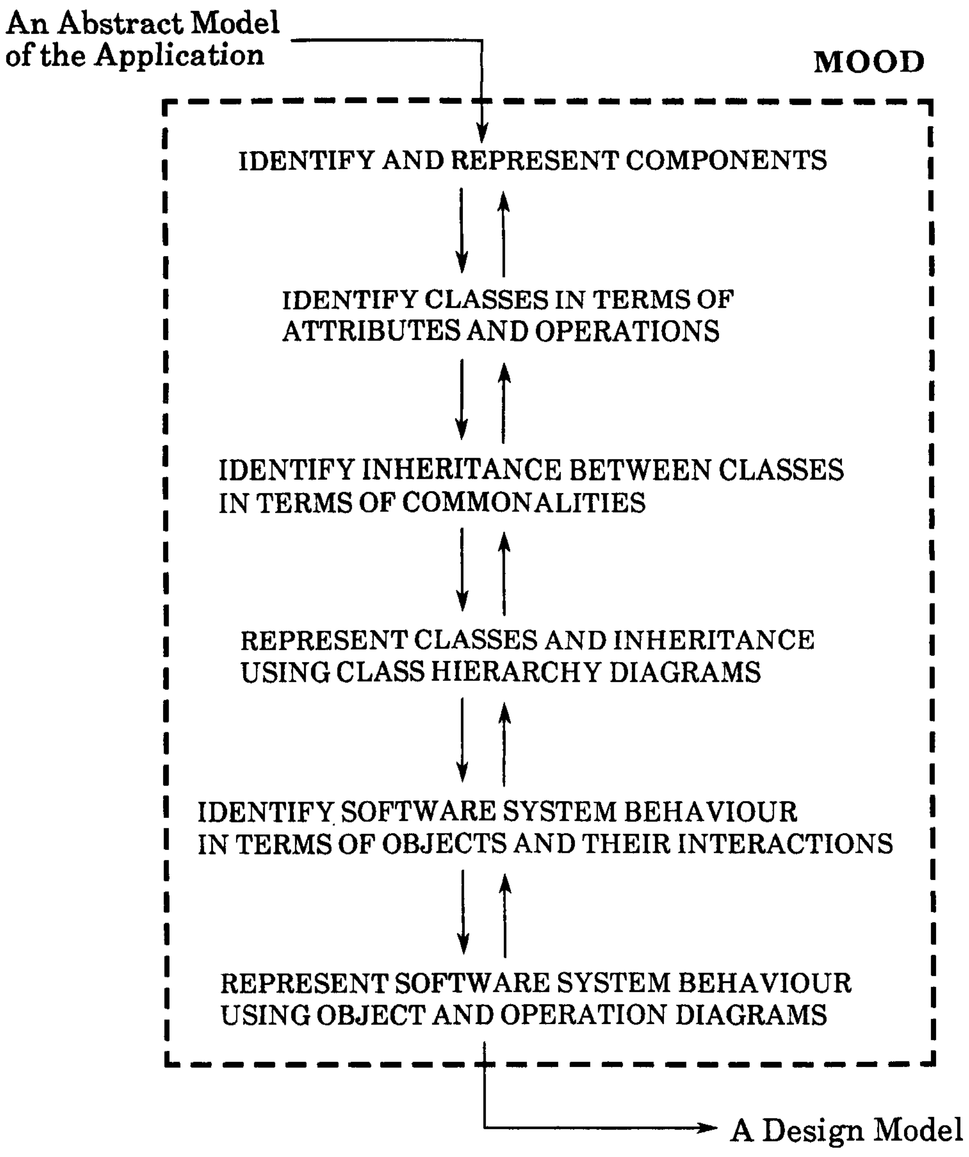

An Abstract Model of the Application
MOOD
IDENTIFY AND REPRESENT COMPONENTS
IDENTIFY CLASSES IN TERMS OF ATTRIBUTES AND OPERATIONS
IDENTIFY INHERITANCE BETWEEN CLASSES IN TERMS OF COMMONALITIES
REPRESENT CLASSES AND INHERITANCE USING CLASS HIERARCHY DIAGRAMS
IDENTIFY SOFTWARE SYSTEM BEHAVIOUR IN TERMS OF OBJECTS AND THEIR INTERACTIONS
REPRESENT SOFTWARE SYSTEM BEHAVIOUR USING OBJECT AND OPERATION DIAGRAMS
A Design Model

Figure 4.14 The Steps Recommended by MOOD

The steps emphasise design before implementation and are independent of any programming language. The design process originates with an abstract model of the application and culminates with a design model ready to be implemented. The path connecting these two models is not a straight line, but rather an iterative refinement process. In the course of building an object-oriented software system, the steps involved in the design of the necessary classes and objects are almost certain to be repeated many times.

The designer can use MOOD to represent a software system by using only a fixed set of diagrams. Class hierarchy diagrams are graphical notations depicting how classes are arranged hierarchically. Object diagrams and operation diagrams are used to represent the behaviour of a software system. More detailed examples of MOOD diagrams from a particular software system are given in Chapter Six.

The graphical notation proposed could be used manually but a prototype of an environment that provides automated support for MOOD is also presented in Chapter Six. In that prototype there are also tools to provide consistency and completeness checks, browsers and configuration management facilities. The next chapter discusses how to consider reusability issues within the MOOD context and a software development life cycle model that encompasses MOOD.

Chapter 5

REUSABILITY AND SOFTWARE LIFE CYCLE ISSUES

This chapter discusses reusability and software life cycle issues which arise during the design of object-oriented software systems. Such issues are related to the previously described topics of domain analysis and application domain, and placed in the context of the MOOD methodology presented in the previous chapter.

The first section of the chapter outlines existing mechanisms for achieving reusability, such as composition and inheritance. The section also concentrates on the main reasons why software is not reused and examines the problems associated with reusability during the design phase. Additionally, in this section reusability is added to the context of the MOOD methodology.

In section two, the main issues concerning an alternative software life cycle model are discussed. The purpose of this section is twofold: firstly, to discuss how different levels of knowledge that the designer has about the application domain can affect software development; and secondly, to present a software development life cycle which encompasses MOOD and takes reusability into account. The chapter ends with comments on the object-oriented software development process with regard to reusability and the software development life cycle proposed.

5.1 REUSABILITY WITHIN MOOD

The concepts of reusability and object-oriented methodologies are so interrelated that it is often difficult to talk about one without mentioning the other. This subsection presents an approach to software reuse while the design process is performed. The approach to be presented is in accordance with the philosophy of MOOD, in such way that while the MOOD steps are carried out, reusability through the composition, generalisation and specialisation mechanisms is taken into account.

The approach focuses on a collection of software systems within a particular application domain, and encourages reuse of components from an existing reusable library within that application domain. Such an approach addresses the mechanisms used when components are reused from the reusable library. Moreover, the approach recognises the iterative nature of software development, hence iterations are incorporated into the design process where appropriate.

A diagrammatic representation of reusability within MOOD is shown in Figure 5.1. The reuse process is well-suited for an object-oriented software development because the composition, generalisation and specialisation mechanisms, which are part of the object-oriented design process (as presented in Chapter Two) are considered. Underlying the reuse of software components is the process which addresses both the identification of reusable components and their deployment into the developing software system.

Reusability involves design with reuse and design for reuse. Initially, the designer identifies potentially reusable components from an existing reusable library, the components are then selected and reused through composition, generalisation and specialisation mechanisms. At the end of software development, there may be many new reusable components which need to be classified and then to be stored into a reusable library. In the future, such components can be reused in other software systems. The second issue will be discussed later in this section.

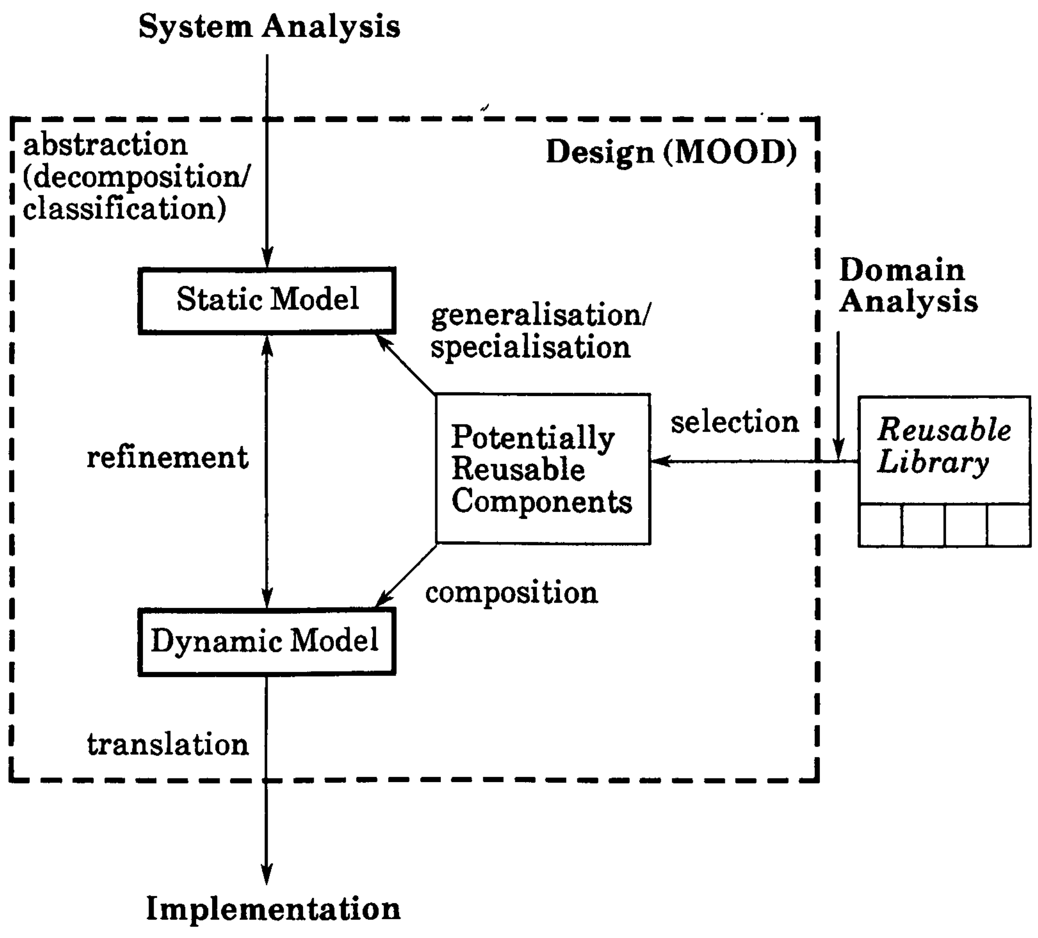


Figure 5.1 Reusability within MOOD

5.1.1 Relationships between Components

So far, most of the work which has been done in the reusability arena involves storing and recovering components from reusable libraries, but there are still many problems related to reusing such components. For instance, as a software system becomes mature, the libraries may grow as domain-specific libraries and reusable components can be added over time. It does not take long for such libraries to expand to enormous proportions and often with multiple versions of a component, which makes it difficult for designers to look for components which might meet their needs.

Reusable libraries are usually large and their organisation makes it problematic to find potentially reusable components. One of the great difficulties in identifying reusable components lies in the fact that there is a discordance in terminology between different designers so that components which designers are looking for are described in the libraries by unfamiliar or unexpected terminology.

In order to reuse a component, the designer must first find it. Therefore, a good classification scheme for arranging components is central to the selection process. This classification can be an aid to understanding a component when software reuse demands adaption of that component to match new requirements. The learning curve for reusable components may be substantial. Besides, the search for a component is a difficult task in that the designer must select one which requires the least effort to adapt, with the goal being an exact matching between what is needed and what is available. The time and effort required for this selection process is decreased by the presence of semantic information within the reusable library. The potential *re-user* of software must be able to find a connection between what is needed and what is available. Relationships between components could be used to facilitate the search for potentially reusable ones.

The semantics supported in the diagrams proposed by MOOD can be expressed by relationships between components. For instance, *has-a* relationships can be found in composition diagrams, *is-a* relationships are presented in class hierarchy diagrams, *uses-a* relationships can be shown in operation diagrams, and *is-part-of* relationships can link a component to a particular context. Such relationships can be used as a classification scheme to provide a network of pre-defined links between components, thus introducing some semantic information and a vocabulary into a reusable library.

One way to express relationships between components of a reusable library involves organising them through a set of pre-defined **relations**. Such relations allow components to be classified and correlated to others which could also be reused. In addition, relations can be used to express a link between different components, facilitating the understanding of the components. Relations used to represent design information between two reusable components can help to solve the problem of discordance of terminology between designers because the relations can establish some fixed semantic concepts between components. Four different relations to link components and express concepts related to MOOD are proposed:

1) **Compose** (<component-1>, <list-of-components>): This relation represents <component-1> as a composition of components in a <list-of-components> (*has-a* relationship). This information is available from the composition diagrams.

2) **Inherit** (<component-1>, <component-2>): This relation indicates that <component-1> is a generalisation of <component-2> or the other way round that <component-2> is a specialisation of <component-1> (*is-a* relationship). This information can be found in the class hierarchy diagrams.

3) **Use** (<component-1>, <list-of-components>): This relation indicates that <component-1> interacts with components in a <list-of-components> (*uses-a* relationship). It means that any operation of <component-1> uses any operations defined in any component in a <list-of-components>. A complete link of dependencies among operations of different components can be built with information from the operation diagrams.

4) **Context** (<component-1>, <context-1>): This relation associates a <component-1> with a <context-1> defined by the designer (*is-part-of* relationship). The <context-1> can be a particular application domain or a framework. This process involves classification of components and depends on the experience of the designer.

The *compose*, *inherit* and *use* relations can be perceived straight from the MOOD diagrams or, if necessary, complemented with other information provided by the designer, whereas the *context* relation should be given by the designer. The designer may use the relations proposed above as an alternative textual representation to describe a software system. Furthermore, enquiries to a reusable library which stores such relations and manipulates the same concepts as MOOD can be undertaken.

Given that the information present in a reusable library is compatible with the information dealt with in MOOD, the MOOD diagrams proposed in the previous chapter can be used to represent reusable components in the reusable library. Moreover, because the semantics of the MOOD diagrams can depict relationships between reusable components, reusability is naturally encouraged by the use of MOOD.

5.1.2 The Process of Component Reuse

The decisions involving the reuse of a component are very important in that the designer must select the component which requires the least effort to adapt, with an exact match between what is needed and what is available being the goal. Basically, the selection of a component from a reusable library involves four steps:

1) identifying the required (target) component;
2) selecting potentially reusable components;
3) understanding the components;
4) adapting (specialising, generalising, composing or adjusting) the components to satisfy the needs of the developing software system.

The search for a component in a reusable library can lead to one of the following possible results:

- An identical match between the target and an available component is reached.
- Some fitting components are collected (only a partial match), then adaptations are necessary.
- The requirements are changed in order to fit available components.
- No reusable component can be found, then the target component should be created from scratch.

Following a procedure which helps select potentially reusable components is vital to the reuse process. The procedure shown in Figure 5.2 illustrates a typical attempt to reuse a component from a reusable library. The procedure describes only the selection of reusable components; classification and storage issues are discussed later in this section. By properly classifying a component using the relations proposed previously, the chance of finding potentially reusable components is increased. Furthermore, the effort required to get a suitable component is reduced because the classification scheme based on relations can guide the designer through the various relations quickly and efficiently.

While searching for components it is necessary to address the equivalence between the required (target) component and any near matching components. The best component selected for reuse may also require specialisation, generalisation or adjustment to the requirements of the new software system in which it will be reused. Sometimes, it is preferable to change the requirements in order to reuse the available components. The adaptability of the components depends on the difference between the requirements and the features offered by the existing components, as well as the skill and experience of the designer. The process of adapting components is the least likely to become automated in the software reuse process.

5.1.3 The Lifetime of Reusable Components

Reusability not only involves reusing existing components in a new software system but also designing components which are meant for reuse. When a software system has been developed, the designer may realise that some components can be generalised and reused in future software developments. An important consideration in the quest of reusability is how to make a potentially reusable component available to other designers. The component must be understandable, well-written and well-documented. Finally, the component must be easily adaptable for different uses, either in original or in modified form. Therefore, developing reusable components is considerable more difficult and involves much greater expense then producing ordinary components, although it may still be worth the investment.

```
Begin

    // The process of component reuse

    given a key name (the name of the target component),
    search library for potentially reusable components and
    their relations

    if identical match between the target and an available component
    then
        // reuse by composition
        retrieve it and reuse it

    else
        collect fitting components
        for each collected component
            assess the degree of matching
        endfor
        rank and select the best component
        if the target can be a sub-class of the best component
        then
            // reuse by specialisation
            put the target as a sub-class and inherit commonalities

        else
            if the target shares commonalities with the best component
            then
                // reuse by generalisation
                create a generic super-class,
                put the target and the best component as sub-classes

            else
                // specialisation and generalisation are not convenient
                if possible
                then
                    adjust the best component to the requirements or
                    adjust the requirements to the best component
                else
                    create the target component from scratch
                endif
            endif
        endif
    endif

end
```

Figure 5.2 A Procedure to Reuse a Component

In an object-oriented approach, classes are the most important reusable components. Since an object-oriented software system is developed essentially as an interrelated collection of independently developed classes, it is important to understand the stages which such components go through. The stages should reflect the activities involving the identification, design, implementation, validation, classification, storage and reuse of the component. Figure 5.3 depicts the lifetime of a reusable component.

