

**NEW INHERITANCE MODELS THAT FACILITATE
SOURCE CODE REUSE IN OBJECT-
ORIENTED PROGRAMMING**

By

HISHAM M. AL-HADDAD

**Bachelor of Science
Yarmouk University
Irbid, Jordan
1986**

**Master of Science
Northrop University
Los Angeles, California
1988**

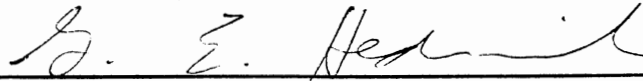
**Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
DOCTOR OF PHILOSOPHY
July, 1992**


NEW INHERITANCE MODELS THAT FACILITATE
SOURCE CODE REUSE IN OBJECT-
ORIENTED PROGRAMMING

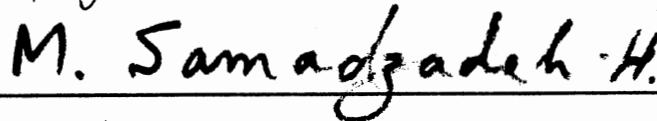
Thesis Approved:

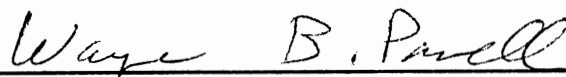


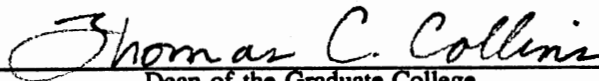
Thesis Advisor











Dean of the Graduate College

PREFACE

Code reusability is a primary objective in the development of software systems. The *object-oriented programming methodology* is one of the areas that facilitate the development of software systems by allowing and promoting code reuse and modular designs. Object-oriented programming languages (OOPLs) provide different facilities to attain efficient reuse and reliable extension of existing software components. *Inheritance* is an important language feature that is conducive to reusability and extensibility. Various OOPLs provide different inheritance models based on different interpretations of the inheritance notion. Therefore, OOPLs have different characteristics derived from their respective inheritance models.

This dissertation is concerned with solutions for three major problems that limit the utilization of inheritance for code reusability. The range of object-oriented applications and thus the usage of object-oriented programming in general is also discussed. The three major problems are: 1) the relationship between inheritance and other related issues such as encapsulation, access techniques, visibility of inheritance, and subtyping; 2) the hierarchical structure imposed by inheritance among classes; and 3) the accessibility of previous versions of the modified methods defined in classes located at higher levels of the inheritance structure than the parent classes.

The proposed solutions for these problems are presented as new inheritance models that facilitate code reuse and relax the restrictions imposed on inheritance models by languages. A survey and taxonomy of the conventional inheritance models, and a comparison and analysis of some of the common OOPLs are also presented in the dissertation.

ACKNOWLEDGMENT

First, I would like to express my sincere appreciation to my dissertation advisor, Dr. K.M. George, for his continuous and consistent guidance, dedication, kindness, and valuable instruction throughout my graduate work. His encouragement and motivation has been the driving force in the completion of this work.

Special thanks are due to my advisory committee member, Dr. Mansur Samadzadeh, for his consistent support, valuable suggestions, and critical review of my research work. I would like also to thank other members of my advisory committee, Dr. G.E. Hedrick, J.P. Chandler, and W.B. Powell, for serving on my graduate committee and for their advice and support. I would like to extend my thanks to all of my friends, colleagues, and staff members, specially Anna Ventris and Janice Bryan, for their help and assistance.

I would like to take this opportunity to thank my family. My deepest love and thanks go to my parents for their prays and spirit of encouragement throughout my academic studies. I also take this opportunity to extend my appreciation and thanks to my brothers and sisters for their moral support. Special appreciation and thanks to my brothers, Ali and Khalid, for their moral and financial support during my undergraduate and graduate studies. Finally, my love and thanks to my wife, MingChing, for her patience and moral support that gave me the motivation and inspiration to complete my graduate studies. Thank you all.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
1.1 Prologue	1
1.2 Classes	2
1.3 Objects	3
1.4 Inheritance	4
1.5 Overview of the Dissertation	9
II. LITERATURE REVIEW	10
2.1 Introduction	10
2.2 Classes	11
2.2.1 Instance Variables and Methods	16
2.3 Instantiation	19
2.4 Inheritance	21
2.4.1 Subtyping	31
2.4.2 Problems with Current Inheritance Models	33
2.5 Message Passing	36
2.6 Encapsulation	38
2.7 Polymorphism and Binding	40
2.8 Definitions	42
III. INHERITANCE IN OBJECT-ORIENTED PROGRAMMING LANGUAGES: A TAXONOMY AND SURVEY	45
3.1 Introduction	45
3.2 A Survey of Inheritance Mechanisms and Access Techniques ...	46
3.2.1 Trellis/Owl	47
3.2.2 C++	50
3.2.3 Eiffel	54
3.2.4 CommonObjects	57
3.2.5 CLOS	60
3.2.6 Flavors	63
3.2.7 Smalltalk-80	66
3.2.8 Simula	70

Chapter	Page
3.3 Subtyping and Inheritance	74
3.4 A Binary-Tree Taxonomy Model	75
3.5 Discussion	81
3.6 Summary	83
IV. APPROACHES TO REUSABILITY IN C++ AND EIFFEL	88
4.1 Introduction	88
4.2 Design Objectives and Highlights of the Two Languages	90
4.2.1 Eiffel	90
4.2.2 C++	94
4.3 Inheritance and Reusability	95
4.3.1 Eiffel	96
4.3.2 C++	99
4.4 Discussion	105
4.5 Summary	108
V. AN OBJECT-BASED INHERITANCE MODEL	110
5.1 Introduction	110
5.2 The Proposed Object-Based Inheritance Model (TIM)	111
5.2.1 Semantics and Composition of an Object	113
5.2.2 Internal Structure of an Object	115
5.2.3 Object Interfaces	117
5.2.4 Message Passing Techniques	117
5.2.5 Multiple Inheritance	118
5.2.6 Object Creation and Deletion	119
5.2.7 Inheritance Hierarchy	122
5.3 Examples Represented in the Proposed Model	124
5.4 Discussion	127
5.5 Summary	129
VI. A FEEDBACK INHERITANCE MODEL	130
6.1 Introduction	130
6.2 Definitions	131
6.3 Two Examples and Their Hierarchical Representations	133
6.4 The Proposed Feedback Inheritance Model	138
6.4.1 Definitions	138
6.4.2 Feedback Inheritance	140
6.4.3 Semantics of Feedback Inheritance	143
6.4.4 Message Passing	145
6.5 Examples Represented in the Proposed Model	148

Chapter	Page
6.6 Discussion	152
6.7 Summary	154
VII. AN IMPLEMENTATION INHERITANCE MODEL	156
7.1 Introduction	156
7.2 Background	157
7.2.1 Eiffel	157
7.2.2 C++	158
7.2.2.1 Overloading of Functions	158
7.2.2.2 Virtual Functions	160
7.2.2.3 Abstract Classes	162
7.2.3 CLOS	162
7.2.4 Smalltalk-80	164
7.3 Classes and Slots	166
7.4 Behavior Slots	168
7.4.1 Syntax of Behavior Slots	170
7.5 Aggregates and Behavior Slots	171
7.6 General Messages	174
7.6.1 Procedure Calls	175
7.7 The Proposed Implementation Inheritance Model	176
7.8 Conceptual View of Implementation Inheritance	177
7.9 An Implementation Scheme for Implementation Inheritance	181
7.9.1 Organization of the I-Index	183
7.9.2 Size of the I-Index	184
7.9.3 Construction of the I-Index	186
7.9.3.1 Single Inheritance	186
7.9.3.2 Multiple Inheritance	188
7.9.4 Optimization of the I-Index	189
7.9.5 Selection Mechanism	191
7.10 Discussion	194
7.10.1 Implementation Inheritance and Encapsulation	194
7.10.2 Multi-Methods and Generalized Messages	196
7.11 Summary	197
VIII. SUMMARY, CONCLUSIONS, AND FUTURE WORK	199
BIBLIOGRAPHY	203

LIST OF TABLES

Table		Page
3.1	Languages considered in the classification tree	77
3.2	Node expansions of the classification tree	78
3.3	Features of the selected languages (1)	85
3.4	Features of the selected languages (2)	86
3.5	Features of the selected languages (3)	87
4.1	Inheritance and related issues in C++ and Eiffel	91
4.2	Visibility and interfaces in C++ and Eiffel	92
6.1	Instances of the class RDF1	135

LIST OF FIGURES

Figure		Page
1.1	An example of classes and subclasses	2
1.2	Definition of the class CAR	3
1.3	A conceptual view of an object	3
1.4	Illustration of the object Ford_Car of the class CAR	4
1.5	An example of single inheritance hierarchy	5
1.6	An example of multiple inheritance hierarchy	5
1.7	The inheritance hierarchy of the class CAR	6
2.1	Representation of the data type queue	13
2.2	Pseudo code of the class QUEUE	14
2.3	The super/subclass relationship between QUEUE and DEQUE	15
2.4	A classification of graphical classes	16
2.5	Initialization of the instance variables of the class QUEUE	17
2.6	Representation of the interface of the class QUEUE	18
2.7	Examples of creating object Q1 in different OOPLs	20
2.8	Representation of single and multiple inheritance relationships	22
2.9	Pseudo code of the class POLYGON	24
2.10	Pseudo code of the class RECTANGLE	25
2.11	Class CAR is an aggregation of its superclasses	26
2.12	Representation of the data type deque	26
2.13	The class STACK is a generalization of the class DEQUE	27

Figure	Page
2.14 CommonObjects' definition of the class DEQUE	27
2.15 CommonObjects' definition of the subclass STACK	28
2.16 Abstraction and implementations of stack and deque	36
3.1 Definition of the type STACK	48
3.2 DEQUE is a subtype of the type STACK	49
3.3 Definition of the class DATE	51
3.4 Class BIRTH_DAY is a public subclass of the class DATE	52
3.5 Definition of the class STACK	55
3.6 Class DEQUE inherits from the class STACK	55
3.7 Definition of the class DATE	58
3.8 Class BIRTH_DAY partially inherits from the class DATE	58
3.9 Definition of the class EMPLOYEE	61
3.10 Class MANAGER is a subclass of the class EMPLOYEE	61
3.11 Definitions of classes	62
3.12 Definition of the flavor 3D_MOVING_OBJECT	64
3.13 The flavors SPACE_SHIP and COMET inherit from the flavor 3D_MOVING_OBJECT	65
3.14 Definition of class PERSONAL_FINANCES	67
3.15 Class DEDUCTIBLES inherits from the class PERSONAL_FINANCES	67
3.16 Definition of the class PLACE	70
3.17 Class TOWN is a subclass of the class PLACE	70
3.18 Class PLACE using the virtual procedure write	73
3.19 The classification Tree	78
3.20 The subtree n0	79
3.21 The subtree n1	80
3.22 The subtree n2	80

Figure	Page
3.23 The subtree n3	80
3.24 The subtree n4	80
3.25 The subtree n5	82
3.26 The subtree n6	80
4.1 Hierarchy of software development information	89
4.2 Definition of the class EFFECTIVE_LIST[T]	97
4.3 The class STACK[T] is a subclass of the class EFFECTIVE_LIST[T]	97
4.4 Definition of the class FIXED_LIST[T]	98
4.5 Definition of the class LIST	101
4.6 Class STACK is derived from the class LIST	101
4.7 The main program	102
4.8 Definition of the class ARRAY	103
4.9 Class ARRAY_LIST inherits from both ARRAY and LIST classes ..	104
4.10 The definition of the abstract class LIST	104
5.1 Object types and possible interfaces in TIM	113
5.2 Lists that an object can contain	116
5.3 A methods list entry	118
5.4 An example of inheritance hierarchy in TIM	122
5.5 Methods and inheritance lists of the objects in Figure 5.4	123
5.6 Definitions of the classes DATE and BIRTH_DATE	125
6.1 Record structure of a relationship	133
6.2 A hierarchical inheritance representation of the record structure in Figure 6.1	134
6.3 An NCA with two machines M-1 and M-2	135
6.4 Class representations of the NCA in Figure 6.3	136
6.5 Two possible object-oriented representations of the NCA in Figure 6.3	137

Figure	Page
6.6 Examples of clans of a set of related classes	140
6.7 Representation of single feedback inheritance among classes	141
6.8 Representation of multiple feedback inheritance among classes	142
6.9 Representation of feedback inheritance among classes C_1 , C_2 , and C_3	143
6.10 A feedback and hierarchical inheritance interfaces among classes C_1 , C_2 , and C_3	144
6.11 The structure of a descriptor	146
6.12 Descriptors of the classes A and B	148
6.13 Bi-directional inheritance between the classes Department and Faculty	149
6.14 Relating classes through hierarchical and bi-directional inheritance . .	150
6.15 Feedback inheritance representation of a three-node NCA	152
7.1 Definition of the class STACK[T]	158
7.2 Example of an overloaded function	159
7.3 Inheritance of non-virtual functions	160
7.4 Inheritance of virtual functions	161
7.5 Definitions of selected functions	163
7.6 Different implementations of the method area	163
7.7 Usage of the method area	164
7.8 Representation of a behavior slot	169
7.9 A hierarchy of Bell/AT&T modem types	170
7.10 Declaration of the class STACK using behavior slots	171
7.11 Representation of an aggregate (conceptual slot)	172
7.12 Cartesian and polar representations of a point	173
7.13 Aggregate representation of the slot setpoint	173
7.14 A generalized message	175
7.15 Inheritance hierarchy among some classes	178

Figure	Page
7.16 Aggregate representation of the slots f(), g(), and h() in Figure 7.15 .	179
7.17 The relationships among handlers of aggregates in Figure 7.15	180
7.18 Inter-relationship between aggregates and I-inheritance	180
7.19 Mapping the concept and implementation of I-inheritance	181
7.20 Class USE_ANY using implementations of the slot setpoint	182
7.21 An I-index example	183
7.22 The C-list of a multiple inheritance hierarchy	184
7.23 Representation of I-indices of classes	185
7.24 Illustration of an I-index	187
7.25 Multiple inheritance among classes	188
7.26 I-indices of the classes in Figure 7.25	189
7.27 Application of algorithm select	193

CHAPTER I

INTRODUCTION

1.1 Prologue

Programming languages may be classified into several groups based on the programming methodologies (paradigms) they support. Some of these paradigms are procedural, functional, logic, and object-oriented programming (OOP). These methodologies present different approaches to program design and implementation. The first approach is based on the idea that programmers instruct the computer *how* to process each piece of data. As a result, programs in the procedural approach are rigidly bound to the types of data they process. Introducing new types of data requires changing the structure and logic of the programs. Therefore, the procedural approach limits the programmers' ability to reuse code since code is intimately tied to the data upon which it will operate. Some functional programming languages also have this weakness to certain extent. Functional and logic programming follow the declarative approach (i.e., emphasis is on *what* is to be done rather *how* it is done).

OOP provides a new approach of thinking about data, procedures, and the relationship among them. It combines the imperative and message passing paradigms. OOP has been promoted as a methodology that will expedite the development of software systems by allowing and promoting code reuse and modular design, and will support the extension of existing software. OOP is based on three basic concepts: class, objects, and inheritance. These concepts are illustrated in the following sections using examples.

1.2 Classes

A *Class* serves as a template from which objects can be created. It is a description of the state and behavior of a set of objects. Objects fall into classes and *subclasses* based on their similarities. For example, all numbers fall into a class named NUMBERS, and all integers fall into a class named INTEGERS, which is a subclass of the class NUMBERS. Furthermore, one may make positive and negative integers fall into two different classes named P_INTEGERS and N_INTEGERS as subclasses of the class INTEGERS. This classification is illustrated in Figure 1.1.

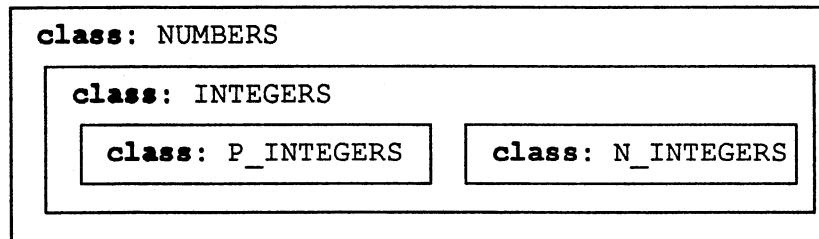


Figure 1.1: An example of classes and subclasses

A class consists of two sets: A set of instance variables, and a set of operations. Instance variables represent the state of objects belonging to that class. Operations are represented by *methods* that determine the interface and behavior of objects of the class. For example, consider the class CAR described in Figure 1.2. Class CAR describes the characteristics and behavior of cars. Each car has attributes such as maker, serial number, color, make year, transmission (either manual or automatic), and others. Additionally, all cars perform a set of operations including start, drive, turn left or right, stop, and other operations.

<p>Class: CAR</p> <p>Instance variables: Make, Serial_Number, Color, Year, Transmission</p> <p>Operations: Start, Drive, Turn_left, Turn_right, Stop</p>

Figure 1.2: Definition of the class CAR

1.3 Objects

An *object* is an instance of a class. An object is a concrete realization of a class abstraction. The instance variables of an object represent values that constitute its state. The state of an object is accessed by operations of the defining class. These operations determine the messages (calls) to which the object may respond to. The state of an object is hidden from the outside world and is accessed only through the interface of that object provided by the corresponding class. Each object belongs to one class. Figure 1.3 illustrates a conceptual view of an object. Figure 1.4 shows the state of an object named Ford_Car of the class CAR illustrated in Figure 1.2.

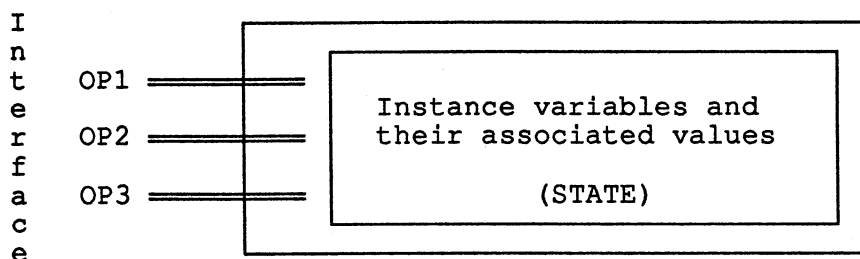


Figure 1.3: A conceptual view of an object

Each instance of the class CAR has different values for its instance variables. Some instances may have similar values but they differ at least in the Serial_Number

Instance Variables	Ford_Car
Make	Ford
Serial_number	NS223-1191
Color	blue
Year	1991
Transmission	manual

Figure 1.4: Illustration of the object Ford_Car of the class CAR

value. All car objects can perform the set of operations specified in the class CAR. Objects perform operations in response to messages. A *message* is a request sent to an object to perform certain behavior. Objects respond to messages according to the operations that have been defined in their classes. For example, the object Ford_Car illustrated in Figure 1.4 can respond to the messages start, drive, turn_left, turn_right, and stop. The responses to these messages represent the behavior of that object.

1.4 Inheritance

Inheritance is a relationship among classes that share common properties. It is a concept that is conducive to reusability and extensibility. A subclass inherits the operations and instance variables of its superclass(es) and adds new operations and instance variables. Inheritance can be single or multiple. In single inheritance, the subclass inherits from one superclass; while in multiple inheritance, the subclass inherits from two or more superclasses. Related classes are organized in a *hierarchy* representing their shared behaviors. At the top of the hierarchy are the most general classes and at the bottom are the most specific classes. For example, biologists group organisms into classes as illustrated in Figure 1.5 adopted and modified from [Wegner 90].

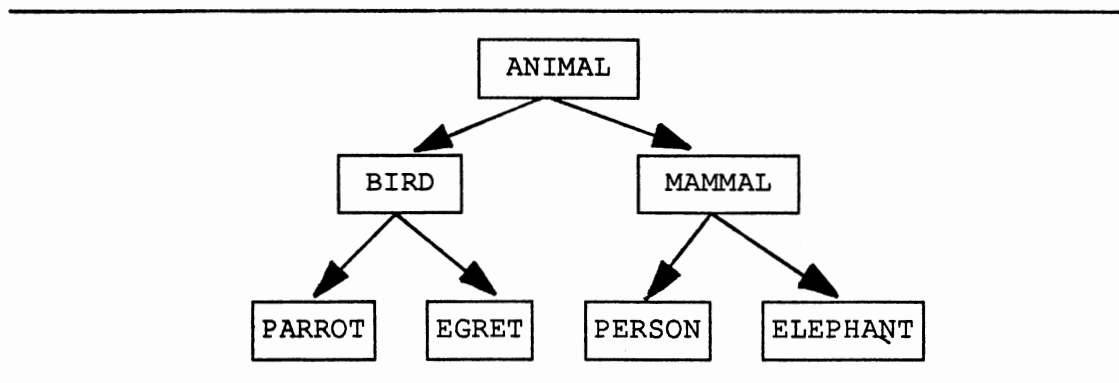


Figure 1.5: An example of single inheritance hierarchy

In Figure 1.5, a subclass specializes its superclass, and conversely, the superclass generalizes its subclass(es). Inheritance increases specificity. A more specific class (e.g. the class PERSON) inherits properties from a more general class (e.g. the class MAMMAL), and it can also add specific properties. In other words, a subclass extends and/or modifies an existing class by adding more properties that specialize its behavior. The behavior of any given class in the hierarchy is an amalgamation of the behaviors of all of its ancestor classes. Single inheritance among classes is represented by a tree structure. To illustrate multiple inheritance, consider the Pie hierarchy illustrated in Figure 1.6 adopted from [Moon 86].

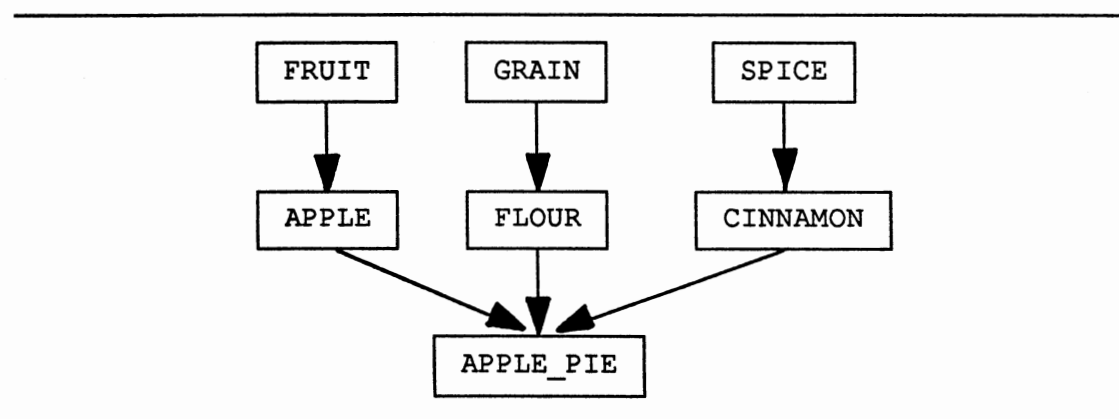


Figure 1.6: An example of multiple inheritance hierarchy

In Figure 1.6, the class `APPLE_PIE` inherits from three superclasses that represent the main components of an apple pie. It inherits the properties of its superclasses, and adds more properties. Unlike the inheritance hierarchy illustrated in Figure 1.5, the subclass is not a specialization of its superclass(es), and the superclass is not a generalization of its subclass(es). This example shows another view of inheritance among classes. Multiple inheritance is represented by a directed acyclic graph (DAG).

To follow up the car example, the class `CAR` in Figure 1.2 defines what cars are. There are more specialized cars such as Ford cars including `Tempo`, `Escort`, and `Taurus`. Ford cars have different characteristics from cars of other makers. At the same time different brands of Ford cars have different characteristics. Since inheritance supports code reusability and extensibility in the sense that a subclass inherits (uses) the code provided in its superclass(es), we define the class `FORD` to represent the common characteristics of all Ford brands. Moreover, we define the classes `TEMPO`, `ESCORT`, and `TAURUS` to represent the characteristics of cars of these brands. Figure 1.7 illustrates the class hierarchy of Ford cars.

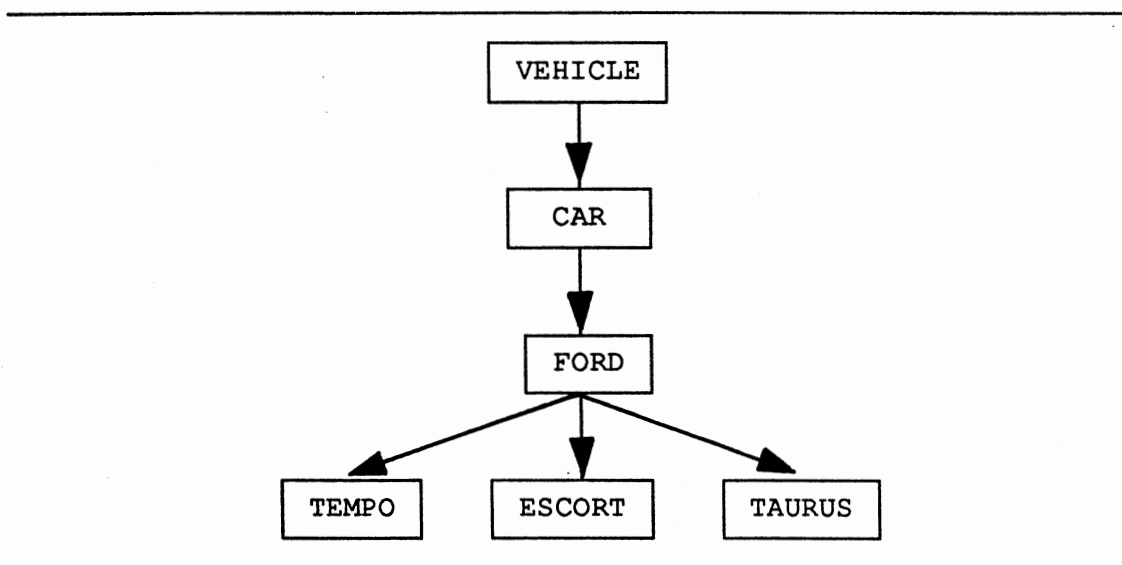


Figure 1.7: The inheritance hierarchy of the class `CAR`

In Figure 1.7, the classes TEMP, ESCORT, and TAURUS define Ford cars in a much more specialized manner than does the class FORD. Class FORD inherits the general properties of the class CAR, and adds properties specific to Ford cars. Class ESCORT inherits the properties of the classes CAR and FORD, and adds properties specific to Escort cars. Therefore, the behavior of an ESCORT car is an amalgamation of the behaviors of the classes FORD, CAR, and VEHICLE. Similarly, the classes ESCORT and TAURUS represent specific behaviors.

A non object-oriented programming language (OOPL) may simulate the OOP style by providing facilities such as encapsulation, genericity, and code reusability. Edelson [Edelson 87] has described how non-OOPLs can simulate the OOP style. He examined the ways in which the three languages C++, Modula-2, and Smalltalk implement object-oriented and abstraction mechanisms in order to help programmers to build large software systems. Klint [Klint 86] and Cook [Cook 86] addressed the relationship between conventional languages and OOP. They reviewed the language features that are required to support the OOP style in a non-OOPL. Such features include encapsulation, inheritance, dynamic binding, genericity, and automatic storage management.

The OOP literature includes many articles that compare and contrast OOPLs. Here we highlight some of the comparisons [Blaschek 89] [Gabriel 89] [Klint 86] [Madsen 89] [Micallef 88] [Siedewitz 87] [Strom 86] [Wolf 89].

Blaschek et al. [Blaschek 89] provided a comparison criteria and compared some OOPLs including C++ [Stroustrup 86,91], Eiffel [Meyer 88], and Smalltalk [Goldberg 83,89]. They compared these languages from the perspectives of inheritance mechanisms, reliability, uniformity of data structures, documentation values, memory management, efficiency, and languages complexity. Gabriel [Gabriel 89] discussed the differences between the object-oriented computational model and the imperative model from the message-passing perspective. He also compared the message passing and generic functions in CLOS [Keene 89] [Bobrow 88] as a Lisp-based language.

Klint [Klint 86] compared OOPLs and conventional programming languages from the perspective of their support for reusability. He used the common example queue data type to illustrate the differences between these languages. He compared Smalltalk implementation of Queue data type against Pascal, Modula-2, and Ada implementations. Madsen and Pederson [Madsen 89] compared the use of virtual classes in the languages BETA [Kristensen 87], Simula [Kirkerud 89], C++ [Stroustrup 86,91], and Smalltalk [Goldberg 83,89]. They presented the notion of a virtual class as a general language mechanism as opposed to the characteristic of a specific language.

Micallef [Micallef 88] provided a comparative survey of OOPLs from the encapsulation, reusability, and extensibility point of views. He outlined some basic concepts and terminologies of OOP. He also addressed and compared the languages Simula [Kirkerud 89], C++ [Stroustrup 86,91], Smalltalk [Goldberg 83,89], Flavors [Moon 86], and CommonObjects [Snyder 86b] in terms of their support for encapsulation, reusability, and extensibility. Siedewitz [Siedewitz 87] compared the basic properties of Ada [Ada 79,83] and Smalltalk using examples in both languages. These properties are encapsulation, inheritance, and binding. He also highlighted the strengths and weaknesses of both types of languages from an object-oriented perspective.

Strom [Strom 86] compared the object-oriented and process paradigms with emphasis on their usefulness for development of large systems. He indicated that both paradigms have computational models based upon message passing; both provide a clear separation between external interfaces and internal algorithms with local data. On the other hand, they differ in many details including their type systems. He also presented and contrasted the mechanisms of each paradigm needed to support dynamic code binding, code reuse, and access control. Wolf [Wolf 89] compared C++ [Stroustrup 86,91] and Flavors [Moon 85,86] from their design perspectives. He compared their data abstraction, inheritance, and method determination (polymorphism). He also discussed the importance of typing and memory management in OOPLs.

1.5 Overview of the Dissertation

The remainder of this dissertation is organized as follows: An extended literature review of the basic concepts of OOP and a highlight of these concepts as realized by OOPLs are provided in chapter 2. chapter 3 provides a survey and taxonomy of the inheritance models adopted by the most common OOPLs including Trellis/Owl, C++, Eiffel, CommonObjects, CLOS, Flavors, Smalltalk-80, and Simula. Chapter 4 addresses the support for code reuse in the languages C++ and Eiffel in terms of inheritance, polymorphism, and other related issues. Chapter 5 describes the object-based inheritance model TIM that supports encapsulation with inheritance along with other related issues. Chapter 6 describes the feedback inheritance model that relaxes the hierarchical model by allowing superclasses to access the methods and instance variables provided in their subclasses. chapter 7 describes the implementation inheritance model that facilitates code reuse among classes by allowing access to implementations of methods provided in ancestor classes. Finally, chapter 8 is the conclusion of the dissertation and suggested future work.

CHAPTER II

LITERATURE REVIEW

2.1 Introduction

OOP incorporates an important set of techniques that facilitate the development of efficient and reliable software systems by allowing code reuse and modular design. OOP has its roots in programming languages such as Simula [Kirkerud 89] and Smalltalk-80 [Goldberg 83,89]. The object-oriented paradigm, which has evolved from OOP, is built on the concepts of structured programming, data abstraction, and software reuse. The role of block structure in OOPLs is discussed by Madsen [Madsen 86]. He examined the block structure in the languages Simula and Smalltalk from the locality, scope rules, and syntax perspectives. The history and the basic concepts of OOP also are discussed extensively in the literature [Alws 85] [Bezivin 87] [Goguen 86b] [Kerr 86] [Love 86] [Nygaard 86].

OOP can be traced back to the concept of "object" defined in Simula. In Simula, a program is a collection of objects (system objects). Objects of a common structure are described by a class declaration. The term "object-oriented" refers to the use of objects associated with behaviors rather than code and data structures. Therefore, object-oriented programs are not seen as a collection of code but as a collection of objects that exchange messages to activate their behaviors [Nierstrasz 86] [Stroustrup 87].

OOP is also centered around the concept of building programs from reusable software components. Packaging (encapsulation), user-defined data types (classes), inheritance, and polymorphism are the major tools for modular design [Meyer 88].

Encapsulation allows the decomposition of large systems into small encapsulated subsystems that can be maintained easily and are portable. *Classes* are user-defined data types that encapsulate data and operations. Classes are the building blocks in the construction of object-oriented programs, in the sense that new classes can be built from old classes [Parnas 76] [Zhong 88]. *Inheritance* as defined in the previous chapter is a relationship among classes that have common properties. Related classes form the *inheritance hierarchy* of a system [Snyder 86a]. *Polymorphism* is the method determination mechanism that adds the power to choose and invoke methods at run time [Nierstrasz 89].

The object-oriented paradigm provides a practical programming methodology. It aids programmers by using their time, skill, and creativity more efficiently in order to develop large software systems. This view is discussed by Edelson [Edelson 87]. Gabriel [Gabriel 89] also outlined the benefits of the object-oriented paradigm including code reusability, abstraction, separation of specification from implementations, prototyping, modularity, and distributed procedure definition. The advantages and disadvantages of the OOP paradigm are also outlined by Klint [Klint 86].

The following sections provide a review of the major concepts of OOP as realized by current OOPs and presented in the literature. Even though the primary focus of this dissertation is on inheritance, for the sake of completeness, this review includes mutually related concepts of class, instantiation, inheritance, message passing, encapsulation, and polymorphism and binding.

2.2 Classes

The notion of *object* was first used in Simula [Kirknerud 89], a language designed for simulation. An object is a collection of private data and a set of methods (operations) that manipulate the data. An object's methods operate on its data responding to incoming

messages that tell the object *what* to do [Stroustrup 87] [Nierstrasz 86]. In Simula, an object is used in the simulation of real-life systems represented as software components. This usage of objects was extended to include prototyping and application development as described by Nierstrasz [Nierstrasz 89]. This direction was pursued also by Smalltalk [Goldberg 83,89].

The concept of "object" is addressed by Nierstrasz [Nierstrasz 86] and Stroustrup [Stroustrup 87]. They have outlined some guidelines that help to distinguish "truly" object-oriented systems from others. A discussion of objects, their definitions, and their roles in OOP appears in many research articles and books in the literature [Booch 91] [Borning 86] [Budd 91] [Cox 86b] [Gabriel 89] [Geoffrey 88] [Kaiser 90] [Nierstrasz 89] [Nygaard 86] [Stein 87] [Snyder 86a] [Wegner 90].

Klint [Klint 86], Madsen [Madsen 86], and Snyder [Snyder 86a] viewed an object as a collection of the instance variables of a class. Many objects have the same behavior and they respond to the same message the same way. Objects of a class are independent of each other [Cox 86] [Klint 86] [Stein 87]. Freeman [Freeman 83] viewed an object to be any information that the developer and designer need in the process of creating software systems.

The notion of *class (object class)* is used to describe the collection of data structures and methods that implement objects. Snyder [Snyder 86a] described a class as a set of methods that can be performed on objects of that class. Methods are implemented as processes that can access and update the instance variables of the target object. Parnas's [Parnas 76] view is that a class is the definition of an abstracted data type that consists of data and methods. Zhong [Zhong 88] also addressed the integration of abstract data types and OOP. He showed that the integration of the abstract data types and the OOP paradigm can be used to achieve high productivity and reliability. He used Smalltalk [Goldberg 83,89] to implement an algebraic specification that takes a hierarchy of specifications and automatically generates Smalltalk classes.

As an example, consider the data type queue illustrated in Figure 2.1. A queue is a first-in-first-out list of elements on which the following operations are defined.

<i>empty</i>	True if the queue is empty; false otherwise.
<i>full</i>	True if the queue is full; false otherwise.
<i>insert</i>	Insert an element at the rear of the queue.
<i>delete</i>	Delete an element from the front of the queue.

The pseudo code of the class QUEUE is given in Figure 2.2.

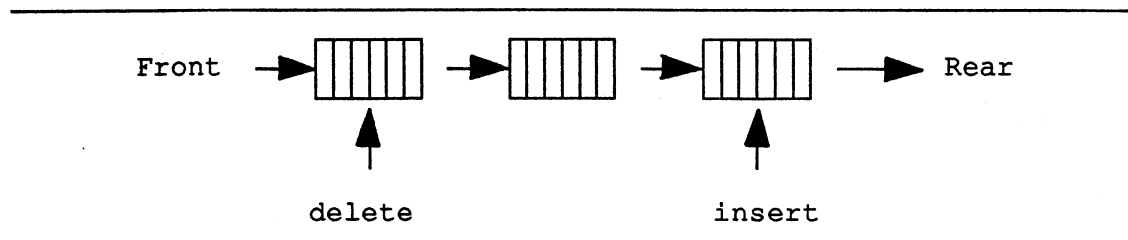


Figure 2.1: Representation of the data type queue

In the literature, a class is viewed in a number of different ways. For instance, a class is viewed as a *template*, *factory of objects*, or *type* [Cox 86] [Nierstrasz 89]. A class is also viewed as an *encapsulated user-defined data type* that abstracts data and their operations. Data structures and the implementations of operations are hidden from users of the class. A class also is called a *flavor* in the programming language Flavors [Schaffert 86]. A flavor defines some instance variables, methods, and specifies other flavors that it inherits. These views are repeated in different places in the literature [Borning 86] [Cannon 82] [Cox 86] [Geoffrey 88] [Keene 89] [Madsen 86] [Moon 85,86] [Schaffert 86] [Snyder 86b] [Wegner 90].

In some OOPs, such as Smalltalk [Goldberg 83,89] and CLOS [Bobrow 88], a class whose instances themselves are classes is called a *metaclass*. A metaclass controls the representation of instances of its instances; while a class controls the structure of its instances. Metaclasses provide methods (called classes methods in Smalltalk) used by their instances (classes). Briot and Cointe [Briot 89] discussed the limitations of

```

Class: QUEUE
Instance Variables: first, last, size, maxsize, elems
Methods:
  create(n) begin
    first = last = 0, maxsize = n
    elems = Array [n]
  end
  empty() if size = 0 return TRUE, else return FALSE
  full() if size = n return TRUE, else return FALSE
  insert(e) begin
    if full return "Overflow"
    else begin
      size = size + 1
      last = (last+1) mod maxsize
      elems[last] = e
    end
  end
  insert(e) begin
    if empty return "Underflow"
    else begin
      size = size - 1
      first = (first+1) mod maxsize
      return elems[first]
    end
  end

```

Figure 2.2: Pseudo code of the class QUEUE

metaclasses in Smalltalk from the private class/metaclass perspective and the non-uniform protocol of instantiating objects. They have described how one can extend standard Smalltalk programming to provide programming with explicit metaclasses. They also showed how explicit metaclasses are supported in the languages CLOS [Keene 89] and ObjVLisp [Cointe 87]. Metaclasses and their use are also addressed elsewhere in the literature by Bobrow [Bobrow 88], Cox [Cox 86], Gabriel [Gabriel 89], Geoffry [Geoffry 88], and Wegner [Wegner 90].

A class can be constructed from scratch or by using and/or modifying some other existing classes. When using a class to construct a new class, the former is called a *superclass* and the latter is called a *subclass*. For example, consider the data type double ended queue (deque). Deque is a queue with two additional methods: deleting an element from the rear of the queue, and inserting an element at the front of the queue. Here, the class QUEUE is used in the construction of the class DEQUE. Class QUEUE is the

superclass; while class DEQUE is the subclass as graphically illustrated in Figure 2.3.