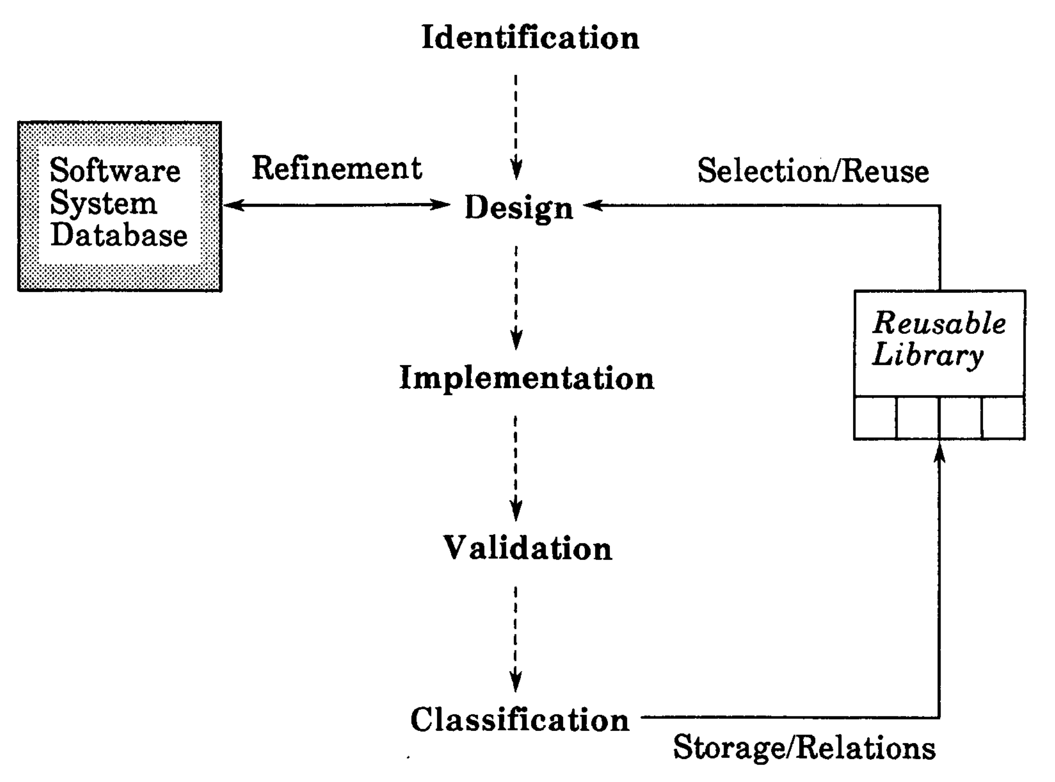


Figure 5.3 The Lifetime of a Reusable Component

Through the introduction of explicit stages of specialisation, generalisation and composition during the design phase, application-dependent classes are revised so that they can be sufficiently generic to be of use in a wider range of software systems rather than the single software system for which they were originally developed. This generality requires extra effort during the design and implementation phases in the short term, but in the long term, when a sufficiently broad reusable library has been constructed, it will lead to a significant reduction in overall software development time and effort.

There must be in fact two main libraries: the reusable library from which components of interest can be picked up (and to which new generic reusable components can be added), and the software system database (or MOOD database as used in Chapter Six) which stores information concerning a particular software system under development.

If a newly implemented component does not exist in the reusable library, then a decision has to be made as to whether the new component should be classified as reusable component and to be put in the reusable library. Before a component can be added to the reusable library, it must be validated and frozen. The validation is just applied to that component, not to the whole software system and should include treatment of exceptional conditions. For instance, for a class to be a viable candidate for inclusion into the reusable library, it must first:

- be well-defined in terms of its attributes and operations;
- have a reasonable performance in terms of time and space required to carry out its operations;
- be a generic abstraction, which means that the functionality it provides must be sufficient to model the real-world entities abstracted;
- have a robust behaviour when it is misused or pushed to its limits, that is, exceptions must be handled.

It is also very important to separate classes of client objects and server objects. Client objects are often highly application dependent and they make decisions and switch the control flow among several server objects. Client objects should not directly perform calculations or implement complex algorithms. On the other hand, server objects perform specific and detailed operations, executing general computations to implement a particular self-contained algorithm and rarely change the control flow. Therefore, server objects are more likely to be reused in other software systems than client objects since the former are more application-independent and basically wait to receive requests from client objects. Thus, as far as design for reuse is concerned, classes of server objects are preferable.

It is not recommended that components in the reusable library be modified, but a copy of a component should be taken into the software system database and adaptations carried out there. Tools can manipulate the reusable library by storing, selecting and browsing the library. Storing a component involves classifying it, getting it from the software system database, relating it to other components and putting it into the reusable library. Selection involves browsing to find a component, retrieving it and transferring it from the reusable library to the software system database.

5.2 MOOD WITHIN A SOFTWARE LIFE CYCLE MODEL

This section presents an alternative object-oriented software development life cycle that encompasses MOOD. The proposed software life cycle model is influenced by the knowledge that the designer has about the application domain and also addresses reusability within an object-oriented approach to software development.

5.2.1 The Role of the Knowledge about the Application Domain

Software system designers hardly ever solve a new problem from scratch. Instead, they try to identify similarities between the new application and previous applications and their solutions. By making suitable transformations from previous experience, designers attempt to solve the new problem. This process is referred to as solving by analogy, which is considered to be a natural way in which people learn. The successful use of analogy depends on recognising the similarities between two problems and recalling the solution of the analogous problem. Therefore, it can be assumed that the knowledge designers have about an application domain increases the chance of reusing solutions from previous experience. However, many object-oriented methodologies do not take this human feature into account.

When designers are developing software systems in an unfamiliar application domain they do not have the same knowledge and skills available to them as when they are developing software systems in a familiar domain. Of course, there are differences in the approach to developing software systems if designers can apply their knowledge and skills acquired when software systems in a known domain have been developed before.

Adelson and Soloway (1985) claim that the experts explicitly construct high level abstractions of a software system whereas novices think about low level entities and their behaviour within a software system. Therefore, the knowledge that designers have about an application domain affects the way software development is carried out. The experts tend to think in more abstract and high level terms following a top-down approach whereas the novices usually start thinking about low level abstractions of a software system and software development is thus predominantly bottom-up.

A strictly top-down or bottom-up approach to software development is not quite appropriate for an object-oriented software development. This section proposes a top-down or bottom-up approach to software development, depending on the knowledge that designers have about an application domain. This knowledge naturally determines the predominant approach to subsequent software development.

If designers know the application domain, they should start thinking about high level abstractions (such as class hierarchies), whereas when designers do not know much about the application domain, they should start instantiating some objects and trying to understand the low level behaviour of a software system. As software development proceeds, designers may sometimes utilise a mixed approach to software development, switching between a top-down and a bottom-up approach as designers may explore options. However, the predominant approach is determined by the designers' knowledge about the application domain.

5.2.2 The Proposed Software Development Life Cycle

Software development is characterised by change and instability, and therefore any diagrammatic representation of a software life cycle model should take into account overlapping and iteration between phases. Based on the steps proposed by MOOD, a consensus may be drawn on the phases pertinent to a software development life cycle and the common tasks carried out following MOOD. A proposed software development life cycle which encompasses MOOD and takes reusability into account is represented in Figure 5.4. Although the main phases may overlap each other and iteration is also possible, the proposed phases are: system analysis, domain analysis, design (static and dynamic) and implementation. Maintenance is an important operational phase, in which bugs are corrected and new requirements met; it will not be further discussed.

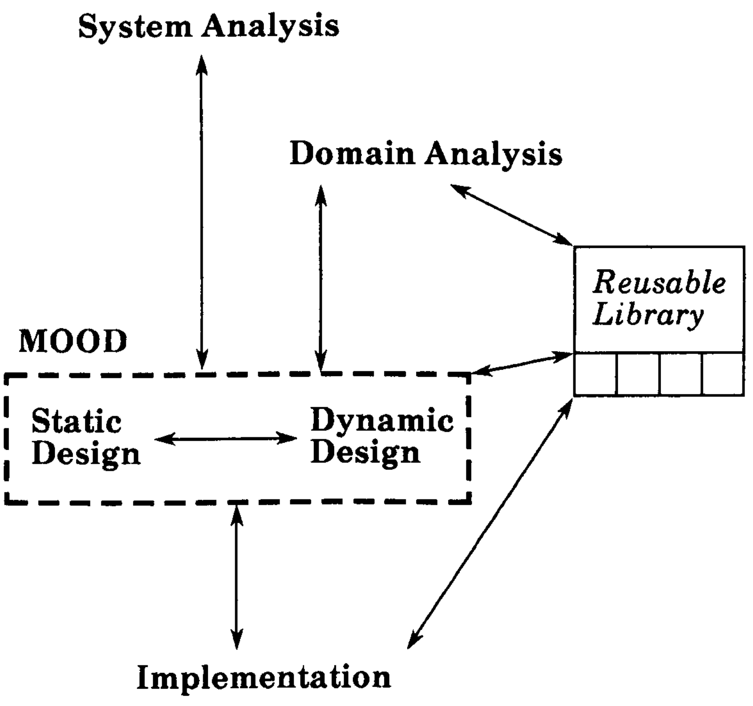


Figure 5.4 MOOD within a Software Development Life Cycle

Another outcome of the proposed software life cycle model is the emphasis on reusability during software development and the production of reusable components meant to be useful in future software developments. This is naturally supported by the object-oriented paradigm because of inheritance and encapsulation. Reusability also implies using composition techniques during software development. This is achieved by initially selecting reusable components and aggregating them, or by decomposing the software system to a point where it is possible to identify components that can be reused from a reusable library.

Figure 5.4 presents a diagrammatic representation of how the system analysis, domain analysis, design and implementation phases proceed iteratively over time and how reuse of components from a reusable library is taken into consideration within the proposed software development life cycle. Reusability within the proposed software life cycle model is smoother and more effective than the traditional waterfall software life cycle model because MOOD integrates at its core (through relations) the concern for reuse and its mechanisms, as shown in the previous section.

System Analysis

This phase involves high level analysis of the application in terms of the functionality that the software system should provide. The system analysis phase requires the designer to:

- understand the application;
- understand the requirements expected to be satisfied by the software system;
- propose an abstract model of the application in which such requirements are accomplished.

The main purpose of the system analysis phase is to provide a graphical or textual description of an abstract model of the application. This phase may conduce to the identification of the major parts of the application so that the software system may be divided into large components, based on the functionality that should be provided. A first idea of the classes and objects which model the application could also be picked up. If several similar relevant objects have been found, a class should be established. In the same way, if correlative classes have been identified, class hierarchies should be worked out.

The functionality the software system should afford in fact defines the subsystems and modules. However, as compared to functional decomposition, this phase is not concerned with the details of one function or procedure in terms of the algorithms they implement, nor that one function can be refined in terms of other subfunctions but it is concerned with decomposition in terms of classes and objects.

The result of this phase is an object-oriented abstract model of the application, which may be graphical or textual, using a formal or informal method. At this phase it seems unnecessary to detail classes in terms of their attributes and operations, or look for communication patterns among objects of different classes, since what is more important is a high level view of the software system.

Domain Analysis

The domain analysis phase primarily seeks to abstract and classify concepts which form the vocabulary of the application domain. During this phase a common terminology is drawn from the application domain, as discussed in Chapter Two. Large applications should be broken into parts, so that specialists in a particular application domain can carry on the domain analysis in that application domain.

During this phase, the abstract model of the application comprising high level abstract representations of software components may be refined and new classes and objects to encompass these components can be defined. Therefore, the boundary between system analysis and domain analysis may at times seems fuzzy because identifying key abstractions in the application domain may be viewed as part of system analysis or part of domain analysis. Nevertheless, at this level, domain analysis is more concerned with the identification and organisation of potentially reusable components.

Selection of reusable components from previous software systems or from a reusable library should be considered. When doing domain analysis, there might be a sketchy idea about candidate classes and some attributes and operations. Sub-classes can be derived from the classes in a particular application domain, and objects can be instantiated from classes in a reusable library and composed with other objects. In the context of the proposed software development life cycle, the main result of the domain analysis phase should be the reuse of software components which have already been developed (see Table 5.1).

Design

Object-oriented design is an exploratory process. The designer looks for classes and objects trying out a variety of schemes in order to discover the most natural and reasonable way to model the application. During the design phase the primary concern is to build a design model which will fulfil the overall software system functionality. The construction of the design model involves defining relevant classes and objects, and producing both the *static design* and the *dynamic design.*

The static design comprises a static model of the software system and basically makes use of class hierarchy and composition diagrams. The dynamic design consists of a dynamic model showing the behaviour of the software system with object and operation diagrams. During the design phase those abstractions identified in previous phases are turned into concepts which represent the software system in more detail and afford the functionality that the software system should provide.

The static design captures the generic and essential features of a software system and can be expanded for other software systems within the same application domain. In contrast, the dynamic design captures dynamic aspects of a particular software system and is therefore more difficult to generalise to other software systems. To do both designs, the MOOD graphical notation introduced in the previous chapter must be used.

When object-oriented designers face an application, they should not ask "How do I work out a solution to this problem?". Instead designers should ask, "Where are the classes and objects that I can directly or indirectly reuse to solve this problem?". A procedure to guide the designer over this task was presented in the previous section. At this point the designer should be able to examine a reusable library and to select components which closely match the classes and objects needed to build the developing software system.

A good browser is useful to trace the class hierarchies and see the attributes and operations possible to apply to a particular object. One of the biggest problems with object-oriented software design is that in order to understand the class hierarchies it might be necessary to scan three or four super-classes to find out all attributes and operations which make up each sub-class. Thus, a browser is necessary to locate information in the class hierarchies. A browser could copy automatically all inherited attributes and operations into a sub-class together with the related super-class names. In this way, once the designer is browsing a class, all the features of that class and of its super-classes are available. Although browsers are very useful for discovering and understanding the characteristics of classes in a library, they are by no means a sufficient tool for finding suitable candidate classes for reuse.

As more classes are identified, re-evaluation of the complete set of classes is required and decisions whether new sub-classes or super-classes should be made. The iterations are not unusual, since good design usually takes many iterations. The number of iterations also depends on the insight, experience and knowledge about the application domain. A bottom-up approach (instantiation of some objects) should be considered if the designer does not have a good perception of the application domain.

The classes and objects which are identified during the design phase might undergo another stage of refinement (for example, treatment of exceptional conditions could be considered) until they become generic and robust enough to be placed in a reusable library. This may add an overhead to software development, but this overhead should be more than compensated for by the long term saving when future software systems reuse such components.

Implementation

The implementation phase is characterised by the translation into an object-oriented programming language of the static concepts and the dynamic behaviour represented by the output from the design phase, that is, the MOOD diagrams. In this phase the major tasks are to implement the identified classes, along with the cooperation among the objects in order to fulfil the software system functionality.

The line between design and implementation is also a thin one. Implementing a class requires definitions of the data structures corresponding to attributes and the algorithms corresponding to operations of that class. It is also necessary to implement the control flow which realises the interaction between objects and defines the overall behaviour of the software system. The best approach is to isolate a class or an object and decide whether a component which matches that class or object can be reused, or whether it has to be implemented from scratch.

Table 5.1 summarises the phases of the software development life cycle proposed. The table shows the input, tasks performed and output of each phase. Such phases evolve dynamically as the understanding the designer has about the software system grows. The phases can be traced during software development and determine an object-oriented software life cycle model.

Phases ↓	Input	Tasks	Output
System Analysis	application: user needs and software system requirements	create an abstract model of the application	abstract model of the application
Domain Analysis	abstract model of the application	identify possible reusable components	potentially reusable components
Design	abstract model of the application and potentially reusable components	build static and dynamic models	static (generic) and dynamic (behaviour) models
Implement.	static (generic) and dynamic (behaviour) models	implement the models	software system solution for the application

Table 5.1 Input, Tasks and Output of each Phase

5.3 COMMENTS ON THE SOFTWARE DEVELOPMENT PROCESS

The phases of the proposed software development life cycle are highly integrated. At the system analysis and domain analysis phases, user needs, requirements, functionality, objectives and the constraints of the software system are very much of interest. Thus, it is important to understand the real-world application, and an abstract model of that application should be accomplished. When the design phase is entered, the abstractions are detailed. The design process should stop when the key generic abstractions and the software system behaviour are detailed enough to be translated to a programming language. Thus, the design phase generates the templates for the implementation phase of the software system.

Because experts tend to think in terms of high level abstractions, their prevailing approach to software development is top-down, whereas novices tend to use a predominantly bottom-up approach. When designers have good knowledge about the application domain, it is easier for them to divide the software system into large components and think about high level abstractions. On the contrary, for designers who have no previous knowledge about the application domain, it is easier to think in terms of low level abstractions and instantiate some objects at that level to depict the overall behaviour of the software system. In broad terms, it could be concluded that everything should be built top-down, except for the first time.

The top-down and bottom-up approaches have a significant effect on reusability because a top-down approach applies reusability predominantly in terms of generalisation and specialisation of abstractions, whereas a bottom-up approach uses reusability predominantly as aggregation of components. Following an object-oriented approach, reusability not only becomes easier as a consequence of the correspondence between real-world entities and its software components, but it will also become enhanced by mechanisms such as inheritance and composition which are supported by MOOD and its tools.

Nevertheless, a software system will not be produced out of reusable components only. On the contrary, usually, components selected and derived from reusable libraries will be combined with newly written components, and these components have to be bound together in a particular software system. With some of the components of such a software system, the designer will face the decision of whether to reuse them straightforwardly, adapt them and reuse, or write them from scratch. It has been argued that the break-even-point of reusing versus redoing would be where the cost of search plus adaptation exceeds the cost of producing the respective piece of software.

Putting such arguments into an object-oriented software development framework, and particularly relating them to MOOD produces the following observations. If the application domain is known by the designer then it is easier to start abstracting classes, finding commonalities between them, and building the class hierarchies. The use of the class hierarchy diagrams, as recommended by MOOD, is likely to be more appropriate for this task. On the other hand, if the designer wants to start a design by instantiating some objects, then the object diagrams and operation diagrams, as recommended by MOOD, are the most appropriate to represent the abstractions.

The implication of such observations on tools which automate software development, and particularly tools supporting MOOD, is relevant. The designer should be aided by a whole range of tools. However, which tool is most helpful depends on the level of knowledge that the designer has about the software system being developed and on the application domain as well. Therefore, all tools should be available to the designer at any time during software development. The designer picks up those tools which will help deal with the concepts manipulated at that moment, such as classes or object behaviour. By using a variety of tools, the designer is able to decide which of them is the most appropriate for each task. Therefore, an environment should allow both bottom-up and top-down approaches to an object-oriented software development. Some tools which are indispensable at different stages of an object-oriented design are presented in the next chapter.

Chapter 6

THE MOOD PROTOTYPE

Computer-Aided Software Engineering (CASE) environments gained significant impetus in the 1980s. With the need to produce large and complex software systems, the provision of computer-aided tools to support the use of design methodologies has become increasingly imperative. Fortunately, the advent of inexpensive powerful workstations with graphical capabilities has permitted such tools to be built and made available.

The tools can take a wide variety of forms, including diagram editors which help document the use of graphical notations relating to an underlying design methodology, type checkers for particular notations, checkers for the consistency and completeness of a design, and even transformation tools for assisting in the conversion of elements of a design from one representation to another.

This chapter discusses how MOOD can be supported by a CASE environment and presents the **MOOD prototype.** This is the prototype of an environment which provides automated support for MOOD through diagram editors, browsers and checkers. The primary aim of the MOOD prototype is to offer its users (software designers) a compatible set of tools for supporting object-oriented design. This is achieved by building a full environment containing the tools discussed in Chapter Four.

The first section of this chapter presents the overall issues related to the implementation of the MOOD prototype. This section also describes the particular InterViews (Linton, 1989) classes relevant to the context of that implementation. Some of those classes were used as super-classes during the design and implementation of the **MOOD interface**. In the second section, the main features of the prototype interface are outlined. The section, as a whole, also presents an approach for building other environment tools.

Section three describes the design and implementation of the **MOOD database** which supports persistency for MOOD objects, and shows how integration among tools is accomplished using a unified representation model, in this case an object-oriented model. In the fourth section, the semantics and behaviour of the MOOD environment tools are presented, showing examples of their use. The chapter finishes with considerations on the use of the prototype during the design process. It should be noticed that all windows shown in this chapter were generated using the MOOD prototype, which helped to evaluate the MOOD environment interface.

6.1 OVERALL IMPLEMENTATION ISSUES

The MOOD environment is more than a set of different tools put together to support object-oriented design. Integration of tools is achieved by bringing them together under an object-oriented framework, enabling a software system to be designed within this unified framework, rather than using a collection of disjointed tools. Integration allows the user to freely interchange information between one tool and another, facilitating communication among tools. The MOOD prototype can use multiple views to display windows of different tools simultaneously and it is possible to change from one tool to another so that the output of one tool can be used as input to another.

Integration is also accomplished under a single database, the MOOD database, and through a common interface. The MOOD prototype has a database structure and a unified representation model to describe the design information in the database, in this case a uniform object-oriented model. When a unified representation model is used to describe information, it has the added advantage of supporting traceability throughout software development because that model deals with uniform concepts continuously. That is, the same concepts can be carried from the system analysis and design phases to the implementation phase, even though the concepts change as they gain additional details during the later stages.

The interface is probably the most important factor in producing an environment that is acceptable. It is especially imperative for MOOD because of its graphical notations and iterative behaviour. Hence the MOOD prototype needs interface features driven by mouse, icons and menus, not command-line driven. In an environment like this, interactions with the tools can be seen as a set of actions without any prescribed order but driven by events from the mouse. Therefore, the MOOD interface is based on mouse interaction and multi-windows, one for each tool, where the user can create, edit and delete diagrams. The MOOD interface provides a uniform behaviour for all tools in the environment and should support a consistent view independent of which tool is being used.

The entire implementation of the MOOD prototype has been constructed using C++ (Stroustrup, 1986) and the reusable InterViews library (Vlissides, 1989) which provides classes of interactive objects, and uses X Window (Scheiffler, 1986) primitives for presenting them. The prototype implementation also contains a database in which the design information associated with the developing software system resides. The database provides basic functionality for object management such as creation, access and deletion of objects, and has also been implemented in C++; naturally, persistence for MOOD objects is supported.

The reusable InterViews library contains a variety of pre-defined classes and components. Only some of those which were used in the implementation of the MOOD prototype are outlined below, so that it becomes easier to understand how the MOOD prototype objects behave and why the interface classes were inherited from some particular InterViews classes. The MOOD prototype uses many features of InterViews and the C++ inheritance mechanism. A strategy pursued during the design and implementation of the MOOD prototype, and particularly its interface, has been to reuse the classes provided by the InterViews library as much as possible.

Each tool in the MOOD prototype uses InterViews classes and utilises the database operations to store and retrieve MOOD objects. As distinct from several tools which mostly act as a mere drawing editor of polygons, the tools presented in this chapter reinforce software engineering principles of the MOOD methodology and provide syntactic and semantic checks in the software development life cycle.

6.1.1 InterViews

InterViews (Interactive Views) is an object-oriented framework which assists in the design and implementation of graphical user interfaces. InterViews allows user interfaces to be defined by the composition of existing objects and new classes to be inherited from existing ones. InterViews provides a rich set of primitive and composition objects (such as scroll bars, dialogue boxes and buttons) which promote flexibility in the implementation of different user interfaces. Additionally, it offers a broad range of interface styles such as pop-up, pull-down and pull-right menus which can be customised on a per-user basis and according to the philosophy required by different styles of user interfaces.

InterViews also gives the choice of graphical and/or textual manipulation, and allows the separation of the user interface from the functionality of a software system (Linton, 1988). InterViews has been primarily chosen for this implementation because there is a close correspondence between its classes and the objects and concepts which are manipulated by the MOOD environment tools, such as graphical elements, texts and menus.

The *interactor* class is the most important class of InterViews. All interactive classes, such as *button*, *stringeditor* and *menu* are sub-classes of the *interactor* class. An *interactor* object manages some area of potential input and output on the screen and typically interprets input events and produces graphical output. The *interactor* class has associated attributes which define the shape and size of its objects. In addition, the *interactor* class defines a set of operations (which characterises the behaviour of its objects) such as draw, resize, redraw, update and handle events.

Figure 6.1 partially depicts the *interactor* class hierarchy. (For the sake of clarity, a slight simplification of attribute and operation names has been made). *Button* is an *interactor* sub-class which manipulates the value of an associated *button-state* attribute. A *button* object, when clicked, may initiate some actions. *Stringeditor* is an *interactor* sub-class which provides a convenient mouse-based interactive editor for text strings. It is suitable for incorporating into interfaces in order to get textual input.

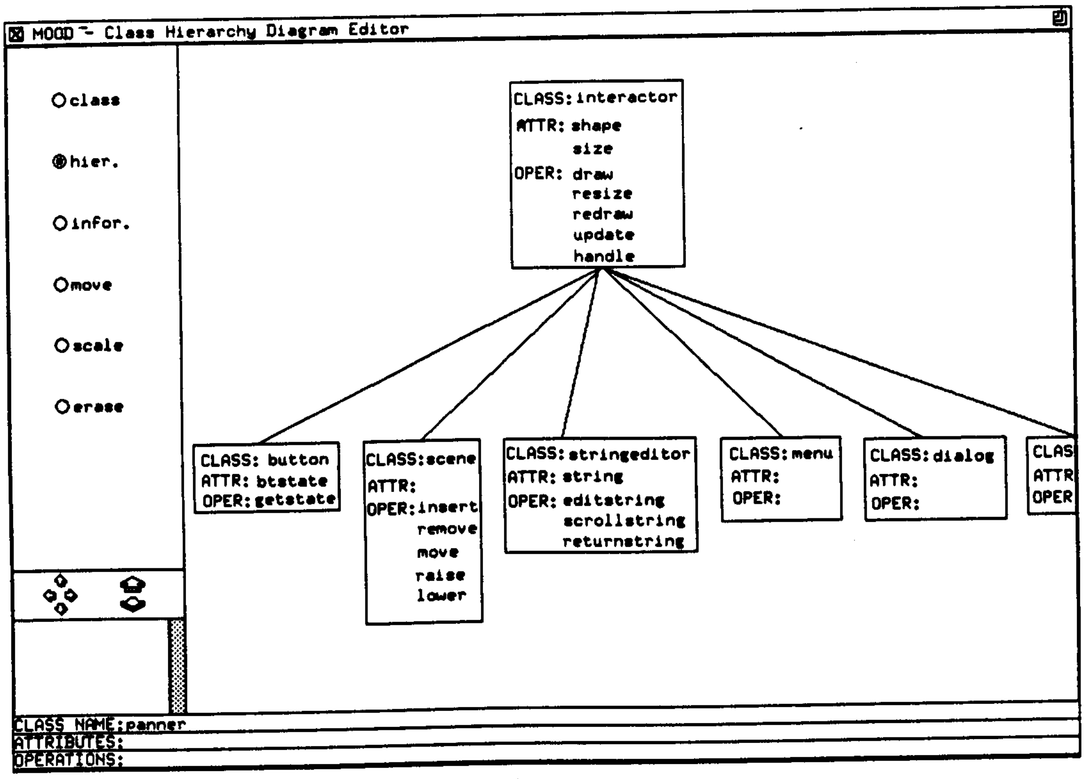


Figure 6.1 The Interactor Class Hierarchy

The *scene* sub-class defines basic operations, such as insert, remove, move, raise and lower, for managing compositions of *interactor* objects. *Scene* objects are used to compose a group of one or more *interactor* objects and, for instance, to tile them within *box* objects separated by *glue* objects. Therefore, complex software system behaviour can be accomplished by building compositions which combine the simple behaviour of several types of objects. Figure 6.2 presents a *scene* object composed of two *box* objects, and a *box* object composed of a *button* object and a *stringeditor* object separated by a *glue* object.

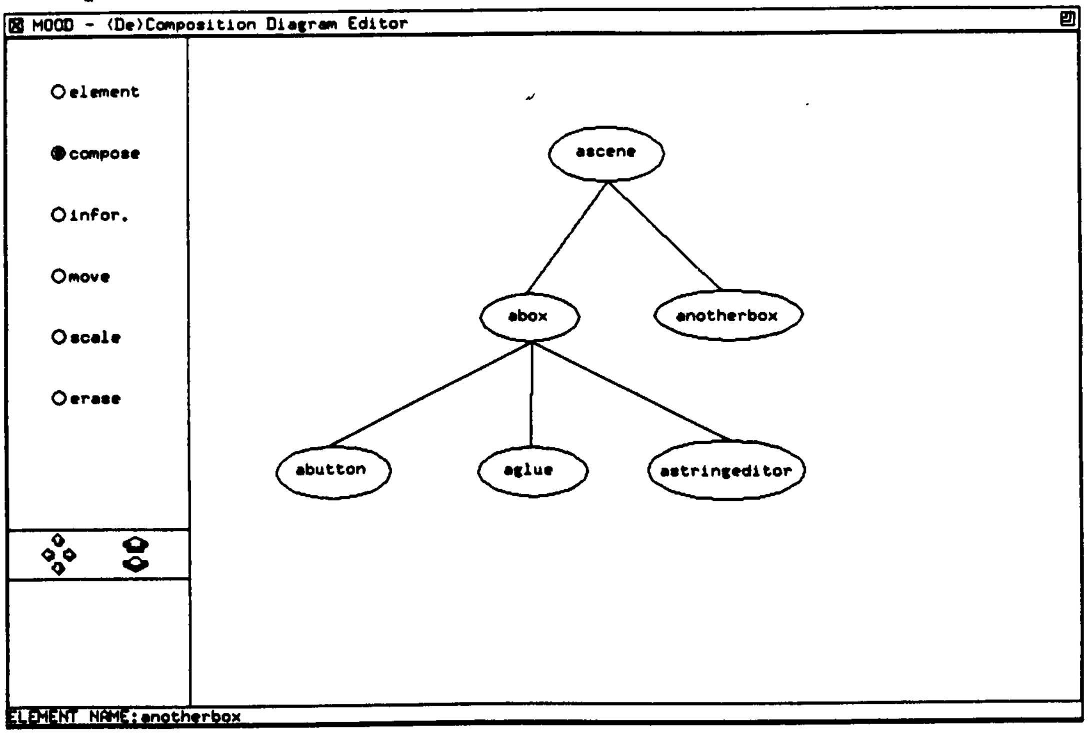


Figure 6.2 A Scene Composition

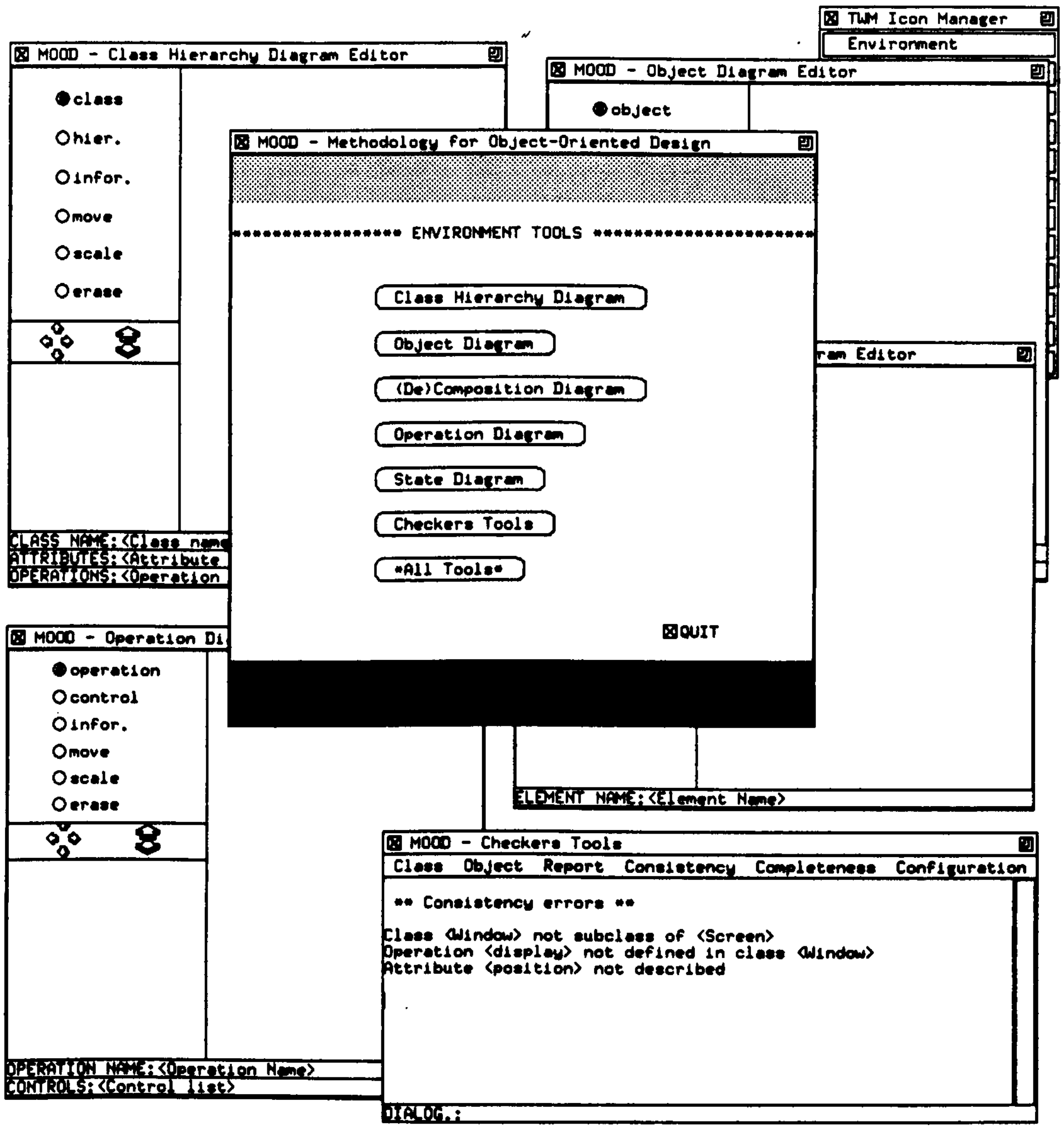
The *rubberband* class provides graphical feedback to a user during an interactive manipulation. This is very useful to implement the idea of animation of objects that are drawn, moved, resized, and scaled on the screen. A *rubberband* object varies its appearance as its tracking point is changed following mouse motion. For example, there are *rubberband* sub-classes for animating the creation of a rectangle defined by two points corresponding to opposing corners, the sliding of an ellipse from one point on the screen to another and the scaling of a circle around its centre. InterViews also provides mechanisms to handle scrolling, zooming and panning operations on a window (see the arrows at the bottom left of Figure 6.1).