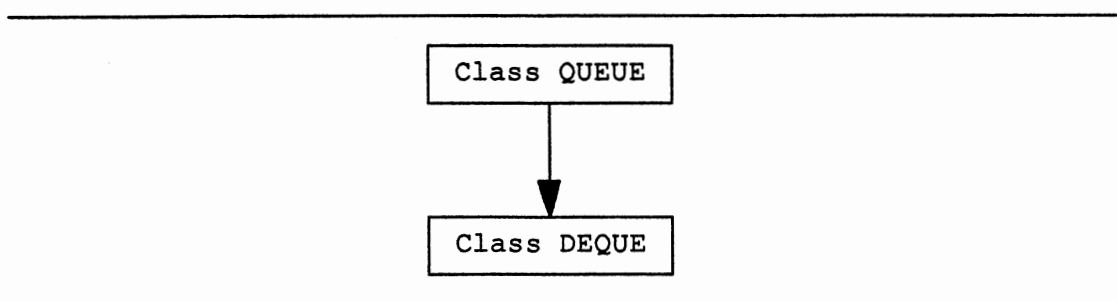


Figure 2.3: The super/subclass relationship between QUEUE and DEQUE

The superclass/subclass notion is used in Smalltalk [Goldberg 83,89]. Other similar notions are *base/derived* used in C++ [Stroustrup 86,91], *parent/child* used in CommonObjects [Snyder 86b], and *type/subtype* used in Trellis/Owl [Schaffert 86]. The terms *ancestor* and *descendant* are used in the obvious way in most OOPs.

A subclass may modify methods of its superclass and may add new methods and variables of its own. Gabriel [Gabriel 89] uses the term *classification* to denote the mechanism for attributing behaviors to classes of objects. That is, grouping of objects denotes the process of building classes. For example, consider the classification of closed figures adopted from [Meyer 88] and illustrated in Figure 2.4. A subclass uses and/or modifies its superclass. A subclass may add new methods and variables of its own.

While classes are descriptions of format and instantiation, prototypes are examples of objects. Prototypes are used to improve the users' understanding of objects. With classes, users can produce many objects of the same behavior; while with prototypes, users can produce unique objects of unique behaviors. Prototypes are used in the object-based languages Self [Ungar 87] and Emerald [Black 86]. Self is a programming language based on the ideas of prototypes and slots. Prototypes combine both inheritance and instantiation, and slots represent the state and behavior of an object. Emerald is an object-based language for the construction of distributed applications. It provides a uniform

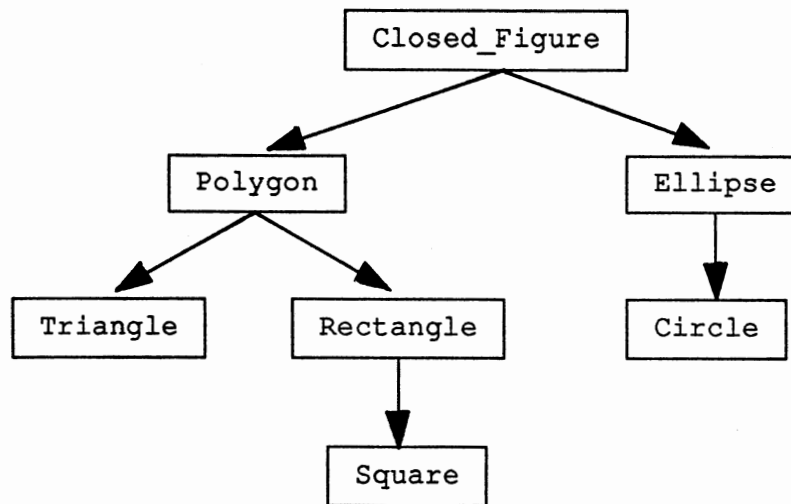


Figure 2.4: A classification of graphical classes

object model for programming both private local objects and shared remote objects. Objects can move among nodes of a network. Black et al. discussed the structure, programming, and implementation of Emerald. Prototypes are widely discussed in the literature [Borning 86] [Lieberman 86] [Madsen 86] [Vines 89].

2.2.1 Instance variables and Methods

A class mainly contains variables (called *instance variables*) and methods. The instance variables of a class represent the private data of the objects that are instances of that class. The instance variables of an object are initiated either at definition time (i.e., compile time) or at creation time (i.e., run time). The "state" of an object is given by the values of its instance variables at any point in time. For example, the instance variables of the class QUEUE in Figure 2.2 are initialized at creation time (i.e., when method create is executed). One may initialize the instance variables first and last at definition time as shown in Figure 2.5.

```

Class: QUEUE
Instance Variables: first=0, last=0, size, maxsize, elems
Methods:
    create(n) begin
                maxsize = n; elems = Array [n!;
                end;
    ... as before

```

Figure 2.5: Initialization of the instance variables of the class QUEUE

OOPLs allow users to declare variables of different scopes and visibilities. For instance, the private, public, and protected variables in C++ [Stroustrup 86,91] have different scopes in the object that contains them. For a defining class, the public variables are visible to methods of inheriting classes, the private variables are visible to methods of the defining class, and the protected variables are visible to methods of the defining class and methods of any class immediately inherits from the defining class. Variables of different scopes are provided also in other languages including CommonObjects [Snyder 86b], Eiffel [Meyer 88], Flavors [Moon 85,86], Trellis/Owl [Schaffert 86], Self [Ungar 87], and Smalltalk [Goldberg 83,89].

Snyder [Snyder 86a] indicated that a class defines the behavior (functionality) of its objects. Methods of a class represent the behavior of the objects created from that class. They are the procedures that perform different functions on the values of the instance variables. Methods can have different scopes. A method has a *specification* and an *implementation*. Specifications of methods are made visible to the users of a class, and they represent the *interface* of the defining class. For example, the interface of the class QUEUE in Figure 2.2 is illustrated in Figure 2.6.

The implementations (often called realizations) of methods are hidden in the class [Cox 86] [Gabriel 89] [Wegner 90]. Methods of a class can be *instance methods* that belong to specific objects of that class, and/or *class (universal) methods* applicable to all objects of that class. Both types of methods are used in Smalltalk and Trellis/Owl.

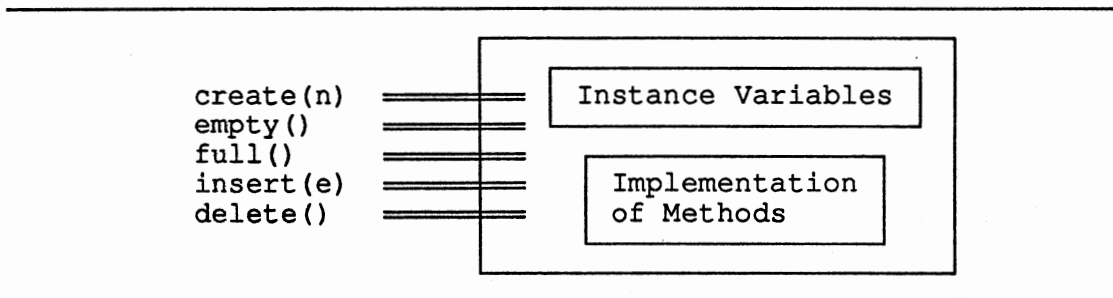


Figure 2.6: Representation of the interface of the class QUEUE

The object-based programming language Self [Ungar 87] uses *slots* instead of instance variables. An object contains a set of named slots that may represent state or behavior. Ungar has argued that using instance variables with classes limits the power of inheritance. That is, the names and the order of the instance variables restrict the format of objects. Moreover, accessing variables through methods, rather than through sending messages, limits the power of the message passing system. Accessing via messages makes inheritance more powerful and allows sharing of the state among objects [Ungar 87]. Slots are also used in the class-based programming language CLOS [Bobrow 88].

Stein [Stein 87] pointed out that inheritance allows objects to share instance variables and methods but not values since values are stored in the instance variables of objects and not in the class itself. The values of instance variables of an object do not affect other objects [Stein 87]. For example, the instance variables of the class QUEUE illustrated in Figure 2.2 are all indirectly accessed through the interface methods shown in Figure 2.6.

Accessing instance variables by methods does not violate encapsulation. This issue is addressed by Snyder [Snyder 86b]. He indicated that restricting descendant classes to access the instance variables using methods is a desirable approach. This approach is used in the languages Trellis/Owl [Schaffert 86], C++ [Stroustrup 86,91], Simula [Kirkerud 89], and CommonObjects [Snyder 86b]. Cannon [Cannon 82] has also indicated that methods must be defined in order to modify the instance variables in Flavors.

2.3 Instantiation

Freeman [Freeman 83] explained the objectives of reusable software engineering and defines classes of information that can be reused. He also highlighted the processes and conditions surrounding reusability that improve the ability of software reuse. He indicated that the following characteristics are essential for reusable information. These characteristics are:

- 1) Items being used are pieces of executable code.
- 2) The definition of a piece of code is system- or organization-specific.
- 3) A reusable piece of code in a collection has little or no operational meaning without being part of that collection.
- 4) The focus of reusability is on reducing the number of lines of code that the programmer needs to write to build new applications.

Various other issues on reuse, which is growing rapidly specially in the OOP area, are discussed in the literature [Biggerstaff 89] [Goguen 86a] [Tracz 88 a,b] [Wegner 83].

Meyer [Meyer 87] also discussed approaches to reusability. He outlined some simple approaches including reusability of source code, personnel, designs, and subroutine libraries. He also addressed the issues of overloading and genericity, and showed that these techniques do not solve all of the issues of reusability. He concluded that isolating users of modules from the internal implementations is the required technique for capturing commonalities within the implementations of related data structures. Klint [Klint 86] also discussed the concepts of modularization and reusability in both algorithmic and OOPLs using examples. His research showed that reusability is limited in conventional programming languages and is more general in OOPLs.

Instantiation is one form of reusability in OOP, as described by Wegner [Wegner 90], Cox [Cox 86], and Nierstrasz [Nierstrasz 89]. It is the process of creating *objects* (often called *instances*) from classes. The precondition of instantiation is existence of a

class containing instance variables and methods. For example, the pseudo code to create the queue Q1 of size 50 elements is "Q1 is QUEUE(create(50))". OOPLs provide different syntax for creating and initializing objects of classes. Examples of creating Q1 of size 50 in some OOPLs are given in Figure 2.7. Note that, a language may provide different syntax to create and initialize objects of a class.

C++:	QUEUE Q1(50);
CLOS:	(setq *Q1* (make_instance 'QUEUE :name "Q1" :size 50));
Eiffel:	Q1:QUEUE[<element_type>; Q1. create (50);
Smalltalk:	Q1 <-- QUEUE new
Trellis/Owl:	Q1 := create (QUEUE [50])
CommonObjects:	(setf Q1 (make_instance 'QUEUE :size 50));

Figure 2.7: Examples of creating object Q1 in different OOPLs

Conceptually, objects of a class are created at run time in response to creation requests sent to that class [Borning 86] [Cox 86] [Goldberg 83,89]. Instantiation implies memory allocation for variables of an object, and linkage between methods and the code segments representing their implementations. This perspective is also addressed by Nierstrasz [Nierstrasz 89], Schaffert [Schaffert 86], and Snyder [Snyder 86 a,b].

Objects can be instantiated by the user or the programming language itself. Objects instantiated by users (often called *user objects*) are objects of user-defined classes that contain hidden instance variables and visible methods. These objects are responses to the creation requests declared in a user's program such as the creation statements illustrated in Figure 2.7. The disposal of these objects is also the response to user-defined destruction requests [Nierstrasz 89] [Wegner 90] [Goldberg 83,89]. For example, method

dispose can be defined in class QUEUE in order to deallocate the queue structure of a queue object when it is invoked. Such a method deallocates the memory assigned to an object at creation time.

Similar to built-in data types, programming languages can instantiate objects either statically (allocated at compile time and remain during program execution) or dynamically (allocated and de-allocated during run time). These so-called *system objects* (instances of system-defined classes) are instantiated and disposed in response to system-defined requests. For example, the Smalltalk-80 system [Goldberg 83] provides system classes such as "data structures", "control structures", and "input/output facilities". These classes are used to create system objects that provide the functionality and environment of the language. These objects are not visible (accessible) to users of the language [Nierstrasz 89] [Wegner 90].

Another alternative for object creation is *cloning* (copying) prototypical objects. In Self [Ungar 87], objects are created by cloning prototype objects that behave like classes. Unlike instantiation, cloning results in a new object whose initial state is the current state of the prototype object at creation time. Prototypes and their applications are addressed in detail by Liberman [Liberman 86] and Vines [Vines 89]. The differences between classes and prototypes are outlined by Borning [Borning 86] and Liberman [Liberman 86].

2.4 Inheritance

One of the important features of OOPLs is the *inheritance* mechanism they support. In addition to instantiation, Cox [Cox 86] and Nierstrasz [Nierstrasz 89] have described inheritance as another form of reusability in OOP. The concept of inheritance provides a natural mechanism for code sharing among classes. A general view of inheritance is that it is a mechanism for code sharing among classes. Another view of

inheritance is that it is an extension of data abstraction definition [Gabriel 89]. Snyder [Snyder 86a] pointed out that inheritance involves objects and classes where a class defines the behavior of its objects. The concept of inheritance and its models are addressed in many research articles [Borning 86] [Cannon 82] [Cox 86] [Edelson 87] [Gabriel 90] [Geoffrey 88] [Hailpern 87] [Keene 89] [Moon 85,86] [Nierstrasz 89] [Pedersen 89] [Schaffert 86] [Siedewitz 87] [Snyder 86 a,b] [Stein 86] [Ungar 87] [Wegner 90] [Wolf 89].

Inheritance relates classes to each other. As described in the first chapter, inheritance includes two relationships: One-to-one and many-to-one. The one-to-one relationship is called *single inheritance* and it relates a subclass to only one superclass. Smalltalk as proposed by Goldberg [Goldberg 83] and the first version of C++ [Stroustrup 86] support single inheritance. The many-to-one relationship is called *multiple inheritance* and it relates a subclass to two or more superclasses. Multiple inheritance is obviously an extension of single inheritance. A graphical illustration of single and multiple inheritance is given in Figure 2.8. Some of the languages that support multiple inheritance are Smalltalk-80 as proposed by Borning [Borning 80], the second version of C++ [Stroustrup 91] [AT&T 89 a,b], CommonObjects [Snyder 86b], CLOS [Bobrow 88], Eiffel [Meyer 88], Flavors [Moon 85,86], and Trellis/Owl [Schaffert 86].

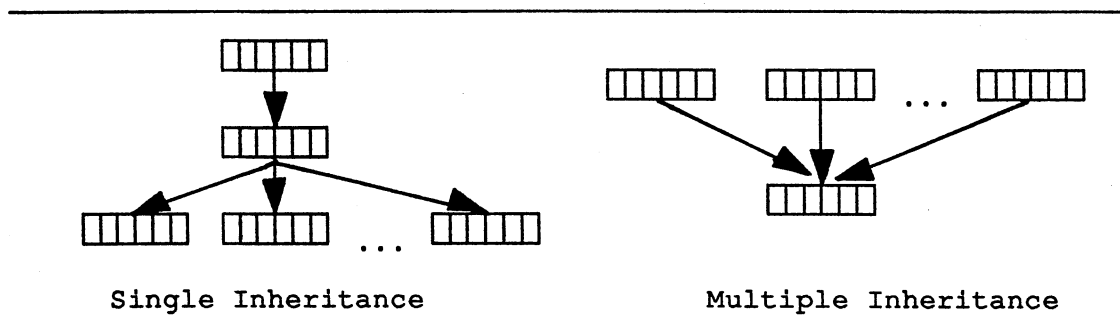


Figure 2.8: Representation of single and multiple inheritance relationships

In single inheritance, a class can only inherit from one superclass but it can have any number of subclasses. It is reasonable to be able to construct an object that is a composition of different types of behaviors (classes). Therefore, the behavior of an object is not limited to things inherited from one superclass. This is accomplished by using multiple inheritance [Gabriel 89]. Multiple inheritance increases the reusability of software components and it encourages users to combine simple software components to build new complex ones [Cox 86] [Snyder 86a].

Inheritance has different forms and meanings depending on *when* and *how* it takes place. The most common form of inheritance is called *class inheritance* and it distinguishes OOPs from other programming languages. It is often referred to as *static inheritance*. This form of inheritance takes place when classes are defined. It is the mechanism that allows the definition of new classes from existing classes. The simplest form of static inheritance is called *extension*. Here, a subclass inherits all of the methods and instance variables of its superclass(es). The subclass does not override inherited methods; it may add new methods and/or instance variables [Hailpern 87] [Snyder 86a].

Another form of static inheritance is called *variation*. Variation is the same as extension with the added capability to override inherited methods. A subclass can modify inherited methods and/or add new methods and instance variables [Nierstrasz 89] [Wegner 90]. A combination of extension and variation is called *specialization* [Nierstrasz 89]. In this form of static inheritance, a subclass inherits all of the methods and instance variables of its superclass(es), and possibly modifies some of the inherited methods and adds new methods and instance variable. An object of the subclass is an object of the superclass(es) since features of the superclass(es) hold for objects of the subclass [Gabriel 90] [Snyder 86a] [Wegner 90].

As an example of variation inheritance, consider the POLYGON and RECTANGLE classes adapted from [Meyer 88]. Class POLYGON is a description of

general polygons (a polygon has at least three vertices). It provides the methods *move* to move a polygon horizontally and vertically, *rotate* to rotate a polygon with its centroid as the center of rotation, *display* to display a polygon on screen, and *perimeter* to compute the perimeter of a polygon. Class POLYGON is illustrated in Figure 2.9.

```

Class: POLYGON
Instance Variables: vertices, perimeter_length, ...
Methods:
  move(hrz, ver) begin
                  "Move horizontally using hrz and
                  vertically using ver"
                  end
  rotate(center, angle) begin
                       "Rotate around center using angle"
                       end
  display() begin
            "Display polygon on screen"
            end
  perimeter() begin
              "loop through vertices and sum
              the edge lengths"
              end

```

Figure 2.9: Pseudo code of the class POLYGON

Now consider the class RECTANGLE, illustrated in Figure 2.10, as a special form of the class POLYGON. A rectangle can be moved, rotated, or displayed on screen in the same way as a polygon. Additionally, a rectangle has four vertices, diagonal, and perimeter that can be easily calculated than a polygon. All features of polygon are applicable to rectangle, and rectangle is a specialization of polygon. Class RECTANGLE modifies inherited methods and adds new methods and instance variables. This is a variation inheritance.

The above example illustrates the variation form of inheritance between the classes POLYGON and RECTANGLE. Class RECTANGLE overrides method perimeter inherited from the class POLYGON. To illustrate extension inheritance using these classes, one can exclude method perimeter from the class POLYGON and define it in subclasses of the

```
Class: RECTANGLE
Instance Variables: vertices, sidel, side2, ...
Methods:
  create(center, s1, s2, angle)
    begin
      "Create a rectangle centered at center,
      and with sides s1 and s2 and
      orientation angle"
    end
  perimeter()
    begin
      "Compute perimeter using [2*(side1+side2)].
      This is a redefined version of the method"
    end
  diagonal() begin "Compute diagonal" end
```

Figure 2.10: Pseudo code of the class RECTANGLE

class POLYGON as a new method. This way represents extension inheritance since class RECTANGLE inherits all methods of class POLYGON without modifying any of them. Class RECTANGLE adds new methods. If a given programming language allows users to relate the classes POLYGON and RECTANGLE in either extension or variation inheritance, this language is said to be supporting specialization inheritance.

The opposite view of specialization is called *aggregation*. This view is provided by Nierstrasz [Nierstrasz 89]. As an example, consider class CAR adapted from [Nierstrasz 89] and illustrated in Figure 2.11. A car is an aggregation of different components represented by different classes. Class CAR inherits from the classes BODY, FRAME, WHEELS, and ENGINE. It inherits all methods and instance variables of its superclasses. The aggregation of all inherited features defines the characteristics of the class CAR. Unlike specialization inheritance, we cannot view an object of the subclass as an object of its superclass(es). A car is neither a body, frame, wheels, nor an engine.

Another notion promoted as the opposite of specification is provided by Pedersen [Pedersen 89] and is called *generalization*. Generalization allows users to create superclasses for already existing classes, and thus enabling exclusion of methods and creation of classes that describe the commonalities among existing classes. For

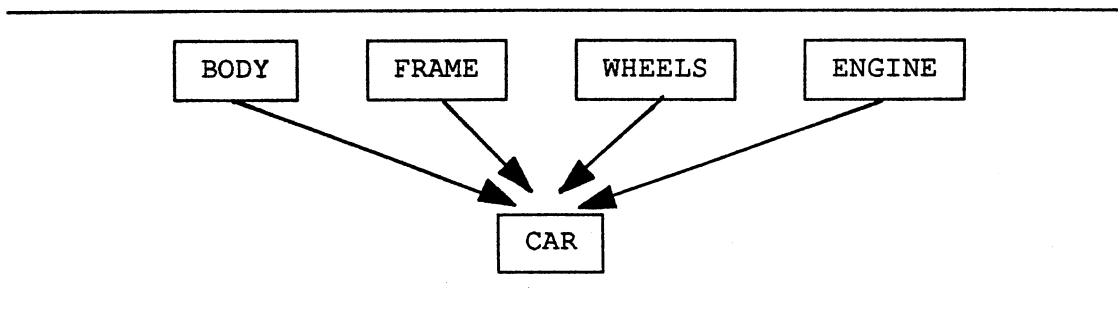


Figure 2.11: Class CAR is an aggregation of its superclasses

illustration, we consider Pedersen's example of stack as a generalization of the data type deque illustrated in Figure 2.12.

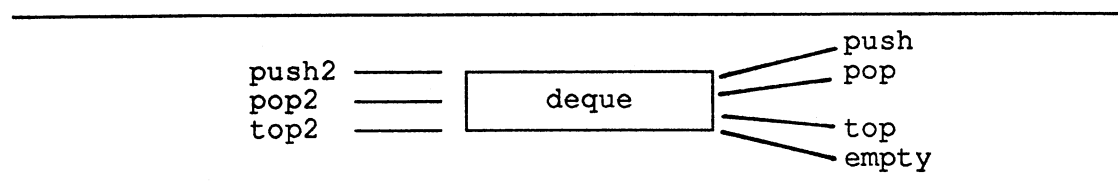


Figure 2.12: Representation of the data type deque

The idea of this example is to reuse selected methods of an already existing class. A deque is a stack in which elements are added and removed from both ends. Assuming that the class DEQUE is already implemented, methods push2, pop2, and top2 need to be excluded in order to convert deque to a stack. As pointed out by Pedersen, this cannot be done using specialization inheritance. In normal cases, we think of deque as a specialization of stack. Using generalization, the class STACK can be a superclass of the class DEQUE. Therefore, class STACK is a generalization of the class DEQUE as illustrated in Figure 2.13 that is adopted from [Pederson 89].

Pedersen indicated that generalization together with specification improves class reusability. He also showed that generalization can coexist with specialization without introducing the problem of naming conflicts [Hailpern 87] [Nierstrasz 89] [Wegner 80].

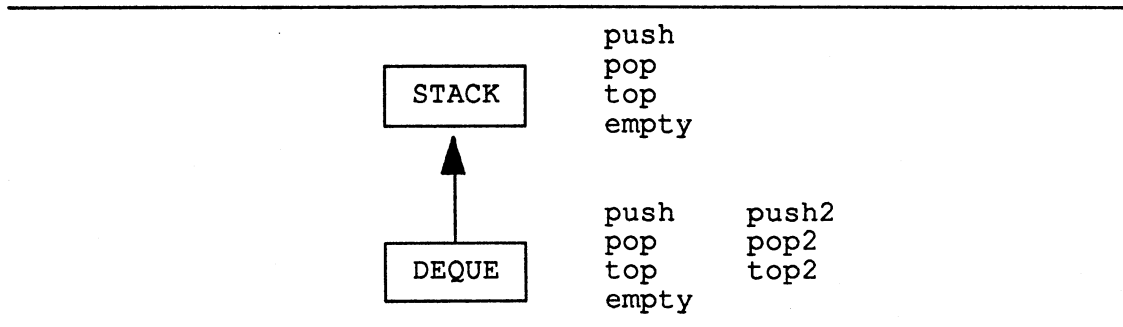


Figure 2.13: the class STACK is a generalization of the class DEQUE

Another form of static inheritance is called *partial (selective)* inheritance. In partial inheritance, a subclass inherits parts of the instance variables and methods of its superclass(es). This form of inheritance is supported by the languages Eiffel [Meyer 87,88], C++ [Stroustrup 86,91], and CommonObjects [Snyder 86b]. Selective inheritance is addressed by Nierstrasz [Nierstrasz 89] and Snyder [Snyder 86b]. As an example, consider CommonObjects' definition of the class DEQUE adopted from [Snyder 86b] and illustrated in Figure 2.14.

```

(define-type DEQUE
  (:var size (:type integer) (:init 100)
         :gettable :initable)
  (:var contents (:type vector)
                 (:init (make-array size)))
  (:var front (:type integer) (:init 1))
  (:var back  (:type integer) (:init 0))
  (:var count (:type integer) (:init 0)))

(define-method (DEQUE :empty?) () (...))
(define-method (DEQUE :full?) () (...))
(define-method (DEQUE :front-push) () (...))
(define-method (DEQUE :front-pop) () (...))
(define-method (DEQUE :front-top) () (...))
(define-method (DEQUE :back-push) () (...))
(define-method (DEQUE :back-pop) () (...))
(define-method (DEQUE :back-top) () (...))
  
```

Figure 2.14: CommonObjects' definition of the class DEQUE

Figure 2.14 describes a deque of maximum size of 100 elements. The instance

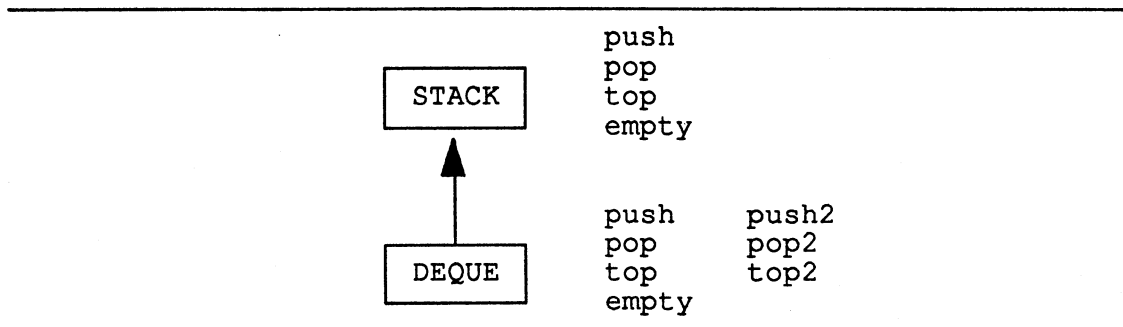


Figure 2.13: the class STACK is a generalization of the class DEQUE

Another form of static inheritance is called *partial (selective)* inheritance. In partial inheritance, a subclass inherits parts of the instance variables and methods of its superclass(es). This form of inheritance is supported by the languages Eiffel [Meyer 87,88], C++ [Stroustrup 86,91], and CommonObjects [Snyder 86b]. Selective inheritance is addressed by Nierstrasz [Nierstrasz 89] and Snyder [Snyder 86b]. As an example, consider CommonObjects' definition of the class DEQUE adopted from [Snyder 86b] and illustrated in Figure 2.14.

```

(define-type DEQUE
  (:var size (:type integer) (:init 100)
         :gettable :initable)
  (:var contents (:type vector)
                 (:init (make-array size)))
  (:var front (:type integer) (:init 1))
  (:var back  (:type integer) (:init 0))
  (:var count (:type integer) (:init 0)))

(define-method (DEQUE :empty?) () (...))
(define-method (DEQUE :full?) () (...))
(define-method (DEQUE :front-push) () (...))
(define-method (DEQUE :front-pop) () (...))
(define-method (DEQUE :front-top) () (...))
(define-method (DEQUE :back-push) () (...))
(define-method (DEQUE :back-pop) () (...))
(define-method (DEQUE :back-top) () (...))
  
```

Figure 2.14: CommonObjects' definition of the class DEQUE

Figure 2.14 describes a deque of maximum size of 100 elements. The instance

environment. Moving that paragraph to a footnote environment results in inheriting different font type, font size, and other features [Nierstrasz 89].

Dynamic inheritance is supported by systems based on prototypical objects as described by Liberman [Liberman 86]. The concept of prototypes is used in the "Object-Based Inheritance Model" outlined by Hailpern and Nguyen [Hailpern 87]. They proposed an inheritance model for code sharing based on objects rather than classes, where objects are processes that communicate through messages. In the "Object Model for Shared Data" described by Kaiser and Hailpern [Kaiser 90], a new object model is proposed in order to support shared data in distributed environment and a language called PROFIT based on that model. Their model accommodates the idea that same data may logically belong to multiple objects and may be distributed over multiple nodes of the network in certain applications. Dynamic inheritance is also used in the languages Self [Ungar 87] and Emerald [Black 86] described earlier.

In class-based programming languages the "is-a" relationship indicates that an object of a subclass can be viewed as an object of its superclass(es): a human is a mammal; and rectangle is a polygon. This relationship is a set inclusion. For example, humans are a subset of mammals; similarly, rectangles are a subset of polygons. The relationship "has-a" indicates that an object of a subclass possesses some properties of its superclass(es). For example, a car has an engine but is not an engine; similarly, a radio has a speaker but is not a speaker. In prototype systems, the relationship "inherit from" describes how objects share behaviors: an object inherits from its prototype object. These interpretations of the inheritance relationship among classes and prototypical objects have been addressed by a number of researchers [Meyer 88] [Nierstrasz 89] [Ungar 87] [Wegner 90] [Zdonik 88].

Another alternative method for incremental definition and sharing is called *delegation*. Delegation allows incremental definition of objects. An object can be defined in terms of other objects. This method is used with object-based programming languages

where objects are not instances of classes, but copies of prototypical objects. Liberman [Liberman 86] indicated that delegation can capture the behavior of inheritance. Stein [Stein 87] has also indicated that inheritance and delegation are alternative methods for incremental definition and sharing. He supported Liberman's argument using examples. Stein demonstrated that there is a natural model for inheritance, which captures all of the properties of delegation. Also, he outlined a framework that captures both delegation and inheritance.

In delegation, objects delegate messages and responsibility instead of inheriting from each other. They can share variables and methods since classes are not present. Objects created from different prototypes can delegate to the same prototype object, and two or more objects of the same prototype can delegate to different prototype objects as described by Stein [Stein 87]. He also provided formal proofs that inheritance and delegation can be used to model each other.

Borning [Borning 86] also discussed prototypes as an alternative for classes and metaclasses. He introduced two problems associated with the use of classes and metaclasses. First, different interfaces for objects require different class definitions. Second, the use of classes requires the user to move to the abstract level of class and write a class definition, and then instantiate and test objects. He suggested the use of prototypes as one alternative to the use of classes in graphic and visual systems. He also discussed the difficulties of classes in Smalltalk [Goldberg 83,89] and the need for metaclasses. He proposed a prototype-based language to illustrate the differences between classes and prototypes.

Gabriel [Gabriel 86] argued that inheritance along with late binding allows users to extend code without having the source code. Methods that did not exist when a code segment is compiled may be called within that segment using dynamic binding. He also indicated that inheritance involves both behavior and structure inheritance. Behavior inheritance means that a class can inherit a method when it does not have it (it is not

associated with it). Inheritance of structure is similar to the definition of the specialization form of inheritance. When objects have similar structures, their classes can be related such that an object of the subclass is an object of its superclass(es). For example, a car has the structure and behavior of an automobile (i.e., a car is an automobile).

In addition to the users of a class that create objects from that class, inheritance adds a new category of users (often called *clients*, as used in Trellis/Owl [Schaffert 86] and Eiffel [Meyer 88]) that inherit from the class. In Eiffel, the term client denotes inheriting classes of a "has-a" relationship. Note that, Snyder [Snyder 86] uses the term client to denote users of a class rather than inheriting classes. This produces another external interface provided by the class to its subclasses. This situation affects encapsulation and limits the ability to change the class contents safely [Snyder 86a].

The formal semantics of inheritance are also discussed in several articles. Cardelli [Cardelli 84] addressed the semantics of multiple inheritance to justify it and solve the problems associated with multiple inheritance. He distinguished between horizontal polymorphism (that has to do with inheritance) and vertical (ordinary) polymorphism. Cook and Palsberg [Cook 89] presented a denotational model of inheritance. They demonstrated the correctness of their model by proving that it is equivalent to the operational semantics of inheritance.

2.4.1 Subtyping

The term *subtyping* is used to denote the specialization form of inheritance [Wegner 90]. When objects of a subclass are seen to be objects of a superclass, the subclass is called a *subtype* of the superclass and the superclass is called a *supertype*. For example, to capture the notion that every manager is an employee, we say that the class MANAGER is a subtype of the class EMPLOYEE. This notion is used in the language Trellis/Owl [Schaffert 86]. The concepts of subtyping, type theories, and type conversions

are addressed in details by Bruce [Bruce 86] and Cardelli [Cardelli 86], Danforth [Danforth 88], and Gannon [Gannon 87]. For a discussion of subtyping in OOPLs, see also [Cox 86], [Gabriel 89], [Nierstrasz 89], [Sethi 89], and [Snyder 86a].

Some OOPLs relate inheritance and subtyping [Halbert 87] [America 87] [Oucournan 87]. Wolf [Wolf 89] discussed the importance of subtyping in OOPLs. He gives two reasons to show that type checking is more important in OOP than in conventional languages. First, a type checker can identify misuse of message names which is hard for the programmer to detect. Second, it may not be obvious whether a method is available to a particular class since inheritance distributes the method(s) defined for a class.

The use of subtyping varies in OOPLs. For instance, in Trellis/Owl [Schaffert 86] and Simula [Kirkerud 89], a subclass is a subtype of its superclass(es). Class STACK is a subtype of class DEQUE if and only if STACK is a subclass of DEQUE. In commonObjects [Snyder 86b], a subclass is not necessarily a subtype of its superclass(es); while in C++ [Stroustrup 86,91], a class is not allowed to be a subtype of its superclass(es) unless public derivation is used. This issue is addressed by Nierstrasz [Nierstrasz 89], Snyder [Snyder 86a], and Wegner [Wegner 90].

Understanding subtyping helps to understand the structure of classes and their inheritance relationships. In Trellis/Owl, Schaffert [Schaffert 86] observed that subtyping is based on behavior and not implementation. That is, methods of a subtype and its supertype may be implemented differently. He characterized this as the definition of specification inheritance. That is, for a given supertype and subtype, objects of the supertype behave like those of the subtype.

Snyder [Snyder 86 a,b] indicated that when subtyping is associated with inheritance, as done in Trellis/Owl, it allows more flexibility and improves efficiency. Thus, declaring a variable O to denote an object of a class C means that O may denote an object of the class C or descendants of C. On the other hand, he indicated that

subtyping in CommonObjects [Snyder 86b] is not related to inheritance and no attempt is made for optimization based on subtyping.

Snyder has also addressed subtyping and its impact on encapsulation and inheritance [Snyder 86a]. He claims that the subtyping rules and their relationship with inheritance in the languages Trellis/Owl [Schaffert 86] and C++ [Stroustrup 86,91] compromise the benefits of encapsulation and limits the language designers' freedom to make changes without affecting the inheriting classes. His proposed solution, which was used in the design of CommonObjects [Snyder 86b], is discussed in the following section.

2.4.2 Problems with Current Inheritance Models

In multiple inheritance, a major problem is that a method may be inherited more than once from an ancestor class through different inheritance paths. Hence the inheriting class can contain multiple instances of the inherited method. Some researchers call this situation *collision*. When a collision occurs in multiple inheritance, there are different decisions one can make [Gabriel 89] [Nierstrasz 89] [Snyder 86a].

- 1) **Shadowing:** Using the properties inherited from the most recent or highest precedence class. This approach is used in Smalltalk [Goldberg 83] to shadow both instance variables and methods, and in CLOS [Bobrow 88] to shadow instance variables.
- 2) **Combination:** Combining all collided properties into one property. This approach is used in CLOS [Keene 89] to combine methods, and in Flavors [Moon 85,86] to merge instance variables.
- 3) **Signalling an Error.** This approach is used in CommonObjects [Snyder 86b] to signal an error when an attempt is made to inherit the same method from different classes.
- 4) **Explicit Selection from among the Collided Properties.** This approach is self-explanatory.

In addition to the above solutions, Snyder [Snyder 86a] outlined three more solutions that deal with the inheritance graph. These solutions are called Graph-oriented

solution, Linear solution, and Tree solution. The first solution deals directly with the inheritance graph. The other solutions flatten the inheritance graph into a linear chain and then deal with the chain using single inheritance rules.

Naming conflicts between instance variables of different classes is a potential problem. Cannon [Cannon 82] indicated that there are several ways to solve this problems. One way is to limit the scope of instance variables at declaration time. Thus, instance variables are accessible to certain classes and not to every class. This approach is an explicit import of variables. Shadowing is another way. These approaches are discussed in details by Cannon [Cannon 82]. In Flavors, another problem outlined by Moon [Moon 85,86] is that when inheriting the same flavor along different inheritance paths, the flavor system eliminates duplicated flavor names by imposing an order on the inherited flavors.

Snyder [Snyder 86a] provided a description of inheriting instance variables, and outlined the problem called *direct access of instance variables*. In most OOPLs the code of a class may access directly all instance variables of its objects including those inherited from its ancestor classes. This approach compromises the encapsulation characteristics since the instance variables of a class should not be explicitly accessible to the inheriting classes. As a result, he indicated that any changes to the instance variables of the superclass(es) (such as renaming or removing) may affect the inheriting classes and hence the language designers' freedom to make changes. The solution he proposed is that the external interface should not include the instance variables. Also, to protect instance variables from direct access, Snyder suggested providing methods to users to access instance variables. These methods are meant to be used by both users and inheriting classes. More details of these solutions are provided in [Snyder 86a].

Another problem outlined by Snyder is called *visibility of inheritance*. That is, should a subclass know about the use of inheritance in its superclass(es)? In other words, should inheritance be part of the external interface of a class? If so, changes to a

superclass's use of inheritance may require changes in the subclass(es) of that class. This approach is used in CommonObjects [Snyder 86b]. CommonObjects is different from other OOPs because of allowing inheritance to be hidden from all users and inheriting classes of a class. That is, a class does not know about the use of inheritance in its superclass(es). This is applied to both methods and instance variables in order to prevent the exposure of the use of inheritance outside the class definition. CommonObjects achieves this characteristic by passing all inherited information through all intervening classes. An error signal is issued when a class attempts to inherit the same method from multiple superclasses.

Relating subtyping to inheritance is another problem outlined by Snyder [Snyder 86a]. He indicated that subtyping exposes the use of inheritance through subtyping rules. His suggested solution is that subtyping should not be related to inheritance. Subtyping should be based on the behavior of objects, and the subtyping hierarchy should be independent of the inheritance hierarchy. For example, consider the stack/deque example provided in [Snyder 86a]. This example demonstrates the separation of inheritance and subtyping hierarchies as illustrated in Figure 2.16. Class STACK inherits from the class DEQUE but is not a subtype of the class DEQUE because it excludes some inherited methods. On the other hand, the class DEQUE is a subtype of the class STACK but does not inherit from the class STACK. The subtype and inheritance relationships between the abstraction and implementation of the stack and deque data types are illustrated in Figure 2.16 adopted from [Snyder 86a].