6.2 THE MOOD INTERFACE

The objective of this section is to describe the uniform user interface for the MOOD environment. It is desirable to maintain a consistent "look and feel" fashion across the environment tools so that the user is not presented with completely different interface styles for each different tool used. Nevertheless, graphical tools (such as the Class Hierarchy Diagram Editor and the Object Diagram Editor) place emphasis upon icons and graphical elements on the screen. On the other hand, the Checkers are driven by options chosen from pop-up, pull-down and pull-right menus, and texts are displayed on the window. Naturally, there have been some slight differences in the look of some of the tools' interfaces because the nature of the tools is different.

The first window which appears on the screen when the MOOD prototype is executed allows the user to choose from the tools in the environment. The user can choose any diagram editors (Class Hierarchy Diagram Editor, Object Diagram Editor, Composition Diagram Editor, Operation Diagram Editor) or the Checkers, or invoke all of them. An example of this first window is given in Figure 6.3 which shows how the screen might appear when several tools are active at the same time. A tool can be active but not displayed; in this case that tool appears as an icon entry at the top right of the screen.

Figure 6.3 also shows that all diagram editors look similar because all of them manipulate mainly graphical elements, and one, two or three text lines. In contrast, at the bottom of that figure, the Checkers deals mainly with textual descriptions of design information. These two trends are discussed separately in the following subsections which describe the interface of the Class Hierarchy Diagram Editor as a representative example of a diagram editor, and the interface of the Checkers with some options available.



```
CLASS NAME:<Class name
ATTRIBUTES:<Attribute
OPERATIONS:<Operation
ELEMENT NAME:<Element Name>
OPERATION NAME:<Operation Name>
CONTROLS:<Control list>
```

```
** Consistency errors **

Class <Window> not subclass of <Screen>
Operation <display> not defined in class <Window>
Attribute <position> not described
DIALOG.:
```

Figure 6.3 The Tools of the MOOD Environment

6.2.1 The Interface of a Diagram Editor

The purpose of this subsection is to present the interface of the Class Hierarchy Diagram Editor. Other tools such as the Object Diagram Editor, the Composition Diagram Editor and the Operation Diagram Editor have been designed in a similar fashion.

Figure 6.4 presents the window of the Class Hierarchy Diagram Editor which provides services that allow the user to:

- draw, move, scale and erase rectangles representing classes;
- establish a class hierarchy;
- scroll the window in four directions;
- zoom in and out a class hierarchy diagram in its entirety.

This tool also allows the user to enter the text for the names of the classes, and their attributes and operations. This information defines the semantics of a class, is kept in the MOOD database and can be manipulated by other tools, such as the Checkers.

Each of the features described above supports animated feedback (follows the mouse motion) as the user creates, points at and manipulates classes as rectangles on the screen. Animation helps users to believe (or deceives them with the illusion!) that real-world objects are being dealt with, and in this way, it improves the friendliness of the interface. There is also a pop-up menu containing commands to open a new diagram, clear the canvas, invert the canvas background, save the diagram into the MOOD database, return the diagram to its original size if it has been zoomed in or out, delete the diagram, and to exit from that tool. The elements of this interface can be composed as illustrated in the MOOD composition diagram presented in Figure 6.5.

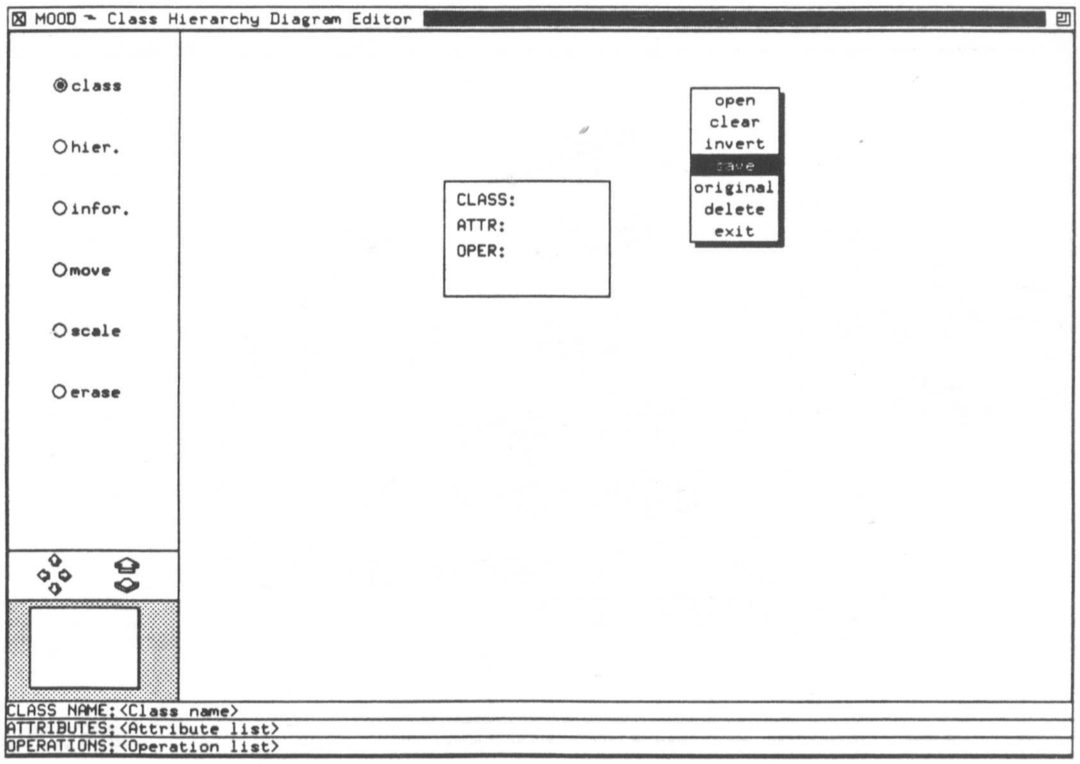


Figure 6.4 The Class Hierarchy Diagram Editor Window

6.2.2 The Interface of the Checkers

The Checkers are basically driven by menus. Figure 6.6 shows how its window appears on the screen. From a menu bar, the user can select a tool to activate a particular checking option. The menu bar is composed of a list of tools represented by menu items. (Figures 6.7 and 6.10, which will appear in the next section, show the class hierarchy diagram and the composition diagram for this interface). A tool is invoked by clicking on the corresponding menu item. A menu item can also be used to activate a pull-down or pull-right menu presenting a list of sub-options (commands) offered by that tool. When an option is chosen, a request is sent to the appropriate tool to perform the corresponding functionality.

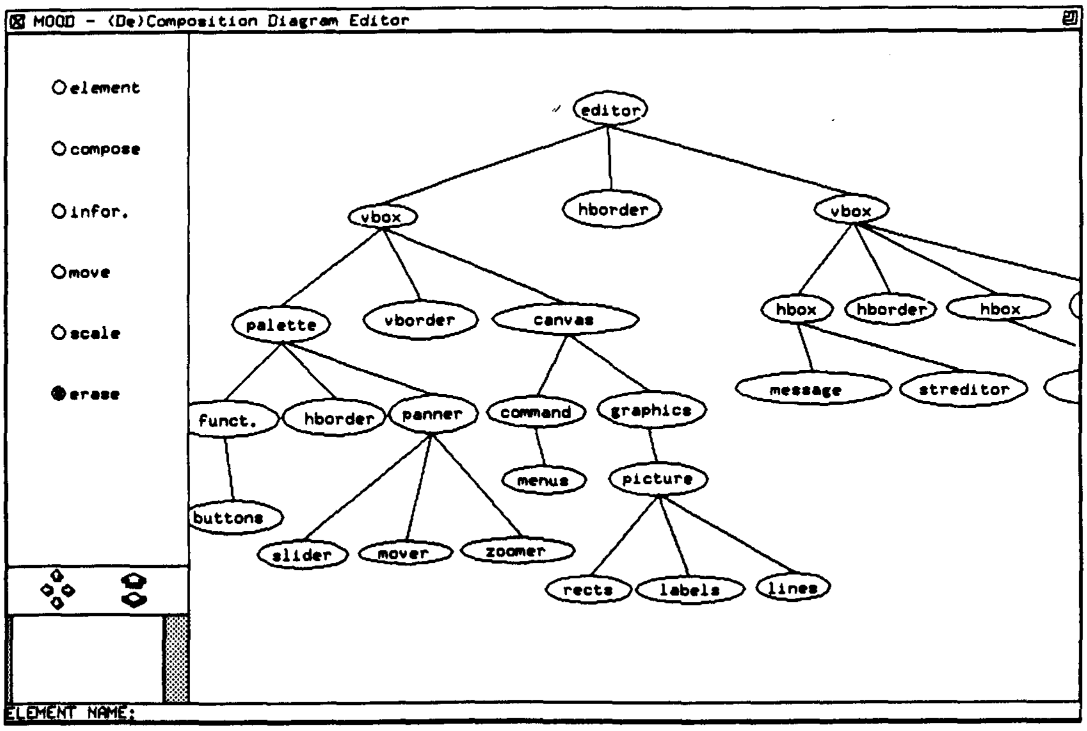


Figure 6.5 A Composition Diagram for an Editor

There is also a pop-up menu which can be pinned to or unpinned from the canvas and contains commands relevant to all Checkers. Such commands can be used to open and save the canvas in the MOOD database, edit (insert, move, copy and delete) the contents of the canvas, and to exit from the tools. These facilities let the user, for instance, select a class with the class browser, edit its attributes and operations, and then save the new version of that class in the MOOD database.

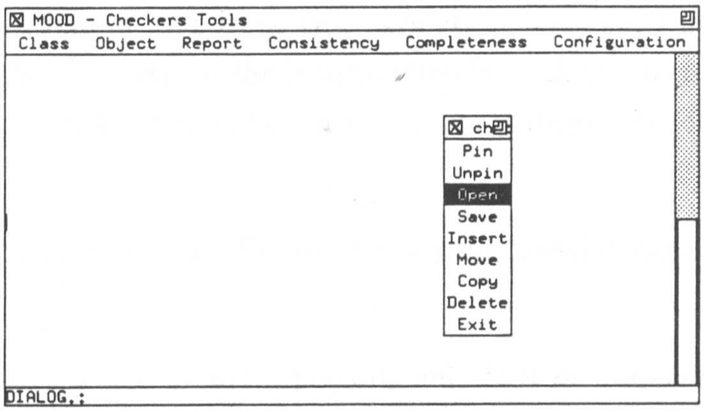


Figure 6.6 The Interface for the Checkers

6.3 THE MOOD DATABASE

The MOOD database offers a flexible way to store, recover and update design information which semantically represents classes, attributes, operations and objects related to the software system being designed. There is a straight mapping between the structure of the information kept in the tool memory and that stored in the database. Essentially, the philosophy of this implementation is record-oriented and the main advantage is that the database supports random access of any record location for a read or a write transaction and, of course, persistence.

6.3.1 Persistence

Persistence is the property of an object by which that object continues to exist after its creator ceases its execution. The object can also be moved from the address space in which it was created to another place.

There are various ways of providing persistence in software systems, and the use of a file is the most common. In this case, persistence may be seen as a storage mechanism for transient structures in memory. The basic idea is to be able to transfer information from transient structures to persistent storage and back. There should be explicit operations which write and retrieve information to and from the persistent storage. Such operations must preserve the structure of the information in both memories so that after each transaction the transient and the persistent structures remain consistent.

There are four main reasons for implementing persistence in the MOOD database:

- It provides a mechanism for passing objects from one tool to another.
- Persistent objects can resume their existence in a later use of the environment.
- Persistent objects free users from worrying about storage.
- Persistent objects are often useful for historical purposes.

6.3.2 Manipulation of the Design Information

The MOOD database consists of two classes: the *filesystem* super-class and the *database* sub-class. The *filesystem* class deals with the physical aspects of the storage mechanism, and the *database* class copes with logical aspects. By creating two general-purpose classes the designer was forced to consider which were the essential features of the software system and to describe these features in the highest super-class.

The *filesystem* class has been designed and implemented to encompass the necessary transactions on the MOOD database. It offers operations to open and close a file, to seek any record in a file, to read and write a record in a file, and to handle errors. The *filesystem* class is a parameterised or generic class. Parameterised classes represent an ordered collection of objects, in general. Good examples of such classes are linked-lists, trees, and naturally, files of records. Indeed, the *filesystem* class represents a logically ordered collection of fixed length records within a file.

The *database* class provides operations to add, delete and update design information in the MOOD database. The *database* class is more that a simple structured file system and supports most of the services used to manipulate the semantic information of the software system under design. This class affords addition, access, update and delĕtion of classes, attributes, operations and objects of a developing software system in the MOOD database.

These operations are called by tools when a MOOD diagram is being edited, or checks are being performed. All design information displayed by a tool must be stored or updated in the MOOD database. This updating is based on the information kept in the tool memory, which is directly manipulated by that tool, and also stored in the MOOD database. The advantage of using a uniform object-oriented representation model is to facilitate the manipulation of the design information because inside the MOOD database everything is an object, and only objects, not combinations of different concepts such as functions, data, archives, and so forth.

6.4 THE TOOLS

Good designs come from good designers, not just from good tools. Although methodologies can be empowered by tools, tools by themselves cannot guarantee the production of a good design. There are some things which tools can do well and others which they cannot do at all. For instance, tools can make it easy to discover the class in which an operation has been defined, although tools cannot guide designers to invent new classes, which would simplify the class hierarchies.

This section presents prototypes of tools that are useful for the object-oriented design methodology which underlies the MOOD environment. This environment is an integrated CASE environment that facilitates the design of object-oriented software systems. The tools described below compose the MOOD prototype that has been implemented. The tools interact with each other sharing the design information retained in the MOOD database. Such tools can be used in a generic object-oriented design and are, therefore, independent of any particular programming language.