Marcke [Marcke 88] observed that inheritance implementations in OOPs are complicated. He claimed that the complexity of inheritance results from the desire to express many different concepts by means of one inheritance lattice. He analyzed the complexity problems and their causes associated with inheritance in some of the existing OOPs. He argued in favor of simple inheritance mechanisms that allow users to build complicated information retrieval architectures explicitly, and contribute to the simplicity

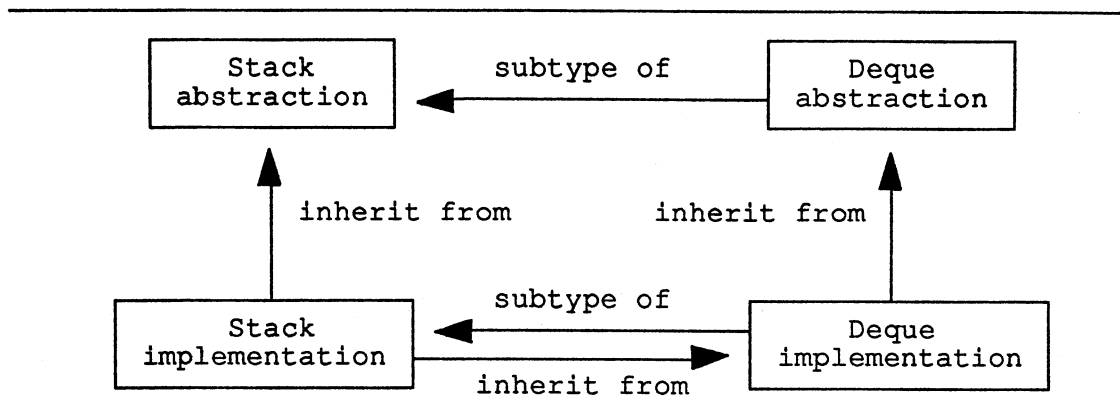


Figure 2.16: Abstraction and implementations of stack and deque

of the language. He presented three alternative schemes for simple inheritance without conflicts. These schemes are called SL1: Multiple Inheritance without Conflicts, SL2: The use of Meta-Interpreters, and SL3: Explicit Method_objects.

Lucas [Lucas 89] investigated the problem of handling confusions arising in frame systems with multiple inheritance. He addressed the inheritance of attribute values from classes to objects of classes. He analyzed multiple inheritance from an algorithmic point of view, and developed an algorithm for constructing a special kind of spanning tree for the associated directed graph of a frame taxonomy. He focused on inheritance of attributes by classes. His method amounts to recording the reasoning that takes place in a frame taxonomy by means of so-called inheritance chains, then applying the notion of "in-between" to decide which attribute values, that are derivable by means of inheritance, should be given preference over others. Kreczmar [Kreczmar 89] also addressed the inheritance rules in OOP. He provided a review for inheritance rules in various OOPLs using examples.

2.5 Message Passing

A *message* is a request sent to an object to perform some activities. It tells the object *what* is to be done rather than *how* it is done. The issue of message passing is

addressed in almost every research that discusses inheritance and objects [Bobrow 88] [Cannon 82] [Gabriel 89] [Hailpern 87] [Keene 89] [Klint 86] [Nierstrasz 86,89] [Snyder 86 a,b] [Wegner 90].

Users interact with objects through messages. The code executed to answer a message is the implementation of a method defined in the class of the receiving object. A class must define both the methods and their implementations. The terms method and implementation are not identical because one method may cause the execution of two or more implementations. This view is discussed by Wolf [Wolf 89]. Nierstrasz [Nierstrasz 89] stated that every method invocation is a message request to one or more objects to perform some actions.

Objects cannot operate on each other, instead they interact by sending messages. The receiving objects interpret messages differently. A receiving object may respond directly, or it may decide not to answer the message and return an appropriate response. This view is outlined by Nierstrasz [Nierstrasz 86]. He also indicated that message passing is a model for object communication.

With specialization (refinement) form of inheritance, Klint [Klint 86] indicated that an object of a subclass responds to messages sent to objects of the superclass(es). In message passing, each object is capable of answering certain messages. Messages of identical names can be defined for several objects. The behavior of a message depends on the type of the object to which the message is sent.

Some OOPLs translate messages to procedure call when only one object is going to answer the message (i.e., single-thread flow of control). Messages are converted into procedure calls for reliability and implementation simplicity. Klint [Klint 86] indicated that methods performed by means of messages are independent of the amount of work required to use these messages. However, for small operations such as addition of two numbers, the overhead of the message can be relatively high. Therefore, direct procedure calls have their advantages in similar cases.

Self [Ungar 87] is an object-based language that depends on messages heavily. It provides a different perspective on objects and messages. When an object receives a message and it has no matching slot, the search continues through the superclass objects, similar to the search method used in most OOPs including CommonObjects [Snyder 86b], Smalltalk [Goldberg 83], and Trellis/Owl [Schaffert 86]. Objects may also send messages to "self" to access the values of stored slots.

A *protocol* is a set of messages that specify the external behavior of an object. The protocol does not define how the behavior is to be implemented by an object. This is outlined by Cannon [Cannon 82]. In a message passing system, Gabriel [Gabriel 89] indicated that the method that handles a message is determined by the class of the objects to which the message was sent.

OOPs vary in their implementation of the message passing system. For instance, in Lisp Machine system [Cannon 82], messages are converted to function calls, and Trellis/Owl [Schaffert 86] employs the standard procedure call notation for invoking methods. On the other hand, other languages such as Smalltalk [Goldberg 83,89], Self [Ungar 87], and CommonObjects [Snyder 86b] use messages rather than procedure/function calls. Messages are also used in other models including the "Object-Based Inheritance Model" proposed by Hailpern and Nguyen [Hailpern 87] and the "Object Model For Shared Data" proposed by Kaiser and Hailpern [Kaiser 90].

2.6 Encapsulation

Abstract data is a set of data items (values) and a set of methods that manipulate the data items [Cardelli 86] [Madsen 86] [Nierstrasz 86]. The behavior of an abstract data item is defined by its set of methods. This approach facilitates the program development and maintenance by allowing the program designer to safely change the implementation details of an abstract data type without affecting its users. The less the exposure of

implementation details, the more *encapsulation* is achieved. Data abstraction goes along with encapsulation. A programming language supports encapsulation if users of a module are restricted to access that module only through its external interface (methods).

Meyer [Meyer 88] addressed modularity to assess what it means for a software construction method to be modular. He discussed modular composability, continuity, and protection. He also outlined and examined the principles that ensure proper modularity. These principles are: linguistic modular units, few interfaces, small interfaces, explicit interfaces, and information hiding.

In OOPLs, a class definition is a module with its own external interface. An object with a defined external interface is an abstract entity such that its users do not need to understand how the methods are implemented and how the data is represented. This concept and the relationship between data abstraction and encapsulation have received a lot of attention [Cardelli 86] [Cox 86] [Edelson 87] [Gabriel 89] [Klint 86] [Sethi 89] [Siedewitz 87] [Snyder 86a] [Wegner 90].

Klint [Klint 86] observed that encapsulation is the foundation aspect of OOP. It involves the separation of functionality from the underlying implementations of a structure. The functionality of a structure is provided to users through the interface, while the underlying implementations are hidden. Therefore, unnecessary access to data is prevented and the life-time of the structure is increased. Long life-time implies that details of the implementations can be changed any time without affecting users of the structure.

Most OOPLs support data abstraction by preventing objects from being manipulated except through their defined external interfaces, as discussed by Snyder [Snyder 86a]. He also pointed out that the support for data abstraction is one of the prime features of OOP. That is, users have the ability to define new objects of a behavior abstractly without any reference to implementation details. Implementations of methods are called the internal view of an object as described by Schaffert [Schaffert 86]. He outlined that encapsulation facilitates the modification of software and improves the

understandability of programs. Also, encapsulation minimizes the interdependency among separated models by defining interfaces such that changing the implementations does not affect the users of a module.

OOPLs vary in their support for encapsulation based on the kinds of changes that can be made safely to the implementation of a module. For instance, CommonObjects [Snyder 86b] provides support for encapsulation with respect to inheritance as outlined by Snyder, and Smalltalk [Goldberg 83] is designed to provide total access to everything and nothing is hidden, as outlined by Gabriel [Gabriel 89]. Nierstrasz [Nierstrasz 89] indicated that the lack of encapsulation results in allowing an object to change the state of another object. This phenomena occurs in the language Self [Ungar 87] because no private slots are defined for objects.

2.7 Polymorphism and Binding

Polymorphism is a Latin word for "may shapes". In OOPLs, the term polymorphism means different forms [Cardelli 86] [Cox 86] [Gabriel 89] [Klint 86] [Meyer 88] [Nierstrasz 89] [Sethi 89] [Stroustrup 86,91] [Wegner 90] [Wolf 89]. Polymorphism is the multiplicity of form for a single method name. For example, consider the shape classes illustrated in Figure 2.4. The method **print** can be defined to print different shapes, and thus the same method name has different forms (implementations) in different classes. Polymorphism is a powerful tool for generalizing a single process among many kinds of objects. A polymorphic function is the one that can be applied to different objects. For example, the function Add can be applied to integer and real objects. Here, the operator "+" is called an *overloaded* operator.

Binding is the point in a program's life when the address of a method is determined. *Early binding* occurs at compile time. This requires that the address of a method be known at compile time. Early binding is one strong characteristic of strongly-

typed languages. In *late binding*, the address of the called method is given to the caller when the actual call takes place during run time.

Gabriel [Gabriel 89] addressed polymorphism in OOPLs. He stated that Polymorphism depends on binding. He also viewed binding as the process of determining which version of the invoked method is to be applied to an object. When polymorphic methods are not supported by the language, the compiler determines (binds) the methods at compile time. He called this approach *static* binding. In *dynamic* binding, methods to be applied to objects are determined at run time. While static binding is efficient and reliable, most OOPLs such as Smalltalk, Simula, Object Pascal, and C++ [Budd 91]. Dynamic binding makes polymorphism possible in OOPLs.

Nierstrasz [Nierstrasz 89] discussed the usage of polymorphism and its relation to inheritance. He stated that, with inheritance, polymorphic methods applied to objects of a subclass are also applicable to objects of the superclass(es). Moreover, polymorphism enhances reusability. It allows users to define generic methods that can be used with existing and newly-added objects.

Wolf [Wolf 89] also addressed the issue of polymorphism. He indicated that Polymorphism is a mechanism that determines which methods are to be selected to answer a message sent to an object. Methods are selected during execution based on the type of the object to which the message is sent. Klint [Klint 86] also stated that dynamic binding is one of the major reasons for the flexibility of OOPLs.

Polymorphism and binding also are discussed in detail by Meyer [Meyer 88]. He indicated that polymorphism is controlled by inheritance in Eiffel. Some languages allow users to have both late and early binding. For example, Eiffel allows both static and dynamic bindings, and C++ [Stroustrup 86,91] also offers static methods bound at compile time and virtual methods bound at run time.

A generic function is an ordinary function implemented as a set of methods that are selected based on the types of the supplied arguments when a method is invoked. The

concept of generic functions is used in Lisp-based OOPLs such as Flavors [Moon 85,86], CLOS [Bobrow 88], and CommonLoops [Bobrow 86] [Kempf 89]. These languages used generic functions to achieve polymorphism.

2.8 Definitions

In the literature, different authors have provided the object-oriented terminology different views from different perspectives, and hence terms are given different definitions. To avoid multi-definition terms and to avoid ambiguity, this section is devoted to provide definitions for terms used in this dissertation.

Instance Variable:

An instance variable is a variable declared in a class. It is associated with a value in an object of the class. Instance variables of a class are initialized either at class definition time or at creation time of an object.

Slot:

A slot is a variable whose value represents either a state or a behavior. The value of a slot can be accessed by messages. Unlike instance variables, a slot can be viewed as an instance variable name or a method name.

Method:

A method is an operation defined in a class to represent a specific behavior. Methods represent the behavior of objects of the class. A method has specification and implementation. Specifications of methods of a class represent the interface of its objects. Implementations are procedures (code segments) associated to specifications. Specifications are visible to users of the class; while implementations are hidden.

Class:

A class is a description (template) of a set of objects that share similar properties. A class consists of a set of instance variables and a set of methods. Instance variables have no values associated to them in the class (i.e. a class has no state).

Metaclass:

A metaclass is a class whose instances are classes.

Prototype:

A prototype is an object that can be viewed and used as a class to produce new objects. A prototype maintains state at any given time. The state of a prototype at any given time can be assigned to a newly created object. The behavior of the new object is similar to that of its prototype.

Object:

An object (instance) of a class is a collection of the values of the instance variables defined in that class. These values represent the state (private data) of the object at any given time of its life. They are accessed through the methods defined in that class. Objects of the same class have different states, and have the same behavior (methods).

Instantiation:

Instantiation is the process of creating new objects (instances) from classes. It is performed either by users or the programming language itself. Users instantiate objects from user-defined classes; while a programming language instantiates objects from system-defined classes that are not accessible to users of the language. All newly instantiated objects have the same initial state. Instantiation is used in class-based programming languages.

Cloning:

Cloning (copying) is the process of creating new objects from prototypes. A new object is a copy of its prototype. The initial state of the new object is the state of its prototype at cloning time. Cloning is used in prototype-based languages that use prototypes as classes.

Inheritance:

Inheritance is a mechanism for code sharing (methods and instance variables) among classes that have common behavior. When two classes have common behavior, they can be related to each other such that one class inherits (uses) code representing the common behavior from the other class. This relation is called inheritance. The inheriting class is called subclass (child); while the other class is called superclass (parent). Single inheritance relates a subclass to only one superclass; while multiple inheritance relates a subclass to two or more superclasses. The subclass inherits methods and/or instance variables from its superclass(es), and adds new methods and/or instance variables to specify its behavior. Inheritance is a static relation defined at program development time.

Inheritance Hierarchy:

Inheritance hierarchy is a hierarchical relation representing the inheritance relationship among related classes. Single inheritance forms a tree structure; while multiple inheritance forms a directed acyclic graph.

Delegation:

Delegation is a mechanism that captures the behavior of inheritance. It is used in prototype-based systems. An object delegates a message together with the responsibility of answering the message to another object. In addition to objects of the same prototype, objects of different prototypes can delegate messages among each other.

Message:

A message is a request sent to an object to invoke certain method. The receiving object may not answer the message directly. The response (if the object chooses to answer) is the result of executing the implementation associated with the invoked method. The set of messages that an object can answer are determined by its interface, and is called a *protocol*. All objects of a class or a prototype have the same protocol. Messages of identical names can be used with objects of different classes or prototypes.

Subtyping:

Subtyping is a relationship between classes such that objects of one class (subtypes) can be used in places where objects another class (supertype) are expected. Objects of the subtype have the characteristics and behavior of objects of the supertype, but not vice versa.

Polymorphism:

Polymorphism is a mechanism for applying a method on different objects of different classes or prototypes. It generalizes a method among different types of objects. It also determines which method to be used to answer a message sent to an object. Polymorphism is tied to binding.

CHAPTER III

INHERITANCE IN OBJECT-ORIENTED PROGRAMMING

LANGUAGE: A TAXONOMY AND SURVEY

3.1 Introduction

Encapsulation, accessing the instance variables, and the visibility of inheritance are important issues in OOP. Various OOPLs apply different restrictions on these issues. Some languages, such as Smalltalk-80 [Goldberg 83,89] and Flavors [Moon 86], include their instance variables in the class external interface definition. Other languages, including CommonObjects [Snyder 86a] and C++ [Stroustrup 86,91], hide their instance variables and allow the inheriting classes to access them only via operations provided in the external interface definition. More recent languages, such as Trellis/Owl [Schaffer 86], apply more restrictions by providing the ability to define private operations dedicated for inheriting classes and not by inclusion in the external interface.

The notion of subtyping may impact the inheritance issue based on the subtyping/inheritance relationship. Some languages view subtyping between classes through their inheritance relationship [Stroustrup 86] [Goldberg 83,89]. That is, subtyping is based on the implementation hierarchy of the class. Other languages view subtyping explicitly based on the class behavior rather than structure [Schaffer 86] [Snyder 86b]. The effect of these perspectives on the inheritance mechanisms in the selected languages are discussed later in this chapter.

In this chapter we address three major issues in a number of widely-used OOPLs:

Inheritance mechanisms, access techniques, and the notion of subtyping and its relation to inheritance. The OOPLs considered in this chapter are Trellis/Owl, C++, Eiffel, CommonObjects, CLOS, Flavors, Smalltalk-80, and Simula. These languages are compared in terms of these issues. The strength and weakness of their support for Snyder's criteria are also considered and discussed. Additionally, a binary tree taxonomy for OOPLs [Al-Haddad 91b] is presented in this chapter.

The rest of this chapter is organized as follows: Section 2 is a survey of the inheritance mechanisms and the access techniques in the selected languages. Section 3 describes how OOPLs relate inheritance to subtyping. Section 4 provides a binary tree taxonomy model of OOPLs based on the major characteristics of the inheritance mechanisms. And in Section 5, the selected languages are analyzed based on Snyder's criteria, and summary tables of their features are provided. Finally, Section 6 is the summary.

3.2 A Survey of Inheritance Mechanisms and Access Techniques

In OOPL environments, systems are composed of objects and objects are instances of classes. When several classes share common abstract properties, it is inconvenient to duplicate the code of the shared properties in several classes. Therefore, inheritance is used as a mechanism for code sharing between classes to construct new software components from existing ones. The different forms of inheritance are discussed in the previous chapter.

The *access technique* for instance variables determines how inheriting classes access the instance variables of the super and ancestor classes [Snyder 86a] [Strom 86]. Various OOPLs have different semantics for this issue. This issue may impact the encapsulation of inherited information when inheriting classes have full access to the instance variables of the superclass. In this case, changing the superclass implementation

may become unsafe for its descendants. The class contents must be hidden and accessed only through defined methods to maintain information hiding and provide a flexible program development.

The rest of this section is organized as follows. Each language subsection has two parts (A) and (B) associated with it. The (A) part discusses the inheritance mechanism of the corresponding language. Here, inheritance is interpreted to have different meanings in different languages. For instance, inheritance in Smalltalk-80 is a technique to construct complex classes from simple ones; whereas in CommonObjects, inheritance is utilized to define objects by hiding the internal representations and exposing the defined operations via the external interfaces of a class. Moreover, in part (A), features provided by the selected languages to support inheritance are also discussed using examples.

Part (B) discusses the access technique of the corresponding language. Here, we examine how the language allows inheriting classes to access the instance variables of ancestor classes. Accessing the instance variables has a major impact on information hiding. Restricted access provides stronger encapsulation in which the inheritance hierarchy is more flexible for changes. Allowing descendant classes to access the instance variables of a superclass limits the designer's freedom to change the class implementation without affecting its inheriting classes [Snyder 86a]. OOPLs provide varying degrees of access. Some languages allow full access to the instance variables of a class, while other languages impose strong restrictions on the external interface provided for inheriting classes.

3.2.1 Trellis/Owl

(A) Inheritance Mechanism. Trellis/Owl is a statically typed OOPL. It combines a multiple inheritance type hierarchy with static type checking [Schaffer 86]. Trellis/Owl provides the conventional syntax of programming languages and uses the standard

procedure call notation for method invocation. In Figures 3.1 and 3.2, definitions of STACK and DEQUE (double ended queue) types (classes) are presented to show their inheritance relationships. Figure 3.1 describes the type STACK with one private operation (method) and two public operations. Figure 3.2 describes the type DEQUE as a subtype of the type STACK. It inherits the public operations of STACK and defines new operations.

```

type_module STACK;
operation is_empty(me)
! return True if stack is empty, False otherwise
...
operation push (me, elem : element_type)
public
! insert element elem into stack if not overflow
...
operation pop (me)
public
! return the top of stack if not underflow
...
end type_module

```

Figure 3.1: Definition of the type STACK

Trellis/Owl subclassing is based on behavior rather than specification hierarchy. That is, a subtype must behave like its supertyp(es), but it may have different implementation. For example, the type DEQUE behaves like the type STACK and provides more operations. Since the subtype specializes the supertype, types cannot hide the use of inheritance from the inheriting types.

Trellis/Owl provides two kinds of operations, instance and class operations. Instance operations such as push and pop operations in Figure 3.2 are applied to individual objects of the type. Class operations such as the creation operations are applicable to the types themselves rather than their objects. Trellis/Owl operations can be declared as public or private operations. Public operations of a type are available to its clients (inheriting classes (users) rather than subtypes); while private operations are used

```

type_module DEQUE;
! deque must define its own operations for boundary
! check since they are not inherited form type STACK.

operation push_top (me, el: element_type)
public
is begin STACK'push (me,el); end;

operation pop_top (me)
public
is begin STACK'pop (me,el); end;

operation push_end (me, el :element_type)
public
! Inject element el into the back end of Deque
...
operation pop_end (me)
public
! Eject element el from the back end of Deque
...
operation create (mytype)
return (mytype)
! Return the newly created Deque ...
! other methods ...

end type_module;

```

Figure 3.2: DEQUE is a subtype of the type STACK

within the defining type and its subtypes, and are not available to clients. A third visibility is called subtype-visibility. Subtype-visible operations are neither restricted as private nor general as public operations. They can be inherited and redefined, but not visible outside the defining type and its subtypes.

The external interface of a type is restricted to include only public and subtype-visible operations provided for inheriting classes. The external interface of the type STACK in Figure 3.1 includes the public operations push and pop. Instance variables and operation implementations of a type are hidden in the type and accessed through operations specified in its external interface. Different types may have different operations with the same name. The keywords **me** and **mytype** are used to distinguish instance operation from class operations. They are called controlling objects; **me** indicates instance operations while **mytype** indicates class operations as illustrated in Figure 3.2.

It is the programmer's responsibility to specify the interface and the implementation of operations as well as solving the inheritance ambiguities. Ambiguity can arise when a type has two or more supertyp(es). When supertypes introduce different definitions for an inherited operation, the programmer needs to specify explicitly the desired definition, or override it by a new definition in the inheriting types.

(B) Access Technique. The instance variables of a class are accessed through the class's public and subtype-visible operations provided in its external interface. When the type's definition of a redefined inherited operation is required, the subtype can directly access the superclass' version of an operation. For instance, if **P** is the supertype and **OP** is the modified operation in the subtype, the notation **P'OP (me,...)** gives direct access to **OP** in the supertype **P**. In Figure 3.2, type **DEQUE** overrides the operations **push** and **pop** inherited from the type **STACK** and directly accesses the superclass's definition using the above notation.

Type **DEQUE** in Figure 3.2 accesses the instance variable **elem** through the definition of the public operations **push** and **pop** defined in the type **STACK** in Figure 3.1. In case of direct access, when a public operation references other operations in the same type, then the public operation still references them after being inherited by other types. A type may refer directly to operations of an ancestor type without being inherited by all intervening ancestors. However, a descendant type must know about its ancestor types.

Subtype-visible operations are inherited and can be redefined, but they are not accessible outside the defining type and its subtypes. Thus, users may define operations that are directly accessible to the subtype and are not exposed in the type's external interface.

3.2.2 C++

(A) Inheritance Mechanism. C++ [Stroustrup 91] provides OOP style with multiple

hierarchical inheritance by means of class declarations. Class declarations divide classes into sections in order to limit the visibility of the class contents to other classes. In C++, changing the implementation of a class does not impact its descendant classes. A class may include public, private, and protected sections. Each section contains variables and methods of the same degree of visibility. An inheriting (derived) class inherits all of its superclass(es) properties and may add new instance variables and methods.

Figures 3.3 and 3.4 provide sample definitions. The classes DATE and BIRTH_DAY are adopted and modified from [Stroustrup 86]. In Figure 3.3, class DATE defines three public instance variables and two public methods. Instance variables and methods are called class members. Class BIRTH_DAY in Figure 3.4 is publicly derived from the class DATE. It inherits methods and instance variables from the class DATE, and defines new methods and instance variables. The main program in Figure 3.4 creates two instances of the class DATE: Today and Christmas.

```

class DATE {
    public:                                // public section
        int month, day, year;              // public variables
        DATE (int,int,int)                 // Constructor
        ~DATE ();                           // Destructor
        void next ();                       // next day
        void print ();                      // print date
};

void DATE::next () {
    if ( ++day > 28 )
        { /* print next month schedule */ };
};

void DATE::print () {
    cout << month << "/" << day << "/" << year <<;
};

```

Figure 3.3: Definition of the class DATE

Private instance variables, such as name and age of the class BIRTH_DAY, are used within the scope of the class. Public instance variables have the opposite situation.

```

class BIRTH_DAY : public DATE {
    char* name;           // private variable
    int age;              // private variable
public:                  // public variables
    void compute_age (int,int,int); //newly defined method
    BIRTH_DAY (char,int,int,int); //constructor
    ~BIRTH_DAY();        // destructor
};

int BIRTH_DAY::compute_age(month, day, year);
{
    DATE::print();      // print today's date
    /* subtract birthday form today's date and return age*/
    ...
};

main (int month, day, year)
{
    DATE today (03, 30, 1990); // instance today
    DATE Christmas (12, 05, 1989); // instance Christmas
    today.print();           // print today's date
    Christmas.next();        // schedule for January
};

```

Figure 3.4: Class BIRTH_DAY is a public subclass of the class DATE

They are made available to other classes. For instance, the variables month, day, and year of the class DATE can be inherited by other classes. Different classes may have methods with the same names. The scope resolution operator "::" helps users avoid the naming conflicts of methods by explicitly specifying the invoked method. For example **DATE::print()** in Figure 3.4 indicates the print method in class DATE not any other class. In addition, the type and number of a method's arguments can specify the invoked method [Pinson 88b].

C++ provides two approaches for inheritance (class derivation): public and private derivation. In public derivation, the derived class inherits both public and protected members of the base class(es), and retains these visibilities. Therefore, the public and protected members of the base class are also public and protected in the derived class(es). For example, class BIRTH_DAY above inherits all public members of the class DATE in Figure 3.3. Inherited members remain public in class BIRTH_DAY and are available

to its descendant classes.

In private derivation, the derived class inherits all public and protected members of the base class(es), but inherited members are private in the derived class and are not available to other classes. For example, If the class BIRTH_DAY is privately derived from the class DATE, then the public members of the class DATE become private members in the class BIRTH_DAY, and are not available to descendants of the class BIRTH_DAY.

The external interfaces provided for instances and inheriting classes include methods defined in the class. An instance of an inheriting class includes only variables used by methods inherited from the superclass, not all the superclass's instance variables [Gorlen 87]. Every object of a derived class has its own copy of the superclass's private variables. For example, each instance of the class BIRTH_DAY has its own copy of the instance variables name and age. These variables represent the state of an instance.

(B) Access Technique. Inheriting classes access the public and protected instance variables of the base class through inherited methods. In public derivation, public variables are accessible to all descendant classes; while protected variables are accessible to only immediate inheriting classes. In private derivation, all public and protected members of the superclass are accessible only to the immediate inheriting classes and are not accessible to other descendant classes.

Using the notation `Class_name::Method_name`, a class can directly access a method of an ancestor class if that method is being inherited (passed down) through all intervening classes including the base class. Class BIRTH_DAY accesses the public instance variables defined in the class DATE through the inherited method `DATE::print`.

C++ provides self-reference by means of pointers to objects. The reserved word **this** allows a class to refer to itself. Redefining a method by descendant classes does not impact this invocation, because the dynamic binding mechanism insures invoking the

correct method. A superclass may authorize a descendant class to access its private variables. One way to allow such access is by declaring the descendant class as a *friend* class of its superclass. The other way is defining *protected* variables in the superclass. Protected variables are hidden from the other classes, but only accessible to immediate derived classes.

3.2.3 Eiffel

(A) Inheritance Mechanism. Eiffel [Meyer 88b] is a statically-typed OOPL. It provides multiple inheritance with strong static type checking and dynamic binding. Eiffel classes provide private and public information to maintain information hiding. Classes also contain instance variables and methods in which implementations of methods are within the class definition. Pre-defined methods (class methods), such as Create, Forget, Clone, and Result are not inherited, and are applicable to all classes. Different classes may have same-named methods, and a class may inherit the same method from different classes.

The client classes are classes that include `ClassType` declaration of the form `var_name:ClassType` where `var_name` is a variable name and `ClassType` is a defined class. Inheriting classes are classes that inherit explicitly from other classes through the `inherit` declaration. Eiffel's inheritance mechanism is illustrated in Figures 3.5 and 3.6.

Figure 3.5 defines the class `STACK` of type integer. `STACK` exports three public methods and one public instance variable. The instance variables `stack_size` and `stack_pointer` are private to the class `STACK`. Figure 3.6 defines the class `DEQUE` that inherits from the class `STACK` and adds new methods. Inheriting classes may rename and redefine inherited methods and instance variables using the keywords `rename` and `redefine`. Class `DEQUE` renames the inherited methods `push`, `pop`, and the instance variable `stack_pointer`. It also retains the method `is_empty`.

```

class Stack[INTEGER] export
  is_empty, push, pop, stack_pointer
feature
  implementation : ARRAY[INTEGER];
  stack_size    : INTEGER;
  stack_pointer : INTEGER;
  ...
  is_empty : BOOLEAN is    -- is stack empty?
  do ... -- return TRUE if stack is empty
  end; -- is_empty

  push(X : INTEGER) is    -- insert element X
  do ... -- add X if not full stack
  end; -- push

  pop is -- pop the top element
  do ... -- pop top if non-empty stack
  end; -- pop
  ...
end; -- class STACK

```

Figure 3.5: Definition of the class STACK

```

class DEQUE[INTEGER] export
  is_empty, push_top, pop_top, push_end, pop_end
inherit STACK[INTEGER]
  rename   push as push_top, pop as pop_top
           stack_pointer as front_pointer
  redefine is_empty
feature
  implementation : ARRAY[INTEGER];
  rear_pointer  : INTEGER;
  ...
  is_empty : BOOLEAN is    -- is stack empty?
  do ... -- different implementation
  end; -- is_empty

  push_end(x : INTEGER) is -- inject element x
  do ... -- inject x if not full stack
  end; -- push_end

  pop_end is -- eject element x
  do ... -- eject element x if non-empty stack
  end; -- pop_end
  ...
end; -- class DEQUE

```

Figure 3.6: Class DEQUE inherits from the class STACK

Eiffel introduces the notion of repeated inheritance. This notion allows classes to inherit more than once from ancestor class(es). The class DEQUE may be written as follows:

```
class DEQUE [INTEGER] export ...
    inherit STACK [INTEGER] rename ... redefine ...
    inherit STACK [INTEGER] rename ... redefine ...
    ...
```

Repeated inheritance may lead to replicated methods if a method has been renamed along the inheritance path in which its code needs to be duplicated in the inheriting class [Meyer 88]. For example, suppose that the class SD inherits from the classes STACK and DEQUE. Since methods push and pop are renamed in the class DEQUE. Class SD may include methods push and pop from the class STACK, and methods push_top and pop_top from the class DEQUE where push_top and pop_top are duplication of push and pop respectively. This might lead to ambiguity.

(B) Access Technique. Eiffel's **export** clause separates the class's private information from the public information offered outside the class definition. It applies information hiding to clients (classes that use **var_name:ClassType** declaration) of a class, and restricts the visibility of the class contents to other classes. For descendant classes, information hiding is not applicable since descendants may depend directly on the implementations of the superclass(es) [Meyer 88].

Clients are restricted to access only public (exported) methods and instance variables. A class may export all of its private and public information to its descendants granting them full access to its contents. A client may directly access (read) an exported variable, but cannot modify its value. However, modification of such a variable can be done through the definition of an exported method that uses the variable. The class DEQUE has full access to the contents of the class STACK through their inheritance relationship.

A class may export methods and instance variables inherited from other classes

in which these methods and variables were private. A class may export its methods and instance variables to certain descendant classes by specifying the destination class(es) in the **export** clause. For instance, one may declare the class STACK in Figure 3.5 as follows:

```
class STACK[INTEGER] export
    is_empty {DEQUE}, push {DEQUE}, pop {DEQUE}
    . . .
```

Class DEQUE is the destination class for the exported methods.

3.2.4 CommonObjects

(A) Inheritance Mechanism. CommonObjects [Snyder 86a] is an extension of CommonLisp. Its inheritance mechanism emphasizes on providing strong support for encapsulation. Similar to other OOPLs, CommonObjects is intended to support a minimal external interface definition and ease the development of software systems. The inheritance mechanism used in CommonObjects can be explained using the definitions given in Figures 3.7 and 3.8.

Figure 3.7 illustrates the definition of the class DATE, which contains four instance variables and defines four methods. In Figure 3.8, the class BIRTH_DAY is defined as a subclass of the class DATE. It inherits method today and adds the instance variable age.

CommonObjects' classes inherit methods and instance variables from each other in hierarchical order. A class inherits all or part of the superclass's methods to be included in its external interface. For instance, the option **methods** in Figure 3.8 allows the class BIRTH_DAY to inherit only necessary methods from the class DATE (method today). Different classes may define same-named methods. Classes encapsulate their instance variables and may have same-name instance variables. Renaming instance variables and changing method implementations of a class do not affect its descendants.

```

(define_type DATE
  ;; Date is represented by month, day, and year variables.
  ;; Variable last_day=1 indicates the last day of the month.

  (:var month      (:type integer) (:init 0))
      :gettable :initable
  (:var day        (:type integer) (:init 0))
      :gettable :initable
  (:var year       (:type integer) (:init 0))
      :gettable :initable
  (:var last_day   (:type integer) (:init 0)))

(define-method(DATE :month_end?) ()
  ;; return true if today is the end of the month
  (= (last_day 1))

(define-method(DATE :year_end?) ()
  ;; return true if current month is December
  (= (month 12))

(define-method(DATE :next_month) ()
  ;; return next month if today is the end of the month.
  (unless (call-method (:month_end?))
    (incm month)
    month))

(define_method(DATE :today) ()
  ;; return today's date
  ...

```

Figure 3.7: Definition of the class DATE

```

(define-type BIRTH_DAY
  (:inherit-form DATE
    (:methods :month_end :next_month)))
  ;; Birth_day is a subclass of class Date, it inherits
  ;; methods today and defines a new method and instance
  ;; variable.

  (:var b_date (type integer) (:init 0)
    :gettable :initable)

  (define-method (BIRTH_DAY :age) ()
    ;; print today's date and return age
    (call-method (DATE :today) ())
    ;; Subtract b_date form today's date and return age

```

Figure 3.8: Class BIRTH_DAY partially inherits from the class DATE

The use of inheritance in a class is allowed to be hidden from the inheriting classes. In Figure 3.8, the **methods** option does not show the status of method today in the class DATE whether defines or inherited. CommonObjects provides external interfaces for both objects and inheriting classes. Inheriting classes cannot directly inherit methods from non-superclasses. Rather, methods must be passed down through intervening classes. For instance, if a subclass of the class BIRTH_DAY wants to inherit method year_end from the class DATE, class BIRTH_DAY must inherit that method and pass it down to its inheriting classes.

CommonObjects provides hierarchical multiple inheritance. When a class inherits from multiple superclasses, it creates a set of instance variables for each superclass. It is an error to inherit same-named methods from different superclasses. When a class inherits same-named instance variables from different superclasses, its instances contain two sets of inherited variables, one from each superclass. For example, if a class (say Daily_Schedule) inherits from both classes DATE and BIRTH_DAY above, its instances include two sets of the instance variables month, day, and year, one from each superclass.

(B) Access Technique. Accessing inherited variables by methods does not compromise encapsulation. Instance variables are accessed and used by methods defined in the class. Inherited methods from a superclass restrict the inheriting class to only access the instance variables used by these methods. Class BIRTH_DAY accesses the instance variables month, day, and year in the class Date through the inherited method today in which encapsulation is preserved.

CommonObjects provides the **call_method** construct for direct invocation of methods defined in ancestor classes but are not inherited by descendant classes. A class can call its own methods by specifying the method's name. A class can access the superclass's un-inherited methods using the superclass and method names as arguments. For example, the statement


```
call_method (DATE :today())
```

in Figure 3.8 includes the arguments DATE and today.

In the case of multiple inheritance, when two superclasses of a class define the same instance variable, an instance of the class contains two instances of that variable in which they are accessed through inherited methods from each superclass. On the other hand, when superclasses of a class share a common ancestor, an instance of the class contains two sets of the instance variables defined in the common ancestor, one from each superclass. These variables can be accessed either through inherited methods or through the direct `Call_Method` construct. An inheriting class needs not know about its superclass's use of inheritance [Snyder 86a].

3.2.5 CLOS (CommonLisp Object System)

(A) Inheritance Mechanism. CLOS [Keene 89] [Bobrow 88] is an object-oriented extension of CommonLisp. It provides a set of tools to help programmers construct highly modular and extensible programs. As inheritance is concerned, CLOS supports multiple inheritance of the class structure and behavior. Every object is an instance of a class and has the same structure, behavior, and type as of its superclass. Each class is an instance of another class called metaclass.

CLOS provides behavior inheritance by means of associating methods with classes. Inheritance of methods is based on the method combination technique, which is a set of calls to applicable methods. If a method is applicable to an instance of a class, it is also applicable to instances of its subclasses. A class may add new methods or override inherited methods. Implementation of methods are disjoint from the class body [Bobrow 88]. Examples are given in Figures 3.9 and 3.10.

Figure 3.9 defines the class EMPLOYEE with three local variables (slots), one shared variable, and two methods. Class EMPLOYEE does not inherit from other classes. Figure 3.10 defines the class MANAGER as a subclass of the class EMPLOYEE. It adds

```

(defclass EMPLOYEE ()
  ((name   :type string   :allocation :instance)
   (age    :type integer  :allocation :instance)
   (hours  :type integer  :allocation :instance)
   (salary :type integer  :allocation :instance)
   (rank   :type string   :allocation :class))

  (defmethod print_name (1 Employee)
    ... )
  (defmethod work_load (1 Employee)
    ... )

```

Figure 3.9: Definition of the class EMPLOYEE

```

(defclass MANAGER (EMPLOYEE)
  (rank type string   :allocation :class)

  (defmethod history (1 Manager)
    ... )
  (defmethod work_schedule (1 Manager)
    ...))

```

Figure 3.10: Class MANAGER is a subclass of the class EMPLOYEE

new instance variables and defines new methods.

CLOS provides two types of instance variables: local and shared. Local instance variables save the state information of a particular instance of the defining class. They are created when creating an instance of a class. For example, the instance variables name, age, hours, and salary of class Employee are local. Shared instance variables save the shared state that is used by all instances of the class. The instance variable rank in Figure 3.9 is shared by all instances of the class EMPLOYEE.

A class specializes its superclasses by inheriting their structure (variables) and behavior (methods). Class MANAGER in Figure 3.10 inherits all methods and instance variables of the class EMPLOYEE and adds a new instance variable and methods. It provides the same structure and behavior of the class EMPLOYEE. The value of the instance variable rank in the class MANAGER is different from that in EMPLOYEE.

CLOS uses the generic function approach to define and invoke methods. A generic function contains a set of methods with relevant information. Its implementation is distributed among its methods, and the appropriate methods are selected based on the function's arguments. Several methods may be related to the same generic function. When two or more methods are defined for an operation, the class precedence list determines which method(s) to be invoked [Gabriel 89].