6.4.1 The Class Hierarchy Diagram Editor

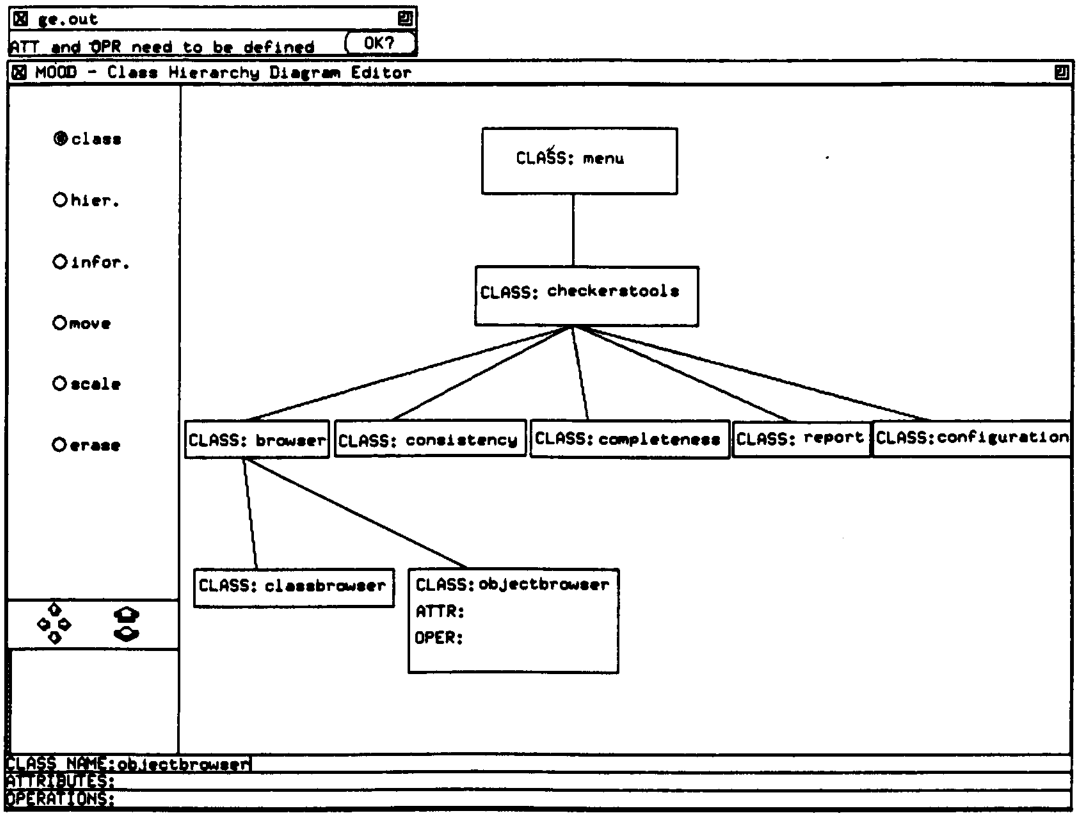
This tool automates the construction of the class hierarchy diagrams as presented in Chapter Four. Such a tool can be used very early in the design process (static design) to enforce the notational conventions of MOOD for that kind of diagrams. Furthermore, the Class Hierarchy Diagram Editor is likely to be used throughout the proposed software development life cycle because as a design evolves from the system analysis phase into the implementation phase, the class hierarchy diagrams are refined.

As an example, Figure 6.7 presents the class hierarchy diagram for the Checkers interface, as discussed previously. At the top of the window a warning message is shown to request the definition of the attributes and operations of the *objectbrowser* class which is just defined. If the requested information is not available at the moment, the user has to click the mouse inside the *ok* button in the *dialog-box* object presenting the warning.

6.4.2 The Object Diagram Editor

Such a tool is used to build object diagrams according to the MOOD notational conventions. The Object Diagram Editor might be the first tool to be used by novice designers, who may decide to experiment with some specific objects in order to understand their behaviour or to begin a bottom-up design.

Figure 6.8 illustrates an example of an object diagram in which an input information is requested by an object named *aneditor* (which is an instance of a *diagram* class) to an *astringed* object through the *returnstring* operation. Eventually, this input information is dispatched to the appropriate object for further processing which might be passing that information to an *acanvas* object to be displayed as a rectangle, and then to an *areclogic* object to be stored in a database as part of a class definition. Of course, at the end of a design, the classes and operations shown in object diagrams must be in accordance with those described in class hierarchy diagrams. These kind of checks are carried out by the Checkers, described later in this section.



ATT and OPR need to be defined

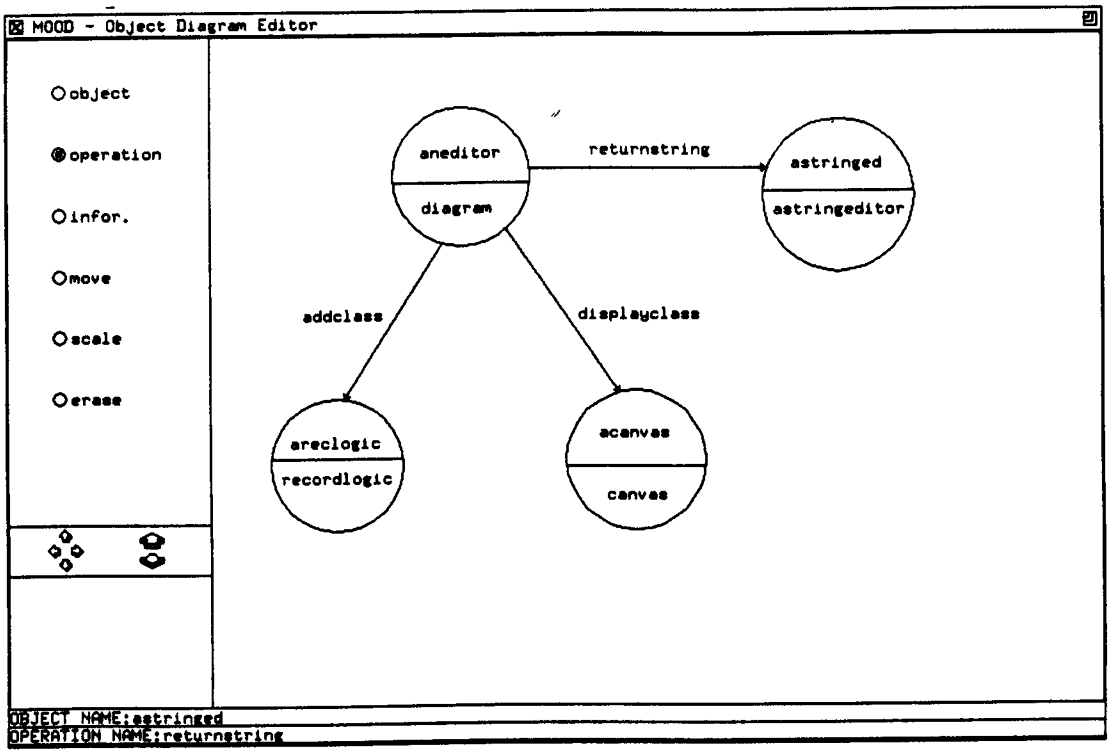
CLASS NAME:objectbrowser
ATTRIBUTES:
OPERATIONS:

Figure 6.7 The Class Hierarchy Diagram Editor

6.4.3 The Operation Diagram Editor

The construction of operation diagrams is supported by the Operation Diagram Editor. Figure 6.9 shows an example of an operation diagram which provides the service of defining a class. The *createclass* operation (for objects of a *diagram* class) passes control to the *returnstring* operation and gets control back with a *string* output parameter.

Afterward, the *drawgraphic* operation is called to draw the graphics associated with a class. This includes the drawing of labels for the class name together with its attributes and operations. Then, the *addclattoper* operation is called to store all the necessary information related to the newly created class, in the MOOD database. The names which appear near the small



OBJECT NAME:astringed
OPERATION NAME:returnstring

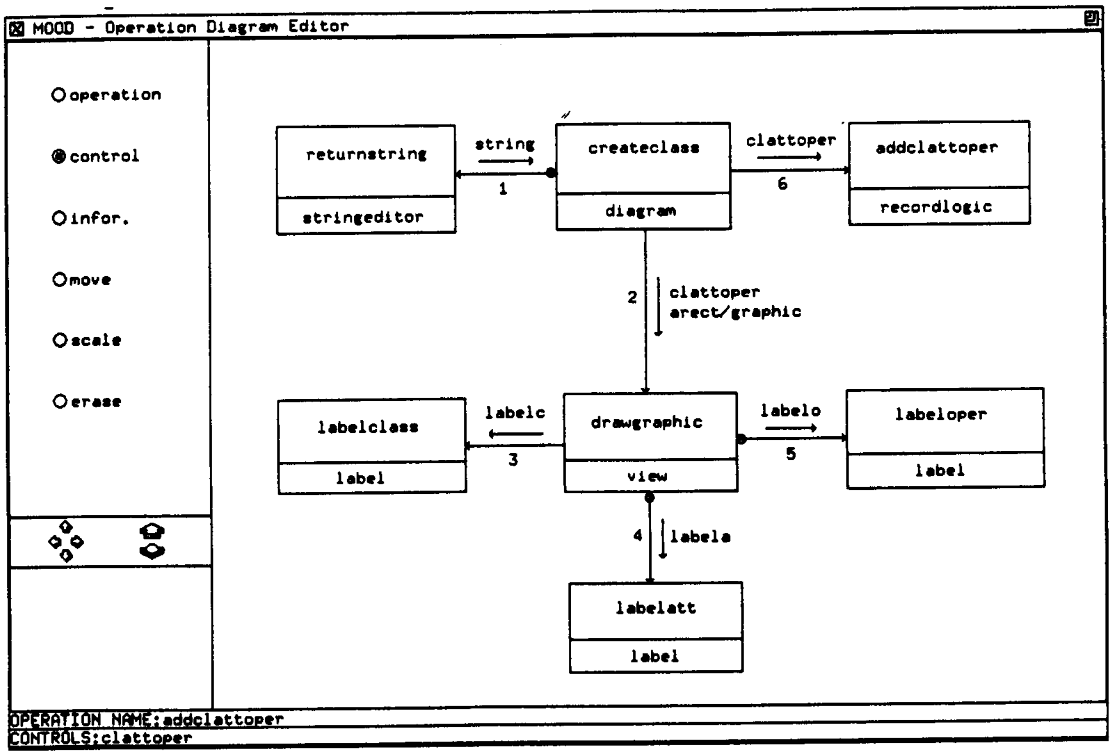
Figure 6.8 The Object Diagram Editor

arrows are parameters for the corresponding operations, as explained in Chapter Four.

6.4.4 The Composition Diagram Editor

This tool is used to produce composition diagrams, which depict the main components (software items) of a software system. For instance, Figure 6.10 shows the composition of an *amenubar* object of the Checkers described previously. That *amenubar* object is composed of six menu components (*classitem*, *objectitem* and so on) which correspond to the options offered by the Checkers.

Some of these menu components are further decomposed into their constituent components, for instance, the *configitem* component is



OPERATION NAME:addclattoper
CONTROLS:clattoper

Figure 6.9 The Operation Diagram Editor

decomposed into the *manageitem* and *versionitem* sub-components and so on. These sub-components can be related to sub-options or commands particular to each tool (as can be seen in the next subsection).

6.4.5 The Checkers

The set of tools known as Checkers have been put together because they use a common interface which deals basically with displaying and editing strings of texts. Some Checkers are used to point out to users any problem related to inconsistencies or incompleteness in a design. The Checkers complain about perceived problems but let the users decide whether the complaints are valid or not and how to fix them.

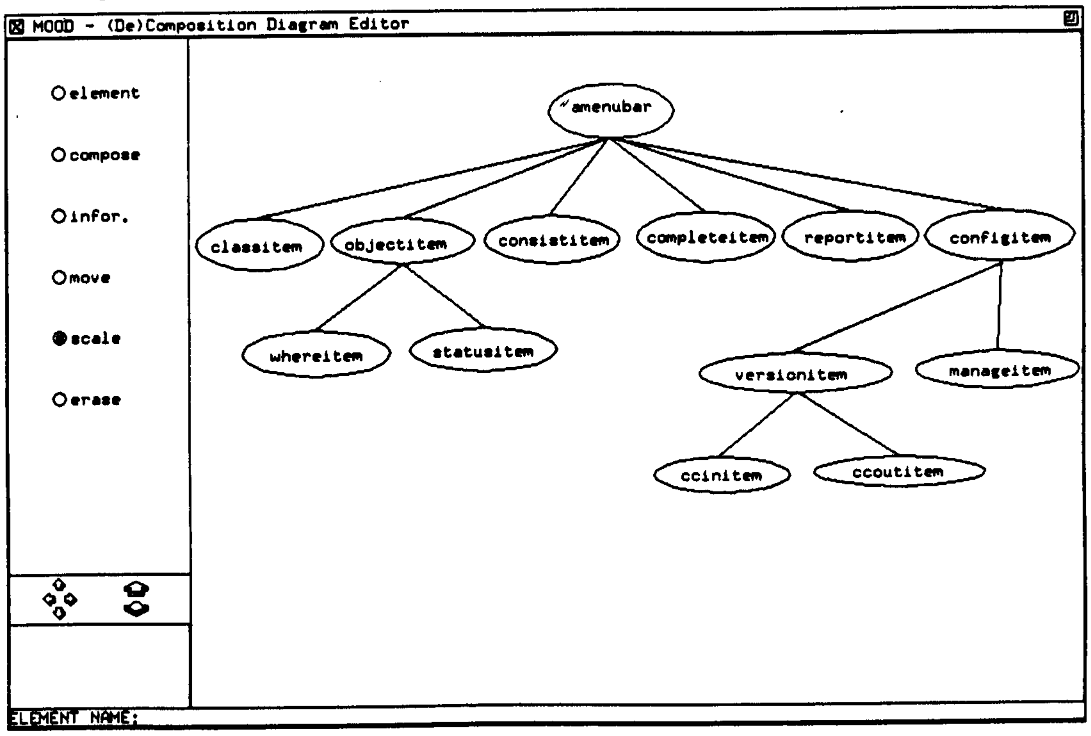
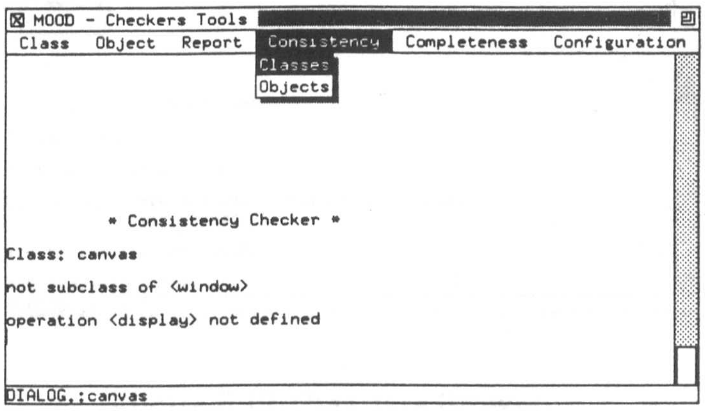


Figure 6.10 The Composition Diagram Editor

In order to resolve such problems, it is also necessary that the user be able to browse through the classes, inspect the objects, and even produce a report from the contents of the MOOD database. Browsers are interactive tools which provide views of the class definitions and class hierarchies. Additionally, browsers help navigate through existing classes and invite the user to create new classes. Furthermore, while using the diagram editors presented in the previous subsections, the user has freedom to speculate among several choices and experiment with different options. Therefore, a configuration management tool is necessary for controlling different versions of a design.

Consistency and Completeness Checkers

These tools are used to evaluate the consistency and completeness of a design which is held in the MOOD database, by checking the interrelationships of the design information in different diagrams, for example, whether class names which appear in an object diagram are defined in any class hierarchy diagram. Checks can be made to ensure that definitions presented in one diagram are consistent with those in another and that rules of the methodology are followed. The consistency tool checks for rules that are not allowed to be violated, which means that any rule of MOOD was disobeyed, as shown in Figure 6.11.



```
* Consistency Checker *
Class: canvas
not subclass of <window>
operation <display> not defined
```

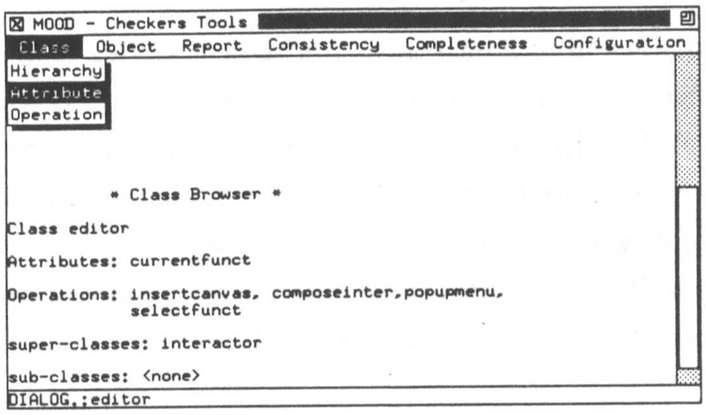
Figure 6.11 The Consistency Checker

The completeness tool basically points out any essential details which have been so far omitted from a design. Typical completeness checks include:

- all classes must have unique names;
- all attributes and operations within a class must have unique names;
- overridden attributes and operations within a sub-class must be defined in any super-class in that class hierarchy.

Browsers

The class browser knows about class hierarchies and allows the user to navigate through classes and their class hierarchies within the MOOD database. Class hierarchies can become so complex that without the help of browsers it is difficult even to find the classes which are part of a design or components which are candidates for reuse. With the class browser the user is able to examine all classes available in the MOOD database. The class browser typically shows class hierarchies, their attributes and operations, as illustrated in Figure 6.12.



```
        * Class Browser *
Class editor
Attributes: currentfunct
Operations: insertcanvas, composeinter,popupmenu,
            selectfunct
super-classes: interactor
sub-classes: <none>
DIALOG.;editor
```

Figure 6.12 The Class Browser

A report generator basically produces reports from the design information retained in the MOOD database. The facilities provided by this tool allow the user to enter the name of a diagram and get all information related to that diagram. Other capabilities include, for instance, the ability to find the class which defines a particular attribute or implements a particular operation.