A class precedence list is a list of defined classes in the program. Classes are ordered based on the given organization of the program. Each class has its own precedence list containing the class itself and its superclasses. A class has precedence over its superclass. In Figure 3.11, the classes X and Y precede the class A, and class B precedes the classes X and Y. Note that the class Y precedes the class X because it is defined after X has been defined. Precedence lists preserve the order of method invocations, and avoid naming conflicts among the applicable methods of the generic function [Keene 89].

```
(defclass A() ())
(defclass X(A) ())
(defclass Y(A) ())
(defclass B(X Y) ())
```

Figure 3.11: Definitions of classes

(B) Access Technique. CLOS provides structured inheritance in the sense that all instance variables defined by a class and its superclasses are accessible in an instance of that class. Instance variables are accessed through methods defined in their classes and included in the generic function definition. Specifiers determine the scope of instance variables. For instance, the specifier `:instance` in Figure 3.9 indicates that each instance of the class `EMPLOYEE` has its own copy of the instance variable; while the specifier `:class` indicates that the instance variable `rank` is shared by all instances of the class.

Methods of a class are accessible to instances of its subclasses. Likewise, methods of a metaclass are accessible to instances of its classes. Class `MANAGER` accesses all of the instance variables used by methods `print_name` and `work_load` defined in the class `EMPLOYEE`.

When modifying the value of an inherited shared instance variable in a subclass, all instances of the subclass access the modified variable. For example, the class `MANAGER` in Figure 3.10 modifies the instance variable `rank` inherited from the class `EMPLOYEE`. All instances of the class `MANAGER` share the new value of the variable `rank`.

The combination of applicable methods of a generic function is called the **Effective method** (implementation) of the generic function. When calling a method of a generic function, the generic function determines and invokes the appropriate method based on the passed arguments. If the method is not applicable, the generic function returns an error message [Steele 84].

In multiple inheritance, when superclasses provide same-named features (methods and slots), the class precedence list is used to solve such conflicts. The precedence list determines which class has precedence over others, and which method to be invoked (the most specific one) when more than one method are applicable. A method is more specific than another if its first argument is a subclass of the first argument in the second method [Gabriel 89]. For example, method `history` of the class `MANAGER` is more specific than method `work_load` of the class `EMPLOYEE` in Figure 3.10 since the argument "`MANAGER`" is a subclass of the argument "`EMPLOYEE`".

3.2.6 Flavors

(A) Inheritance Mechanism. `Flavors` [Moon 86] is a non-hierarchical lisp-based OOP that supports multiple inheritance. The notion of flavor is analogous to the notion

of class. It is an abstraction representing one type of objects. A flavor defines a set of instance variables and methods, and specified inherited flavors. Figure 3.12 illustrates the flavor 3D_MOVING_OBJECT that defines three instance variables and one method. These examples are adopted from [Moon 85].

```

(defflavor 3D_MOVING_OBJECT
  (x_velocity, y_velocity, z_velocity
   x_motion, y_motion, z_motion) ()
  :initable_instance_variables)

(defmethod (speed 3D_MOVING_OBJECT) ()
  (sqrt (+ (expt x_velocity 2)
           (expt y_velocity 2)
           (expt z_velocity 2))))

(defmethod (location 3D_MOVING_OBJECT) ()
  ;; locate the object at time t.

```

Figure 3.12: Definition of the flavor 3D_MOVING_OBJECT

An inheriting flavor specializes its parent (super) flavors by maintaining the same characteristics. It inherits all of its instance variables and methods. It may also redefine inappropriate inherited methods, and define new methods and instance variables. A space ship and comet are specialized 3D moving objects. Figure 3.13 illustrates the flavors SPACE_SHIP and COMET that inherit from the flavor 3D_MOVING_OBJECT above.

Inheritance is defined by means of mixing flavors. When different flavors have common characteristics, users can define them in one flavor called **component flavor**. Other flavors inherit the component flavor instead of duplicating the common characteristics in each of them. The flavor 3D_MOVING_OBJECT in Figure 3.12 represents a common characteristics of all 3D moving objects. Hence, it is a component flavor in the flavors SPACE_SHIP and COMET in Figure 3.13. When different flavors define same-named instance variables, the inheriting flavor combines them into one instance variable, and instances of the inheriting flavors contain one copy of the new

```

(defflavor SPACE_SHIP
  (crew_list name destination) (3D_Moving_Object)
  :initable_instance_variables)

(defmethod (fuel SPACE_SHIP) () ...)
(defmethod (altitude SPACE_SHIP) () ...)

(defflavor COMET
  (percent_iron estimated_mass size)
  (3D_MOVING_OBJECT)
  :initable_instance_variables)

(defmethod (produced_energy COMET) () ...)
(defmethod (friction_force COMET) () ...)

```

Figure 3.13: The flavors `SPACE_SHIP` and `COMET` inherit from the flavor `3D_MOVING_OBJECT`

combined instance variable.

Like other lisp-based OOPLs, Flavors uses the generic function approach to invoke methods. A generic function is a set of methods defined on different flavors. A method combination type combines a list of methods into one method called a **combined method**. A flavor applies ordering on its component flavors to specify the structure and behavior of its inheriting flavors. Ordering component flavors determines the order of the inherited methods from these component flavors, and which methods to be chosen from the method combination type. The combined method calls methods of the generic function in the appropriate order based on the ordering applied to the component flavors [Cannon 82].

Flavors exposes the instance variables to the inheriting flavors in which instances of the inheriting flavors include all instance variables of the parent flavor(s). Therefore, Flavors does not hide the use of inheritance since the instance variables and methods of the component flavors are visible to the inheriting flavors. Mixing flavors is defining a flavor in terms of other flavors. It does not affect the order of the component flavors. However, it is an error to violate the ordering constraints when mixing several flavors [Moon 85].

For multiple inheritance, Flavors translates the inheritance graph into a linear chain

of flavors that preserve the order of component flavors. Ancestor flavors appear before the inheriting flavors, and the non-ordered component flavors can appear anywhere on the chain. [Snyder 86a].

(B) Access Technique. The **Defflavor** construct defines a flavor with instance variables, inherited component flavors, and relevant options for accessing the instance variables. These options can be used to initialize the inherited variables, and to customize a flavor's behavior. The instance variables of a flavor are accessed through methods of that flavor as implicit lexical variables. When a flavor inherits other flavors, their instance variables are accessed through the generic function methods defined on the inheriting flavors. When adding or renaming a variable in a flavor, the changes are propagated to all inheriting flavors from the updated one. Thus, users need not update the inheriting flavors.

Flavors exposes the instance variables outside the flavor definition by using them in the method definition (external interface), which grants inheriting flavors full access to the instance variables of the parent flavors. For example, the flavors `SPACE_SHIP` and `COMET` in Figure 3.13 have full access to the instance variables of the flavor `3D_MOVING_OBJECT`. The visibility of the instance variables compromises encapsulation and limits the ability to decompose the program into modules [Moon 85].

3.2.7 Smalltalk-80

(A) Inheritance Mechanism. Smalltalk-80 [Goldberg 83,89] is a dynamically-typed OOP language that provides single hierarchical inheritance. Every class has a superclass, except the system object class (root class) that describes the common properties of all objects in the system. Examples provided in Figures 3.14 and 3.15 are adopted from [Goldberg 83,89].

```

Class name          PERSONAL_FINANCES
Superclass         Object
Instance variables  income, expenses

Class methods
  initialize: amount
    ^super new initialBalance: amount "initialize balance"
  new
    ^super new initialBalance: 0 "create new instance"

Instance methods
  receive: amount
    income <-- income + amount "update income"
  spend: amount
    expenses <-- expenses + amount "update expenses"
  availableCash
    ^income - expenses "return available cash"
  totalIncome
    ^income "return total income"
  totalExpenses
    ^expenses "return total expenses"
  initialBalance: amount
    income <-- amount "initialize the account"
    expenses <-- 0

```

Figure 3.14: Definition of class PERSONAL_FINANCES

```

Class name          DEDUCTIBLES
Superclass         PERSONAL_FINANCES
Instance variables  deductibleExpenses

Class methods
  initialize: amount
    ^(super new initialBalance: amount)
      "initialize balance"
  zeroDeduction "no deductions"
  new
    ^super new zeroDeduction "create new instance"

Instance methods
  amountDeductible: amount
    self spend: amount deducting: amount "deduct amount"
  deductions: amount deductibleAmount
    super spend: amount "update debt"
    deductibleExpense <-- deductibleExpenses + deductibleAmount
  totalDeduction
    ^deductibleExpenses "return total deducted money"
  zeroDeduction
    ^deductibleExpenses <-- 0 "initialize deductibleExpenses"

```

Figure 3.15: Class DEDUCTIBLES inherits from the class PERSONAL_FINANCES

A class defines instance variables to represent private data, and class variables to represent shared data. Class methods are the implementation details of messages sent to the class name; while instance methods are the implementation details of messages sent to an instance of the class. For example, method `initialize:` in Figure 3.14 is an implementation of the initialization message sent to the class `PERSONAL_FINANCES` in order to initialize an account (an instance of class `PersonalFinances`) by certain amount of money. Likewise, the method `receive:` is the implementation of a message sent to an account to deposit certain amount of money.

A subclass inherits the instance variables and methods of its superclass, and may add new ones. In Figure 3.15, the class `DEDUCTIBLES` inherits all methods of its superclass `PERSONAL_FINANCES`. An instance of a subclass responds to all messages of the superclass' instances in addition to those newly defined ones. For example, class `DEDUCTIBLES` can answer all messages sent to the class `PERSONAL_FINANCES`. Upon sending a message to a class, the search for the invoked method starts at the receiving class and continues up the superclasses until reaching the invoked method or the root class, which returns an appropriate message to the sender.

Smalltalk-80 treats classes as objects. They are instances of other classes called `Metaclasses`. The external interface (protocol) of a class, provided for instances of the class, is a list of messages understood by its instances. This interface is limited to include only the instance methods of the defining class. For example, the external interface of the class `PERSONAL_FINANCES` in Figure 3.14 includes the methods `receive:`, `spend:`, `availableCash`, `totalIncome`, `totalExpenses:`, and `initialBalance:`.

The class external interface provided for inheriting classes includes all methods defined in the class. Instance variables of a class are not encapsulated. They are fully exposed outside the class definition. Therefore, an instance of an inheriting class contains all of the instance variables defined in the superclass [Meulen 87]. For example, an instance of the class `DEDUCTIBLES` contains the variables `income` and `expenses` from

the class `PERSONAL_FINANCE` in addition to the instance variable `deductibleExpenses`. Also, classes expose the use of inheritance to other classes.

Borning and Ingalls [Borning 82] provided an extended version of Smalltalk. They described and implemented multiple inheritance in Smalltalk-80. They added a new feature called **Compound selector** for method invocation. It allows programmers to specify the method's name and the corresponding superclass when invoking a method. Multiple inheritance uses the dynamic binding approach for methods on the chain of inheritance instead of copying the inherited methods into each subclass.

When two or more superclasses define the same instance variable, an instance of the class contains only one instance of that variable even though it is inherited through different paths. Besides, it is an error to have two or more different inherited instance variables with the same name.

(B) Access Technique. The class description protocol in Smalltalk-80 includes information about the instance and class variables, as well as about the instance and class methods. Including all these information in the interface grants inheriting class full access to the instance variables of the superclass. For example, the class `DEDUCTIBLES` in Figure 3.15 accesses all instance variables defined in the class `PERSONAL_FINANCES`. An instance has its own list of the class instance variables. These variables are accessible to the instance methods defined in the class.

Full access to the instance variables restricts the inheriting class to use variable's names similar to the inherited ones. Additionally, removing or renaming an instance variable may affect inheriting classes since dependency exists between them. Thus, changing the implementation details of the superclass may require changing the inheriting classes.

3.2.8 Simula

(A) Inheritance Mechanism. Simula [Kirkerud 89], the grandfather of all OOPLs, is a statically-typed OOPL. A class is a construct used to declare a template for a set of objects that contain the same set of attributes (data and operators). Simula provides single hierarchical inheritance with strong type checking. The subclass inherits all attributes of its superclass, and may add new attributes. Objects of the subclass are a subset of objects of the superclass. For illustration consider the classes PLACE and TOWN adopted from [Kirkerud 89] and shown in Figures 3.16 and 3.17 respectively.

```

class PLACE;
begin
  real longitude, latitude;
  procedure read;
    begin
      latitude := prompt_for_real ("degrees north?");
      longitude := prompt_for_real ("degrees east?");
    end of Place' read;

  procedure write;
    begin
      outfix(latitude,2,5); outtext("degrees north,");
      outfix(longitude,2,5); outtext("degrees east.");
    end of Place' write;
end of PLACE

```

Figure 3.16: Definition of the class PLACE

```

PLACE class TOWN;
begin
  integer num_of_inhabitants;
  <Declaration of other Town attributes>;
end of TOWN;

```

Figure 3.17: Class TOWN is a subclass of the class PLACE

In Simula, objects are created using an object generator expression of the form "new Class_name". A declaration of the class Class_name introduces a new type called a **reference type**, and is denoted by **ref(Class_name)**. Users can declare variables of reference types that are called **reference variables**. For example,

```
ref(PLACE) country_side, tallest_mountain;
ref(PLACE) array mountains (1 : number_of_mountains);
```

are declarations of reference variables. A reference variable either contains no reference "none", or contains the address of an object of the class Class_name.

Assigning a new value to a reference variable is called a **reference assignment**. A reference assignment is the creation process of new objects using the expression "new Class_name". For example,

```
country_side :- new PLACE;
tallest_mountain :- mountains(3);
```

are expressions that create the objects country_side and tallest_mountain. Note that the object tallest_mountain is the third element of the array mountains.

A subclass inherits all attributes of the superclass. The syntax of constructing a class hierarchy is given as follows:

```
class AA; begin < Declaration of AA-attributes > end of AA;
AA class BB; begin < Declaration of BB-attributes > end of BB;
```

Here, class BB is a subclass of the class AA. The declarations

```
ref(AA) AA_var; ref(BB) BB_var;
AA_var :- new AA; BB_var :- new BB;
```

indicate that the reference variable AA_var points to an object containing the AA-attributes, and the reference variable BB_var points to an object containing both the AA- and BB-attributes. Consider the class TOWN illustrated in Figure 3.17 as a subclass of the class PLACE.

Class TOWN inherits the attributes of the class PLACE and adds the new attribute num_of_inhabitants. A reference variable of the type Town points to an object containing three attributes: longitude, latitude, and num_of_inhabitants. When declaring attributes that

have the same name as attributes in the superclass, the declarations in the subclass override the declarations inherited from the superclass. In the case of inheritance, a reference assignment expression is legal if the right-hand-side of the operator ":" refers to an object that is in the class of the left-hand-side variable or in one of the subclasses of this class. For example, consider the following reference assignments [Kirkerud 89].

```

ref(PLACE) P_var;
ref(TOWN) T_var;
P_var :- new TOWN; ! This is legal assignment since TOWN
                  ! is a subclass of the class PLACE.
T_var :- new PLACE;! This is not legal since new PLACE
                  ! is not an object in Town or in one
                  ! of its subclasses.

P_var :- new TOWN;
T_var :- P_var;   ! This is legal since P_var refers to
                  ! an object in the class of TOWN.

P_var :- new PLACE;
T_var :- P_var;   ! This is not legal since P_var does
                  ! not refer to an object in the class
                  ! TOWN.

```

Simula provides virtual procedures that allow the subclasses of a superclass to have procedures with common name, purpose, and use. For example, the procedure write in the class PLACE may be declared as virtual, and subclasses can have their specialized versions of write. The specification

```

virtual: procedures write;

```

implies that whenever procedure write is invoked in some object, the version declared in the class to which the object belongs should be used. Figure 3.18 [Kirkerud 89] illustrates the virtual declaration of the procedure write.

(B) Access Technique. The external interface of a class includes all of its attributes that are made available to its subclasses. Such approach provides full access to the attributes (instance variables) of the superclass. Class TOWN has full access to all attributes of the class PLACE.

```

class PLACE;
virtual procedure write, sub-write;
begin
  real longitude, latitude;
  procedure read begin < as before > end

  procedure write;
  begin
    outfix(latitude,2,5); outtext("degrees north,");
    outfix(logitude,2,5); outtext("degrees east.");
    sub-write; !write attributes specific to each subclass.
    outimage;
  end of PLACE'write;
end of PLACE;

PLACE class TOWN;
begin
  integer num_of_inhabitants;
  <Declaration of other Town attributes>;

  procedure sub-write;
  begin outtext(" number of inhabitants: ");
    outint(num_of_inhabitants, 0);
  end
  <Other declarations>;
end of TOWN;

```

Figure 3.18: Class PLACE using the virtual procedure write

Simula uses the dot notation for remote access to attributes using the expression `reference_expression.Att_name`. This notation allows direct access to the attribute `Att_name` of the object pointed to by `reference_expression`. If the reference expression refers to "none" or to an object that does not contain the attribute `Att_name`, the evaluation of the expression returns an error. Examples of remote access [Kirkerud 89] are:

```

ref(TOWN) my_town;
my_town.latitude := 37.44;
my_town.longitude := 58.29;
my_town.num_of_inhabitants := 300000;
my_town.write;

```

Simula supports the concept of protected attributes, so that attributes become inaccessible (invisible) from the outside of the class. The declaration "`protected Att_name`" makes the attribute `Att_name` inaccessible to neither subclasses nor users of

the class. This is similar to the private declaration in C++. To make protected attributes visible only to an immediate subclass, the subclass must include the declaration "**hidden Att_name**". Att_name is not visible to descendants of the superclass. This is similar to the protected declaration in C++. The declaration "**hidden protected Att_name**" can be used in the superclass to prevent an immediate subclass from accessing Att_name even though the subclass includes the declaration "**hidden Att_name**". Simula provides self-reference by means of pointers to objects. The declaration "**this Class_name**" refers to the current object of the class Class_name. Such object is called a **local object**. This is similar to the use of **this** in C++.

3.3 Subtyping and Inheritance

One important issue related to inheritance is subtyping. Various OOPLs relate subtyping to inheritance differently. Some languages including Smalltalk-80 and C++ (public derivation), view subtyping through inheritance between classes based on the structure hierarchy. In other languages, such as CommonObjects, subtyping is based on the behavior of the class objects rather than the structure. Below we examine the notion of subtyping and its relation to inheritance in some of the selected languages.

CommonObjects provides explicit behavioral subtyping that is not related to inheritance. A class can be a subtype of any other class, and the subtyping hierarchy can be separated from the inheritance hierarchy. This approach hides the use of inheritance from descendant classes. Changing implementation of a subtype class does not affect its subtyping relationship with its supertype class as long as it provides the same behavior.

In C++, inheritance is implicitly related to subtyping. When a subclass inherits all the superclass's methods (public derivation), it is a subtype of its superclass. A class cannot be a subtype of a non-ancestor class, and may not be a subtype of the superclass. The subtyping relationship between classes may be affected by changing the

implementation of the subtype class. For example, consider the classes `REGULAR_CAR` and `SPORT_CAR`. When the class `SPORT_CAR` inherits all features of the class `REGULAR_CAR`, class `SPORT_CAR` becomes a subtype of the class `REGULAR_CAR`. If the design of the sport car is changed by eliminating some features of the regular car, then the class `SPORT_CAR` is no longer a subtype of the class `REGULAR_CAR`. Therefore, C++ subtyping hierarchy depends on the inheritance hierarchy when public derivation is used.

In Trellis/Owl, behavioral subtyping is implicitly related to inheritance. Class X is a subtype of class Y if and only if X is a descendant of Y. A subtype must inherit all methods of its superclass(es), and provide the same behavior. Thus, subtyping is a specialization of classes. A superclass cannot hide the use of inheritance from its descendant, and cannot separate the subtyping hierarchy from the inheritance hierarchy. Similarly, Smalltalk-80 provides implementation-based subtyping that is implicitly related to inheritance. As a result of the hierarchical subclassing inheritance, the inheritance and subtype hierarchies are the same. That is, a subclass is a subtype of its superclass.

3.4 A Binary-Tree Taxonomy Model

OOPLs provide different inheritance mechanisms. A classification approach is presented in this section to provide a predictive framework to identify a new inheritance model in the space of inheritance models [Al-Haddad 91b]. The classification is based on major characteristics of inheritance in a number of well-known OOPLs (see Tables 3.3 through 3.5 provided at the end of this chapter). Those characteristics are listed below.

- C1) Type system provided (Static or Dynamic).
- C2) Inheritance mechanism (Single or Multiple).
- C3) Inheritance structure (Hierarchical or not).
- C4) Providing restricted external interface.
- C5) Providing semi-restricted external interface.
- C6) Providing restricted access to the instance variables.
- C7) Hiding the use of inheritance.

- C8) Hiding the instance variables.
- C9) Providing private methods.
- C10) Providing metaclasses.

C1 is the type-checking system of the language, C2 is the inheritance relationship among classes, and C3 is the structure of inheritance. In the characteristics C4, C5, and C6, languages provide different external interfaces. Some languages provide un-restricted interfaces and hence exposing the contents of a class to other classes, thus providing full access to the instance variables and methods of a class; while other languages provide restricted interfaces in order to hide the variables of a class by allowing the access only via the external interface specifications. A restricted external interface of a class provides only the method specifications to inheriting classes.

On the other hand, some languages provide semi-restricted external interfaces (in addition to the restricted ones) for certain inheriting classes. This rare situation exists in C++ [Stroustrup 86] and Eiffel [Meyer 88]. In C++, protected variables and friend declaration allow certain classes to directly access the private data of the defining class. Eiffel grants descendant classes (that inherit through the *Inherit* clause) full access to the class implementation; while other inheriting classes are granted restricted external interfaces providing them only the specifications of the superclass's methods.

C7 is hiding the use of inheritance in the sense that inheriting classes do not know about the use of inheritance in the superclass(es) if present. In C8, the instance variables are hidden from the other classes and are accessible via methods of the external interface. In C9, classes may have private methods which are local to the defining class and are not available to other classes. Finally, C10 is that classes are instances of other classes (metaclasses). The variety of languages captured in this taxonomy is sufficient for the well-known languages considered in this chapter. These languages and their reference codes are given in Table 3.1.

TABLE 3.1
LANGUAGES CONSIDERED IN THE
CLASSIFICATION TREE

Language	Code	Language	Code
Trellis/Owl	L1	CLOS	L5
C++	L2	Flavors	L6
Eiffel	L3	Smalltalk	L7
CommonObjects	L4	Simula	L8

The proposed classification is given in the form of a binary tree in which all the characteristics are grouped into 4 sets for the sake of manageability of representation. A complete binary tree with ten characteristics would result in 1024 leaves. The grouping of several characteristics into internal nodes certainly reduces the descriptive and predictive power of the taxonomy. Each set of characteristics occupies one level on the tree. The sets, their characteristics, and their levels are given in Table 3.2.

In the binary tree representation, a node represents one or more characteristics. In the representation of the characteristics C1, C2, and C3, the left edge of the node indicates static type, single inheritance, and hierarchical inheritance structure; while the right edge indicates dynamic type, multiple inheritance, and non-hierarchical inheritance structure. For other characteristics, the left edge indicates the presence of the characteristic(s), and the right edge indicates their absence in the language being classified.

The classification tree illustrated in Figure 3.19 represents the proposed classification of the selected languages in Table 3.1. It has four levels representing the sets of characteristics in Table 3.2. In Figure 3.19, the languages on the left edge of a node provide the characteristics represented in that node; while the languages on the right edge do not provide the represented characteristics. The fourth level (leaves) of the

TABLE 3.2
 NODE EXPANSIONS OF THE
 CLASSIFICATION TREE

Node	Level	Characteristics
n0	0	C8, C6, C1, C10
n1, n2	1	C4, C5, C7
n3 - n6	2	C2, C3
n7 - n14	3	C9

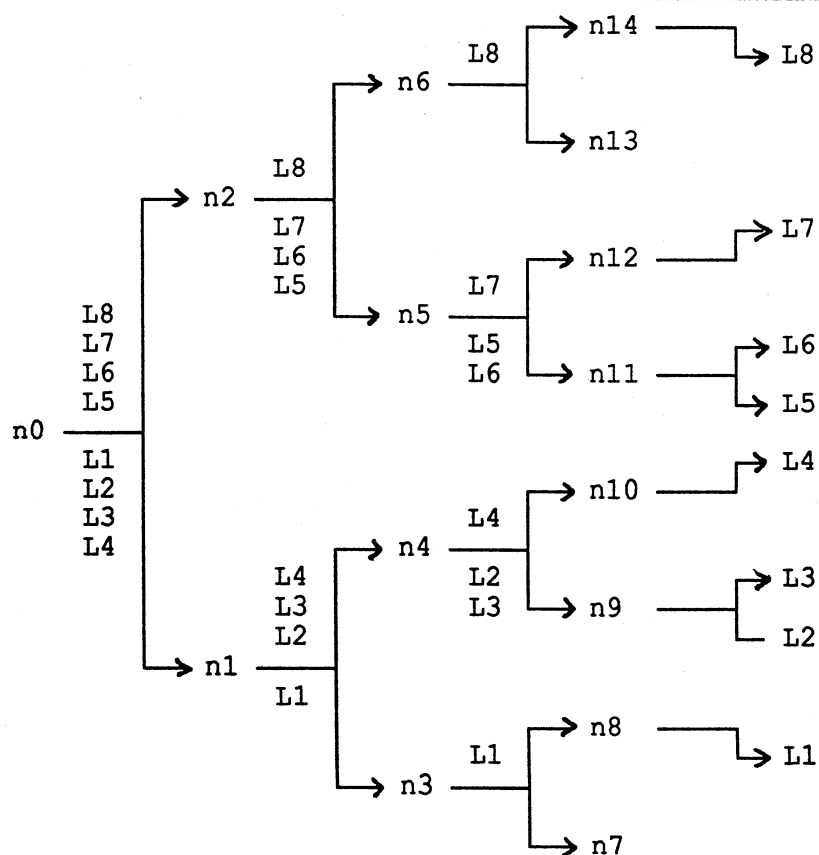


Figure 3.19: The classification Tree

classification tree represents the resulting models in which all characteristics along the path from a leaf node to the root node are supported by the model.

Each node of the classification tree is a subtree of the characteristics it represents. The height of a subtree of a node equals the number of characteristics represented in that node. The trees in Figure 3.20 through Figure 3.26 represent the subtrees corresponding to the internal nodes of the classification tree in Figure 3.19. Here, the nodes are denoted by the corresponding characteristics. The left edge of a node indicates the presence of the characteristic, and the right edge indicates the absence of the characteristic in the classified languages.

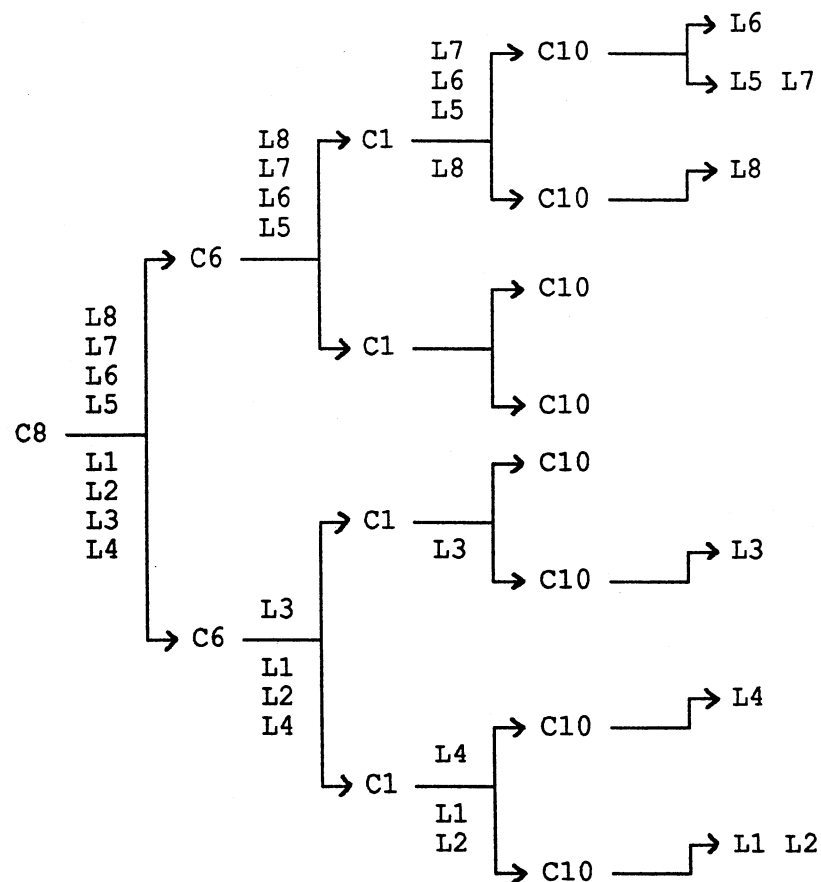


Figure 3.20: The subtree n0

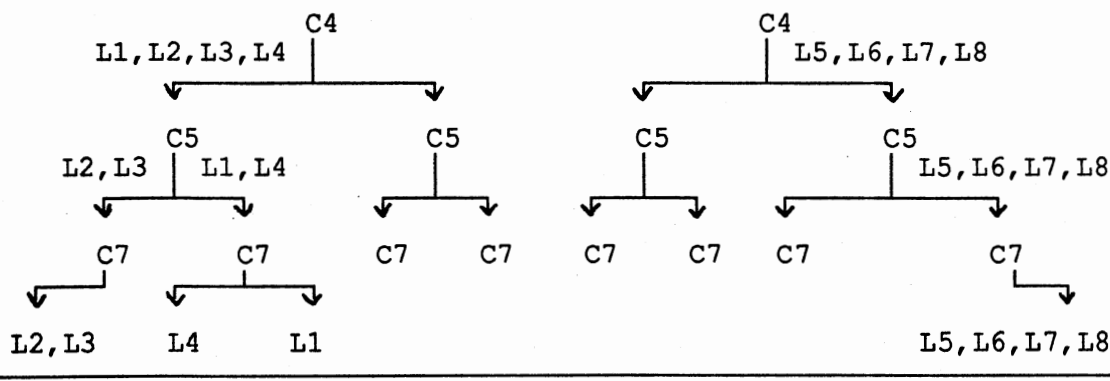


Figure 3.21: The subtree n1

Figure 3.22: The subtree n2

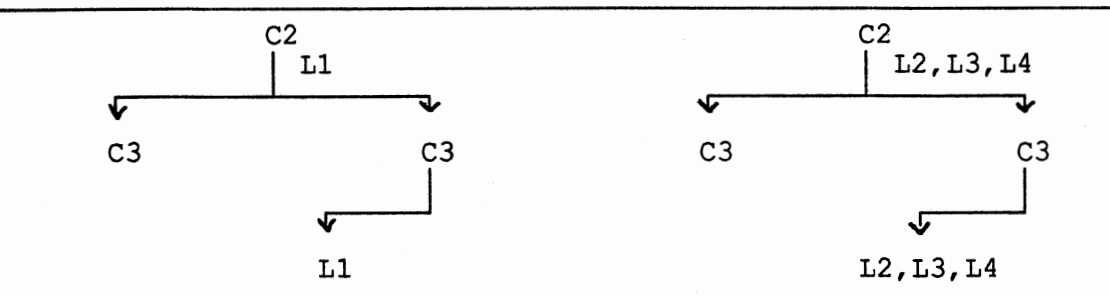


Figure 3.23: The subtree n3

Figure 3.24: The subtree n4

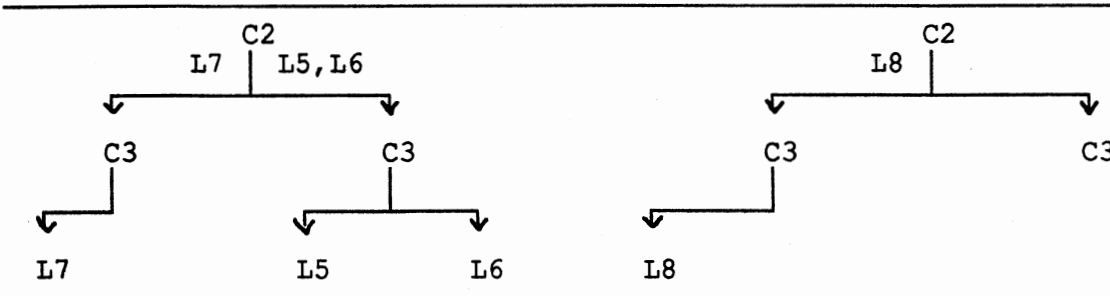


Figure 3.25: The subtree n5

Figure 3.26: The subtree n6

In each of the subtrees, the leaf nodes are grouped into two sets, each set goes along one edge of the node. These sets are the languages annotating the edges of the classification tree. For instance, in the subtree n0 in Figure 3.20, L1, L2, L3, and L4 go to n1 in the classification tree; while L5, L6, L7 and L8 go to node n2. Likewise, the leaf nodes of other subtrees (n1 through n6) are grouped into two sets annotating the edges of the represented node in Figure 3.19.

Note that The left edge of the node C1 in Figure 3.20 indicates statically typed languages and the right edge indicates dynamically typed languages. Moreover, the left edge of node C2 in Figure 3.23 and Figure 3.24 indicates single inheritance and the right edge indicates multiple inheritance. The left edge of node C3 indicates hierarchical inheritance and the right edge indicates non-hierarchical inheritance.

The subtrees corresponding to the nodes n7 through n14 of the classification tree are at most one-node trees since each node represents one characteristic. They are obvious from the classification tree in Figure 3.19. For instance, the subtree n7 is empty, and the subtree n9 is one-node tree in which L2 annotates the left edge and L3 annotates the right edge. The annotations on the edge leading the leaf nodes are the same as the corresponding leaf nodes.

3.5 Discussion

In [Snyder 86a], Snyder examined the relationship between inheritance and encapsulation. He has proposed several requirements for full support of encapsulation with inheritance. In the rest of this section, the selected languages are briefly analyzed with respect to their support for these requirements. These requirements are listed below.

- 1) Providing different external interfaces for both instances and inheriting classes of a class;
- 2) Hiding the instance variables and class implementation from the inheriting classes;

- 3) Accessing the instance variables through methods provided is the external interface;
- 4) Hiding the use of inheritance from inheriting classes;
- 5) The ability of defining private methods for the benefit of inheriting classes; and
- 6) Granting inheriting classes the ability to exclude certain methods from their external interface;

CommonObjects, C++, and Trellis/Owl provide different external interfaces for instances and the inheriting classes of a class. Smalltalk-80 provides the same external interface for instances and inheriting classes. Classes of CommonObjects and C++ hide the instance variables and the use of inheritance from the inheriting classes. Smalltalk-80 and CLOS expose both instance variables and the use of inheritance outside the class definition, and grant inheriting classes full access to the instance variables of the superclass. In Flavors, a flavor cannot hide the use of inheritance since variables and methods of the component flavors are visible to the inheriting flavors.

Since subclassing in Trellis/Owl is based on behavior, classes cannot hide the use of inheritance, and inheriting classes must inherit all methods defined in the superclass. But the instance variables are hidden in the class, and are accessed through methods provided in the external interface. Since C++ and Trellis/Owl provide private data, users may define private methods for the benefits of the inheriting classes, while other languages do not support this feature.

In CommonObjects, C++, and Trellis/Owl, inheriting classes may exclude methods defined in the external interface of the superclass. Users of CLOS and Flavors may select certain methods from different classes through the definition of the generic function. Because Smalltalk-80 grants inheriting classes full access to the superclass and like Trellis/Owl, inheriting classes must inherit all methods defined in the superclass. Other features of the selected languages are provided in the Tables 3.3 through 3.5.

Several comparison surveys are found in the literature. Here, we highlight a few

of them. E. Seidewitz [Seidewitz 87] compared the capabilities of Ada and Smalltalk-80 from an object-oriented perspective. He addressed the basic properties of encapsulation, inheritance, and binding in object-oriented programming. He focused on comparing the object-oriented capabilities in Smalltalk-80 with the object-oriented features of Ada and modula.

J. Micallef [Micallef 88] provides a comparison survey of OOPLs in terms of their support for encapsulation, reusability, and extensibility as objectives of the object-oriented paradigm. Several other papers including [Bezivin 87], [Meulen 87], and [Blaschek 89] discuss simulation experiments in OOPLs, and the flexibility and reliability gained from the inheritance property and other properties of the OOP methodology.

In [Klint 86], Klint addressed a comparison between the algorithmic and OOPLs from a perspective of code reusability which is a form of the inheritance concept. He also reviewed the features required, including inheritance, to obtain OOP in a non OOPL.

3.6 Summary

In this chapter, we have discussed three major aspects of OOP: inheritance techniques, access mechanisms, and relating subtyping to inheritance. Current OOPLs address these aspect differently, and have different deficiencies in their support for other issues such as data encapsulation, information hiding, and the visibility of inheritance.

Inheritance has different interpretations and purposes in the OOP. In the selected languages, inheritance impacts the other issues in varying degrees. In CommonObjects, C++, and trellis/Owl, the inheritance mechanisms and access techniques do not compromise data encapsulation, and provide for flexible software development. While in Smalltalk-80 and Flavors, the inheritance mechanisms and access techniques compromise data encapsulation and limit the flexibility of program development.

Various OOPLs relate subtyping to inheritance differently. In some languages, they

are related based on the class implementation; while in other languages are related based on the behavior rather than the implementation. Behavioral subtyping separates the inheritance hierarchy from the subtyping hierarchy, and does not compromise data encapsulation and the visibility of inheritance. Implementation-based subtyping compromises data encapsulation and the visibility of inheritance, and the inheritance and subtyping hierarchies are dependent.

In Section 5, Snyder's requirements for supporting encapsulation with inheritance are introduced. The selected languages are analyzed in terms of these requirements. The inheritance mechanisms of CommonObjects, C++, and Trellis/Owl support most of these requirements.

TABLE 3.3
FEATURES OF THE SELECTED LANGUAGES (1)

Feature/Language	Trellis/Owl	C++	Eiffel	CommonObjects
Language Type	Static	Static	Static	Dynamic
Multiple Inheritance	Yes	Yes	Yes	Yes
Hierarchal Inheritance	Yes	Yes	Yes	Yes
External Interface	Restricted	Restricted & semi Rest.	Restricted & semi Rest.	Restricted & semi Rest.
Accessing the Inst. Variables	Restricted	Restricted	Full	Restricted
Use of Inheritance	Exposed	Hidden	Hidden	Hidden
Visibility of Inst. Variables	Invisible	Invisible	Invisible	Invisible
Same-named Inst. Variables	Yes	Yes	Yes	Yes
Private Methods	Yes	Yes	No	No
MetaClasses	No	No	No	No

TABLE 3.4
FEATURES OF THE SELECTED LANGUAGES (2)

Feature/Language	CLOS	Flavors	Smalltalk-80	Simula
Language Type	Dynamic	Dynamic	Dynamic	Static
Multiple Inheritance	Yes	Yes	No	No
Hierarchal Inheritance	Yes	No	Yes	Yes
External Interface	Restricted	Un-Restricted	Un-Restricted	Un-Restricted
Accessing the Inst. Variables	Full	Full	Full	Full
Use of Inheritance	Exposed	Exposed	Exposed	Hidden
Visibility of Inst. Variables	Visible	Visible	Visible	Visible
Same-named Inst. Variables	Yes	Yes	No	Yes
Private Methods	Yes	No	No	No
MetaClasses	Yes	No	Yes	No

TABLE 3.5
FEATURES OF THE SELECTED LANGUAGES (3)

Language/Feature	Communication Approach	Direct Invocation Construct(s)
Trellis/Owl	Procedure Call	me / P'OP(me, ...)
C++	Function Call	this / "::"
Eiffel	Procedure Call	x:ClassType / x.method_name
CommonObjects	Message Passing	call_method
CLOS	Generic Function	call_next_method
Flavors	Generic Function	self
Smalltalk-80	Message Passing	self / super
Simula	Procedure Call	this

CHAPTER IV

APPROACHES TO REUSABILITY IN

C++ AND EIFFEL

4.1 Introduction

Concepts in software technology such as commonality, portability, modularity, maintainability, and evolution are closely related to the concept of reusability. In the development of new software systems, Freeman [Freeman 83] suggested that the needed information can be classified into five levels. Figure 4.1 is adopted from [Freeman 83]. It illustrates these levels along with the information types in each level where the arrows imply a reuse relationship between these types. At the lowest level (Code Fragment), reusability of source code is a primary objective of software development.

OOP is one of the methodologies in which reusability can be practiced effectively by utilizing existing programs as well as producing new programs that can be reused in future developments. Reusability helps reduce the cost of software development and makes the software design and development tasks easier and more reliable. The two important language concepts that support code reuse are polymorphism and inheritance. This chapter is devoted to study the language support for code reuse. The languages C++ [Stroustrup 86,91] [AT&T 89a] and Eiffel [Meyer 88] are used as cases to explore how inheritance and polymorphism are incorporated into languages. C++ is an extension of the C language; while Eiffel is a new languages definition. We compare and contrast their support for reuse in terms of inheritance, polymorphism, and other related issues.

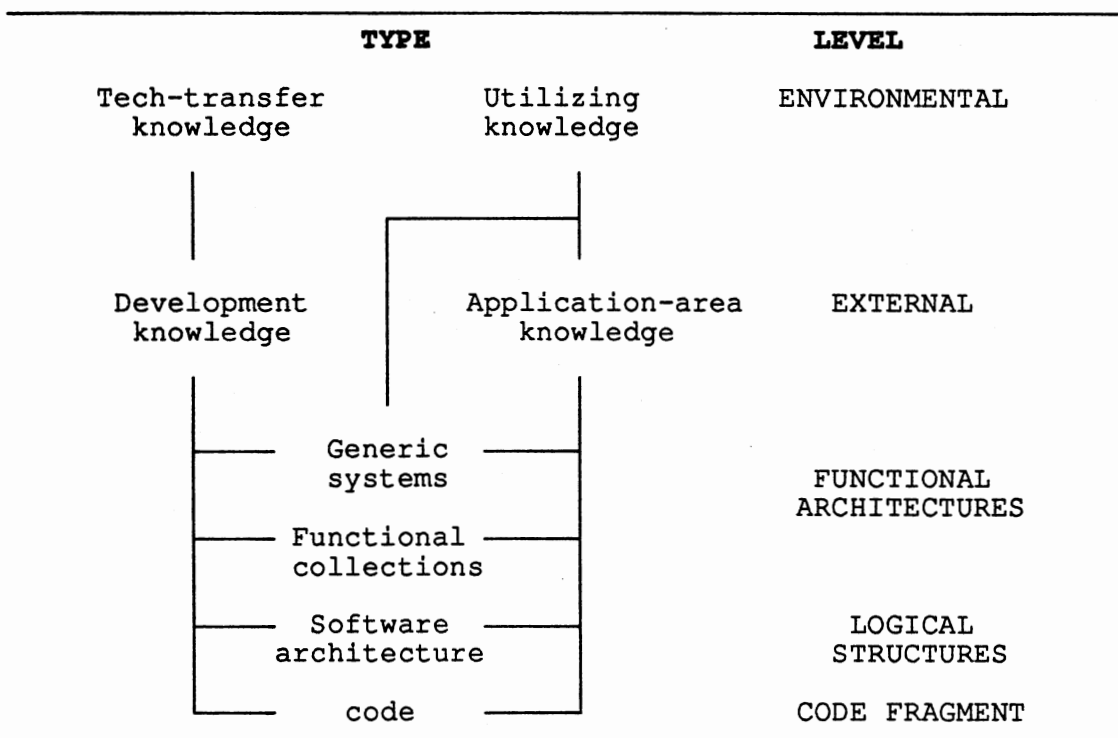


Figure 4.1: Hierarchy of software development information

The class concept is the key for reusability and extensibility [Tracz 88] [Biggerstaff 89 a,b]. Classes can be combined and modified to create new classes (for new applications) by utilizing the concepts of inheritance and polymorphism [Edelson 87]. Unlike object-based inheritance models [Hailpern 87] [Ungar 87], C++ and Eiffel provide class-based inheritance models. Inheritance is used as a mechanism for code sharing among classes to construct new software components from existing ones. The new class differs from the superclass(es) in the way it is derived. A new class may extend its superclass(es) by adding new methods and variables, or may specialize its superclass(es) by redefining some of the inherited methods [Strom 86].

Another concept that supports software reuse is polymorphism. Polymorphism provides the capability of using a function with objects of different types. Polymorphism can be divided into ad hoc polymorphism and parameterized polymorphism. Most programming languages provide some degree of ad hoc polymorphism by providing

overloaded operators and parameterized polymorphism is beginning to be incorporated into programming languages. We follow the Eiffel terminology; the term "generic" is used in place of parameterized polymorphism and polymorphic entities refer to objects of a type whose values may change during the execution time [Meyer 88a]. Ad hoc polymorphism will be referred to as overloading. The term polymorphism will be used in the general sense to include generic, overloaded, and polymorphic entities.

Libraries of specialized routines are created to facilitate code reuse. OOPLs provide libraries of classes (to be imported into specific applications) as an approach to reusability. Libraries provide classes of fixed functionality that cannot be modified in a target application. Libraries of highly parameterized functions would increase code reusability and help programmers to develop new software systems more efficiently [Wegner 83].

In this chapter we describe the major features that support reuse in the selected languages Eiffel and C++ [Al-Haddad 91c]. We describe the concept of inheritance and its related issues (by means of examples) as an approach to reusability. The various language features are compared and contrasted from reusability and extensibility point of views.

4.2 Design Objectives and Highlights of the Two Languages

In this section we describe the major aspects of Eiffel and C++ in terms of their support for inheritance, polymorphism, and other related issues. Tables 4.1 and 4.2 provide a summary of features that impact reusability and extensibility in both languages.

4.2.1 Eiffel

Eiffel is an OOPL designed in late 1985. The main objectives of its design are efficiency, reliability, reusability, extensibility, modularity, and portability. Eiffel is

designed as a new language rather than an extension of an existing language. In addition to its simplicity and design consistency, Eiffel provides a set of powerful tools such as a library of classes for I/O operations and string manipulation, assertions, support for using code written in other languages, generation of portable C packages, graphics package GOOD (Graphics for Object-Oriented Design), and garbage collection.

TABLE 4.1
INHERITANCE AND RELATED ISSUES IN C++ AND EIFFEL

Inheritance and related issues	Language	
	C++	Eiffel
Inheritance type	Multiple Inheritance	Multiple Inheritance
Inheritance mode	Selective (inherit what you need)	Non-selective (inherit all or none)
Inheriting groups	One group (derived classes)	Two groups (clients and descendant classes)
Information hiding	Applied strongly	Applied to clients not to descendant classes
Genericity	Limited by use of macro	Unconstrained and compatible with inheritance
Polymorphic assignment	Not applicable	Constrained by inheritance
Dynamic binding	Provided for virtual routines of a class	Provided for all methods of a class
Abstract classes	Applicable in the new release 2.0	Applicable and called deferred classes
Overriding and renaming of methods	Only overriding is applicable	Both are applicable
Creation routines	Creation routines are inherited	Creation routines never inherited directly

TABLE 4.2
VISIBILITY AND INTERFACES IN C++ AND EIFFEL

Interface entities/features	Visibility	
	C++	Eiffel
Class methods	Public methods are visible to clients and protected methods are visible to derived classes	Exported methods are visible to clients and all methods are visible to descendant classes
Data representation	Public data is visible to derived classes	Public and private data are visible to descendant classes
Instance variables	Hidden and accessed via inherited methods	Visible and accessed directly by descendants
Exclusion of methods	Applicable	Not applicable
Availability of methods	Class methods are equally available to other class(es)	A class can export some features to certain related classes only
Same-named methods	Classes can have same-named methods	Classes can have same-named methods
Naming conflicts of methods	Resolved by the resolution operator "::"	Resolved by the renaming mechanism
Visibility options	Several options are provided based on the base class type and the data declaration	Few options based on whether clients or descendants

Eiffel is a statically-typed language. It provides multiple inheritance reconciled with dynamic typing and strong type checking. A class provides private and public data to maintain information hiding and encapsulation. In addition, a class contains the implementation of its methods. Eiffel provides two groups of inheriting classes. The first group consists of client classes that include declaration of the form `var_name:ClassType` where `var_name` is a variable name and `ClassType` is a defined class. Client's inheritance is a has-a relationship. The second group consists of descendant classes that inherit explicitly from one or more other classes through the `inherit` clause declaration.

This inheritance relationship is characterized as an is-a relationship. Features (methods and variables), such as Create, Forget, Clone, and Result, are not inherited and are applicable to all classes. Eiffel provides two clauses: **Feature** which describes all variables and methods of the class, and **Export** which provides all features available for client classes. Class features listed within the export clause are public while other features are private. Different classes may have same-named methods and a class may inherit the same method from different classes.

Information hiding provided by the export control allows a class to export certain features (variables and methods) to its clients. Eiffel introduces the Open-Closed principle. That is, classes are open for descendants for extension and reuse; while classes are closed to be accessed through their interfaces by clients. The public/private mechanism is applied to clients of a class, exported features are visible and accessible to clients; while features not exported are neither visible nor accessible. For descendants, everything is visible and directly accessible. Therefore, a descendant class may depend on the implementation of its superclass(es). For instance, one may implement the class STACK as an Array, then the class STACK depends on the implementation of the class ARRAY.

Polymorphism and dynamic binding mechanisms allow users to define polymorphic entities that refer to different instances of different classes. Polymorphism is limited by inheritance to achieve type compatibility. If the class Y is a descendant of the class X, entities of type X can be assigned to entities of type Y ($x:=y$), not the other way [Meyer 88b].

A deferred routine is a routine that is defined in a superclass and implemented in a descendant class. A class that contains one or more deferred routines is called a deferred class. Deferred classes do not have instances. They provide common behavior among classes to increase code reusability and class extensibility. The deferred classes also allow users to provide different implementations of abstract data types. Non-deferred classes are called effective classes.

4.2.2 C++

C++ version 2.0 [Stroustrup 91] [AT&T 89b] provides multiple inheritance along with other features as a modification of and extension to the old version [Stroustrup 86]. The new release is intended to provide efficient and reliable use of the class concept and to provide tools for creating libraries of user-defined types for different applications.

C++ provides OOP with multiple hierarchical inheritance by means of class declarations. Class declarations divide classes into sections to limit the visibility of the class contents to other classes. A class may include public, private, and protected sections. Each section contains members (variables and methods) of the same degree of visibility.

C++ inheritance is an is-a relationship. A derived class is a specialization of the base class(es). It combines features of its base class(es) to provide new composite features (concepts). A derived class inherits what it needs from base class(es), it may override undesired inherited methods, and may add new variables and methods. Inheritance supports information hiding and provides restricted and reliable external interfaces for clients and derived classes.

Overloading does not solve the problem of reusability. Overloading operations allows an operation name to provide different meanings. Thus, an operation can have different implementations. The 2.0 version overcomes the ambiguity of overloaded function names in different libraries by providing an improved linkage scheme. This scheme provides satisfactory solution for working with overloaded functions in the construction, combination, and usage of libraries. Moreover, the new linkage scheme allows users to access functions in libraries of other languages using the appropriate linkage specifications.

The external interfaces provided for instances and derived classes include specification of the methods defined in the superclass(es). An instance of a derived class includes only the instance variables used by the inherited methods from the superclass(es)

and not all of the superclass's instance variables [Gorlen 87]. Every object of a derived class has its own copy of the superclass's private variables, as well as its own private variables.

C++ version 2.0 introduces the notions of pure virtual functions, abstract classes, and virtual base classes. A pure virtual function is a function that does not have a definition in the defining class. An abstract class is a class that contains at least one pure virtual function. These features allow users to provide different definitions of a function in different classes. A virtual base class is an extension of the base class concept. They allow a class to share data from its multiple superclasses, which are represented as pointers to the shared data structure for the virtual base class.

Public/protected/private mechanism provides different levels of visibility and data hiding for the class contents. In multiple inheritance, the visibility rules are applied to base classes individually. When a derived class includes private and public base classes, the visibility rules are applied to each base class as in single inheritance. Restricted visibility provides strong encapsulation in which the inheritance hierarchy is more adaptable to change and grants the designer the freedom to change the class implementation without affecting its clients.

The concept of friend classes is another option of visibility that allows a classes to grant access to its private data by other classes. Thus a set of private definitions in a class can be available to specific classes.

4.3 Inheritance and Reusability

In this section we illustrates by means of examples how inheritance and its related issues are used as approaches to reusability in Eiffel and C++. Eiffel examples are adopted and simplified from [Meyer 88b] and C++ examples are the C++ code that correspond to the Eiffel's examples.

4.3.1 Eiffel

Eiffel's single inheritance mechanism is illustrated in Figures 4.2 and 4.3 adopted from [Meyer 88]. In Figure 4.2, the class `EFFECTIVE_LIST[T]` describes the general features of lists using an array implementation; while in Figure 4.3 the class `STACK[T]` is a descendant of the class `EFFECTIVE_LIST[T]`. Class `STACK[T]` modifies inherited features and adds new features. For descendant classes, inheritance is described through the `inherit` clause. Syntactically, the `inherit` clause explicitly provides the names of the superclass(es). For instance, the statement

```
inherit EFFECTIVE_LIST[T]
```

in Figure 4.3 makes all contents of the class `EFFECTIVE_LIST[T]` available for class `STACK[T]`.

```
class EFFECTIVE_LIST[T] export
  nb_elements, position, empty, full, offright,
  offleft, value, change_value, add_new;
  -- Other features ...

feature
  nb_elements: INTEGER; -- number of the list's elements
  max_size: INTEGER;    -- list maximum size
  position: INTEGER;    -- cursor position
  implementation: ARRAY[T]; -- list implementation

create (n: INTEGER) is -- create a list of size n elements
  do if n>0 then max_size := n end;
  implementation.create(1, max_size)
end;

empty: BOOLEAN is      -- is list empty?
  do Result := (nb_elements = 0) end;

full: BOOLEAN is     -- is list full?
  do Result := (nb_elements = max_size) end;

offright: BOOLEAN is -- is cursor off right edge?
  do Result := empty or (position = nb_elements+1) end;

offleft: INTEGER is  -- is cursor off left edge?
  do Result := empty or (position = 0) end;

value: T is          -- value of element at cursor position
  require not offleft; not offright; -- not empty
  do Result := implementation.entry(position) end;
```

```

change_value(v:T) is --assign v to element at cursor position
  require not offleft; not offright; -- not empty
  do implementation.enter(position, v);
  insure value = v; implementation.entry(position) = v;
  end;

add_new(v: T) is -- add new element to the right of list
  require (nb_elements < max_size); -- not full
  do nb_elements := nb_elements + 1;
    implementation.enter(nb_elements, v);
  insure implementation.entry(nb_elements) = v;
    nb_elements := old nb_elements + 1
  end;
  -- ... Many other features

invariant
  0 <= position; position <= nb_elements + 1;
  not empty or else (position = 0);
  empty = (offleft and offright);
  offright = empty or (position = nb_elements + 1);
  offleft = empty or (position = 0);
end; -- class EFFECTIVE_LIST

```

Figure 4.2: Definition of the class EFFECTIVE_LIST[T]

```

class STACK[T] export
  push, pop, top, full, empty, nb_elements
inherit EFFECTIVE_LIST[T];
rename value as top, add_new as push, create as list_create;
redefine top;

feature
  implementation: ARRAY[T];
  create (n: INTEGER) is --allocate a stack of n elements
    do list_create (1, n) end;

  pop: T is -- pop top value
    require not empty;
    do nb_elements := nb_elements - 1;
    ensure nb_elements = old nb_elements - 1;
    end;

  top: T is -- return top element
    require not empty;
    do Result := implementation.entry(nb_elements)
    end;
end; -- class STACK

```

Figure 4.3: The class STACK[T] is a subclass of the class EFFECTIVE_LIST[T]

Since descendant classes have full access to the public and private features of the superclass(es), renaming and redefinition properties allow descendant classes to rename and/or redefine undesired features. For example, the class `STACK[T]` in Figure 4.3 redefines and renames the inherited features `value`, `add_new`, and `create` from the class `EFFECTIVE_LIST[T]` to provide an appropriate implementation for the class `STACK[T]`. Redefinition and renaming have different purposes. Redefinition of a feature associates the name with new feature, while renaming associates the same feature with a new name. Thus, redefinition provides different features under the same name and renaming provides different names for the same feature.

```

class FIXED_LIST[T] export
    ... Same exported features as in class LIST[T]
inherit ARRAY[T] rename create as array_create;
        LIST [T] redefine i_th, change_i_th, swap, go;

feature
    create(n: INTEGER) is -- allocate stack of n elements
        do array_create(1,n);
            check n = size end;
            nb_elements := n;
        end;

    value: T is -- Value of element at cursor position
        do Result := entry(position) end

    change_value(v:T) is --Assign v to cursor position entry
        do enter(position, v);
            ensure value = v; entry(position) = v
        end;

    i_th(i: INTEGER): T is -- Return value of i_the entry
        require 1 <= i; i <= nb_elements;
        do mark; go(i); Result := entry(i); return;
        ensure value = v
        end;

    change_i_th(i:INTEGER, v:T) is -- change the i_th entry
        require 1 <= i; i <= nb_elements
        do mark; go(i); enter(i,v); return
        ensure value = v; entry(i) = v
        end;
    -- Definitions of other features ...
end; -- class FIXED_LIST

```

Figure 4.4: Definition of the class `FIXED_LIST[T]`

A class with no deferred routines is called an effective class. Multiple inheritance allows users to use features of an effective class to implement routines of a deferred class. In Figure 4.4, the class `FIXED_LIST[T]` describes a fixed-length list using an array implementation (complete listing of the classes `LIST[T]` and `ARRAY[T]` can be found in [Meyer 88b, pages 452 to 641]). Class `ARRAY[T]` is an effective class, and the class `LIST[T]` is a deferred class. Class `FIXED_LIST[T]` inherits the properties of the class `ARRAY[T]` and provides implementations for deferred routines in the class `LIST[T]`. Class `ARRAY[T]` sets the bounds of a fixed-length list and provides its methods to access the fixed-length list's entries.

The possibility of redefining inherited feature is provided for descendant classes as a property of multiple inheritance. The class `FIXED_LIST[T]` in Figure 4.4, adopted from [Meyer 88], renames the method `create` from the class `ARRAY[T]` to be invoked in its own `create` method; it also redefines some features of the class `LIST[T]` to fit its implementation since these features are provided to serve unbounded lists.

Eiffel introduces the notion of repeated inheritance. This notion allows classes to inherit more than once from ancestor class(es). For instance, one may rewrite class `FIXED_LIST[T]` as follows:

```

class FIXED_LIST[T] export ...
  inherit ARRAY[T] rename ... redefine ...
  inherit LIST[T] rename ... redefine ...
  inherit ARRAY[T] rename ... redefine ...
  inherit LIST[T] rename ... redefine ...
  ...

```

4.3.2 C++

C++ version 1.0 provides single inheritance where a base class defined the common features of related classes. This is a simple form of inheritance which is provided by all OOPs. To provide the ability of redefining an inherited method, the virtual feature needs to be used. Figure 4.5 and Figure 4.6 provide sample definition of

single inheritance. The classes LIST and STACK are functionally equivalent to those defined in Figures 4.2 and 4.3.

```

#include <stream.h>
#include <string.h>
const char* msg[] = // array of error messages.
{ "Overflow.\n", "Underflow.\n", "Cursor off the left edge.\n",
  "Cursor off the right edge.\n", "List is not empty.\n",
  "Index out of range.\n" };

class LIST
{ public:
  int* lst; // pointer to a list
  int max_size; // maximum size of a list
  int nb_elements; // number of elements in a list
  int position; // cursor position
  virtual int empty(); // is list empty?
  int offrigh(); // is cursor off right edge?
  int offleft(); // is cursor off left edge?
  virtual int value(); // read from current position
  void change_value(int); // write to current position
  void add_new(int); // add to the right of list
  void forth(); // move cursor 1 position ahead
  virtual void back(); // move cursor 1 position back
  virtual void go(int); // Go to the i_th entry
  virtual int i_th(int); // Return the i_th entry
  virtual int change_i_th(int, int); //Change the i_th entry
  virtual void swap(int); //Swap positioned and i_th entries
  void init() // initialize variables
    { nb_elements = 0; position = 0; }
  void error_msg(int e_num) // issue an error message
    { cout << msg[e_num]; }
  LIST(int size) // constructor
    { max_size = size; lst = new int[size]; }
  ~LIST() { delete lst; } // destructor.
};

int LIST::empty() // is list empty?
{ return (nb_elements == 0); }

int LIST::offrigh() // is cursor off right edge?
{ return ((empty()) || (position == nb_elements)); }

int LIST::offleft() // is cursor off left edge?
{ return ((empty) || (position == 0)); }

int LIST::value() // return value at cursor position
{ if (offleft()) error_msg(2);
  else if (offrigh()) error_msg(3);
  else return lst[position]; }

int LIST::change_value(int v)
//change value at cursor position
{ if (offleft()) error_msg(2);
  else if (offrigh()) error_msg(3);

```

```

        else lst[position] = v; }

int LIST::add_new(int v) // add new entry to list
{ if (nb_elements > max_size) error_mag(0);
  else { if (nb_elements == 0) position = 1;
         lst[++nb_elements] = v; } }

int LIST::forth() // move cursor one position ahead
{ if (offright()) error_mag(3);
  else ++position; }

int LIST::back() // move cursor one position back
{ if (offleft()) error_msg(2);
  else --position; }

// ... Definitions of other features such as go, i_th,
// change_i_th, swap, ...

```

Figure 4.5: Definition of the class LIST

In Figure 4.5, the class LIST defines a subset of the methods of a general list. The class STACK is derived from the class LIST since stacks are lists with restricted accesses. The class STACK redefines inherited methods and adds a new method. In this example,

```

class STACK : public LIST
{ public:
  int empty(); // is stack full?
  void back(); // pop top value
  int value(); // return top value
  void push(int); // push on top
  STACK(int size) : (size) {} // constructor
};

int STACK::empty() // is stack full?
{ return (nb_elements == max_size); }

int STACK::back() // pop top value
{ if (nb_elements == 0) cout<<"Stack is empty...\n";
  else --nb_elements; }

int STACK::value() // return top value
{ if (list::empty()) error_msg(1);
  else return lst[nb_elements]; }

int STACK::push(int v) // push onto top
{ list::add_new(v); }

```

Figure 4.6: Class STACK is derived from the class LIST

the main program in Figure 4.7 creates the list L and the stack S of at most three elements. In the class STACK, the method push shows the direct invocation of the base class method add_new. One may remove the function push from the class STACK and directly use the method add_new (in Figure 4.5) on stack S in the main program (e.g., S.add_new(100)).

If the class LIST in Figure 4.6 is defined to be private base-class of the class STACK using the declaration

```
class STACK : Private LIST {...}
```

then the public data of the class LIST are private to the class STACK and are not available to users of the class STACK. Virtual functions imply using the body of inherited method in the derived class. Thus, different versions of the inherited method may be defined for different derived classes. For instance, the virtual function value() is being modified in the derived class STACK, and the new version of the function value is executed when the invocation "STACK::value" is issued. Thus, "LIST::value" returns the entry at the current position and "STACK::value" returns the top element of the stack.

```
main()
{ int s = 3;           // size
  LIST L(s);          // create list L of three entries
  STACK S(s);         // create stack S of three entries
  L.add_new(10);L.add_new(20);L.add_new(30); // initialize L
  S.push(100); S.push(200); S.push(300); // initialize S
  cout << L.value() <<"\n"; // print value 10 from L
  L.forth(); L.forth(); // move cursor ahead twice
  cout << L.value() <<"\n"; // print value 30 from L
  L.back(); // move cursor back once
  cout << L.value() <<"\n"; // print value 20 from L
  cout << S.value() <<"\n"; // return value 300 from S
  S.back(); // pop top value (300)
  cout << S.value() <<"\n"; // return value 200 from S
};
```

Figure 4.7: The main program

C++ classes may have same-named methods. The scope resolution operator "::"

helps users avoid naming conflicts of methods by explicitly specifying the invoked methods (e.g, "LIST::add_new(v)" in the class STACK). In addition, type and number of a method's arguments can specify the invoked method [Pinson 88].

The new release of C++ [Stroustrup 91] [AT&T 89b] introduces multiple inheritance in the sense that a class may have more than one base class. A derived class combines independent features of the bases classes. The class ARRAY_LIST in Figure 4.9 inherits from the classes ARRAY in Figure 4.8 and LIST in Figure 4.5. It combines properties of the classes ARRAY and LIST to provide a fixed-length list using an array implementation. The declaration

```
class ARRAY_LIST : public ARRAY, public LIST
```

implies that the public data of the base classes is public to class ARRAY_LIST.

```
class ARRAY
{ private:
    int area;           // Array entries
  public:
    int lower;         // Array lower bound
    int size;          // Maximum array size
    int upper;         // Array upper bound
    int entry(int);   // Return the i_th entry
    void enter(int, int); // Assign the i_th entry
  // ... Constructor, destructor, and initialization
};

int ARRAY::entry(int i)    // is list empty?
{ // if upper < i < lower then error index out of range
  // else return the i-th element. }

void ARRAY::enter(int i, int v) // is cursor off right edge?
{ // if upper < i < lower then error index out of range
  // else assign value v to the i-th entry. }
```

Figure 4.8: Definition of the class ARRAY

To illustrate the notion of abstract classes, one may define the class LIST in Figure 4.5 as an abstract class. The definition of the abstract class LIST is given in Figure 4.10. The class ARRAY_LIST provides the definition of the pure virtual functions using

an array implementation. Functions `entry` and `enter` of the class `ARRAY` are used to manipulate elements of `ARRAY_LIST`. The abstract class `LIST` provides alternatives for list implementation.

```

class ARRAY_LIST : public ARRAY, public LIST
{ public:
    void go(int); // Go to the i_th entry
    int i_th(int); // Return the i_th entry
    int change_i_th(int, int); // Change the i_th entry
    void swap(int); //Swap positioned and i_th entries
};

int ARRAY_LIST::go(int i) // Go to the i_th position
{ // move cursor to the i-th position }

int ARRAY_LIST::i_th(int i) // Return the i_th entry
{ if (1 <= i) || (i <= nb_elements) error _msg(5);
  else { mark; go(i);
        return entry(i); } }

int ARRAY_LIST::change_i_th(int i, int v)
{ if (1 <= i) || (i <= nb_elements) error _msg(5);
  else
    { mark; go(i); enter(i,v); } }

int ARRAY_LIST::swap(int i) // swap entries
{ // swap the i_th entry the current positioned entry }

// ... Other redefined inherited methods from class list

```

Figure 4.9: Class `ARRAY_LIST` inherits from both `ARRAY` and `LIST` classes

```

class LIST
{ public:
    // ...
    virtual void go(int) = 0; // Go to the i_th entry
    virtual int i_th(int) = 0; // Return the i_th entry
    virtual int change_i_th(int, int) = 0;
    // Change the i_th entry
    virtual void swap(int) = 0;
    //Swap positioned and i_th entries
    // ...
};
// ... Definitions of non-pure virtual functions.

```

Figure 4.10: The definition of the abstract class `LIST`

4.4 Discussion

In this section we contrast C++ and Eiffel in terms of inheritance and related issues as approaches to reusability and extensibility. The notion of abstract classes (along with the notion of pure virtual functions) in the new release of C++, and the notion of deferred classes in Eiffel have significant role in using the inheritance issues to support reuse. They allow users to partially implement abstract data types in order to provide different implementations of data types. Pure virtual function and deferred routines also provide more options for the definitions of functions. Such notions are not provided by most of other OOPs.

The notion of inheritance in C++ can be viewed as an is-a relationship between base classes and derived classes where subtype/supertype relationship may not be appropriate [Danforth 88]. For instance, in Figure 4.6, the class STACK is a list but not a subtype of the class LIST. On the other hand, the notion of inheritance in Eiffel implies behavioral subtyping among classes. The class Y is a subtype of the class X if and only if the class Y is a descendant of the class X. Here, the subtype inherits all of the superclass' features and provides the same behavior. For instance, in Figure 4.3, the class STACK[T] is a subtype of the class EFFECTIVE_LIST[T] and provides the same behavior.

In the case of multiple inheritance, there is a high possibility for the occurrence of naming conflicts. The solution provided for such conflicts is different from one language to another. In C++, classes may have same-named functions. Ambiguity arises when same-named function have the same visibility level (private, protected, or public) in their classes. Moreover, in the case of virtual base classes, a derived class may refer to an ancestor class more than once through its base classes. Thus, more than one copy of an ancestor class may exist in the derived class and ambiguous access to the ancestor class may arise. C++ and Eiffel involve the programmer in the solution and leave to

him/her the responsibility of avoiding such clashes. However, their approaches are different. While C++ provides the resolution operator, Eiffel provides a renaming mechanism to solve name clashes. Eiffel's approach allows users to provide the proper names of inherited features in the descendant class(es) without having to define new features. For example, the names `push` and `top` in Figure 4.3 are more appropriate stack terminologies than the names `add_new` and `value` used in the class `LIST` (Figure 4.5).

C++ and Eiffel implement inheritance differently. C++ allows classes to simply inherit the desired features from their base class(es) and reject the undesired ones. Unlike C++, Eiffel's descendant classes cannot reject undesired features, a descendant class must override undesired features by introducing new definitions. Eiffel follows this approach to maintain its Open-Closed principle as an approach toward reusability.

A significant advantage of the export mechanism in Eiffel is that users can relate a group of classes together by exporting features to certain group of classes (by indicating the destination in the export clause) and not to others. For instance,

```
class LIST[T] export
    is_empty {FIXED_LIST, LINKED_LIST, STACK},
    push {STACK}, pop {STACK},
    ...
```

allows to group the classes `FIXED_LIST[T]`, `LINKED_LIST[T]`, and `STACK[T]` together through sharing the function `is_empty`. Thus, certain features of a class are available to related classes and not to other classes. A higher structuring level can be achieved by using this mechanism. However, C++ does not provide such a feature and all members are left equally available to all other classes. However, by using the friend declaration, users can provide private functions to specific classes and then relate them to their superclass through their friend relationship.

The presence of export control and granting descendant classes full access to superclass(es) in Eiffel may result in side-effects. A descendant class may export features inherited from other classes that were private and may hide inherited features that were

public in the superclass(es). Such side-effects compromise the visibility rule applied to the class contents. C++ takes a different approach, derived classes cannot compromise the visibility of the base class(es) members. This approach maintains support for information hiding and visibility rules, and it comes as a result of providing restricted external interfaces to derived classes.

To maintain the visibility rules, C++ provides its users several degrees of visibility of the class contents. These options add simplicity for structuring a class and provide reliable external interfaces for derived classes. From Eiffel's view, applying restrictions on the external interfaces violates its Open-Closed principle. This principle facilitates the construction of reusable software segments, but it violates the information hiding and encapsulation rules since accessing the superclass contents has a major impact on these issues.

Dynamic binding is one of the features that support polymorphism. While polymorphic entities refer to different instances of different classes at run time, dynamic binding provides the support for realizing polymorphism. C++ provides dynamic binding for virtual functions to support overloading of functions. That is, a class decides which of its functions need to be (and must be) redefined in the derived class(es). On the other hand, Eiffel grants dynamic binding for the whole class to maintain the Open-Closed principle in which a class should remain open for extension. C++ requires that virtual functions to be defined in the original defining class; while pure virtual functions (as well as Eiffel's deferred routines) do not need to have effective definitions in the original class.

Genericity (parameterized polymorphism) and overloading are other approaches to reusability. They imply symmetric functionality. Genericity provides a code fragment of a data structure that applies to different types. Overloading provides different code fragments (implementations) of the same data structure. In Eiffel, genericity is compatible with inheritance to support reusability and provide flexibility. In C++ overloading (ad hoc polymorphism) is achieved by virtual functions. Genericity is limited by the use of macros

and requires a great deal of experience with the language to be utilized.

The notion of repeated inheritance is unique to Eiffel. It may lead to replicated methods if a method has been renamed along the inheritance path in which its code will be duplicated in the inheriting class and might lead to ambiguity [Meyer 88a].

4.5 Summary

As more complex systems are being built, the significance of software reuse is further emphasized by practitioners and researchers. OOP provides support for code reuse. Inheritance and polymorphism are major contributors to reusability. In this chapter we examined these concepts as approached in C++ and Eiffel. The two languages address those issues differently and provide different perspectives of reusability.

As an approach to reusability, inheritance techniques have different interpretations and purposes in C++ and Eiffel. In C++, inheritance technique and visibility rules do not compromise data hiding and provide reliable external interfaces for derived classes and clients. In Eiffel, the inheritance technique and access mechanism compromise the data hiding in order to accommodate the Open-Closed principle. Inheritance and information hiding are parallel in C++; while orthogonal in Eiffel (a descendant classes may hide exported methods, and may export inherited private methods).

Polymorphism is viewed and implemented differently in the two languages. The C++ ad hoc polymorphism provided by overloaded operations serves the syntactic issue. It is implemented in terms of pointers to functions that can be applied to objects of different classes. Eiffel incorporates parameterized polymorphism and provides polymorphic entities that refer to different types of objects during execution.

Naming clashes due to multiple inheritance are also handled differently. Redefinition of inherited methods is provided in both languages. Eiffel grants descendant classes the ability of redefining any inherited method. C++ grants the base class the

control to decide which methods can be redefined in the derived classes using the keyword `Virtual`. Redefinition supports polymorphism and adds more flexibility by allowing classes to provide different definitions for the same method.

Structured libraries of subroutines participate partially in the solution of reusability. Both languages provide libraries of subroutines to be used in different situations, and tools for the construction of custom libraries for specific applications. Eiffel can directly use code written in other languages such as C, and it generates portable C packages. The macro facility in C++ supports reusability. Moreover, the type-safe linkage scheme provided by the 2.0 release of C++ grants users the ability of accessing functions in libraries of other languages, and it supports reusability through construction and combination of libraries.

CHAPTER V

AN OBJECT-BASED INHERITANCE MODEL

5.1 Introduction

Despite the obvious advantages, using the notion of inheriting classes to construct new components may introduce some problems; for instance, exposing the class implementation to the inheriting classes, whether or not to include instance variables in the external interface, and exposing the use of inheritance in the ancestor classes. Snyder [Snyder 86] examined the relationship between inheritance and encapsulation, and outlined a criteria for full support of encapsulation with inheritance. He outlined the following requirements:

- 1) Providing separate external interfaces for the class objects and the inheriting classes.
- 2) Restricting the inheriting classes' external interface by not exposing the instance variables and implementation details of the class outside the class definition.
- 3) Providing methods for the instance variables to be accessed by the inheriting classes.
- 4) Hiding the use of inheritance by not making it a part of the external interface.
- 5) When using inheritance for code sharing, it is adequate to grant the inheriting classes the ability to exclude inherited methods from the external interface (i.e., classes inherit what they need).
- 6) The ability to define private methods for the benefit of inheriting classes when the methods are inappropriate for some instances of a class.
- 7) The ability of directly accessing non-inherited parent's methods and redefined inherited methods.

Current inheritance models and OOPLs provide only partial solutions to these problems and do not provide full support for encapsulation either. Accessing the instance variables and the visibility of inheritance are important issues in this situation.

OOPLs handle inheritance differently based on their interpretation of the inheritance concept, and therefore different approaches to inheritance are found in the literature. These approaches are outlined in chapter 2. The inheritance model incorporated in each language is influenced by the particular inheritance approach adopted by the language. The restrictions placed on these models have prompted researchers to search for better models (e.g., see [Hailpern 87] and [Ungar 87]). In this chapter, a new model that unifies ideas from several existing models is proposed. The new model in addition to and probably in spite of being relatively general is clear and easy to understand.

The proposed Two-faceted object-based Inheritance Model (hereafter referred to as TIM [Al-Haddad 90b]) satisfies most of the requirements mentioned earlier, and provides full support for encapsulation and hiding the use of inheritance. The proposed model provides code sharing inheritance mechanism based on objects rather than classes. TIM also provides a unified approach to inheritance with encapsulation. TIM can be regarded as a generalization of Hailpern and Nguyen's model [Hailpern 87] and it is also consistent with the requirements outlined by Snyder [Snyder 86].

The rest of the chapter is organized as follows: Section 2 provides a detailed description of the proposed model. Simulation of a C++ [Stroustrup 86] example in TIM is provided in Section 3. In Section 4 we compare our model against other models found in the literature. Section 5 is the summary and concluding remarks.

5.2 The Proposed Object-Based Inheritance Model (TIM)

We take a two-faceted orthogonal approach to the design of a unified object inheritance model [Al-Haddad 90 a,b]. We call this new model a Two-faceted object-

based Inheritance Model (TIM). Each object in TIM consists of instance variables, procedures, and methods. As in the case of Hailpern and Nguyen's model [Hailpern 87] and Self [Ungar 87], there are no classes in TIM. Classes and the class hierarchy can be derived from the objects. TIM provides a unified approach to inheritance and subtyping.

In the rest of this section we address the following issues: semantics and composition of an object, the internal structure of an object, object interfaces, message passing technique, multiple inheritance, object creation and deletion, and the inheritance hierarchy among objects.

TIM consists of two orthogonal sets of objects. The first set of objects consists of sets of identical objects, hereafter referred to as M-objects (Member objects). Each set of M-objects have a copy of the variables and methods (access methods) of their parent object. An M-object inherits all methods (except access methods) of its parent object and maintains its own copy of the instance variables. The initial state of an M-object is the current state of its parent object at creation time. The concept of an M-object eliminates the distinction between classes and objects in the traditional sense.

The second set of objects, hereafter referred to as I-objects (Inheriting objects), are objects with a new identity that are not members of other objects. I-objects define new methods and variables. While the behavior of M-objects is similar, the I-objects can have new behavior and can inherit the behaviors of one or more M-objects and/or I-objects. M-objects are analogous to instances of a class in the traditional inheritance models; while I-objects are analogous to inheriting classes of a class.

Despite the similarities of this model to the class-based models, there are significant differences. Inheritance is object-based rather than class-based, which means that an I-object can inherit from any M-object, and an M-object can be derived from an I-object. In class-based models, classes are static templates for dynamic object creation. In TIM, the distinction between static and dynamic entities is removed; a parent object provides its current state (values of the variables), variables, and methods to other objects.

Thus, in TIM, a chain of objects may represent a real process where each object is created from the previous one.