Facilities to browse the MOOD database can save the user many hours of work. Therefore, it is worth spending some time browsing through classes, and if there are no suitable classes for reuse it might be worth building generic classes which could be reused in future software development. The more time is spent in such browsing activities, the more is the payoff for reusability in future software systems. For this reason, browsers are imperative tools for an object-oriented design.

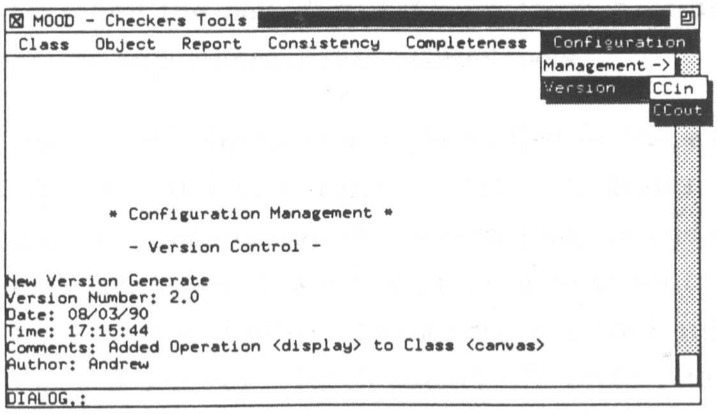
Software Configuration Management

Software configuration management is one of the biggest problems facing designers of large software systems. This activity is concerned with controlling the evolution of software systems (Bersoff, 1984; Tichy, 1985). With large software systems and several designers working together, designers must be sure that having made use of a particular class, the version of that class will remain unchanged.

Classes are the best software configuration items to be controlled in object-oriented software systems, and it is useful to keep track of different versions of classes which make up a particular version of a software system. Therefore, a software configuration management tool is necessary to control those versions. All changes must be automatically logged and the configuration management tool should permit the re-creation and inspection of earlier versions, and it could also point out the differences between versions.

Versioning is a particularly appealing technique for managing class evolution because it enables designers to try different paths when modelling complex real-world applications, and to record the history of class modifications during the design process. The hierarchy of versions is somewhat similar to the class hierarchy. Each version is complemented with generic information and specific information. Generic information includes those features which are common to most of the versions, while specific information contains variations which are associated with a particular version.

The MOOD configuration management tool provides capabilities for the modification of classes and for keeping different versions of classes. The version control which traces the history of class modifications uses a derivation tree which keeps track of the differences between two adjacent versions of the same class. Version numbers are used to distinguish between various versions of a class. The configuration management tool provides facilities to manipulate different versions, as presented in Figure 6.13. The *CCin* command gets any previous version of a class and the *CCout* command stores the current classes as new versions.



```
* Configuration Management *

 - Version Control -

New Version Generate
Version Number: 2.0
Date: 08/03/90
Time: 17:15:44
Comments: Added Operation <display> to Class <canvas>
Author: Andrew
```

Figure 6.13 The Configuration Management Tool

6.5 CONSIDERATIONS ON THE MOOD PROTOTYPE

The design phase is a stage of refinements of an abstract model of the application and it is essentially a creative and iterative process, not a mechanical one. The iterative nature of the design process is reflected in the way in which the MOOD prototype operates and in the chance that users have to choose the most appropriate tool within that environment. There are several independent tools, but with functionality linked in such a way that these tools cover all design stages, and information supplied by a particular tool is accessible to others. The integration between these tools is achieved through the use of a unified representation model, a single database and a uniform interface which establishes the communication protocol between the tools, and between the environment and its users (software designers).

As discussed in the previous chapter, the design phase strongly depends on the knowledge that designers have about the application domain. This experience will guide designers to use the most appropriate tool. During the static design, the most abstract aspects of a software system are picked up, therefore the Class Hierarchy Diagram Editor is more likely to be used. On the other hand, in the dynamic design, the behaviour of the software system is understood in more detail and the use of the Object Diagram Editor and the Operation Diagram Editor are more suitable. During object-oriented software development, these two design sub-phases can become blurred and iterative, and therefore the boundary between them becomes even more indistinct. Hence, the importance of interrelated tools.

The fact that classes and objects can be identified in the early phases of software development, and can undergo detailed design and may be implemented separately means that the system analysis and design phases spawn classes and objects which are to be passed on to an implementation phase. The dealing with a unified representation model makes iteration between the proposed software development life cycle phases a smooth process. Even though a considerable amount of work remains to be done, the design of a prototype capable of showing how MOOD could be supported in a CASE environment has been implemented. The MOOD prototype provides a user-friendly interface and supports a centralised and uniform storage of the design information in a single database.

Chapter 7

EXPERIENCE WITH MOOD

This chapter discusses some of the results from experience with the development of the MOOD prototype using MOOD itself. While the chapter has not attempted to produce a comprehensive study and a detailed design of a software system, it has provided an evaluation of MOOD as a design methodology.

The first section of the chapter highlights the design aspects in which problems have been found and also presents an evaluation of the proposed methodology. Section two covers other general aspects which form a more complete assessment of the whole software development process, not only design. For instance, the abstraction mechanisms most frequently used, and an estimate of the time and effort spent in each phase of the proposed software development life cycle.

7.1 THE SUITABILITY OF MOOD

The main aspects of MOOD, which have been under consideration in this section, relate to its graphical notations and the feasibility of its steps to design object-oriented software systems. Such discussion also helps improve the MOOD methodology by providing the necessary feedback based on the results of a real design.

Partitioning Issues

In order to manage the complexity inherent in the construction of a CASE environment for MOOD, the decomposition of such an environment into a number of smaller tools which can be seen as large components, is recommended. The main purpose of such decomposition is to relate some classes to a context which provides particular functionality. Thus, the division is based on the functionality provided by the various tools. Further decomposition of large components into sub-components was not at all intuitive and accentuated the problems of dividing functionality into classes. Additionally, the methodology emphasises an early decision about separation between attributes and operations. It also recommends early judgment about the choice between the use of composition and inheritance for software reuse.

These observations lead to the following conclusions about the relationship between functionality, classes and reusability. The notion of class hierarchy is not enough to divide object-oriented software systems into manageable pieces. It is important to realise how orthogonal the concepts of inheritance (employed in class hierarchies) and composition (associated with functionality) are. Inheritance can be taken to express an *is-a* relationship (that is, an integer is a kind of a number), whereas composition expresses a *uses-a* relationship (thus, a number uses digits). Hence, while class hierarchy diagrams aim to represent generalisation and specialisation mechanisms, composition diagrams convey decomposition and composition concepts. As a result, composition diagrams were introduced to MOOD in order to supplement class hierarchy diagrams. Together, such diagrams enable MOOD to provide an adequate representation for large software systems and their functionality.

Despite class hierarchies being a crucial aspect of the object-oriented paradigm, some drawbacks associated with a large number of classes and multiple inheritance have been noticed. Experience has shown that large class hierarchies overwhelm designers when they attempt to understand the overriding process, especially when many levels of sub-classes are used and multiple inheritance occurs. Designers of deeply-nested class hierarchies have problems grasping the many different super-classes and sub-classes, and their inherited properties. Possible features which could be integrated in browsers are global views of the system architecture in order to help designers to depict their position in the software system.

Representation of the Control Flow

The decomposition of object-oriented software systems into components disperses their control flow and obscures the operations present in their class hierarchy diagrams. Such dispersion is due to the spread of the control flow among several objects, making the global control flow less visible in object-oriented software systems than in software systems constructed by functional decomposition. A possible reason for this dispersion of the control flow is that unlike functionally decomposed software systems, where a high level function could be seen as an abstraction of lower level subfunctions, object-oriented software systems could be viewed as an aggregation of independent objects, which might be running virtually in parallel.

The software system behaviour depends on the overall interactions among objects as well as on the control flow within each operation. Object diagrams and operation diagrams help designers to understand the control flow within a software system and make it possible to realise interaction patterns to achieve particular functionality. Such diagrams have demonstrated themselves to be a useful design aid for the understanding of the interactions between objects and their interfaces. In particular, operation diagrams have proven a valuable means of representing the control flow of object-oriented software systems.

The dispersion of control flow in object-oriented software systems also brings about difficulties in treating exceptions because the exception handling operations will require recovering from exceptional conditions in many different circumstances. One idea which might have been considered is the possibility of designing the MOOD prototype so that each of its class hierarchies had a handler in order to minimise the negative effect of having one exception handler within each main operation. Trials should be extended to further investigate the effects of exception handling mechanisms upon the overall control flow when exception handlers are put either into super-classes (and then inherited by their sub-classes) or in a special class hierarchy of exceptions.

State transition diagrams are useful to capture aspects of the behaviour of software systems and to identify operations to be implemented by different objects. However, this technique has inherent limitations: in particular, the inability to express recursive applications, and the difficulties of describing algorithms which are distributed between two or more different kinds of objects. As well, in many realistic real-world applications, there are simply too many events to be represented within a state transition diagram.

Object Interactions

In a synchronous communication model, requests are sequential; a client object requests an operation and waits idly for a response from a server object before continuing execution, implying that a single thread of control is passed from one object to another. Such object interactions are simplistic and not rich enough to describe all kinds of interactions between objects.

With asynchronous communications, a client object is free to take further action after requesting a particular operation from a server object. A client object may proceed concurrently without waiting for a server object to reply. Concurrency fits quite well within an object-oriented approach because the autonomy of objects makes them a natural unit for concurrent execution. However, object diagrams do not provide clear notation to show concurrency and no account of timing is made. Therefore, determining an appropriate set of additional notational conventions to represent sequence and concurrency in object diagrams should be investigated.

The methodology does not consider issues concerning dynamic objects, that is, when the number and type of objects varies as the software system runs. As well, it has ignored problems arising from sharing of objects, especially the accidental sharing that can occur in programming languages where two objects can both reference the same third object. These shortcomings are worth examining and appropriate additions to the methodology to cope with such designs should be proposed.

7.2 GENERAL ISSUES RELATED TO MOOD

The remainder of this chapter contains some general issues, and reusability in particular, which are associated with MOOD within the proposed software life cycle model. The aspects to be discussed also include the prevailing abstraction mechanisms employed in each phase of the software development life cycle and an estimate of the percentage of time spent in each phase.

Reusability

There are differences in the mechanisms used to achieve reusability when different kinds of reusable components are involved. The most basic software components (i.e. objects) are often reused by composition which can be seen as a process of building a piece of software from elementary self-contained components. Nevertheless, reusability is better linked, from an object-oriented point of view, with reuse of classes through inheritance. In this case, it takes place by specialisation and generalisation of commonalities between classes. Not all classes identified early in software development are implemented because some of them can be refined during the design phase or taken from a library of reusable components. It is better to reuse high level components such as classes during design because they have fewer implementation details which would constrain their applicability.

There is a need for tools to support the creation of domain-specific collections of components or contexts (also known as frameworks) which could form a reusable library for an application domain. A context could be viewed as a generic structure which provides a skeleton for designing software systems in a particular application domain. It is sometimes necessary to adapt the new developing software system so that it can be fitted within an available context, which results in a tremendous gain in productivity. Many advantages have come from the adoption of contexts. In this aspect, several independent reusable contexts are more effective for reuse than a single universal library of components. Therefore, rather than developing a single library for a centralised repository of components, the strategy should be the development of a reusable library for each particular application domain.

Reusability through inheritance and composition has been largely applied during the design of the MOOD prototype. The primary objective has been to design the environment interface reusing, as much as possible, software components from the InterViews library. Reusable contexts such as InterViews provided a good incentive to implement object-oriented graphical interfaces as an independent software component. Therefore, an object-oriented approach was introduced as early as possible during software development because abstractions were based on classes and objects in an application domain for which reusable components were available, increasing reusability considerably.

Software Development Life Cycle Issues

The experience of using MOOD to design a large and complex software system, such as the MOOD environment, has firstly shown that it was very difficult to follow either a strict top-down or bottom-up approach, and that it was necessary to switch between both approaches. This implies that it is useful to identify high level functionality in the software system along with the identification of some objects and their interactions. As a result, when developing large software systems, it is important to synthesise ideas from both top-down and bottom-up approaches, and to relate how classes and objects provide particular functionality.

One great advantage of using the object-oriented paradigm is the conceptual continuity across all phases of the software development life cycle. The conceptual structure of a software system not only remains the same from system analysis down through implementation, but also remains the same during the refinement of a design. Therefore, when the object-oriented paradigm is used, the design phase is linked more closely to the system analysis and the implementation phases because designers have to deal with similar abstract concepts (such as classes and objects) throughout software development.

Mechanisms Prevalent in each Development Life Cycle Phase

The four phases of the proposed software development life cycle are highly integrated. Table 7.1 shows the mechanisms used most often in each phase of the software life cycle model followed during the construction of the MOOD prototype. These mechanisms are part of an abstraction process inherent to object-oriented software development as discussed early in Chapter Two.

The system analysis phase emphasises classification of high level concepts in a real-world application and decomposition of a software system. Several mechanisms are relevant to the domain analysis phase but specialisation, generalisation and composition are fundamental in order to achieve reusability. During the design phase all mechanisms are of paramount importance as can be realised from the MOOD steps along with its diagrams. In the implementation phase, almost all mechanisms are essential except for decomposition since at this phase the partition of a software system will have been done.

phases → mechanisms ↓	**System Analysis**	**Domain Analysis**	**Design**	**Implement.**
classification	●	●	●	●
instantiation			●	●
generalisation		●	●	●
specialisation		●	●	●
decomposition	●		●	
composition		●	●	●

Table 7.1 Phases Versus Abstraction Mechanisms

Percentage of Time per each Development Life Cycle Phase

Although it is difficult to draw distinct lines between two adjacent software life cycle phases, the mechanisms more frequently used in each phase can be pointed out. Figure 7.1 shows an approximate percentage of the amount of time likely to be spent in each software life cycle phase during the complete development of a software system. These times were taken from the development of the MOOD prototype. Despite the system analysis, design and implementation phases being deeply interrelated, it is clear that the design phase is the longest and most important because most of the tasks are done during that phase.

Domain analysis is relevant to identifying potentially reusable components during object-oriented software development. Consequently, the amount of time spent in this phase, naturally, must not be longer than that spent in other phases. If the perceived cost of finding a certain component is higher than the cost of creating a new component from scratch, then all hope for reuse is lost. For this reason, it is important to have at least some minimal librarian tools which allow designers to locate and add useful components as they are identified.

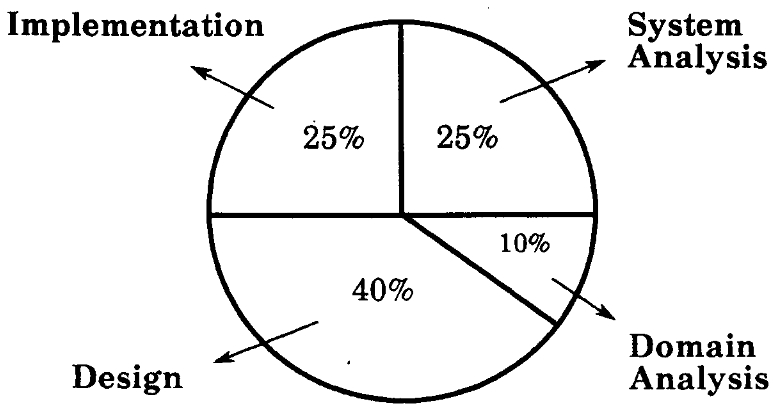


Figure 7.1 Phases Versus Time

Although maintenance accounts for the vast majority of software system costs, it is not considered in Figure 7.1 because it can be seen as an operational phase which succeeds software development. It is felt that the basic reuse issues which MOOD encourages would form a useful basis for supporting software maintenance and evolution, although enhancements to MOOD to encompass these aspects is an area requiring further research.

This chapter has provided some remarks on the experience gained from designing and experimenting with MOOD within the proposed software development life cycle. It can be summarised from such experience that MOOD brings about some characteristics which might be considered inexpedient by some, but it is believed that the benefits from the use of MOOD outweigh its drawbacks.

Chapter 8

CONCLUSIONS

This thesis has concentrated on object-oriented design methodologies for software systems. The purpose of this final chapter is twofold: firstly, to discuss the research which has been done together with its main contributions; and secondly, to suggest other possible directions and ideas for future research concerning object-oriented design of software systems.

8.1 DISCUSSION

With a topic as broad as the object-oriented paradigm, whose literature offers a myriad of different interpretations and points of view, it has been difficult to give a precise definition for object-oriented concepts. Since there have been many subtle flavours which might be combined to give the overall picture of an object-oriented approach, it has been better to characterise an object-oriented model. Therefore, this thesis started by giving a characterisation of an object-oriented model based on what the author felt to be its most relevant concepts (such as classes, objects and inheritance, and their background) along with the philosophy of object-oriented design (see Chapter Two).