Traditional models provide different kinds of variables in which several scoping rules are needed. In TIM, private variables are associated to procedures. Thus, having only instance variables simplifies the scoping rules. Figure 5.1 illustrates the possible inter-relationships between the two types of objects in TIM.

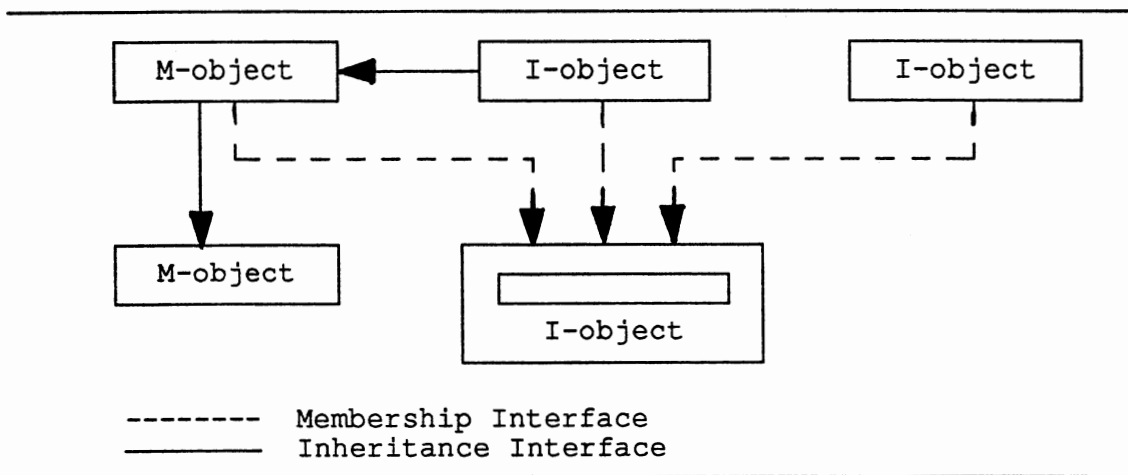


Figure 5.1: Object types and possible interfaces in TIM

To categorize the notion of visibility, we introduce the concept of "degrees of visibility". In TIM, three levels of visibility are distinguished:

- 1) **Level-1 variables:** Invisible variables, where neither the name nor the value associated to the name is visible outside the object.
- 2) **Level-2 variables:** Partially visible variables, where variable names are not visible outside the object but their values can be accessed via defined methods.
- 3) **Level-3 variables:** Visible variables, where both variable names and values are visible to the descendants of the defining object.

5.2.1 Semantics and Composition of an Object

Here, the semantics of object elements including variables, procedures, and

methods are introduced. The visibility levels of object variables are also discussed.

Private Variables: Private variables are related to the procedures of an object rather than to the object itself. They are in effect the local memory of a procedure. They are level-1 variables and are only accessed by the procedures in which they are used.

Instance Variables: Instance variables are level-2 variables. Their values reflect an object's state during its life. An access method is associated with each instance variable. Syntactically, "Access\$" is prefixed to a name to indicate the access method associated with that name. For example, Access\$a is a method to access the instance variable a. Access to instance variables is limited to methods defined in the object and the access methods.

Objects can inherit values of instance variables through their access methods (see methods below) and they may have same-named variables. Therefore, this scheme supports the following features:

- 1) Instance variables are not visible outside an object and are accessed through defined methods.
- 2) Inheriting objects will never complain about undefined methods for accessing inherited instance variables. Since each variable is associated with an access method, the values of instance variables (the states) are available to inheriting objects.

Procedures: Procedures are the executable code bodies of the defined methods. A procedure accesses the instance variables of the object (either a defining or an inheriting object) executing the procedure. When a method is to be executed in response to a message, the associated procedure is obtained and executed by the requested object. Procedure execution, as a consequence of objects responding to messages, may change the requesting object's state. Procedure names are level-1 variables.

Methods: Since objects contain private and instance variables, two groups of methods are defined to provide different external interfaces that support encapsulation and maintain the invisibility of the instance variables outside an object. An object provides the

following methods:

Access Methods: Access methods are created during object creation. Each instance variable is associated with an access method. Access methods are part of the external interface provided for I-objects and intended to enforce level-2 visibility. They do not perform computational tasks and do not use private variables. When creating an M-object, the new object includes a copy of the access methods to be used later by its I-objects.

Computational Methods: This group of methods, which accomplishes a certain computation, is defined on both instance variables of an object and the private variables of the associated procedures. An I-object may define or inherit a method. Defined (original) methods are associated with executable code bodies (procedures) which are defined during object creation. Inherited methods have their executable codes in the defining object(s). Computational methods are included in the external interface provided for both M- and I-objects. Defining objects return the procedures associated with the requested methods to the requesting objects (M- or I-objects) where the code is executed. Method execution results in accessing the private variables of the associated procedure and the instance variables of the executing object. Since objects may have same-named methods, full-name reference is used to avoid naming conflicts. For instance, X.Access\$b means the access method of variable b defined in object X.

5.2.2 Internal Structure of an Object

Each object in TIM can contain several lists. These lists are illustrated in Figure 5.2, and are defined as follows:

- 1) Instance variables list: to store an object's instance variables.
- 2) Methods list: to store all methods specified in the creation message. Each entry of the list includes: method name, method form (original or inherited), and for inherited methods, the source object from which the method is inherited. When

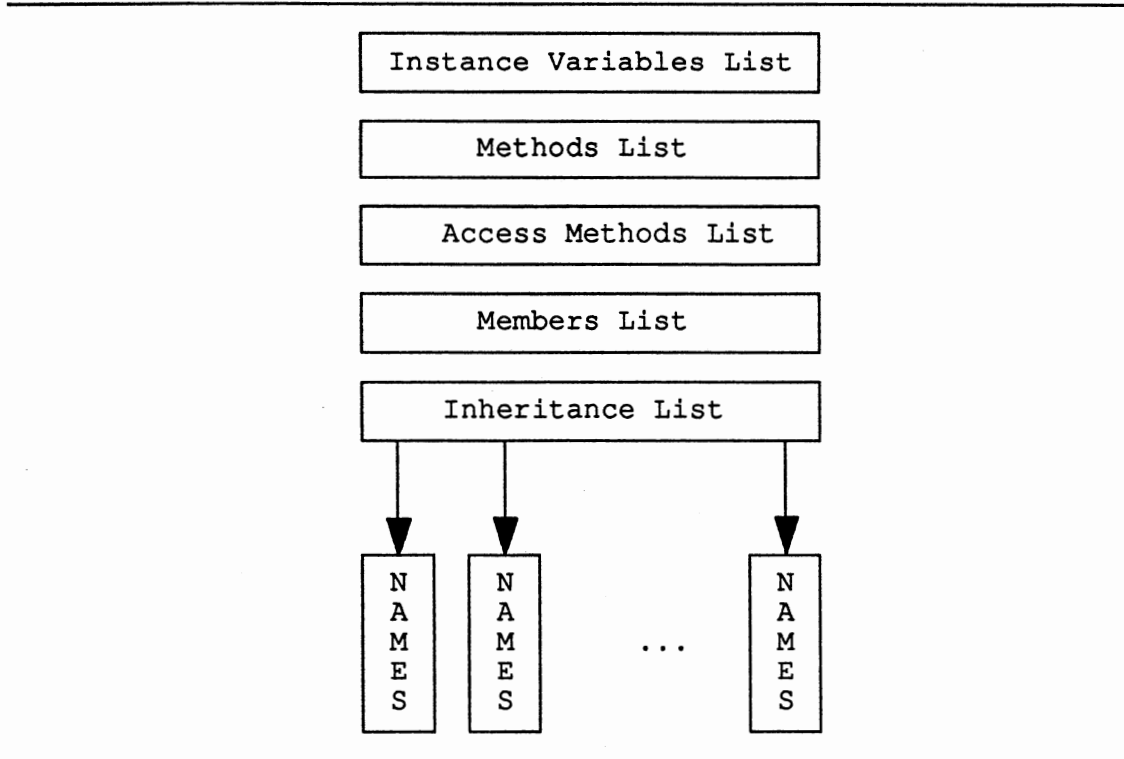


Figure 5.2: Lists that an object can contain

sending a message to another object, the methods list explicitly indicates the type of the message (Request or Inherit) and implicitly indicates the destination object.

- 3) Access methods list: to store the access methods defined for each variable in the instance variables list.
- 4) Members list: to store the names of all related M-objects. This list is used to distinguish M-objects from I-objects when receiving a request for a method.
- 5) Inheritance list: to store the inheritance information of descendant objects. Each entry is associated with a list of inherited methods names. This list is used to indicate whether or not a requesting object for a method is an I-object, and what methods it inherits.

5.2.3 Object Interfaces

In TIM different interfaces are provided for M- and I-objects. These interfaces are illustrated in Figure 5.1. Each object can have two interfaces:

- 1) Membership interface which provides all computational methods defined and inherited in the object to all of its M-objects. An M-object can have only one parent.
- 2) Inheritance interface which provides all methods of the membership interface in addition to the access methods. This interface allows I-objects to inherit both methods and instance variables from ancestor objects. An I-object may have several parents.

5.2.4 Message Passing Techniques

Objects in TIM are communicating processes. Any object can send a message to any other object. The receiving object either returns the associated procedure of the requested method (if available) or passes the message to the object which the method is inherited from; otherwise, it returns an error. Any object can request a method by sending either of the following messages:

```
REQUEST (Source, Method)
INHERIT (Sender, Source, Method_name)
```

where Source is the requesting object, Sender is the sending object, and Method_name is the requested method. The receiving object whose name is used as a qualifier responds by executing the following algorithm:

Algorithm Method retrieval

```
IF the request does not include the receiving object's name
THEN RETURN "ERROR: Unknown object" & EXIT.
```

```
IF the requested method is original
THEN RETURN its procedure to the source object & EXIT.
```

```

IF the requested method is inherited
THEN send the following inheritance message to the
    ancestor object:

```

```

    INHERIT (Sender, Source, Method_name)

```

```

ELSE RETURN "ERROR: Unknown method" & EXIT.

```

5.2.5 Multiple Inheritance

Single inheritance is a special case of multiple inheritance. All of the inherited methods and instance variables come from one parent object. The methods list provides multiple code inheritance by allowing objects to inherit the desired methods from several objects and disallowing unsuitable methods. An object is able to answer several requests at a time and activate several procedures simultaneously since private variables are local to the procedures and not to the object itself.

An object is not allowed to have same-named methods, but different objects may have same-named methods. When two or more parent objects have same-named methods, the I-object will have several methods with the same name. To avoid this conflict, as mentioned previously, full-name method reference approach is used. The "Source" attribute in each entry of the methods list illustrated in Figure 5.3 refers to inherited methods using both the object name and the method name. An I-object can rename its inherited methods, but the methods list maintains the original names of the inherited methods.

Method	Form	Source
m2	Inheritance	X.m1

Figure 5.3: A methods list entry

When objects send a **REQUEST** or an **INHERIT** message, they use the original name of the requested method used in the receiving object. For example, suppose that object X provides method m1 and object Y inherits m1 using a different name, say m2. The methods list entry of m1 in object Y is depicted in Figure 5.3. Object Y may send either of the following messages to object X:

REQUEST (Y, X.m1) or **INHERIT** (Y, Z, X.m1)

The receiving object, X, expects to find its identifier prefixed to the requested method, otherwise it does not respond to the message. The above interpretation of multiple inheritance supports the following features:

- 1) Methods (original and inherited) have distinct names in an object in order to avoid naming conflicts.
- 2) The use of inheritance inside an object is hidden.
- 3) Each method is associated with only one parent object.
- 4) Changing the implementation of a method does not impact the inheriting objects as long as the ancestor object provides the same inheritance external interface.

5.2.6 Object Creation and Deletion

Every object includes a specialized method for creating a new object upon receiving a creation message. The creation message has the following general format:

CREATE (Destination, New, IV, Methods)

where	Destination	: The receiving object,
	New	: The new object name,
	IV	: Set of instance variables, and
	Methods	: Set of method names.

Since there are two types of objects in TIM, the scheme for object creation needs to be specialized for each type.

- 1) Creating an M-object of a destination object: Upon receiving the message

CREATE (Destination, New)

the destination (parent) object creates a new M-object by copying all of its own variables and access methods. The initial state of the new object is the same as the current state of the creator at creation time. In other words, this object reproduces itself. All computational methods are inherited (not copied) from the parent object. In class-based models, new objects are not provided with initial states since classes are merely templates for objects.

2) Creating an I-object with a new identity: The destination object should receive the message:

```
CREATE (Destination, New, IV, Methods)
         (Definition of new methods)
         (Sources of inherited methods)
```

The result of receiving the message is that a new I-object with the specified contents will be created. An access method is created for each defined instance variable but the access methods are not specified in the argument "Methods". The private variables of a procedure are defined during the definition of the corresponding new method. The destination can be any object in the system.

The newly created object's state is based on the variables' initialization and the state of its parent object(s) at the creation time. This is a major difference between class-based inheritance and TIM. The reason behind this approach to creation is to take advantage of having a creation method associated with each object. The advantage is that users need not keep track of the objects to send creation messages. Any existing object can perform the creation since the creator does not have to contribute to the contents of the created object. In the case of multiple inheritance, a creation message can be sent to any object in the system and not necessarily to the parent object(s).

In the first scheme, a clone of the destination object is created; while in the second scheme a new object is created which may contain properties inherited from the creator as well as methods inherited from other objects. The following example illustrates what a complete creation message may look like in the second method (without emphasizing

the syntax).

```
CREATE (X, Y, {a, b}, {m1, m2, m3})
DEF Y.m1 : BEGIN {The code body} END;
Y.m2 = Z.m2
Y.m3 = Z.m1
```

The creation message is sent to object X, m1 is a new method and m2 is inherited method from object Z. As a result of this message a new object Y will be created. Definition of method m1 would correspond to the above **DEF** statement. An assignment-like operator "=" is used to indicate inheritance among objects and renaming the inherited methods. Therefore, Y.m3 = Z.m1 indicates that object Y inherits and renames method m1 from object Z).

Dependent objects are objects that depend fully or partially on the methods and/or variables of an ancestor object (namely, an M- or I-object). Semantics of object deletion is given below.

- 1) Each object is associated with a destruction method that can be invoked by itself.
- 2) An object destroys itself when it has been processed and has no dependent objects of either kind.
- 3) If there are dependent objects, the object remains active serving its dependent objects.
- 4) A deleted object answers to messages sent only by its dependent objects and ignores other messages. An object recognizes messages sent by its dependent objects based on its own members and inheritance lists.
- 5) Any object may send the message **DELETE**(object_name) to any/all of its dependent objects. This message is accepted if the receiving object is related to the sending object as an M- or I-object. This message deletes the receiving object and all of its dependent objects. In the case of multiple inheritance, if a parent of an object is deleted, this object decides to stay active or deletes itself.
- 6) The specification of the object behavior dictates when to issue the **DELETE**

message. Objects may use this message when an object (process) and all of its dependent objects must be terminated for some reason. Upon deletion of an object, the parent object(s) will remove the object's name from its (their) members list(s). Likewise, the ancestor object(s) will remove the object's name from its (their) inheritance list(s).

5.2.7 Inheritance Hierarchy

The class/subclass hierarchy can be derived directly from the object lists as illustrated below by an example in Figures 5.4 and 5.5. Figure 5.4 depicts an example of inheritance hierarchy in TIM.

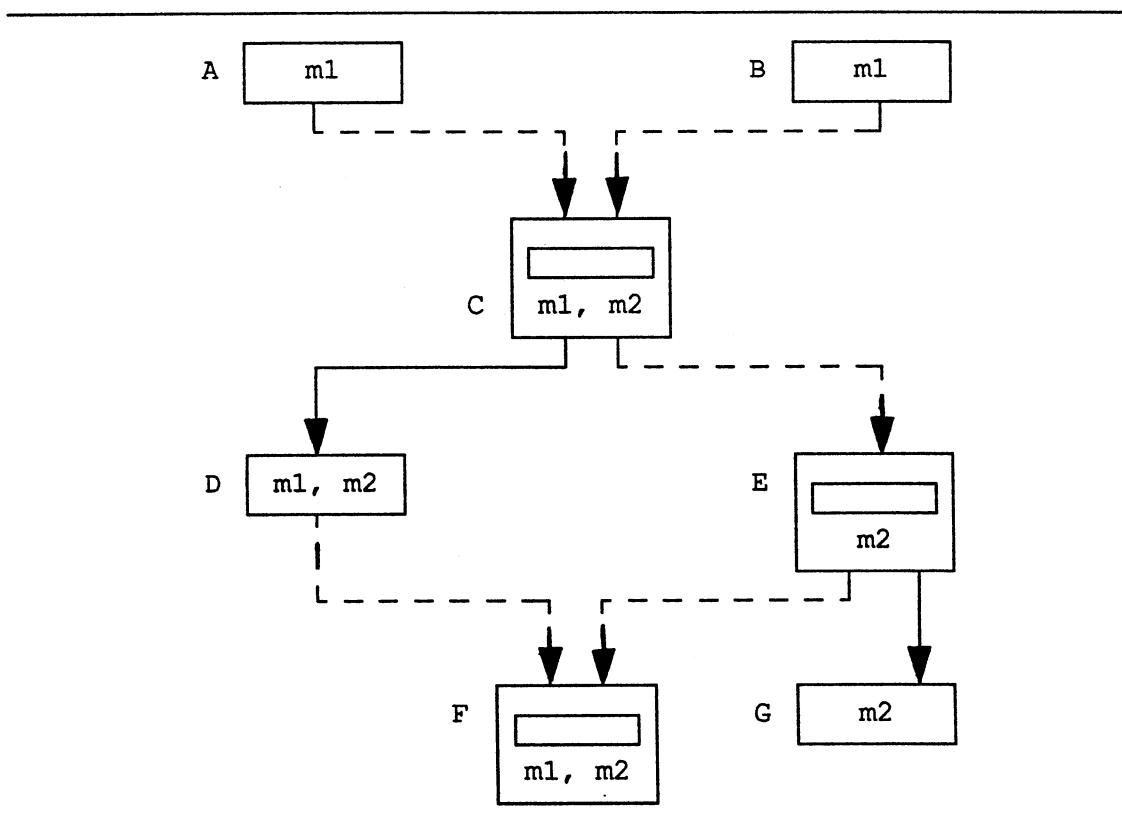


Figure 5.4: An example of inheritance hierarchy in TIM

In Figure 5.5, the inheritance list indicates the inheritance information of the objects in Figure 5.4 and the methods they inherit. Thus, I-objects may be viewed as subclasses and their parent objects as superclasses. The methods list in Figure 5.5 provides all of the inherited methods and the ancestor objects from which they are inherited.

Object	Methods List			Inheritance List	
	Method	Form	Source	Object	Method
A	m1	Original	NIL	C	m1
B	m1	Original	NIL	C	m1
C	m1	Inherited	A.m1	E	m2
	m2	Inherited	B.m1		
D	m1	Inherited	C.m1	F	m2
	m2	Inherited	C.m2		
E	m2	Inherited	C.m2	F	m2
F	m1	Inherited	D.m2	G	m2
	m2	Inherited	E.m2		
G	m2	Inherited	E.m2		

Figure 5.5: Methods and inheritance lists of the objects in Figure 5.4

Objects in Figure 5.4 inherit methods as follows. Object C inherits m1 from A and B. It renames method m1 from B. That is, C.m1 = A.m1 and C.m2 = B.m1. Object D is

an M-object of object C It implicitly inherits all of the computational methods in object C. On the other hand, object E inherits m2 from object C. That is, $E.m2 = C.m2$. Object F inherits and renames m2 from object D, and it inherits m2 form object E. That is, $F.m1 = D.m2$ and $F.m2 = E.m2$. Object G is an M-object of object E.

Figure 5.5 shows both methods and inheritance lists of the objects illustrated in Figure 5.4. Objects in Figure 5.4 exchange messages as follows. Suppose that object G requests m2 by sending the message

REQUEST (G, E.m2)

to object E. Object E executes the method retrieval algorithm described early and sends the message

INHERIT (E, G, C.m2)

to object C (destination object). Object C executes the same algorithm and send the message

INHERIT (C, G, B.m1)

to object B. Since m1 is originally in object B, object B returns the procedure associated with m1 to object G. The "Source" attribute of the Methods list remembers the original method name in the parent object.

5.3 Examples Represented in the Proposed Model

In this section, a demonstration, by means of an example, of how objects in the sense of other models can be realized in TIM is given in this section. In this example, a C++ program is simulated in TIM. The original code is followed by the appropriate simulation including object creation, methods definitions, and object contents. Here, the class BIRTH_DAY is defined as a public derived class of the class DATE adopted and modified from [Stroustrup 86]. It inherits method "print" from the class DATE and

defines new methods and variables. The main program creates the instances Today and Christmas of the class DATE, and the instance My_birthday of the BIRTH_DAY. Definitions of these classes are illustrated in Figure 5.6.

```

class DATE
{ public:
    int month, day, year; // public variables
    void next(); // next day
    void print(); // print date
};

void DATE::next()
{ if (++day > 28)
    { /* print next month's schedule */ }; }

void DATE::print()
{ cout <<month<< "/" <<day<< "/" <<year<<; }

class BIRTH_DAY::public DATE
{ char* name; // private variable
  int age; // private variable
public:
    birth_day(char, int, int, int); // constructor
    int compute_age(int,int,int); // newly defined method
};

int BIRTH_DAY::compute_age(month,day,year)
{ /* subtract birthday form today's date
and return age. */ }

main (int m,d,y)
{ DATE Today(3,30,1992); // instance Today
  DATE Christmas(12,25,1991); // instance Christmas
  BIRTH_DAY My_birthday('Al',1,4,1964); // instance My_birthday
  Today.print(); // print today's date
  Christmas.next(); // schedule for January
  My_birthday.compute_age(3,30,1992); // return my age
};

```

Figure 5.6: Definitions of the classes DATE and BIRTH_DATE

In TIM, the object DATE is treated as an M-object and the object BIRTH_DAY is treated as an I-object. To create the object DATE, one can send the following message to any destination object in the system (say Dest):

```

CREATE (Dest,DATE,{month <-- 0,day <-- 0,year <-- 0},
        {next,print})

```

with the following newly defined methods:

```

DEF DATE.next: BEGIN
    IF (day +1) > 28
    THEN BEGIN
        /* print next month's schedule */
    END;
    END;
DEF DATE.print: BEGIN
    RETURN (month, "/", day, "/", year);
END;

```

The instances Today, Christmas, and My_birthday created in the main program can be derived as M-objects of the object DATE using the following messages:

```

CREATE (DATE, Today)
CREATE (DATE, Christmas)
CREATE (DATE, My_birthday)

```

We can simulate class BIRTH_DAY by sending the following message to a destination object

```

CREATE (Dest, BIRTH_DAY, {name, age <-- 0, month <-- 0,
    day <-- 0; year <-- 0}, {print, compute_age})

```

The instance My_birthday can be created using the following message

```

CREATE (BIRTH_DAY, My_birthday)

```

The C++ instances Today, Christmas, and My_birthday become M-objects in TIM. They inherit all methods of their parent objects. Thus, there are no methods definitions needed for these objects. However, object My_Birthday is an I-object. Therefore, we need to define all of its methods. Those methods can be defined as follows:

```

BIRTH_DAY.print = DATE.print /* inherited from DATE */
DEF BIRTH_DAY.compute_age(month, day, year):
    BEGIN
        /* subtract the given date from today's date
        and return the result */
    END;

```

If necessary, the instance variables can be initialized in the creation message. For illustration, the internal structures of some of these objects are viewed as shown below:

```

M-object DATE:
Object                : DATE
Instance variables List <-- {month, day, year}
Methods list          <-- {next, print}
Members list          <-- {Today, Christmas}
Inheritance list      <-- {BIRTH_DAY [print]}

```

```

M-object Christmas:
  Object           : Christmas
  Instance variables List <-- {month,day,year}
  Methods list     <-- {next,print}
  Members list     <-- {}
  Inheritance list <-- {}

I-object BIRTH_DAY:
  Object           : BIRTH_DAY
  Instance variables List <-- {name,age,month,day,year}
  Methods list     <-- {print, compute_age}
  Members list     <-- {My_birthday}
  Inheritance list <-- {}

M-object My_birthday:
  Object           : My_birthday
  Instance variables List <-- {name,age,month,day,year}
  Methods list     <-- {print,compute_age}
  Members list     <-- {}
  Inheritance list <-- {}

```

5.4 Discussion

The closest inheritance models to TIM are the ones defined by Hailpern and Nguyen [Hailpern 87] and Ungar and Smith in the language Self [Ungar 87]. Hailpern and Nguyen described a network of objects communicating by utilizing a message passing mechanism. In their model, each object consists of private variables, a set of methods, and a set of procedures. Self is an object-based programming language. It provides a set of objects that communicate by utilizing a message passing mechanism. Objects in TIM consist of private and instance to variables as well as methods and procedures.

TIM supports most of the Snyder's requirements and maintains consistency with the other related issues such as encapsulation and visibility of inheritance. Snyder's second and third requirements outlined in Section 1 are not applicable in Hailpern and Nguyen's model since their objects do not include instance variables. Furthermore, in their model they relate the private variables to the defining object in which they can handle only one message at a time. TIM avoids such use of private variables.

In Hailpern and Nguyen's model, list-based multiple inheritance may lead to

naming conflicts among the inherited methods. When an object X inherits a method, m1, from different objects, the inheritance list associated with m1 in object X contains the names of the objects where m1 is inherited from. Suppose another object Y inherits m1 from object X, when object Y sends a message to object X requesting m1, object X does not know which method is requested.

To avoid the naming conflict problem, we suggest two options. The first option is restricting all objects to have distinct method names in the system. This option limits the designer's freedom to use same-named methods in different objects. The second option, which is used in TIM (also used in Eiffel [Meyer 88]), allows objects to rename inherited method and has the ability to remember the given methods' name in the parent objects. This option avoids the above restriction and maintains consistency with data hiding, hiding the use of inheritance, accessing the instance variables, and excluding inherited methods from the external interface. The second approach also avoids the possibility of having same-named methods in different objects.

In order to facilitate multiple inheritance, Hailpern and Nguyen's model artificially includes several copies of "SuperClass" variables with different values. TIM does not use "SuperClass" variables since ancestor objects can be derived directly from the methods list of the inheriting objects.

Some of the feature of TIM are similar to those found in Self. Both TIM and Self eliminate the distinctions between classes and objects. The creation of M-objects and the search mechanism for methods are also similar. However, there are significant differences also. Self does not distinguish between state and behavior. Instead, they are unified into the notion of slots. Therefore, it is not possible to hide the state of an object, hence violating the principle of information hiding. TIM takes a different approach and maintains information hiding.

As a model, TIM does not have the attributes "static" or "dynamic". In TIM, behavior and state are considered conceptual attributes of an object. Also, renaming a

behavior is another feature of TIM that is not found in Self. In Self, any object can alter the state of any other object. In TIM, only the object itself can change its state. Each object has associated with it construction and destruction methods. The notions of creation and deletion have different semantics from other models.

5.6. Summary

Inheritance models in well-known OOPs have different deficiencies in their support for issues such as information hiding, subtyping, and visibility of inheritance. To achieve the goal of OOP style more faithfully, we need an inheritance model which maintains consistency with those other issues [Snyder 86] [Wayne 89].

Upon analyzing the current models, we gleaned their advantages and proposed a unified approach by defining a Two-faceted object Inheritance Model (TIM). TIM combines the selected features to maintain consistency with the above-mentioned related issues; while adding new features to obtain inheritance and support for the OOP style and reuse.

TIM consists of two orthogonal sets of objects. M-objects contain the same instance variables and computational methods of their parent objects. I-objects, on the other hand, inherit behavior, variables, and methods from other objects, they can define new variables and methods as well.

TIM is an object-based inheritance model based on code sharing with the ability to capture the other views of inheritance. It provides single and multiple inheritance based on the message passing paradigm, and provides semantics for object creation and deletion. The notion of visibility of instance variables was categorized and used in the definition of TIM.

CHAPTER VI

A FEEDBACK INHERITANCE MODEL

6.1 Introduction

In practical applications there are many situations where dependency between two objects requires sharing of properties belonging to each other. Such situations cannot be easily represented using the hierarchical inheritance model. Therefore, to adhere to the object-oriented paradigm, users may attempt to use complicated and inefficient approaches to model mutual dependency among objects. In such cases, inefficient and highly expensive software may result due to the lack of control over the dependency and flow of information among objects [Parnas 76].

Parnas et al. [Parnas 76] described a similar situation (in the context of abstract types defined as of classes of variables) as follows:

Our position is that representation of dependent programs will be written whenever cost considerations demand it; it is better to provide a mechanism that allows the control of such dependency than to force the programmer to use dirty tricks.

A mechanism that allows control over mutual dependency among objects makes the programming task simpler and provides more understandable and maintainable software in addition to being a better reflection of the problem. The necessity to generalize the inheritance model is recognized by Pedersen also who defines an extension to include generalization of the inheritance system [Pedersen 83].

Maintenance adds another dimension to the problem because of the possibility of unexpected changes in relations [Freeman 83]. In this chapter, our goal is to define an

inheritance model that has the flexibility to represent real-life problems in a natural way and at the same time facilitate software design and accommodate software maintenance.

In the hierarchical inheritance model, when a superclass needs to use properties from a subclass, users can replicate these properties either in the superclass or in classes that can be reached by the superclass by means of inheritance. Users may also restructure the class system to achieve the objective, which would probably result in a complex and inappropriate class hierarchy. Thus, efficiency may be the victim of such a re-organization and/or data replication.

In this chapter the hierarchical model is relaxed by allowing superclasses to access the properties of their subclass. This idea is derived from real-life situations and models where mutual dependency among objects exists and needs to be controlled. The proposed model is called a "feedback inheritance model" [Al-Haddad 92b].

The rest of this chapter is organized as follows. Section 2 provides definitions and notations to be used in the description of the proposed model. Section 3 provides two motivating examples and their representations using the hierarchical model. Section 4 is a description of the proposed model. Section 5 provides representation of the selected examples in Section 3 using the new model. Section 6 is a discussion of the issues related to inheritance in the new model including information hiding, encapsulation, access mechanism, and visibility. Section 7 outlines the summary of the chapter and some concluding remarks.

6.2 Definitions

In this section we introduce definitions and notations to describe the specification of examples and the proposed feedback model. These definitions and notations are given below. For the sake of clarity, some earlier definitions are repeated.

- 1) A **method** is a name associated with a specification of behavior (routine).

- 2) An **instance variable** is a name associated with a data object (memory value).
- 3) An **attribute** is either a method or an instance variable.
- 4) **Value** of a method attribute is the implementation of that method. Values of method attributes represent a set of behaviors denoted by B . Value of an instance variable attribute is the value held by the instance variable at a certain time. Values of instance variable attributes represent a set of states denoted by S . The value of an attribute which is undefined is denoted by \perp .
- 5) A **domain** D of identifiers is a set of distinct attribute names. We shall denote attributes of classes by the lower case letters a, b, c , etc.
- 6) A **class** is a pair $(C, \{a_1, a_2, \dots, a_n\})$ where C is the class name and $\{a_1, a_2, \dots, a_n\}$ is a set of attributes such that $a_i \in D$ for $i=1, 2, \dots, n$. For notational convenience, a class is denoted by C and the set of attributes by $C.all$. A subset of the attributes of the class C is denoted as $C.[a_1, a_2, \dots, a_k]$ for $a_i \in C.all$, $1 \leq i \leq k$ and a_i is any attribute in $C.all$. For the sake of convenience, a class is represented as $C = \{a_1, a_2, \dots, a_n\}$. It should be noted that C is the name of the class and $C.all = \{a_1, a_2, \dots, a_n\}$. When there is no confusion, we may use C in place of $C.all$.
- 7) An **instance** of a class is an association of values to the attributes of the class. For example, an instance I of a class C (denoted by $I :: C$) with initial values $\{v_1, v_2, \dots, v_n\}$ is represented as

$$I :: C \implies \{a_1 \leftarrow v_1, a_2 \leftarrow v_2, \dots, a_n \leftarrow v_n\}$$
 where the value v_i is associated with the attribute a_i for $i=1, 2, \dots, n$; and $v_i \in S \cup B$.
- 8) Given the classes

$$C = \{a_1, a_2, \dots, a_m\} \text{ and } C_1 = \{b_1, b_2, \dots, b_n\},$$

$$C_1 \text{ is a \textbf{subtype} of } C \text{ (written } C_1 < C) \text{ iff } \{a_1, a_2, \dots, a_m\} \subseteq \{b_1, b_2, \dots, b_n\}.$$

The class C_1 is said to inherit the attributes of C . This is called **subtype inheritance**. On the other hand

$$C_1 \text{ is a \textbf{subclass} of } C \text{ iff } \{b_1, b_2, \dots, b_n\} - \{a_1, a_2, \dots, a_m\} \neq \emptyset.$$

A subclass is a class that fully or partially depends on other class(es). However, a subtype is a subclass that fully depends on other class(es). The term subtype denotes subtype inheritance; while the term subclass denotes the existence of inheritance relationship between two classes.

- 9) A **message** M received by an instance of a class is a request for the receiving instance to perform a specific action. A message M sent to an instance I of the class

$C = \{a_1, a_2, \dots, a_n\}$ is denoted by $I :: C \Leftarrow M[m, p_1, \dots, p_k]$

where m is the method name and p_1, \dots, p_k are actual parameters of the method.

There are situations where modeling using hierarchical inheritance may not give a natural representation of data flow among classes. In the next section we present examples in areas where hierarchical inheritance increases the complexity of modeling in terms of the number of inheritance relationships and the number of classes.

6.3 Two Examples and Their Hierarchical Representations

There are several areas, such as databases and networks, where the hierarchical inheritance model does not provide a natural correspondence between the problem and the model. In the database area [Alagic 89], the example shown in Figure 6.1 illustrates a typical relationship defined by record structure occurring in a relational database (in this case related to a university system).

```

TYPE Department = RECORD
    D_name      : string40
    Location    : string40
    F_members   : SetOfProfessors
END.

TYPE Faculty    = RECORD
    F_name      : string40
    Rank        : (assistant, associate, full)
    Department  : SetOfDeptIds
END.

```

Figure 6.1: Record structure of a relationship

In such a relation, a faculty member may belong to more than one department (joint appointments) and a department can have several faculty members. What this suggests is mutual dependency between the types Department and Faculty. Moreover, nested types may occur when an attribute of a record is defined as a type.

An object-oriented representation of the scenario in Figure 6.1 is shown in Figure 6.2. In this representation, a record structure in an object-oriented relational database is depicted in terms of classes and hierarchical inheritance. A base class contains attributes that represent relevant information of an object type. For instance, the type Department would be represented by the base class DEPARTMENT which contains the attributes D_name, Location, Degrees_offered, and College.

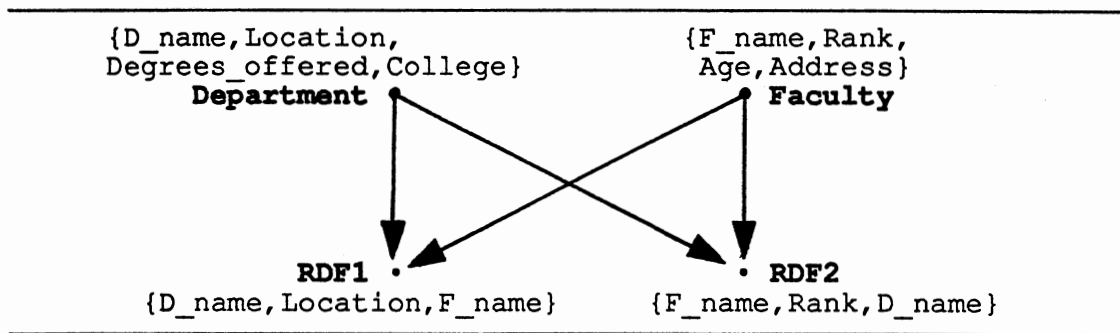


Figure 6.2: A hierarchical inheritance representation of the record structure in Figure 6.1

A relation defined on the base classes is represented by a class together with inheritance relationships and the base classes involved. For instance, as illustrated in Figure 6.2, a relation R1 on the base classes DEPARTMENT and FACULTY can be represented by a new class RDF1 (Relation-Department-Faculty-1) that inherits the attributes D_name and Location from the base class DEPARTMENT, and the attribute F_name from the base class FACULTY. The classes RDF1 and RDF2 in Figure 6.2 are equivalent to the relational types Department and Faculty defined in Figure 6.1. Instances of the class RDF1 are equivalent to tuples of the relational type Department. Examples of instances of class RDF1 are given in Table 6.1.

In the above hierarchical representation, every relation needs to be defined as a class inheriting from other classes (this seems to be the case in any hierarchical representation). Thus, the more relations we define, the more classes we need to define;

TABLE 6.1
INSTANCES OF THE CLASS RDF1

	Department	Location	Faculty
Instance1	COMPUTER_SC	MATH_SC_Bldg	f1, f2, f3
Instance2	CIVIL_ENG	ENGINEERING_Bldg	f2, f4
Instance3	GENERAL_ADM	BUSINESS_Bldg	f3, f5

and the complexity, in terms of the number of classes and inheritance relationships, is increased. Moreover, it is obvious that this representation does not provide a true reflection of the relationship provided by the record structure.

The second illustrative example is selected from the field of Network Computing Architecture (NCA) [Dineen 87]. NCA is an object-oriented framework for developing distributed systems. In NCA, Remote Procedure Call (RPC) is defined as a mechanism that allows programs to call subroutines that run on different machines.

At the lowest level of abstraction, an NCA is a collection of machines located at various remote locations. Each machine provides a set of functions available to other machines through interfaces. Figure 6.3 abstractly represents an NCA with two machines M-1 and M-2. They are connected through an interface. The remote procedure call mechanism allows users of one machine to invoke functions defined in the other machine through their interfaces.

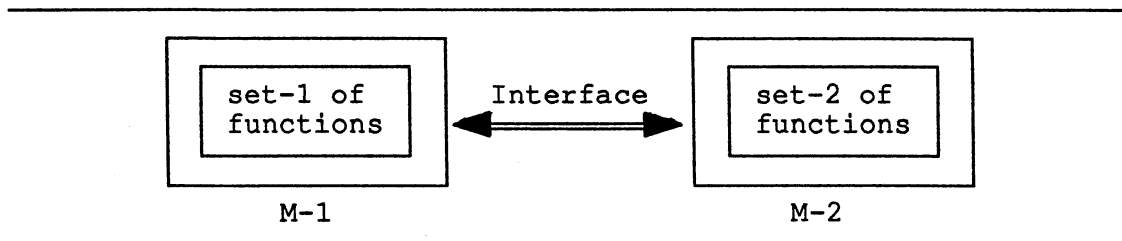


Figure 6.3: An NCA with two machines M-1 and M-2

In the object-oriented paradigm, a network of a set of such nodes can be represented by a set of classes that are related to each other through the inheritance relationships. In an NCA, a machine located at a site can be represented by a class and its functions by a set of attributes. Attributes of a class are available to other classes (machines) through their inheritance relationship. If we attempt a direct mapping of this scenario into an equivalent representation in terms of classes and inheritance, we will get the organizations shown in Figure 6.4.

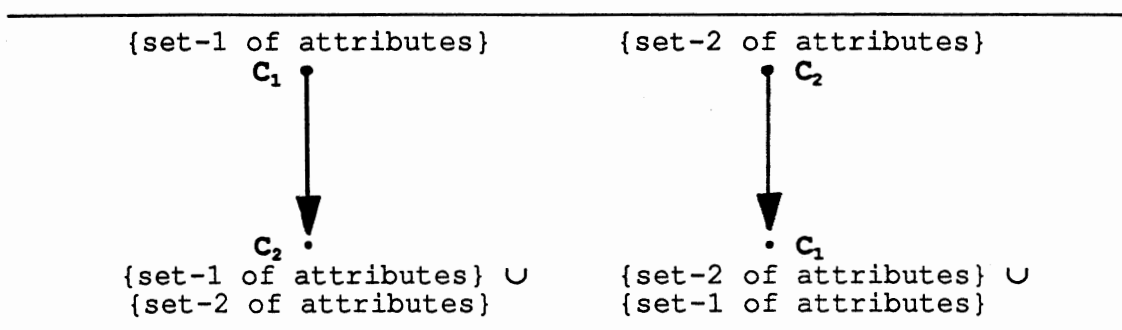


Figure 6.4: Class representations of the NCA in Figure 6.3

Figure 6.4 shows that two hierarchical inheritance relationships between the classes C_1 and C_2 are required to provide a relationship equivalent to the one described in Figure 6.3. To address this seemingly paradoxical situation, an object-oriented approach can be taken. Two different object-oriented representations of the NCA in Figure 6.3 are given in Figure 6.5.

Figure 6.5(a) shows that a machine and its functions are represented by separate classes. This representation provides multiple inheritance and maintains the bi-directional relationship depicted in Figure 6.3. One may group all of the common attributes into one class (e.g., C in Figure 6.5(b)) and let the classes $M-1$ and $M-2$ inherit from that class. In both representations, the number of classes will become in general larger than the number of actual machines in the network and besides the class structure provides a

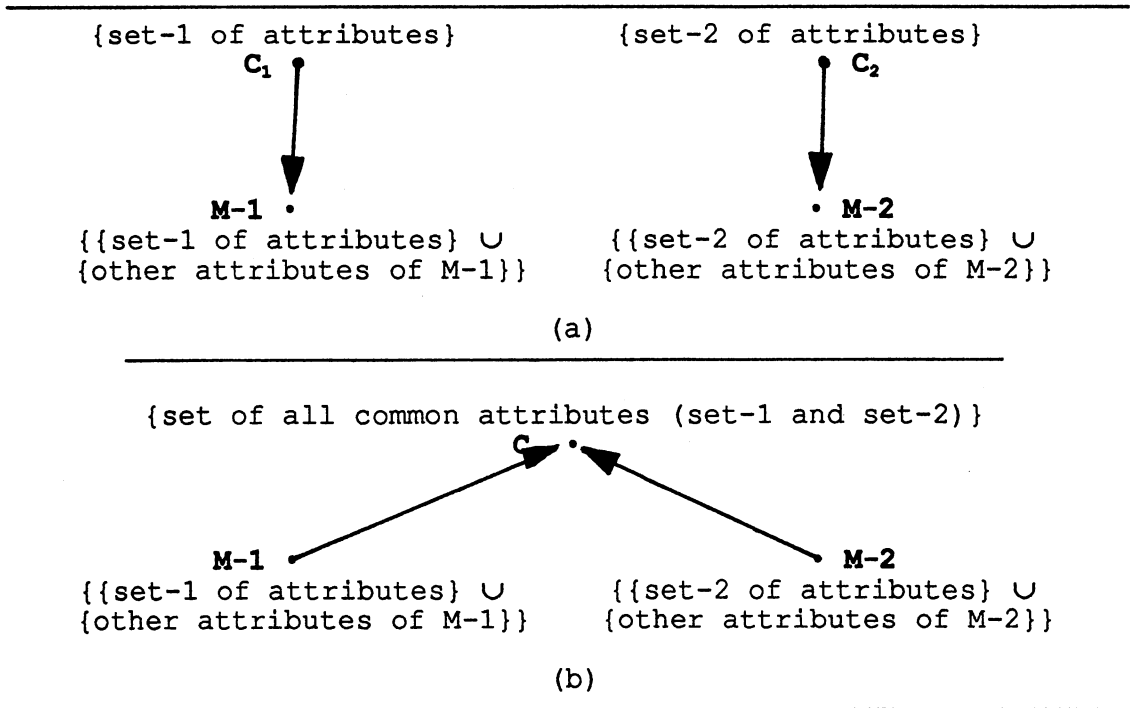


Figure 6.5: Two possible object-oriented representations of the NCA in Figure 6.3

distorted view of the actual configuration.

Hierarchical inheritance applies some form of a partial ordering to the classes in the structure. That is, properties of a superclass are available to its descendants, but properties of descendant classes are not available to the superclass. For instance, one of the two hierarchical structures in Figure 6.4 is needed to represent the non-hierarchical structure in Figure 6.3. Thus, classes are replicated and complexity is increased due to the restricted ordering provided by the hierarchical inheritance among classes.

The representation in Figure 6.5(a) requires a number of inheritance relationships among the sets of attributes and the machine classes. In the case of a network of a large number of nodes, a tremendous number of inheritance relationships would be required to connect a machine to every set of available attributes. Moreover, in Figure 6.5(a), the required number of classes is twice the number of nodes in the network, and Figure 6.5(b) provides an inappropriate representation, in that it does not reflect the nature of the

network because the actual functions of a network are residing at different locations, while this representation gives a common pool of functions shared by all classes.

The above-mentioned scenarios suggest the need for a new model. In the following section we introduce the proposed feedback model. Annotated directed graphs are used to describe the model and the notion of a **clan** is also introduced.

6.4 The Proposed Feedback Inheritance Model

The examples discussed in the previous section strongly suggest that the one-way information flow (inheritance) is inappropriate for situations where bi-directional dependency exists among a superclass and its subclass(es). Therefore, a relaxation of inheritance as a mechanism that represents dependency among classes is called for. A bi-directional model would relieve the programmer from attempting inefficient ad-hoc approaches to achieve such dependency among classes.

6.4.1 Definitions

In addition to the notations and definitions introduced in Section 6.2, we introduce the following notations and definitions. In what follows we assume the existence of an inheritance hierarchy. Suppose a is an attribute defined in the subclass C_1 of the class C . If a is not defined in C but is used in C , we say that class C *derives* the attribute a from class C_1 .

- 1) **Synthesized attributes** of a class C are those derived from its subclass(es).
- 2) Synthesized attributes derived from a subclass are denoted by the lower case letters x , y , etc.
- 3) To allow for synthesized attributes, an extended class definition is given as follows. A class C is defined as a triple

$$(C, \{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\})$$

where C is the class name, $\{a_1, a_2, \dots, a_k\}$ is the set of both defined and inherited attributes, and $\{x_{k+1}, \dots, x_m\}$ is the set of synthesized attributes. For notational convenience, a class C is represented as

$$C = (\{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\}).$$

If there are no synthesized attributes in C , it will be represented as

$$C = \{a_1, a_2, \dots, a_k\} \text{ (see Section 6.2 (6)).}$$

- 4) Given the classes

$$C = (\{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\}) \text{ and } C_1 = \{b_1, b_2, \dots, b_n\},$$

and that C_1 is a subclass of C ,

C is a **synthesized type (syntype)** of C_1 (written $C_1 \succ C$)
iff $x_i = b_j$ for some $x_i \in \{x_{k+1}, \dots, x_m\}$ and $b_j \in \{b_1, b_2, \dots, b_n\} - \{a_1, a_2, \dots, a_k\}$.

Here, C_1 is a subclass of C which means that C_1 inherits from C and possibly defines new attributes. When C uses the newly defined or derived attributes in C_1 , it implies that C is a syntype of C_1 . If class C is a syntype of two or more classes, we denote that by $(C_1, C_2, \dots, C_n) \succ C, n \geq 2$.

- 5) **The Synthesized Interface (SI)** that is provided to a superclass C (denoted by $C.SI$) is a set of synthesized attributes derived from its subclass(es).

- 6) **A Feedback** is a derivation relationship provided through the synthesized interface from a subclass to its superclass(es) (i.e., syntype(s)). Given the classes

$$C_1 = (\{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\}) \text{ and } C_2 = \{b_1, b_2, \dots, b_n\}$$

where C_2 inherits from C_1 and C_1 derives from C_2 ,

$$C_1.SI = \{x_{k+1}, \dots, x_m\} \text{ where } x_i = b_j \text{ for } x_i \in C_1.SI \text{ and } b_j \in \{b_1, b_2, \dots, b_n\} - \{a_1, a_2, \dots, a_k\}.$$

Intuitively, a feedback is a derivation of attributes from subclass(es), and the synthesized interface of a superclass is a description of the set of derived attributes.

- 7) **A clan** is a set of classes related through feedback inheritance. The clan of a class C is constructed as follows:

- i) C is in the clan of C .
- ii) If there exists a feedback relationship between C and its subclass(es), then

the subclass(es) is (are) included in the clan of C .

- iii) If X is in the clan of C and there exists a feedback relationship between X and its subclass Y , then Y is also included in the clan of C .
- iv) No other classes are included in the clan of C .

Each class belongs to at least one clan. Clans provide design modularity and control complexity. The notion of clan is illustrated in Figure 6.6. Solid arrows indicate hierarchical inheritance of attributes and dashed arrows indicate feedback inheritance (derivation) of synthesized attributes.

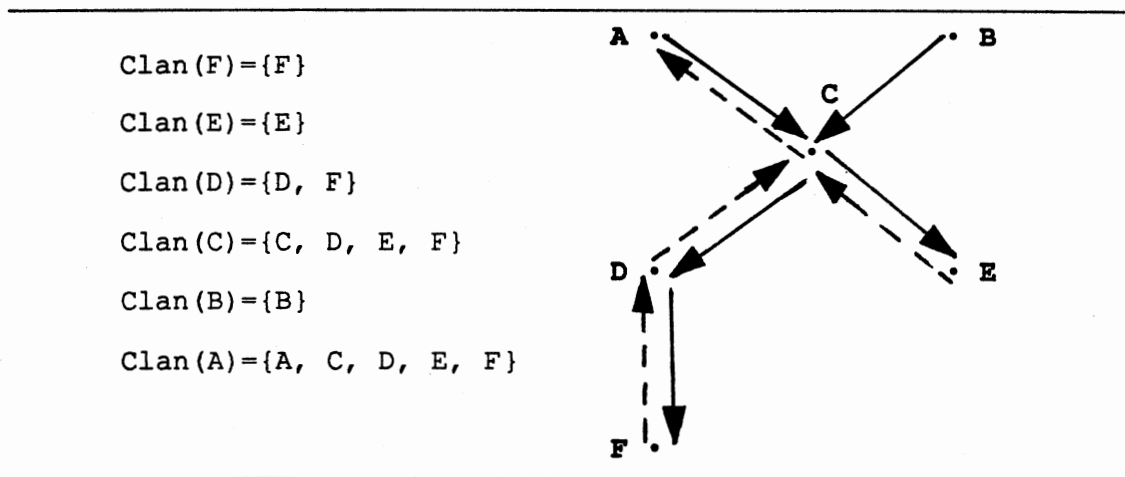


Figure 6.6: Examples of clans of a set of related classes

Hereafter, the term *syntype* is used to denote the superclasses that are provided with feedback relationships. The next subsection provides a detailed description of the feedback model using the above notations and definitions.

6.4.2 Feedback Inheritance

The feedback model [Al-haddad 92b] provides a feedback relationship between a subclass and its syntype class(es), and allows for control over the information flow in that

direction also. The basis of the new model is inheritance relationship among classes. Having hierarchical inheritance does not imply feedback inheritance, but two classes cannot have a feedback relationship unless they have a hierarchical inheritance relationship with each other.

As in the inheritance models, single and multiple feedback relationships may relate classes to one another. In the case of single feedback, the syntype class depends on the attributes of only one subclass. Given the classes

$$C = (\{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\}) \text{ and } C_1 = \{b_1, b_2, \dots, b_n\},$$

and that C_1 is a subclass of C ,

$$C_1 > C \text{ iff } x_i = b_j \text{ for some } x_i \in C \text{ and } b_j \in \{b_1, b_2, \dots, b_n\} - \{a_1, a_2, \dots, a_k\}.$$

A general representation of single feedback is depicted by the annotated directed graph in Figure 6.7.

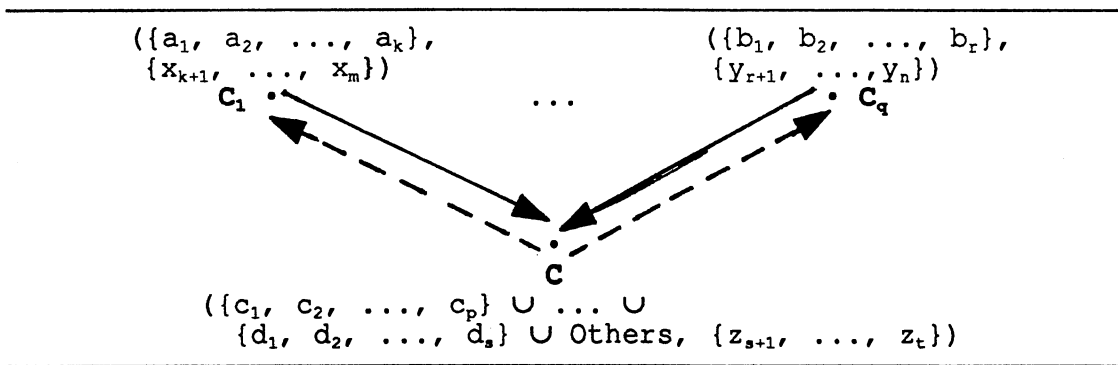


Figure 6.7: Representation of single feedback inheritance among classes

In Figure 6.7, the class C inherits attributes c_i , for $i=1, 2, \dots, p$, from $C_1.[a_1, a_2, \dots, a_k]$ and attributes d_j , for $j=1, 2, \dots, s$, from $C_q.[b_1, b_2, \dots, b_r]$. Class C_1 derives the synthesized attributes x_i , for $i=k+1, \dots, m$, from C , and class C_q derives the synthesized attributes y_j , for $j=r+1, \dots, n$, from C . The synthesized interfaces defined in Figure 6.7 are

$$\begin{aligned} C_1.SI &= C_1.[x_{k+1}, \dots, x_m] \subseteq C.Others \cup \{z_{s+1}, \dots, z_t\} \\ &\dots \\ C_q.SI &= C_q.[y_{r+1}, \dots, y_n] \subseteq C.Others \cup \{z_{s+1}, \dots, z_t\} \end{aligned}$$

In the case of multiple feedback, a syntype class may depend on attributes of one or more subclasses. For example, given the classes

$$C = (\{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\}), C_1 = \{b_1, b_2, \dots, b_n\}, \text{ and } C_2 = \{c_1, c_2, \dots, c_t\},$$

and that both C_1 and C_2 are subclasses of C ,

$$(C_1, C_2) \succ C \text{ iff } x_i = b_j \text{ for } b_j \in \{b_1, b_2, \dots, b_n\} - \{a_1, a_2, \dots, a_k\}$$

$$\text{or } x_i = c_j \text{ for } c_j \in \{c_1, c_2, \dots, c_t\} - \{a_1, a_2, \dots, a_k\}.$$

In general, multiple feedback can be depicted by the annotated directed graph in Figure 6.8.

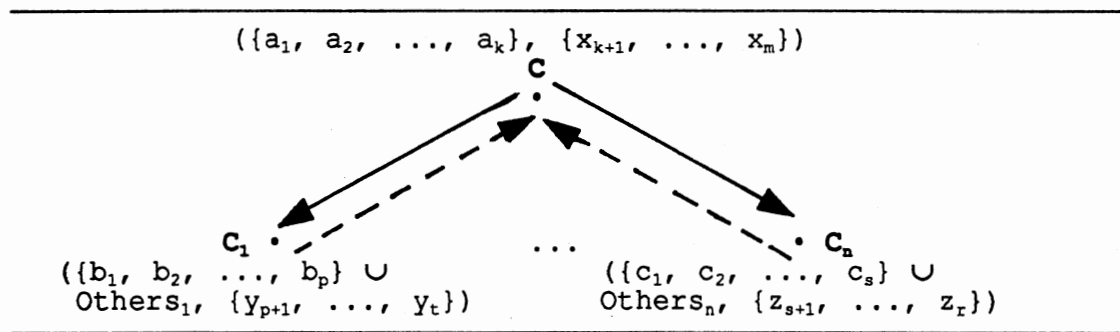


Figure 6.8: Representation of multiple feedback inheritance among classes

In Figure 6.8, the classes C_1 through C_n inherits the attributes b_i , for $i=1, \dots, p$ and through c_j , for $j=1, \dots, s$, from $C.[a_1, a_2, \dots, a_k]$. The class C derives the synthesized attributes x_i , for $i=k+1, \dots, m$, from the classes C_1 through C_n . The set of derived attributes in C is a subset of all attributes defined in the classes C_1 through C_n . The synthesized interface defined in Figure 6.8 is

$$C.SI = C.[x_{k+1}, \dots, x_m] \subseteq (C_1.Others_1 \cup \{y_{p+1}, \dots, y_t\}) \cup \dots \cup (C_n.Others_n \cup \{z_{s+1}, \dots, z_r\}).$$

As an example, consider the following classes.

$$\begin{aligned} C_1 &= (\{a_1, a_2, \dots, a_k\}, \{x_{k+1}, \dots, x_m\}), \\ C_2 &= (\{b_1, b_2, \dots, b_s\}, \{y_{s+1}, \dots, y_n\}), \text{ and} \\ C_3 &= (\{c_1, c_2, \dots, c_p\}, \{z_{p+1}, \dots, z_t\}). \end{aligned}$$

Suppose that C_2 inherits from C_1 and C_3 inherits from C_2 , then $C_2 \succ C_1$ and $C_3 \succ C_2$. The directed graph in Figure 6.9 illustrates two single feedback relationships among the classes C_1 , C_2 , and C_3 . Class C_2 inherits some of the attributes b_i , for $i=1, 2, \dots, s$, from C_1 , and the class C_3 inherits some of the attributes c_j , for $j=1, 2, \dots, p$, from C_2 . The class C_1 derives the attributes x_i , for $i=k+1, \dots, m$, from C_2 , and the class C_2 derives the attributes y_j , for $j=s+1, \dots, n$, from C_3 .

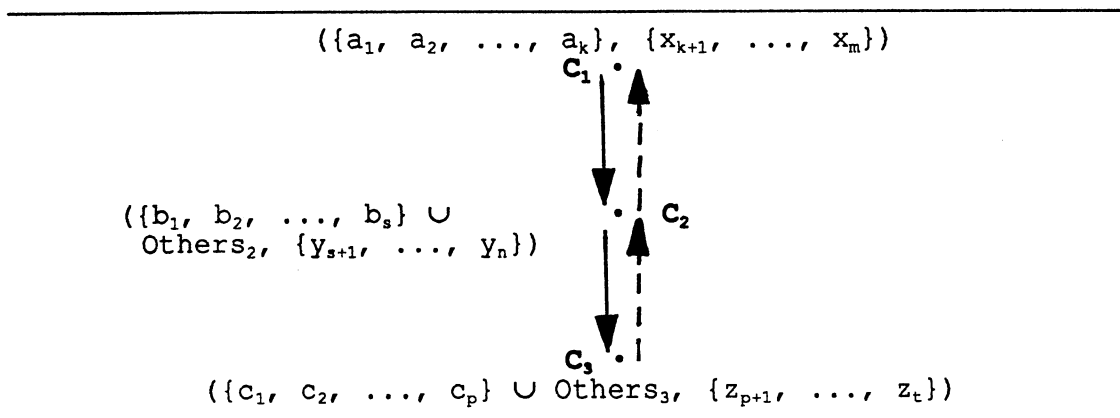


Figure 6.9: Representation of feedback inheritance among classes C_1 , C_2 , and C_3

6.4.3 Semantics of Feedback Inheritance

Hierarchical inheritance among classes means that attributes of the superclass are implicitly (automatically) made available to its subclass(es). In feedback inheritance, attributes of a subclass are not implicitly made available to syntype class(es). However, a synthesized interface of a subclass explicitly provided to a syntype class implies the availability of its contents (derived attributes) to the syntype class, and hence the knowledge about the availability of the subclass's synthesized attributes to the syntype class. A syntype class that is not explicitly provided with a synthesized interface does not have any knowledge about the subclass's attributes and cannot access them. This point is illustrated in Figure 6.10.

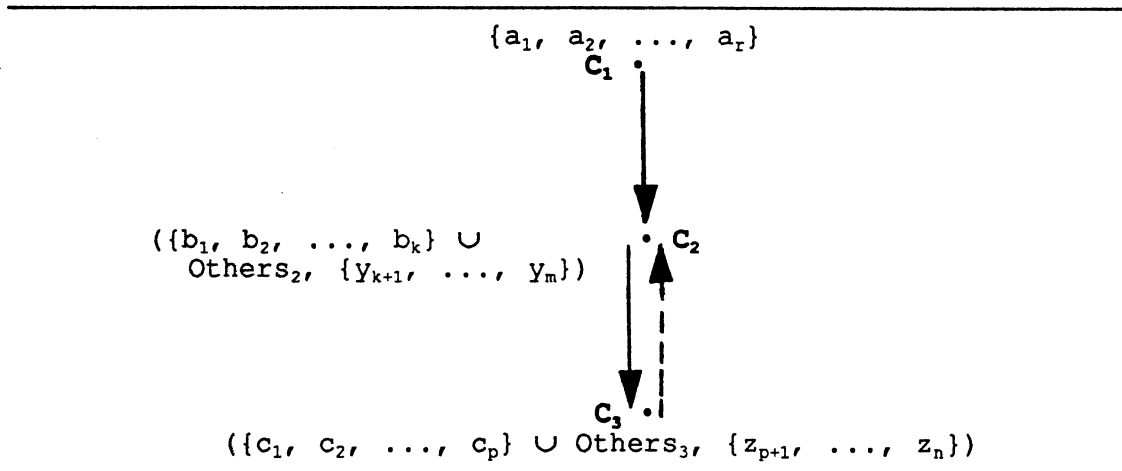


Figure 6.10: A feedback and hierarchical inheritance interfaces among the classes C_1 , C_2 , and C_3

In Figure 6.10, the class C_2 derives the attributes y_i , for $i=k+1, \dots, m$, from $\{z_{p+1}, \dots, z_n\} \cup Others_3$ of C_3 . This derivation implies the availability of attributes of C_3 to C_2 , and hence the knowledge about the attributes of C_3 in C_2 . Since C_1 does not derive attributes from C_2 , there is no feedback relationship between C_1 and C_2 . The availability of the synthesized attributes of C_3 is terminated at C_2 . On the other hand, the hierarchical inheritance interface from C_1 to C_2 and then to C_3 makes attributes of C_1 and C_2 implicitly available to C_3 .

For simplicity of control, the feedback relationship is defined only between a syntype class and its subclass(es). However, a syntype class may include a subclass's synthesized attributes in its synthesized interface. For instance, the class C_1 in Figure 6.10 cannot directly derive attributes from the set $\{z_{p+1}, \dots, z_n\}$ of C_3 , however it can derive from the set $\{y_{k+1}, \dots, y_m\} \cup Others_2$ of C_2 . Moreover, feedback inheritance implies that the contents of the synthesized interface explicitly provided by a syntype class to its ancestor class(es) is not necessarily disjoint from the contents of the synthesized interface provided by a subclass to that syntype class.

6.4.4 Message Passing

Message passing and method determination in feedback inheritance can be described as follows. A message is a request for an instance to execute one of its methods. The receiving instance determines how to execute the requested method. When a message $M[m, p_1, \dots, p_k]$ is sent to an instance of the class C , methods of C are searched for matching method m . If m is not found, the search continues along the inheritance and feedback paths until a matching method is found or an error message is returned. If method m is found, it is executed and the result is returned.

To define the semantics of message passing, we introduce a run-time data structure for method description called a descriptor. A **descriptor** is a data structure associated with each class in the system. It contains information about variables, defined and derived methods, and pointers to super and subclasses. Information about whether a method is derived explicitly by a syntype class when feedback exists among classes, is also included in the descriptors. The structure of a descriptor is depicted in Figure 6.11.

As illustrated in Figure 6.11, the descriptor of a class may include several components. Five of the components are described below.

- 1) **Class Name (CN):** The name of the class that the descriptor represents.
- 2) **SuperClasses (SC):** A table of pointers to the descriptors of all superclasses.
- 3) **Synthesized Interface (SI):** A table of the names of all derived methods from subclasses. Each name is associated with a pointer pointing to the subclass providing that method (i.e., to another descriptor).
- 4) **Instance Variables (IV):** A tables of the names, types, and other attributes of all instance variables defined in the class.
- 5) **Defined Methods (DM):** A table of the names of all methods that are defined in the class. Each name is associated with a pointer to the executable code of the method.

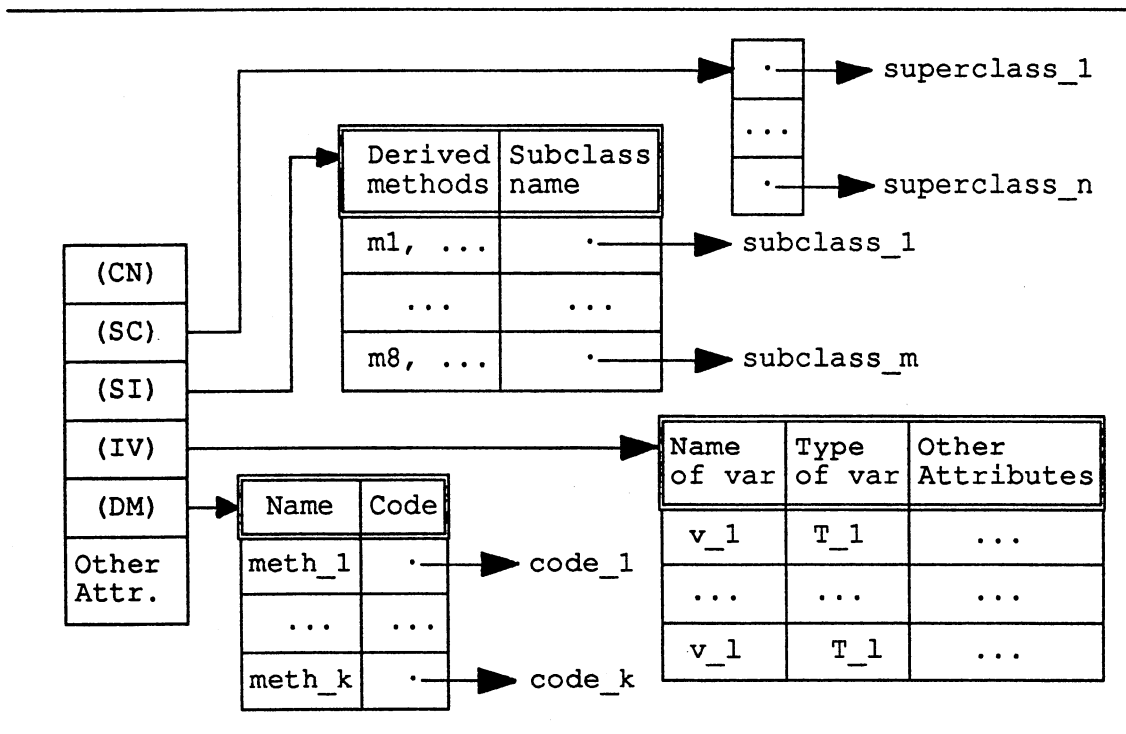


Figure 6.11: The structure of a descriptor

Pointers to super and subclasses are in fact pointers to the descriptors of these classes. Synthesized interface tables provide information about derived methods. Reaching the code of a derived/inherited methods in a class is achieved by following the pointers to the descriptors of super/subclasses.

When a message is sent to an instance at run time, the descriptor associated with the class of the receiving instance is searched for a matching method. If a matching method is not found, the descriptor(s) of the superclass(es) is (are) searched, and so on. If a matching method is found, it is invoked through the pointer to its compiled code. Otherwise, an error message is returned.

If the class is a syntype class of subclass(es) (i.e., feedback inheritance exists among the class and its subclass(es)), its descriptor will include the names of the subclass(es) from which methods are derived. When an instance of a syntype receives a messages, the descriptor(s) of the superclasses and the syntype interface are searched for

a matching method.

Example:

In this example, the class A is the superclass of the class B. Class B is the superclass of the classes R and S. A derives the attributes m5 and m8 from B. B derives the attributes m8 and m9 from classes R and S, respectively. The classes A and B can be described as follows:

```

Class A
  Superclass      : nil;
  Derived methods : B [m5], B [m8];
  Instance Variables: int x, int y;
  Defined methods : m1, m2;
end;

Class B
  Superclass      : A;
  Derived methods : R [m8], S [m9];
  Instance Variables: int a, real b;
  Defined methods : m4, m5;
end;

```

Figure 6.12 illustrates the descriptors of the classes A and B.

Suppose that the instance A-1 of the class A receives the message $M[m5]$. A-1 will react by searching the defined methods table in the descriptor of A. Since m5 is not defined in A, m5 is either inherited or derived. A search of the synthesized interface table (derived methods) indicates that m5 is derived from B. The search continues to the descriptor of B. The method m5 is defined in B and its executable code is obtained through a pointer in the defined methods table of B. If m8 has been requested, the search for m8 would have continued to the descriptor of the class R, and so on.

In a different scenario, if A-1 receives the message $M[m20]$, the search for m20 starts at the descriptor of A. Since m20 is neither defined nor derived, the search continues along the inheritance path to the descriptors of ancestor classes, unless m20 is found or an error message is returned indicating that no such method is defined along the inheritance path.

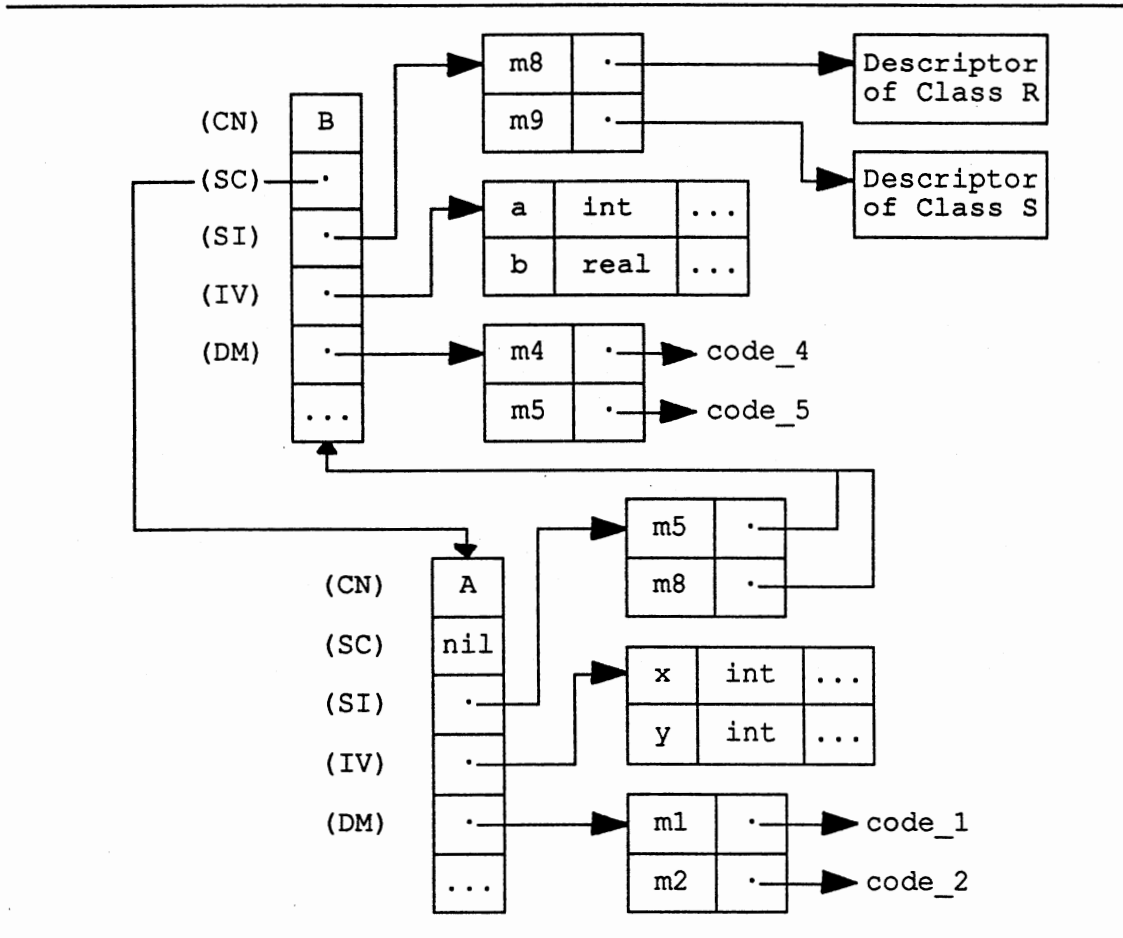


Figure 6.12: Descriptors of the classes A and B

6.5 Examples Represented in the Proposed Model

Examples are used in this section to illustrate the novel notions of a **clan** and the feedback relationship. To see the usage of the new model, we refer to the database and network examples mentioned in Section 6.3 and provide a representation in the context of the new model.

In the case of databases, as in hierarchical inheritance, a class contains a set of attributes that represent the state and behavior of its instances. Attributes represent the relevant information of an entity in the system. For instance, the class `DEPARTMENT` contains the attributes `D_name`, `Location`, `Degrees_offered`, and `College`. The class

attributes, their values are returned as a result. It can be observed that the instance Dept returns results equivalent to tuples created by the type Department defined in Figure 6.2. It is not necessary that all classes should have feedback relationship, only classes whose attributes appear in relations defined on them (e.g., Department and Faculty in Figure 6.2). One may relate the classes DEPARTMENT, FACULTY, STUDENT, and COURSE (in the university system) as shown in Figure 6.14. In this figure, the sets

$$\begin{aligned} Others_D &= \{D_name, Location, Degree_offered, College\}, \\ Others_S &= \{S_name, S_age, S_address\}, \\ Others_F &= \{F_name, Rank, F_address\}, \text{ and} \\ Others_C &= \{C_name, C_Level\} \end{aligned}$$

are the sets of the newly defined attributes in the classes DEPARTMENT, STUDENT, FACULTY, and COURSE, respectively. The attributes defined in the classes DEPARTMENT, STUDENT, and FACULTY are all available to the class COURSE through the hierarchical inheritance. The second set of attributes at the nodes Department, Student, and Faculty is the set of synthesized attributes from their subclasses. The four classes in Figure 6.14 constitute the clan of the class DEPARTMENT.

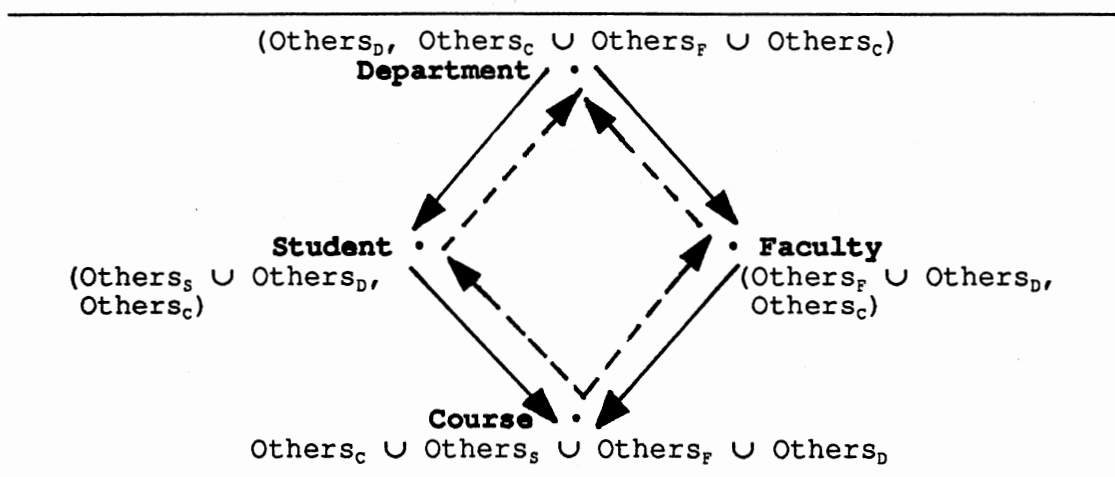


Figure 6.14: Relating classes through hierarchical and bi-directional inheritance

Any relation (data collected from different classes) on the classes shown in Figure 6.14 can be represented by sending message(s) to an instance of the appropriate class. For

instance, sending the messages

$M[C_name]$, $M[F_name]$, and $M[D_name]$

to the instance OP_SYS of the class Course, returns, for example, the values

OPERATING_SYSTEMS, {f1, f2}, and COMPUTER_SC.

The feedback relationship reduces the number of classes in the university system and provides a flexible technique for the definition of relations on classes. A message can be sent to different instances of different classes and return similar results. For example, the above messages, which were sent to the class COURSE, can be sent either to the class DEPARTMENT or the class FACULTY and return similar (not necessary identical) results.

Feedback is efficient and simple in terms of relations that can be defined on classes. Shared attributes among related classes is provided and a better reflection of the original system is produced. The addition of new attributes to a class makes the new attribute automatically available to the inheriting classes and explicitly available to the syntype class(es). Deletion of an attribute simply requires a de-synthesis (i.e., removing it from the synthesized interface) of that attribute since it is provided explicitly.

Considering the NCA example, as in the hierarchical inheritance approach, each node of the network is represented by a class. Classes are related through their feedback and inheritance relationships in which they share attributes (functions). An abstract representation (utilizing classes, hierarchical inheritance, and feedback relationship) of a network consisting of three machines is given in Figure 6.15.

In Figure 6.15, the feedback interface makes attributes of a subclass available to its syntype class. Class M-2 derives {e, f} from the class M-3, and class M-1 derives {c, d, e, f} from the class M-2. However, the attributes defined in a node are available to all other nodes. A network of n nodes can be represented by n classes such that every two adjacent nodes share attributes through their inheritance and feedback relationship. All the classes constitute one clan and any instance of any class can answer the same message.

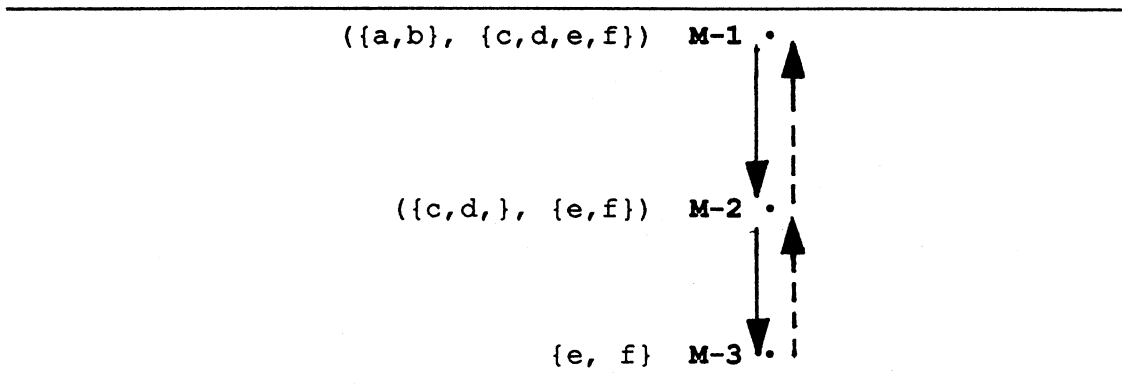


Figure 6.15: Feedback inheritance representation of a three-node NCA

This representation reflects the NCA in a more natural fashion than the one in Figure 6.5.