This thesis faces the object-oriented paradigm from a methodology standpoint, rather than from an implementation standpoint. The research was prompted by the perceived inadequacy of existing object-oriented methodologies for designing object-oriented software systems, and has sought to establish a viable and comprehensive object-oriented design methodology which obtains the benefits of the object-oriented paradigm, such as encapsulation and inheritance. To reiterate, the four major contributions which have been achieved by this research are:

- The setting of a classification scheme for object-oriented methodologies and the presentation of the state-of-the-art of object-oriented analysis and design methodologies, outlining their problems and limitations (Chapter Three).
- The development of a methodology for object-oriented design (MOOD) (Chapter Four) which can be used to design software systems based on an object-oriented approach; and to evaluate its applicability by partially designing two software systems (Chapter Seven).
- The proposal of an alternative software life cycle model for object-oriented development which takes reusability into account and considers how the knowledge that designers have about the application domain affects software development (Chapter Five).
- The implementation of a prototype of an integrated CASE environment, and its associated tools for object-oriented design, which supports the MOOD methodology (Chapter Six).

There have been many claims about the object-oriented paradigm regarding power of representation, maintainability, and clarity and correctness of object-oriented software systems. However, at present few of such claims can be fully validated because most of the attention has been focussed on the implementation phase. From the point of view of programming languages, the road to an object-oriented approach is an evolutionary step, whereas from the point of view of software development methodologies, the differences which exist between traditional structured development methodologies (based on functional decomposition) and those based on an object-oriented approach suggest that a revolution is taking place.

Although a number of object-oriented design methodologies are becoming available and gaining increasing use in order to answer a broad range of software engineering questions, such methodologies are still at a relatively early stage of growth. It is clear that even more experimentation is required (particularly in developing very large software systems using an object-oriented approach) before this paradigm can claim to be a mature subject.

Additionally, new experiments will provide several case studies to evaluate the various claims made about the object-oriented paradigm, which can only be fully tested when applied to developing substantial software systems.

The outcome of such experience will progress towards a better understanding of the strengths and weaknesses of an object-oriented approach to software development and might also lead to a re-evaluation of some claims made about it in recent years. Therefore, further experimentation is expected to result in a better understanding of object-oriented methodologies independent of any programming language; the consolidation of object-oriented concepts and terms; the dissemination of well-accepted notations and the appearance of several libraries of reusable software components for particular application domains.

The Assessment of MOOD

As presented in Chapter Three, there are design trends which try to integrate object-oriented concepts with different methodologies such as Structured System Analysis and SADT. There also are other methodologies which lead to implementation using Ada. Nevertheless, this research has sought to show that only one approach, in this case an object-oriented approach, is enough to develop software systems and it does not need to be complemented with other approaches or methodologies. This point of view encourages the designer to conform to the object-oriented paradigm and to benefit from features such as abstraction, encapsulation, inheritance and reusability, from the beginning of software development.

MOOD produces a design by progressive refinement, adding details to the same design model which is strictly object-oriented, and remains consistent through the design phase. Another result of the research described in this thesis has been the creation of a graphical notation to represent object-oriented design, which makes use of class hierarchy diagrams (identifying attributes and operations), composition diagrams, object diagrams and operation diagrams. In addition, since the use of MOOD results in an object-oriented design, it encompasses many of the benefits claimed to be inherent in any object-oriented software system, such as clarity, modularity and extensibility.

Methodologies are especially important for developing large scale software systems. Only recently however, have object-oriented methodologies been exposed to many designers. MOOD has already been applied by the author to the design of a small number of medium size software systems, but naturally it can evolve and mature from additional experience with its application to develop large software systems. Of course, further use of MOOD may indicate some areas of refinements, for instance in the separation of class features into attributes and operations, although such refinements would not constitute a major change to MOOD. So far, MOOD has proved to be beneficial and has led to a better understanding of object-oriented design.

The MOOD approach for design could be thought of as a rigorous one as it lies between an informal and a formal approach. Informal approaches are based on natural language descriptions but suffer from the ambiguity intrinsic in the use of a natural language. Rigorous approaches have a well-defined syntax and may not be ambiguous. However, they do not have a well-defined semantics so it is difficult to prove their correctness. Formal approaches can derive proofs by rules of logic that a design is correct, but they are difficult to be used by ordinary designers. MOOD, as currently defined, practised and used is a pragmatic and systematic approach to designing software systems.

This research has also shown how an object-oriented approach can permeate the entire software development process from system analysis to implementation, and can affect the whole software engineering one way or another. Hitherto, there is no object-oriented software life cycle model which has yet gained great acceptance. However, by trying to incorporate the advantages of an object-oriented approach and encompassing MOOD, the phases proposed in Chapter Five seem appropriate. Therefore, a new software development life cycle, linking system analysis, domain analysis, design (static and dynamic) and implementation to form a coherent software development life cycle, has been put forward in this thesis.

The proposed software development life cycle considers iteration, takes into account the knowledge that designers have about the application domain (which dictates a bottom-up or top-down approach to software development) and incorporates reusability as a natural part of the design phase. Reusability is a main concern of MOOD and domain analysis is a good framework to encourage it, built upon the mechanisms of inheritance and composition. The use of reusable libraries is a major plus for productivity; however, there is an overhead cost in developing a component for use in a specific product while making it generic and robust for future reuse.

The employment of a uniform object-oriented model from system analysis to implementation also facilitates a consistent binding between the various phases of software development. Therefore, it would be better if the implementation of an object-oriented design were carried out using an object-oriented programming language because concepts presented by the proposed notational conventions could be more easily mapped into such a programming language. In achieving this, it becomes easier to transform one representation into another, from system analysis to implementation, and MOOD would be able to support traceability from system analysis to design and from design to implementation, bridging the gaps between these phases.

Experience with the Development of the MOOD Prototype

MOOD has been applied to design its own prototype, but a full implementation of the MOOD environment and the support to manipulate libraries of reusable components are still necessary in order to get MOOD fully automated. In concluding the implementation of the MOOD prototype some experience can be reported. First, the applicability of any methodology within an environment is limited by the capacity of the environment to automate that methodology, which means that, although automated support is imperative, it is difficult to implement some aspects of MOOD such as small details of its graphical notation.

Although InterViews is another layer of abstraction between the prototype and the user interface, it provides an elegant abstraction of graphical objects, and the gain in productivity and quality provided by reusing the InterViews library has paid off. Experience with the prototype design showed the importance of the inheritance and composition mechanisms during the design phase, and how these concepts can be easily mapped into C++ and InterViews during the implementation phase. In spite of the interface aspects of the MOOD environment being more complicated than those of the database, both (the interface and database implementations) took the same effort to build because InterViews facilitated the job tremendously, and provided further evidence of the importance of software reuse.

In order to improve the implemented prototype, a distributed architecture for the MOOD environment would be appropriate because software development tends to involve groups of designers cooperating on given tasks which would require an environment running on a number of workstations interconnected by a network. It is therefore envisaged that the MOOD environment has to evolve from a single machine to multiple machines, thus multi-user and multi-access features are desirable features for the MOOD database. Nevertheless, the issues of distributed object-oriented software systems and sharing objects with multiple users are complicated areas of work and much still has to be done to solve the problems of distributed inheritance, consistency of objects states, migrations of objects and persistence.

As far as persistence is concerned, there are several researchers working on this issue and important results have been accomplished. In particular, McCue and Shrivastava (1990) present a structure for persistent object-oriented software systems in a distributed environment. Nevertheless, the general problem of what happens to persistent objects when their definitions are changed is very much a topic of research. The long lifetime of persistent objects implies that mechanisms might be needed to allow objects to evolve, so that they may use new operations and attributes which have been added to their classes by different designers. Further developments in this field should be incorporated to the MOOD environment.

8.2 DIRECTIONS FOR FUTURE RESEARCH

This section describes potential areas of future work related to the subject of object-oriented design methodologies. Actually, in some of the areas identified, significant progress is already emerging. Some key areas particularly relevant to the scope of this thesis, which is believed to be worthy of further consideration include: object-oriented analysis, metrics, formal methods and exception handling mechanisms.

Methodologies such as MOOD have led to a better understanding of object-oriented design and topics related to the identification and representation of classes, objects and inheritance during the design phase. However, object-oriented analysis methodologies covering requirements definitions and specification are still incipient and research in this direction should continue. They should be compatible with object-oriented principles so that traceability is achieved between software life cycle phases. Thus, the object-oriented paradigm could be extended to cover the whole software life cycle model. Appropriate object-oriented analysis methodologies might therefore be integrated with MOOD in the future.

Most large software systems need to be partitioned early so that designers have reasonably sized parts of the application to work with. Unfortunately, methodologies for large scale object-oriented software systems remain untested. Currently, this decomposition is done on the basis of experience and intuition. Therefore, there is still a great need to investigate this partitioning problem and it is hoped that in the future this important branch of object-oriented software development can be probed more deeply.

Although a considerable amount of information concerning metrics exists, until now, there are no widely accepted metrics for evaluating object-oriented designs in terms of their complexity, quality, size and costs. Therefore, new directions in metrics are needed which cater for the features of object-oriented software systems and reusability. There remain certain measures which this research might benefit from; metrics could be used in order to get measurements from large object-oriented designs and compared with other paradigms. For instance, there could be comparisons regarding: size and performance of software systems created in an object-oriented approach versus similar features of software systems created using other approaches; the differences between schedules and estimating when software systems are built pursuing an object-oriented software development life cycle and when other trends are followed.

Another area of future work related to metrics which might be examined is whether new forms of software metrics can be applied to the object-oriented paradigm to improve the prediction of time and costs involved in an object-oriented software development. The number of constructs in an object-oriented design (for instance, the number of operations for each class) could be used to estimate the size of a software system and might be good predictors of the total effort involved in its implementation. Laranjeira

(1990) argues that when an object-oriented approach is used it is easier to estimate at early stages of the software life cycle model the time and costs involved in the whole software development, because of the traceability between specification and implementation.

The MOOD tools may collect information on how software systems are developed, for example, the number of classes, objects and operations defined. When it comes to reviewing a particular object-oriented design, metrics could also be used to measure qualitative and quantitative aspects of that design, for instance the number of changes requested on classes, how many times a specific diagram has been edited and the number of inconsistencies reported. These measures are useful to assess the overall quality of a design.

An interesting area for further research would be that of providing a more formal treatment for object-oriented design. Formal methods are important because they could provide a solid base for the semantics of an object-oriented methodology and enforce a coherent use of a design language across the design phase. This would not only improve the verification of object-oriented software systems but it would also help to clarify the semantics of object-oriented design methodologies. Some research has already appeared in this area. For instance, the work of Harel (1988) on visual formalisms could be incorporated into MOOD in order to enrich its graphical notations.

Another reason for using formal approaches is that software development involves several tasks where there is a significant risk of human and machine faults occurring. Since human beings may make mistakes, software development is therefore susceptible to human faults. Nevertheless, software systems should be robust enough to deal as far as possible with human and machine faults. A robust software system is one which continues to behave reasonably and in a well-defined way even in the presence of a fault. If a software system is unable to deal with some exceptional conditions, at least it should report the problem and avoid harmful consequences.

Further research to understand exception handling mechanisms within an object-oriented approach is still required. As Lee and Anderson (1990) state: "Software is prone to design faults because, despite modern software construction methodologies, most software systems continue to be enormously complex". Designers should be forced to think about which exceptions might occur and what should be done about them before the implementation phase, and this may lead to the uncovering of more faults during the design phase than might otherwise have been the case. As far as exception handling is concerned, within an object-oriented framework, the following topics should be considered:

- How to deal with exceptions during object-oriented design.
- How to respond to exceptions from an operation point of view.
- How to cope with propagation of exceptions across an object boundary.
- How to achieve modularity in the presence of exceptions.
- How to represent exceptions in an object-oriented software system.

If designers think about exceptional conditions during the design phase, it becomes easy to map these exceptions into a programming language during the implementation phase, although the complexity of a design is increased. Therefore, the effect of exception handling in object-oriented software systems must be further investigated, and the inclusion of exception treatment as part of the design process would be very useful.

8.3 IN CONCLUSION

As with most topics is computer science, a multitude of additional research fields can easily be identified. The most important topics following from the author's research have been outlined above. Although the object-oriented paradigm has an evidential learning curve and the adoption of an object-oriented approach does not represent the solution for all software engineering problems, the author is confident that the object-oriented paradigm is here to stay because it is a step towards better quality software and reusability.

The idea of software reuse based on the exploitation of the inheritance and composition mechanisms during the design phase has been put forward in this thesis. Reusability is a technique for improving software productivity and quality which is finally finding general acceptance. The object-oriented paradigm is especially important for software reuse because it provides important facilities, such as inheritance and encapsulation, which make reusability feasible. This suggests that the object-oriented paradigm may contribute significantly to the solution of the so-called *software crisis.*

The object-oriented paradigm is such a powerful set of concepts that eventually it will get completely absorbed into the software development culture, in the same way that structured development methodologies and to some extent abstract data types concepts have been. This is evident in the abundance of research looking at various aspects of the paradigm. Consequently, the 1990s are likely to be a period of gradual acceptance of the object-oriented paradigm which will become the main approach to developing software systems in this decade. Despite its limitations, MOOD is a significant step forward in this direction.

The future of object-oriented software engineering might well be to accept a hybrid trend with other approaches and mapping of concepts between such approaches. However, the author believes that the object-oriented paradigm should (and hopefully will!) pervade the entire software life cycle. The object-oriented paradigm has needed an organised and disciplined view of software development, and to be extended to cover more phases of the software life cycle model. MOOD is believed to represent an important step forward in the understanding and promotion of object-oriented design methodologies.

References

Abbott R. J. (1983) "Programming Design by Informal English Description", *Communications of the ACM*, **26**(11), November 1983, pp. 882-894.

Ackroyd M. and Daum D. (1991) "Graphical Notation for Object-Oriented Design and Programming", *Journal of Object-Oriented Programming*, **3**(5), January 1991, pp. 18-28.

Adelson B. and Soloway E. (1985) "The Role of Domain Experience in Software Design", *IEEE Transactions on Software Engineering*, SE-11(11), November 1985, pp. 1351-1360.

Agha G. (1986) "An Overview of Actor Languages", *ACM SIGPLAN Notices* **21**(10), October 1986, pp. 58-67.

Alabiso B. (1988) "Transformation of Data Flow Analysis Model to Object-Oriented Design", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '88*, San Diego, California, September 1988, *ACM SIGPLAN Notices*, **23**(11), November 1988, pp. 335-353.

Alford M. W. (1977) "A Requirements Engineering Methodology for Real-Time Processing Requirements", *IEEE Transactions on Software Engineering*, SE-**3**(1), January 1977, pp. 60-69.

America P. (1987) "POOL-T: A Parallel Object-Oriented Language", Yonezawa A. and Tokoro M. (ed.) *Object-Oriented Concurrent Programming*, The MIT Press, Cambridge, Massachusetts, pp. 199-220.

Arnold P., Bodoff S., Coleman D., Gilchrist H. and Hayes F. (1991) *An Evaluation of Five Object-Oriented Development Methods*, Hewlett-Packard Laboratories, Bristol, United Kingdom, May 1991.

Bailin S. C. (1989) "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, **32**(5), May 1989, pp. 608-623.

Beck K. and Cunningham W. (1989) "A Laboratory for Teaching Object-Oriented Thinking", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'89*, New Orleans, Lousiana, *ACM SIGPLAN Notices*, 24(10), October 1989, pp. 1-6.

Beichter F. W., Herzog O. and Petzsh H. (1984) "SLAN-4 - A Software Specification and Design Language", *IEEE Transactions on Software Engineering*, SE-10(3), March 1984, pp. 155-162.

Berard E. (1986) *An Object-Oriented Design Handbook*, EVB Software Engineering Inc., Rockville, Maryland.

Bersoff E. H. (1984) "Elements of Software Configuration Management", *IEEE Transactions on Software Engineering*, SE-10(1), January 1984, pp. 79-87.

Bifferstaff T. and Richter C. (1987) "Reusability Framework, Assessment and Directions", *IEEE Software*, 4(3), March 1987, pp. 41-49.

Blair G., Gallagher J., Hutchison D. and Shepherd D. (1991) *Object-Oriented Languages, System and Applications*, Pitman Publishing, London.

Boehm B. W. (1988) "A Spiral Model of Software Development and Enhancement", *Computer*, 21(5), May 1988, pp. 61-72.

Booch G. (1983a) *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, California.

Booch G. (1983b) "Object-Oriented Design", Freeman P. and Wasserman A. I. (eds.) *Tutorial on Software Design Techniques*, Fourth Edition, IEEE, Silver Spring, Maryland, pp. 420-436.

Booch G. (1986) "Object-Oriented Development", *IEEE Transactions on Software Engineering*, SE-12(2), February 1986, pp. 211-221.

Booch G. (1987) *Software Components with Ada*, Benjamin/Cummings, Menlo Park, California.

Booch G. (1991) *Object-Oriented Design with Applications*, Benjamin/ Cummings, Redwood City, California.

Borgida A. (1985) "Features of Languages for the Development of Information System at the Conceptual Level", *IEEE Software*, 2(1), January 1985, pp. 63-72.

Bulman D. M. (1989) "An Object-Based Development Model", *Computer Language*, 6(8), August 1989, pp. 49-59.

Buzzard G. D. and Mudge T. N. (1985) "Object-Based Computing and the Ada Language", *Computer* 18(3), March 1985, pp. 11-19.

Cardelli L. and Wegner P. (1985) "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys*, 17(4), December 1985, pp. 471-522.

Cardelli L., *et al.* (1989) *Modula-3 Report*, Systems Research Center of Digital Equipment Corporation, Palo Alto, California.

Chen P. P. (1976) "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, 1(1), March 1976, pp. 9-36.

Coad P. and Yourdon E. (1990) *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey.

Cox B. J. (1986) *Object-Oriented Programming - An Evolutionary Approach*, Addison-Wesley, Readings, Massachusetts.

Cunningham W. and Beck K. (1986) "A Diagram for Object-Oriented Programs", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, 21(11), November 1986, pp. 361-367.

Dahl O.-J., Myhrhaug B. and Nygaard K. (1970) *SIMULA67 Common Base Language*, Publication No. S-22, Norwegian Computing Centre, Oslo October 1970.

DeMarco T. (1979) *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, New Jersey.

DeMichiel L. G. and Gabriel R. P. (1987) "The Common Lisp Object System: An Overview", Bézivin J., Hullot J.-M., Cointe P. and Liberman H. (eds.) *Proceedings of the European Conference on Object-Oriented Programming - ECOOP'87*, Paris, June 1987, *Lecture Notes in Computer Science*, No. 276, Springer-Verlag, Berlin, pp. 151-170.

DeRemer F. and Kron H. H. (1976) "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, SE-2(2), June 1976, pp. 80-86.

Embley E. W. and Woodfield S. N. (1987) "A Knowledge Structure for Reusing Abstract Data Types", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, March 1987, IEEE Computer Society Press, Washington, D.C., pp. 360-368.

Freeman P. (1984) "Reusable Software Engineering: Concepts and Research Directions", Freeman P. and Wasserman A. I. (eds.) *Tutorial on Software Design Techniques,* Fourth Edition, IEEE, Silver Spring, Maryland, pp. 63-76.

Gane C. and Sarson T. (1979) *Structured System Analysis: Tools and Techniques*, Prentice-Hall, Englewood Cliffs, New Jersey.

Ghezzi C. and Jazayeri M. (1982) *Programming Language Concepts*, John Wiley & Sons, New York, New York.

Goldberg A. and Robson D. (1983) *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts.

Goldberg A. (1984) *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts.

Gossain S. and Anderson B. (1990) "An Iterative-Design Model for Reusable Object-Oriented Software", Meyrowitz N. (ed.) *Proceeding of the Conference on Object-Oriented Programming: Systems, Languages and Applications and the European Conference on Object-Oriented Programming - OOPSLA/ECOOP'90*, Ottawa, *ACM SIGPLAN Notices*, **25**(10), October 1990, pp. 12-27.

Harel D. (1988) "On Visual Formalisms", *Communications of the ACM*, **31**(5), May 1988, pp. 515-530.

Heitz M. (1989) *HOOD Reference Manual*, Issue 3.0, European Space Agency, Noordwijk, The Netherlands, September 1989.

Henderson-Sellers B. and Edwards J. M. (1990) "The Object-Oriented Systems Life Cycle", *Communications of the ACM*, **33**(9), September 1990, pp. 142-159.

Henderson-Sellers B. and Constantine L. L. (1991) "Object-Oriented Development and Functional Decomposition", *Journal of Object-Oriented Programming*, **3**(5), January 1991, pp. 11-17.

Hoare C. A. R. (1974) "Monitors: an Operating Systems Structuring Concept", *Communications of the ACM*, **17**(10), October 1974, pp. 549-577.

Hopkins T. P., Williams I. W. and Wolczko M. I. "MUSHROOM - a Distributed Multi-user Object-oriented Programming Environment", presented at the British Computer Society joint Object-Oriented Programming and Systems and Parallel Programming Specialist Groups Workshop on Parallel Processing, October 1987.

Hopkins T. P. and Wolczko M. I. (1989) "Writing Concurrent Object-Oriented Programs Using Smalltalk-80", *The Computer Journal*, **32**(4), August 1989, pp. 341-350.

Horowitz E. and Munson J. B. (1984) "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering*, SE-10(5), May 1984, pp. 477-487.

Hull M. E. C., Zarca-Aliabadi A. and Guthrie D. A. (1989) "Object-Oriented Design, Jackson System Development (JSD) Specification and Concurrency", *Software Engineering Journal*, 4(2), March 1989, pp. 79-86.

Jackson M. A. (1975) *Principles of Program Design*, Academic Press, New York, New York.

Jackson M. A. (1983) *System Development*, Prentice-Hall, London.

Jacky J. P. and Kalet I. (1987) "An Object-Oriented Programming Discipline for Standard Pascal", *Communications of the ACM*, 30(9), September 1987, pp. 772-776.

Jacobson I. (1986) "Language Support for Changeable Large Real Time System", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, 21(11), November 1986, pp. 377-384.

Jacobson I. (1987) "Object Oriented Development in an Industrial Environment", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '87*, Orlando, Florida, October 1987, *ACM SIGPLAN Notices*, 22(12), December 1987, pp. 183-191.

Jalote P. (1989) "Functional Refinement and Nested Objects for Object-Oriented Design", *IEEE Transactions on Software Engineering*, SE-15(3), March 1989, pp. 264-270.

Johnson R. E. and Foote B. (1988) "Designing Reusable Classes", *Journal of Object-Oriented Programming*, 1(2), June/July 1988, pp. 22-35.

Jones C. B. (1986) *Systematic Software Development Using VDM*, Prentice Hall, Englewood Cliffs, New Jersey.

Kerth N. (1988) "A Methodology for Structured Object-Oriented Design", Tutorial presented at the *Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '88*, San Diego, California, September 1988.

Kristensen B. B., Madsen O. L., Moller-Pedersen B. and Nygaard K. (1985) Multi-Sequential Execution in the Beta Programming Language", *ACM SIGPLAN Notices*, **20**(4), April 1985, pp. 57-70.

Lanergan R. G. and Grasso C. A. (1984) "Software Engineering with Reusable Design and Code", *IEEE Transactions on Software Engineering*, SE-**10**(5), May 1984, pp. 498-501.

Laranjeira L. A. (1990) "Software Size Estimation of Object-Oriented Systems", *IEEE Transactions on Software Engineering*, SE-**16**(5), May 1990, pp. 510-522.

Lauber R. J. (1982) "Development Support Systems", *Computer*, **15**(5), May 1982, pp. 36-46.

Ledgard H. F. and Taylor R. W. (1977) "Two Views of Data Abstraction", *Communications of the ACM*, **20**(6), June 1977, pp. 382-384.

Lee P. A. and Anderson T. (1990) *Fault Tolerance - Principles and Practice*, Second Edition, Springer-Verlag, Wien.

Linton M. A., Calder P. R. and Vlissides J. M. (1988) *InterViews: A C++ Graphical Interface Toolkit,* Technical Report CSL-TR-88-358, Computer Systems Laboratory, Department of Electrical Engineering and Computer Systems, Stanford University, July 1988.

Linton M. A. (1989) *InterViews Reference Manual, Version 2.6*, Computer Systems Laboratory, Department of Electrical Engineering and Computer Systems, Stanford University, November 1989.

Liskov B. and Zilles S. N. (1975) "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering*, SE-1(1), March 1975, pp. 7-19.

Liskov B., Snyder A., Atkinson R. and Schaffert, C. (1977) "Abstraction Mechanisms in CLU", *Communications of the ACM*, **20**(8), August 1977, pp. 564-576.

Lorensen W. (1986) *Object-Oriented Design, CRD Software Engineering Guidelines*, General Electric Co., Corporate Research and Development Center.

Loy P. H. (1990) "A Comparison of Object-Oriented and Structured Development Methods", *Software Engineering Notes*, 15(1), January 1990, pp. 44-48.

Madsen O. L. and Moller-Pedersen B. (1988) "What Object-Oriented Programming May Be and What It Does Not Have to Be", Gjessing S. and Nygaard K. (eds.) *Proceedings of the European Conference on Object-Oriented Programming - ECOOP'88*, Oslo, August 1988, *Lecture Notes in Computer Science,* No. 322, Springer-Verlag, Berlin, pp. 1-20.

Masiero P. and Germano F. S. R. (1988) "JSD as an Object-Oriented Design Method", *Software Engineering Notes*, 13(3), July 1988, pp. 22-23.

McCue D. L. and Shrivastava S. K. (1990) "Structuring Fault-Tolerant Object Systems for Portability", Fourth ACM SIGOPT Workshop, Bologna, September 1990.

Meyer B. (1988) *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, New Jersey.

Micallef J. (1988) "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, 1(1), April 1988, pp. 12-36.

Minsky M. (1975) "A Framework for Representing Knowledge", Wiston P. (ed.) *The Psychology of Computer Vision*, McGraw-Hill, New York.

Moon D. A. (1986) "Object-Oriented Programming with Flavors", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, 21(11), November 1986, pp. 1-8.

Neighbours J. M. (1984) "The DRACO Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, SE-10(5), May 1984, pp. 564-574.

Nygaard K. (1986) "Basic Concepts in Object Oriented Programming", *ACM SIGPLAN Notices*, 21(10), October 1986, pp. 128-132.

Oosthuizen G. D., Bekker C. and Avenant C. (1990) "Managing Classes in Very Large Classes Repository", *Proceedings of the Second International Conference TOOLS*, Angkor, Paris, pp. 625-633.

Parnas D. L. (1972) "On the Criteria to Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12), December 1972, pp. 1053-1058.

Pascoe G. A. (1986) "Elements of Object-Oriented Programming", *Byte* 11(8), August 1986, pp. 139-144.

Power L. (1988) "Workshop on the Specification and Design of Objects", Power L. and Weiss Z. (eds.) *Addendum to the Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '87*, Orlando, Florida, October 1987, *ACM SIGPLAN Notices*, 23(5), May 1988, pp. 7-16.

Prieto-Diaz R. (1988) "Domain Analysis for Reusability", Tracz W. (ed.) *IEEE Tutorial: Software Reuse: Emerging Technology*, IEEE Computer Society Press, Washington, D.C., pp. 347-353.

Pun W. W. Y and Winder R. L. (1989) "A Design Method for Object-Oriented Programming", Cook, S. (ed.) *Proceedings of the European Conference on Object-Oriented Programming - ECOOP'89*, Nottingham, United Kingdom, July 1989, Cambridge University Press, Cambridge, pp. 225-240.

Raj R. K. and Levy H. M. (1989) "A Compositional Model for Software Reuse", Cook S. (ed.) *Proceedings of the European Conference on Object-Oriented Programming - ECOOP'89*, Nottingham, United Kingdom, July 1989, Cambridge University Press, Cambridge, pp. 3-24.

Rentsch T. (1982) "Object Oriented Programming", *ACM SIGPLAN Notices*, 17(9), September 1982, pp. 51-57.

Robson D. (1981) "Object-Oriented Software Systems", *Byte*, 6(8), August 1981, pp. 74-86.

Ross T. R. and Schoman K. E. (1977) "Structured Analysis for Requirements Definitions", *IEEE Transactions on Software Engineering*, SE-3(1), January 1977, pp. 6-15.

Royce W. W. (1987) "Managing the Development of Large Software Systems", *Proceedings of the Ninth International Conference on Software Enginnering*, Monterey, California, March 1987, IEEE Computer Society Press, Washington, D. C., pp. 328-338.

Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. (1991) *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey.

Schaffert C., *et al.* (1986) "An Introduction to Trellis/Owl", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, 21(11), November 1986, pp. 9-16.

Scheiffler R. W. and Gettys, J. (1986) "The X Window System", *ACM Transactions on Graphics*, 5(2), April 1986, pp. 79-109.

Seidewitz E. (1989) "General Object-Oriented Software Development: Background and Experience", *The Journal of Systems and Software*, 9(2), February 1989, pp. 95-108.

Shaw M. (1984) "Abstraction Techniques in Modern Programming Languages", *IEEE Software*, 1(4), October 1984, pp. 10-26.

Shlaer S. and Mellor S. J. (1988) *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, Englewood Cliffs, New Jersey.

Sincovec R. F. and Wierner R. S. (1987) "Modular Software Construction and Object-Oriented Design Using Ada", Peterson G. E. (ed.) *Tutorial: Object-Oriented Computing*, IEEE Computer Society Press, Washington, D.C., pp. 30-36.

Sixtensson A. and Wenchuan Y. (1990) "Object-Oriented Technology and Reuse in Telecommunication Application - Practical Experience", *Proceedings of the Second International Conference TOOLS*, Angkor, Paris, pp. 433-441.

Sommerville, I. (1989) *Software Engineering*, Addison-Wesley, Wokingham, England.

Stay J. F. (1976) "HIPO and Integrated Program Design", *IBM System Journal*, 15(2), April 1976, pp. 143-154.

Stefik M. and Bobrow D. G. (1986) "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, 6(4), April 1986, pp. 40-62.

Stroustrup B. (1986) *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts.

Stroustrup B (1987) "What is Object-Oriented Programming?", Bézivin J., Hullot J.-M., Cointe P. and Liberman H. (eds.) *Proceeding of the European Conference on Object-Oriented Programming - ECOOP'87*, Paris, June 1987, *Lecture Notes in Computer Science,* No. 276, Springer-Verlag, Berlin, pp. 51-70.

Tarumi H., Agusa K. and Ohno Y. (1988) "A Programming Environment Supporting Reuse of Object-Oriented Software", *Proceedings of the Tenth International Conference on Software Engineering*, Singapore, April 1988, IEEE Computer Society Press, Washington, D.C., pp. 265-273.

Teichroew D. and Hersey E. A. (1977) "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, SE-3(1), January 1977, pp. 41-48.

Tesler L. (1985) *Object Pascal Report*, Apple Computer, Santa Clara, California.

Thomas D. (1989) "What's in an Object", *Byte*, 14(3), March 1989, pp. 231-240.

Tichy W. F. (1985) "RCS - A System for Version Control", *Software - Practice and Experience*, 15(7), July 1985, pp. 637-654.

Tracz W. (1988) "Software Reuse Myths", *Software Engineering Notes*, 13(1), January 1988, pp. 18-22.

Tsichritzis D. (1989) "Object-Oriented Development for Open Systems", Ritter G. X. (ed.) *Proceedings of the IFIP 11th World Computer Congress*, San Francisco, August 1989, North Holland, Amsterdam, pp. 1033-1040.

Vlissides J. (1989) *A Tutorial for InterViews Programmers - Part III: An Application Using Structured Graphics*, Computer Systems Laboratory, Department of Electrical Engineering and Computer Systems, Stanford University.

Ward P. and Mellor S. (1985) *Structured Development for Real-Time Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

Ward P. (1989) "How to Integrate Object Orientation with Structured Analysis and Design", *IEEE Software*, **6**(2), March 1989, pp. 74-82.

Wasserman A. I., Pircher P. A. and Muller R. J. (1990) "The Object-Oriented Structured Design Notation for Software Design Representation", *Computer*, **23**(3), March 1990, pp. 50-63.

Wegner P. (1987) "Dimensions of Object-Based Language Design", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '87*, Orlando, Florida, October 1987, *ACM SIGPLAN Notices*, **22**(12), December 1987, pp. 168-182.

Wirfs-Brock R., Wilkerson B. and Wiener L. (1990) *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, New Jersey.

Wirth N. (1971) "Program Development by Stepwise Refinement", *Communications of the ACM*, 14(4), April 1971, pp. 221-227.

Wirth N. (1976) "Modula: A Language for Modular Multiprogramming", *Software - Practice and Experience*, 7(1), January/February 1977, pp. 3-35.

Wolczko M. (1988) *Semantics of Object-Oriented Languages*, Ph.D. Thesis, Manchester University, United Kingdom.

Yokote A. and Tokoro M. (1987) "Concurrent Programming in ConcurrentSmalltalk", Yonezawa A. and Tokoro M. (eds.) *Object-Oriented Concurrent Programming*, The MIT Press, Cambridge, Massachusetts, pp. 129-158.

Yonezawa A., Shibayama E., Takada T. and Honda Y. (1987) "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1", Yonezawa A. and Tokoro M. (eds.) *Object-Oriented Concurrent Programming*, The MIT Press, Cambridge, Massachusetts, pp. 55-90.

Yourdon E. and Constantine L. L. (1979) *Structured Design*, Prentice-Hall, Englewood Cliffs, New Jersey.

Yutaka I. and Tokoro M. (1986) "A Concurrent Object Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation", Meyrowitz N. (ed.) *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA '86*, Portland, Oregon, September 1986, *ACM SIGPLAN Notices*, **21**(11), November 1988, pp. 232-241.