6.6 Discussion

This section is devoted to a discussion of issues such as encapsulation, visibility, and the access of attributes, that are related to inheritance in the context of the new model. Encapsulation minimizes the dependency among classes by providing external interfaces that contain the attributes of a class that are available to its subclass(es). In the hierarchical inheritance model, a class is encapsulated if its clients are constrained to access its attributes only via its external interface [Danforth 88]. In the feedback inheritance model, the inheritance and synthesized interfaces are independent and can be constrained to varying degrees.

Encapsulating classes facilitates maintenance and provides the capability of making safe changes in a class without affecting its subclass(es). Inheritance allows a class to include inherited attributes in its interfaces that are provided for its subclass(es). If inheritance is part of the interface (i.e., visible to descendants), then changing the implementation of a superclass, that affects this interface, may require changes in the inheriting classes at several levels [Snyder 86a]. In the feedback model, the exclusion of inherited attributes from being derived by syntype classes prevents the spread of the side

effects of changes made in the ancestor classes. Also, providing synthesized interfaces explicitly implies that the values of the derived attributes themselves are not part of the interface (not visible to the syntype class(es)), and changes made to these attributes will not affect the syntype class(es). Therefore, we conclude that encapsulation is not compromised by the new model.

In some OOPLs, the designer of a class is granted full access to the implementation of ancestor class(es) [Strom 86]. Full access to class implementation compromises encapsulation and limits the ability of safely changing the class implementation without affecting its inheriting class(es). Instance variables of a class are allowed to be inherited among classes. Providing methods to access the instance variables of a class is a solution for safe access and is an implementation issue that differs from one language to another [Stein 86]. In the feedback model, the same concept is applicable. At the implementation level, the values of attributes (state and behavior) of a class can be hidden from other classes and access is allowed only through attributes provided in the external interface.

The issue of visibility of inheritance is implementation dependent [Snyder 86a]. It depends on several factors that the designer may consider. These factors are:

- 1) The capability of excluding inherited attributes;
- 2) The naming conflicts that may arise as a result of multiple inheritance;
- 3) The direct invocation of attributes from ancestor classes; and
- 4) The subtyping rules and their relation to inheritance.

The first and second factors are not applicable in the feedback model. Excluding attributes and providing same-named attributes are avoided since attributes are explicitly provided to the syntype class(es). The third factor cannot occur in the new model since the availability of the synthesized interface terminates at the syntype class and derivation is provided explicitly. The fourth factor depends on the subtyping rules that are language

dependent.

Therefore, encapsulation, visibility, and safe access are maintained in the new model due to the restricted and explicitly provided synthesized interface to the syntype class(es). These issues facilitate the development and maintainability of software systems.

6.7 Summary

Generally, a preferred representation of a problem is that provided by a model that faithfully reflects the structure of the problem. The OOP paradigm has been the paradigm that provides a better correspondence between a problem and its representation. Even so, many real-life situations still cannot be well reflected in the object-oriented paradigm. The hierarchical inheritance model does not provide a satisfactory representation for situations where the dependency among objects is bi-directional.

A relational database represents one area where variables are replicated among object types (record structures). Networking is another area that illustrates the situation where attributes need to be shared in both directions. Such situations are hard to represent in the hierarchical inheritance model. A feedback inheritance model that allows both a superclass and its subclass(es) to exchange attributes along with the notions of synthesized attributes, synthesized interfaces, and clans are presented. It is also shown that annotated directed graphs provide a simple and clear representation of the model. This model facilitates the programming task in situations similar to the above examples, and relieves the users from having to use tricky and inefficient approaches in the hierarchical inheritance model. Moreover, the notion of clan relaxes the message passing technique and potentially increases the probability of answering a message as well as the use of attributes among classes.

Examples of a relational database and a network are used to illustrate the merits of the new model. First, possible representations in the hierarchical model are presented,

then the feedback representations are provided and compared with the hierarchical representations. The advantages and side effects in each representation are highlighted too.

Feedback inheritance provides an opportunity for better and simpler implementation of systems that include bi-directional dependency among classes. Also, fewer classes are used and simpler structure results. In general, feedback inheritance avoids replicating functions among classes, increases reusability, eases software maintenance, and facilitates the sharing of functions in a distributed environment.

CHAPTER VII

AN IMPLEMENTATION INHERITANCE MODEL

7.1 Introduction

The inheritance models supported by the currently available OOPs [Cardelli 84] [Hailpern 87] [Pedersen 89] [Stein 87] are characterized as a "specification inheritance". A class provides the specifications of its methods in the external interface for other classes to use, and for the subclasses to inherit. Subclasses may provide new implementations for the inherited methods. Therefore, a method may have more than one implementation in different classes. In order to avoid ambiguity, the philosophy adopted by current OOPs is that an object shall use the most recent implementation of an inherited method, and that the object is prohibited from using any previous implementation provided by ancestor classes. Subclasses may not be able to select freely any available implementation of a method since a class inherits the specifications of the methods rather than their implementations. When a subclass redefines an inherited method *m*, it is desirable to allow instances of that class to access any previous implementation of the method *m* provided in ancestor classes.

This approach is convenient when a method has different implementations in different classes. Otherwise, if an implementation different from the most recent one is needed, it has to be explicitly provided even if that implementation is available in an ancestor class. Such restrictions prohibit code reuse. As it turns out, some OOPs do provide some limited programming features that allow a class to associate a method with different implementations. In the following section, we examine those features in the

languages Eiffel, C++, CLOS, and Smalltalk.

The goal of this chapter is to generalize the current philosophy of inheritance in order to provide classes with the ability to choose any available implementation of a method in ancestor classes and to facilitate code reuse among classes. A language implementation scheme to support the proposed approach is provided.

7.2 Background

In this section four OOPs are examined based on a number of their respective features that allow the association of a method with different implementations. The languages examined in this section are Eiffel, C++, CLOS, and Smalltalk.

7.2.1 Eiffel

Eiffel [Meyer 88] provides a feature called **deferred class** that allows users to provide different implementations of a method in different classes. Figure 7.1 illustrates a deferred class using an example adopted from [Meyer 88].

In Figure 7.1, the deferred class `STACK[T]` represents an abstract data type in which methods are deferred for later implementations. The inheriting classes of the class `STACK[T]` provide the appropriate implementations of the deferred methods to suit their needs. For example, one subclass may implement stack as an array and then implement the deferred methods to accommodate the array implementation. Another subclass may implement stack as a linked list, in which case the implementations of the deferred methods would be different from those of the array implementation.

The class `STACK[T]` is a partial implementation of the abstract data type stack. It has no instances. The "deferred class no-instantiation rule" of Eiffel prevents creating instances from deferred classes in order to avoid the use of incomplete methods. Thus, the concept of deferred methods allows subclasses to have different implementations, but

```

deferred class STACK[T] export
    nb_elements, empty, full, top, push, pop, ...
features
    nb_elements: INTEGER is
        deferred
        end; -- nb_elements
    empty: BOOLEAN is
        do result := (nb_elements = 0)
        ensure result := ~(nb_elements = 0)
        end; -- empty
    full: BOOLEAN is
        deferred
        end; -- full
    top: T is
        require not empty
        deferred
        end; -- top
    push(x: T) is
        require not full
        deferred
        insure not empty; top := x;
            nb_elements := old nb_elements + 1;
        end; -- push
    pop is
        require not empty
        deferred
        insure not full; nb_elements := old nb_elements - 1;
        end; -- pop
    ... -- Other features
end; -- class STACK

```

Figure 7.1: Definition of the class STACK[T]

subclasses still use the most recent implementation of an inherited method.

7.2.2 C++

C++ [Stroustrup 86,91] provides several features that allow a method to be associated with different implementations: function overloading, virtual function, and abstract class. These features are described in the following subsections.

7.2.2.1 Overloading of Functions. C++'s overloading is another language-specific feature that can be used to provide different meanings and different implementations for functions. In programming languages such as Pascal, functions with arguments of

different types must have different implementations and distinct names. With the availability of overloading, different functions (implementations) in C++ can have the same name. In this case, the C++ compiler and run-time support determines (at compile time and based on the number and/or type of arguments) the specific function to invoke. However, overloading is in fact a syntactic issue that allows several meanings to be given to a symbol or a function name in order to re-implement its inherited description from ancestor class(es) and to avoid name replication. Overloading in C++ is illustrated in Figure 7.2 where the function `ave_3_grades` is overloaded. Figure 7.2 adopted and modified from [Pohl 89].

```

#include <stream.h>
class AVERAGE
  { public:
    float ave_3_grades (float, float, float);
    int ave_3_grades (int, int, int);
  };

float average::ave_3_grades (float g1, float g2, float g3);
  { float avg;
    avg := (g1+g2+g3) / 3.0;
    return avg; }

int average::ave_3_grades (int g1, int g2, int g3);
  { int avg;
    avg := (g1+g2+g3) / 3;
    return avg; }

main ()
  { float a,b,c,ave_1; int x,y,z,ave_2;
    AVERAGE ave;
    cout << "\g1= "; cin >> a;
    cout << "\g2= "; cin >> b;
    cout << "\g3= "; cin >> c;
    cout << "\g1= "; cin >> x;
    cout << "\g2= "; cin >> y;
    cout << "\g3= "; cin >> z;
    ave_1 = ave.ave_3_grades (a,b,c);
    ave_2 = ave.ave_3_grades (x,y,z);
    cout << "ave_1 = " << ave_1 << "\n";
    cout << "ave_2 = " << ave_2 << "\n";
  }

```

Figure 7.2: Example of an overloaded function

Figure 7.2 associates the name `ave_3_grades` with two implementations for different types of arguments. In one case, the arguments are of type integer; while in the other case, they are of type float. When an object of the class `AVERAGE` receives a message invoking the method `ave_3_grades`, the type of the arguments determines the appropriate implementation.

7.2.2.2 Virtual Function. Virtual function [AT&T 89 a,b] is another C++ feature that allows classes to provide different implementations for inherited functions. The new implementation of an inherited function in the subclass dominates old implementations in ancestor class(es). Instances of the subclass use the new implementation. Non-virtual functions can be illustrated using the inheritance hierarchy shown in Figure 7.3.

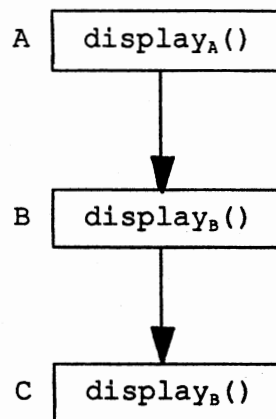


Figure 7.3: Inheritance of non-virtual functions

Let us assume that Figure 7.3 represents the following code declaration:

```

Class A { public: display_A() };
Class B : public A {public: display_B() };
Class C : public B {};
  
```

Let us further assume that `display_A()` and `display_B()` are different implementations of the function `display()`. Class C inherits but does not redefine `display_B()`. `display_B()` dominates

`displayA()` and instances of class C use `displayB()` defined in class B. However, the principle of domination prevents instances of classes B and C from using the version of `displayA()` defined in class A.

In the case of overloading, overloaded functions are selected statically based on type matching arguments; while in the case of virtual functions, the appropriate invoked function is determined dynamically from among class and its ancestor class(es) [Pohl 89]. Instances of subclasses use the virtual functions defined in the ancestor class(es) unless subclasses provide new implementations for the inherited virtual functions. The principle of virtual functions states that instances of a subclass choose implementations provided in the closest ancestor class (it is the responsibility of the programmer to avoid conflicts). We illustrate this principle using the inheritance hierarchy adopted from [AT&T 89b] and given in Figure 7.4.

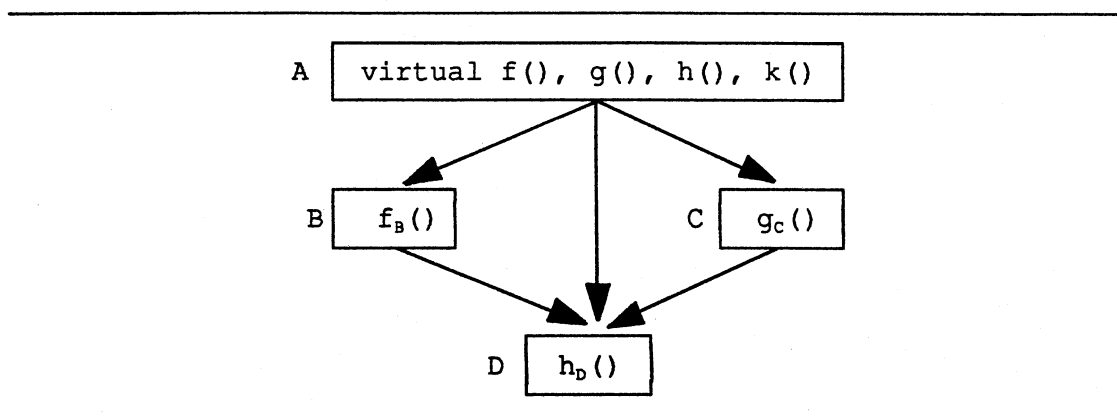


Figure 7.4: Inheritance of virtual functions

Figure 7.4 represents the following code declaration:

```

Class A : { public: virtual f(); virtual g();
           virtual h(); virtual k() };
Class B : public virtual A { public: f_B() };
Class C : public virtual A { public: g_C() };
Class D : public B, public C, public virtual A
         { public: h_D() };
  
```

The functions $B::f_B()$, $C::g_C()$, and $D::h_D()$ are re-implemented versions of the corresponding functions defined in class A. An instance of class D uses versions $f_B()$, $g_C()$, and $h_D()$, and does not have access to versions $f()$, $g()$, and $h()$ defined in A.

The C++ implementation strategy for virtual functions is based on creating tables for virtual functions. For a given class, C++ creates a virtual table that contains pointers to the appropriate implementations of virtual functions. Every instance of a given class includes a pointer to that table. For example, any instance of class D in Figure 7.4 includes a pointer to a virtual table that contains pointers to the functions $A::k_A()$, $B::f_B()$, and $C::g_C()$. However, instances of class D cannot choose other implementations of any of the inherited functions.

7.2.2.3 Abstract Classes. C++ abstract classes [Stroustrup 91] are similar to deferred classes in Eiffel. Abstract classes allow subclasses to provide different implementations of general functions. For example, the function `display()` can be used with a variety of shapes. Hence, the function `display()` can be defined in an abstract class and allow the subclasses that represent different shapes to inherit and redefine the function `display()` to suit their needs.

7.2.3 CLOS

CLOS [Keene 89] uses the generic function approach to invoke methods. A generic function is a function whose implementation is distributed across a set of different methods that belong to different classes. Unlike the message passing approach in which the invoked method is determined by the class type of the object to which the message was sent, the generic function approach determines the invoked method using the class type of the arguments to which the invoked method is applicable. Since the implementation of a generic function does not, in general, exist in one place, CLOS uses a *generic dispatch* mechanism for invocation. Generic dispatch is the process of

determining the applicable methods and invoking them. A method is applicable if the arguments of the generic function match the corresponding arguments of that method. For illustration, consider the following example adapted from [Winston 89] and illustrated in Figure 7.5.

```

(defun triangle_area (figure)
  (* 1/2 (triangle_base figure)
     (triangle_altitude figure)))

(defun rectangle_area (figure)
  (* (rectangle_width figure)
     (rectangle_height figure)))

(defun circle_area (figure)
  (* pi (expt (circle_radius figure) 2)))

```

Figure 7.5: Definitions of selected functions

The functions given in Figure 7.5 belong to the classes TRIANGLE, RECTANGLE, and CIRCLE, respectively, which calculate the area of the appropriate shape. Let the function `area` be a generic function that retrieves the implementations of these functions upon receiving the appropriate arguments. Consider the one-parameter methods illustrated in Figure 7.6

```

(defmethod area ((figure triangle))
  (* 1/2 (triangle_base figure)
     (triangle_altitude figure)))

(defmethod area ((figure rectangle))
  (* (rectangle_width figure)
     (rectangle_height figure)))

(defmethod area ((figure circle))
  (* pi (expt (circle_radius figure) 2)))

```

Figure 7.6: Different implementations of the method `area`

In Figure 7.6, each method is automatically applied when passing arguments of

the appropriate shape type. Note that all of these methods have the same name, and together they define the generic function `area`. The expression "(figure triangle)" names the parameters, and specifies the method used when the parameter `figure` is bound to an instance of class `TRIANGLE`. The argument `triangle` is called *parameter specializer*. Each method is applied when the argument matches the parameter specializer. Examples of usage are given in Figure 7.7.

```
* (setf triangle (make_instance :base 2 :altitude 3))
* (setf rectangle (make_instance :width 5 :height 7))
* (area triangle) ;Matching method triangle
3
* (area rectangle) ;Matching method rectangle
35
```

Figure 7.7: Usage of the method `area`

When the applicability of a method depends on the classes of two or more arguments, it is called a *multi-method*. Therefore, a multi-method in CLOS is a method that specializes more than one parameter [Keene 89]. For example, suppose that `meth` is a multi-method with two parameters, then the following method definitions

```
(defmethod meth ((x Class1) (y Class2)) ...) [1]
(defmethod meth ((x Class3) (y Class4)) ...) [2]
```

define the generic function `meth`. Method 1 is applicable when the first argument is of type `Class1` and the second one is of type `Class2`. Method 2 is applicable if the two arguments are of types `Class3` and `Class4` respectively. When two or more methods are applicable, they are ranked in order of precedence based on the order of the arguments from left to right. CLOS uses lexicographic ordering to determine the most specific method. Therefore, the most recent implementation is always used. This is the case in other languages also.

7.2.4 Smalltalk-80

In Smalltalk [Goldberg 83,89], a subclass inherits all variables and methods of the ancestor classes. It may add new variables and methods of its own. If the subclass adds a method whose specification is similar to a method in the superclass, instances of the subclass use the implementation in the subclass when receiving a message invoking that method. This is called **overriding** a method. Like other languages, the most recent implementation of a method is used. The search for an implementation starts in the class corresponding to the receiving instance, and continues up along the inheritance path until an implementation is found or an error message is returned.

In Smalltalk, the variable **super** allows an instance of a class to use a method's implementation provided in a superclass. In this case, the search for the implementation of the invoked method starts in the superclass of the class containing the implementation that uses the variable **super**. Therefore, this technique allows the use of one particular implementation of a method that uses the variable **super**, and not any implementation of that method provided in the ancestor classes.

Analogous to Eiffel and C++, Smalltalk provides abstract classes. They contain the specifications of methods shared by classes that are unrelated to one another through inheritance. These classes provide appropriate implementations for shared methods. These methods are similar to deferred methods in Eiffel. A method specified in an abstract class can be implemented once in a subclass. Instances of the inheriting classes use the most recent implementation, or the inheriting classes can override the method to be used by their instances. Abstract classes have no instances.

All features of different OOPs discussed in the above subsections have a common factor: a method may have several implementations in different classes and objects cannot choose freely any previous implementation of that method. Another approach to the definition of classes and methods is based on the concept of slots. A slot

is a repository associated with a set of values such that it holds one value at a time. For example, an integer slot can be associated with a set of integer values holding one value at a time. What happens when all variables and methods of a class are replaced by slots? In the next section this issues is briefly examined and its effect on the design of languages is discussed.

7.3 Classes and Slots

Classes contain variables and methods. Slots unify variables and methods into a single construct. Therefore, when replacing variables and methods of a class by slots, the class becomes a set of one type of entities. A slot may represent a state or a behavior. Both class-based and object-based languages (such as CLOS [Keene 89] and Self [Ungar 87] [Chambers 90] respectively) use slots.

CLOS [Keene 89] uses slots to determine the structure of a class. A slot has a name and a value. CLOS provides local and shared slots: a local slot has different values in different instances of the class and a shared slot has a single value shared by all instances of the class. CLOS implicitly generates accessors, `:accessor accessor_name`, that read and write the values of a slots.

Self [Ungar 87] is a language based on prototypes, slots, and behaviors. Slots are constructs that unify variables and methods. Therefore, Self does not distinguish between the state and behavior of an object. It describes slots as containers of objects that return themselves as results. The name of slot `S` reads the value of `S` as a state, and the accessor `S:` updates the value of `S`. Slots representing methods have values as executable code. Each such value executes when the corresponding slot receives a message.

Replacement of variables and methods of a class by slots eliminates the distinction between the state and behavior of the instances of the class. This may affect the specification of features that require this distinction. The required changes, as a result of

using slots, may vary from one language to another.

An instance of a class containing variables and methods has its own copy of the variables and uses methods of its class. When using slots, an object has its own copy of all slots that may have initial values representing the initial state of the object.

Objects communicate via messages [Chambers 90]. A receiving object determines whether to answer a message or not and how. When using slots, accessing a slot of an object may be accomplished by sending a message. The receiving slot determines whether to answer the message or not. In this case, the slot is given the responsibility of determining the match between the message and itself. A slot that accepts a message may return its value (state), or it may execute its value (implementation) and return the appropriate result.

Some languages, such as C++ [Stroustrup 91] and CommonObjects [Snyder 86b], prohibit direct access to instance variables. When using slots, slots that represent the state are associated with accessor slots to manipulate them. Accessor slots may be implicitly generated by the language as in CLOS or explicitly defined by the user as in Self.

Some languages, such as C++ [Stroustrup 91] and Smalltalk [Goldberg 89], provide different types of variables with different scopes. When using slots, either slots have different scopes or scopes are eliminated and slots themselves determine their own scopes by answering only the matching messages. For example, private, public, and protected variables in C++, and local, instance, class, pool, and global variables in Smalltalk can be unified into slots, and each slot can determine its scope. A slot answers a message based on its parameters and the identity of the sender.

Inheritance allows sharing common behaviors among classes. Shared slots can be made available through inheritance and sharing common behaviors among classes is not affected by the use of slots. On the other hand, some languages provide variables shared by all instances of a class. Such variables remain in the class and are made available to all instances of the class. For example, the class variables in Smalltalk are shared by all

instances of the defining class. When using slots, class slots (that correspond to class variables) remain in the class and are made accessible to all instances of a class. Another example is the use of global slots in CLOS. Therefore, using slots does not affect sharing instance variables among instances of a class.

To summarize, a slot is associated with a set of values, holding one value at a time. Slots of a class represent its variables and methods, and are responsible for handling messages. The values of a slot represent different implementations. Therefore, a slot can be associated with a number of implementations in different classes and a subclass may associate a new implementation with an inherited slot.

In the following section, a class model that consists of variables and slots is proposed. We use slots in place of methods and introduce the notion of behavior slots.

7.4 Behavior slots

The focus of this model is limited to methods and the reuse of their implementations. Slots are used to replace methods of classes. The distinction between the terms method and slot is in their association with implementations. Each method name is associated with one implementation; while a slot name may be associated with several implementations holding one implementation at a time. In this context, a class is a collection of variables and slots are called **behavior slots**. The values of class variables represent the state of an instance of the class, and the values of behavior slots represent implementations of methods of the class. A behavior slot (different from the definition of slots in other languages) is defined as follows:

Definition:

A behavior slot SL is a pair (spec, imp) where spec is the specification(s), and imp is the implementation(s) of the behavior slot (see Figure 7.8).

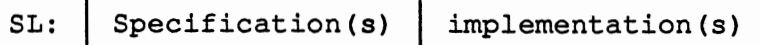


Figure 7.8: Representation of a behavior slot

In the above definition, a behavior slot may have several specifications associated with different implementations. In this model we consider a specification as a constant and an implementation as a variable. Therefore, a behavior slot has one specification that may be associated with a set of implementations. The set of implementations of a specification consists of all possible implementations of that specification in different classes. Hereafter, the term slot implies a behavior slot.

As an example we can consider the behavior of modems. Modems with higher baud rate can communicate with modems of lower baud rate at the lower rate. This suggests that a modem with the higher baud rate has several behaviors to choose from. We can represent this situation using classes and behavior slots. A specific modem can be viewed as an object. A modem object is an instance of a class that contains the behavior of modems of different baud rates. The behavior of an object can be represented by a slot named Connect. The slot Connect of a modem object responds to messages received from other modem objects to establish a communication line. The transmission rate is determined by the baud rate of the slower modem. We can view the determination of the baud rate as choosing an implementation of the slot Connect. For illustration, consider the following Bell/AT&T modem types [Black 87].

```

Type 103J of speed 300 bps
Type 202T of speed 1200 bps
Type 201C of speed 2400 bps
Type 208A of speed 4800 bps
Type 209  of speed 9600 bps

```

Classes that represent these modems are represented hierarchically in Figure 7.9.

These classes inherit the implementations of the slot Connect to communicate with the slower modems. An object of type 209, for example, is able to communicate with all

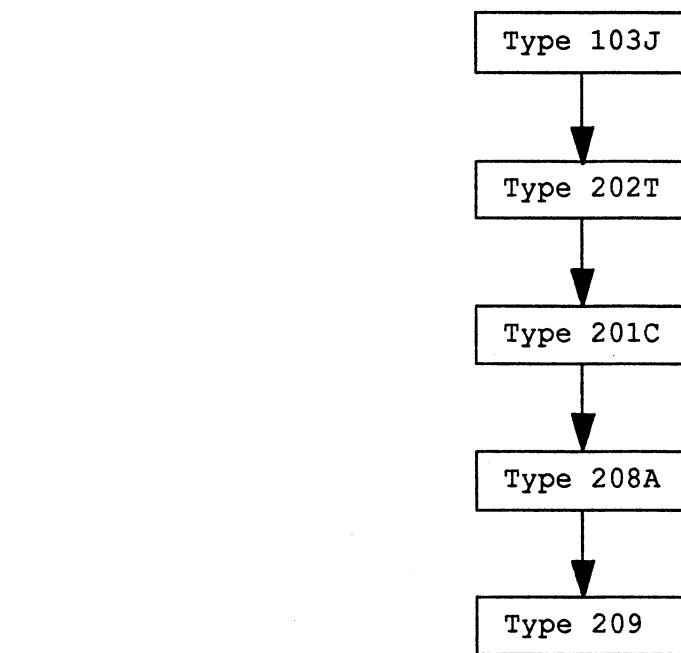


Figure 7.9: A hierarchy of Bell/AT&T modem types

other types. Therefore, it inherits all implementations of the slot `Connect` provided in the other types. The slot `Connect` has the same specification (e.g., `Connect(integer: speed)`) in all classes, and is associated with different implementations. Therefore, `Connect` is a behavior slot of one specification and several implementations. (This is an adaptation of an example suggested by Professor Gail Kaiser). The next section outlines the syntax used in this model.

7.4.1 Syntax of Behavior Slots

A class using behavior slots consists of three sections: variables, slots, and the implementations of the slots. The "variables" section is devoted to variable declarations. The "slots" section consists of the specifications of all newly defined slots and inherited slots that require new implementations. Re-implemented slots are preceded by the keyword `re_imp`. Inherited slots that do not require re-implementation are not included

in the slots section. The last section includes the actual implementations of all slots specified in the slots section. Figure 7.10 illustrates the class STACK using behavior slots (The code of examples used in this section is Pascal-like syntax).

```

Class STACK
Cbegin
  variables: max_len: constant := 10;
               top   : integer;
               s     : array [max_len] of char;

  slots: reset(): void;      push(char) : void;
          pop()  : char;      top-of()   : char;
          empty(): boolean;  full()     : boolean;

  implementations:
    slot reset() begin top:=0; end;
    slot push (c:char) begin top:=top+1; s[top]:=c; end;
    slot pop() begin return (s[top]); top:=top-1; end;
    slot top-of begin return (s[top]); end;
    slot empty() begin return (top == 0); end;
    slot full() begin return (top == max_len); end;
Cend.

```

Figure 7.10: Declaration of the class STACK using behavior slots

In this model we are dealing with multi-implementation slots. Therefore, it is convenient to treat all implementations of a slot as one group. In the following section we present the concept of an aggregate and apply it to the implementations of slots.

7.5 Aggregates and Behavior Slots

Generally speaking, an aggregate is a collection of objects referenced by a single name. Aggregates have different usages in OOPs. They allow building "un-serialized hierarchies of abstractions" and incorporate several language features (including concurrency, delegation, and message passing) that may simplify the programming task [Chien 90]. In the context of behavior slots, we view a slot conceptually as an aggregate. An aggregate is defined as follows.

Definition:

An aggregate (conceptual slot) is a collection of implementations associated with a set of handlers, and is referenced by a single name. Implementations are procedures (programs) that are independent of each other and can execute simultaneously. Handlers are selection functions that determine the acceptance of messages received by the slot and select the appropriate implementations. The aggregate name is the slot name (see Figure 7.11).

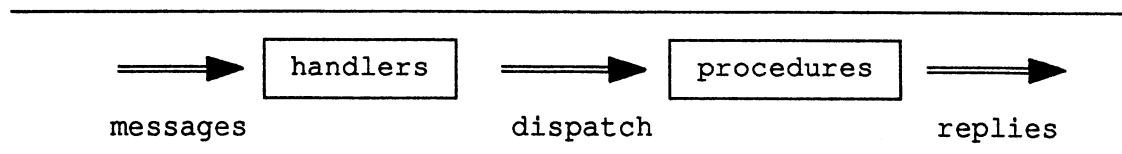


Figure 7.11: Representation of an aggregate (conceptual slot)

A slot receives messages from different objects and handlers select the invoked procedures. (An important characteristic of an aggregate of implementations is that it is a multi-access entity that may accept more than one message at a time and may return a number of simultaneous results as well). For illustration, consider the classes of the Cartesian and polar representations of a point illustrated in Figure 7.12.

In Figure 7.12, the class `CART_POINT` is the superclass of the class `POLAR_POINT`. The slot `setpoint` initializes a point in either representation, and it requires two arguments for either representation. It has the same specification in both classes, but different implementations. Specific types are not provided in the specification, rather a generic type is used. For an instance of the class `POLAR_POINT`, there are two implementations for the slot `setpoint` to choose from; whereas instances of the class `CART_POINT` have access to one implementation only. The aggregate representation of the slot `setpoint` in the class `POLAR_POINT` is illustrated in Figure 7.13.

In Figure 7.13, the set of procedures consists of the two implementations of the classes `CART_POINT` and `POLAR_POINT`. The slot `setpoint` may receive messages from different objects. The handlers analyze these messages (in this case based on the qualifier or class name and the type of arguments) and select the appropriate procedure

```

Class CART_POINT
Cbegin
  variables: Xval : integer;
             Yval : integer;
  slots: setpoint (type,type) : void;
         offset (integer,integer) : void;
  implementations:
    slot setpoint(x : integer, y : integer)
      begin Xval:=Xval+x; Yval:=Yval+y; end;
    slot offset(i : integer, j : integer)
      begin Xval:=Xval+i; Yval:=Yval+j; end;
Cend.
Class POLAR_POINT child of CART_POINT
Cbegin
  slots: re_imp setpoint (type,type) : void;
  implementations:
    slot setpoint(length: real, angle: real)
      begin
        Xval:=int (length*cos(angle));
        Yval:=int (length*sin(angle));
      end;
Cend.

```

Figure 7.12: Cartesian and polar representations of a point

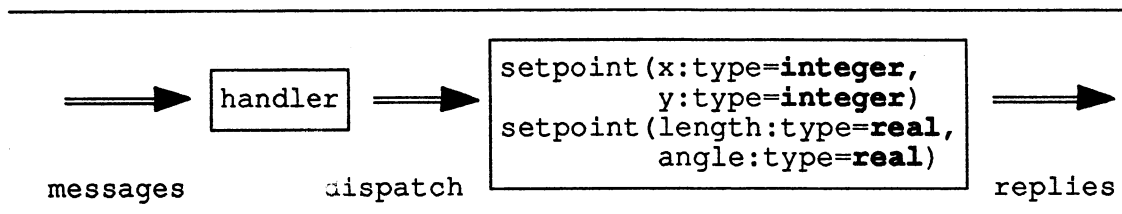


Figure 7.13: Aggregate representation of the slot setpoint

(implementation). Here, the handler is a function that takes the qualifier and arguments as inputs and produces calls to the invoked procedures as outputs.

As mentioned earlier, we are concerned with the implementations of slots and their reuse. The concept of message as used by current OOPs discourage if not prohibit this type of reuse. Therefore, it is necessary to modify the notion of a "message" in order to accommodate the goal of this model, which is the capability to selection of any previous implementation of a slot provided in ancestor class(es). In the following subsection we present a generalization of the concept of a message.

7.6 Generalized Messages

Message passing is a communication model for objects to interact with one another. The receiving object can determine whether to answer a message or not and how [Nierstrasz 86]. An object invokes a method of another object by sending a message. We present the notion of a generalized message in the context of behavior slots.

The notion of a generalized message is founded on the philosophy that it should be possible to select any slot's implementation that is available along the inheritance hierarchy. Therefore, it is necessary that all implementations (not the details of the implementation) above a receiving object be visible (known) at any point in the inheritance hierarchy. Selection of a particular implementation of a slot is application-dependent. Here we develop the idea of a generalized message. The syntax of a generalized message may have the form:

$$\text{message } (O, S, D)$$

where O is the receiving object, S is the requested slot, and D is the discriminant (additional information) that may be required by the receiving object for selecting an implementation of the slot S .

The discriminant D is an information packet. Such information is necessary when the receiving object O has knowledge about several implementations (for the invoked slot S) distributed over different classes along the inheritance path. In this case, the given information is used by the receiver to select the appropriate implementation. The discriminant D may be missing when there are no alternative implementations for the invoked slot S , which the receiving object O can select from. In this case, only one implementation of the slot S is known to object O . The approach adopted by current OOPLs falls into this specific case.

As an example for illustrating the point, we have adopted the **login** procedure in UNIX [Sobell 89], which is facilitated by the process **getty**, and have modified it to fit

the framework of objects. Two objects `logobj` and `gettyobj` are used to represent the login procedure and the `getty` process respectively. It is assumed that they communicate through messages.

Object `gettyobj` invokes the slot `I/O-control`. `I/O-control` is associated with several procedures. Two of these procedures are: `Output-Upper-Case` that turns on the uppercase output mode, and `Output-Lower-Case` that turns on the lower case output mode. Object `gettyobj` activates the `Output-Upper-Case` procedure if the user's login name is non-empty and it does not include any lowercase letter. Otherwise, the procedure `Output-Lower-Case` is activated. The objects `logobj` and `gettyobj` along with their message communication are illustrated in Figure 7.14.

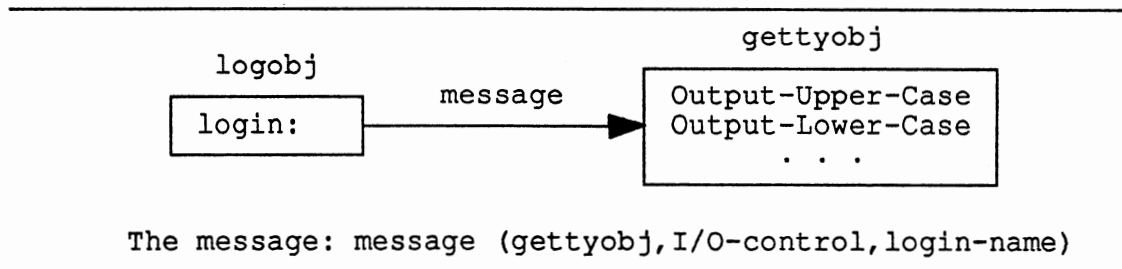


Figure 7.14: A generalized message

In Figure 7.14 the receiving object is `gettyobj`, the slot is `I/O-control`, and the discriminant is the case of the user's login name sent by `logobj`. `Gettyobj` has two implementations associated with the slot `I/O-control`: uppercase and lowercase outputs. It uses the case of the user's login name as the factor to select the appropriate implementation. (In the UNIX system, the procedure `getty` calls the login procedure).

7.6.1 Procedure Calls

In most OOPLs, a message has the form "message (O,m)", and only one implementation is available for method `m`. This approach has the advantage of efficient

implementation. For example, the message "message (O,m)" can be translated into a procedure call of the form: `O.m(actual parameters)`. This format eliminates the discriminant parameter since the decision about the invoked method is made implicitly in advance using the "." operator.

In the procedure call format, the receiving object and the appropriate implementation of the invoked method are determined at compile-time. Pointers to objects can be used to determine the receiving object and the invoked method at run time. The appropriate implementation of the invoked method is determined based on the object pointed at by the pointer. For example, C++ provides virtual functions that may be associated with different implementations in subclasses. C++ dynamically determines the appropriate implementation for each call of a virtual function. The selection depends on the object pointed at rather than the type of the pointer itself [Pohl 89]. Therefore, in some existing languages, we can manipulate the available features in order to invoke an implementation provided in ancestor class.

The concept of a generalized message provides a formal approach rather than an indirect one. It also facilitates some compilation checks. A class may use a particular method implementation in an ancestor class by using a full-name reference to that implementation. In this case the ancestor class name is the discriminant. A full-name reference is a form of a procedure call that includes the class name as a qualifier for the selection among the possible implementations of the invoked method.

In the following section the concept of implementation inheritance is introduced based on the concepts discussed in the previous sections.

7.7 The Proposed Implementation Inheritance Model

In this section we bring together the concepts of behavior a slot, an aggregate, and a generalized message to develop the idea of implementation inheritance. In the object-

oriented paradigm, inheritance is a mechanism for sharing common features among classes. A subclass inherits the specifications of slots provided by its superclass(es). A subclass either uses the implementations used in the superclass(es) or it may provide new implementations for some of the inherited slots. Therefore, the subclass has to use the most recent implementation of an inherited slot and has no knowledge of any previous implementation provided by ancestor classes.

One of the objectives of this model is to relax the above restriction and develop a new approach to inheritance. The new approach grants classes the knowledge of previous implementations of the inherited slots. The idea behind this approach is that whenever a slot is associated with different implementations at different levels of the inheritance hierarchy, these implementations should be known to classes located at lower levels of the hierarchy. Therefore, a class can inherit the specification of a slot along with the desired implementation. We call this approach **Implementation Inheritance (I-inheritance)**.

The semantics of I-inheritance simply states that an instance of a class may use any slot implementation provided by the ancestor class(es) in addition to the implementations in the class itself. This ability allows a programmer to achieve efficient reuse of existing implementations provided for inherited slots because one of the important concepts in facilitating code reuse - identification and access [Biggerstaff 89] - is combined with the concept of message. In the following section we present the conceptual view set forth by the idea of I-inheritance.

7.8 Conceptual View of Implementation Inheritance

Since a behavior slot can be associated with several implementations in different classes, it seems appropriate to think of all implementations of a slot as a collection, and treat them as one set of possible values for that slot. Therefore, we are conceptually

dealing with multi-implementation slots. To provide a conceptual view for multi-implementation slots, we adopt the concept of an aggregate [Chien 90] and apply it to the set of implementations of a slot. Each multi-implementation slot is represented by one aggregate. The concept of I-inheritance is described by the relationship among aggregates representing multi-implementation slots of an inheritance hierarchy. For illustration, consider the inheritance hierarchy given in Figure 7.15.

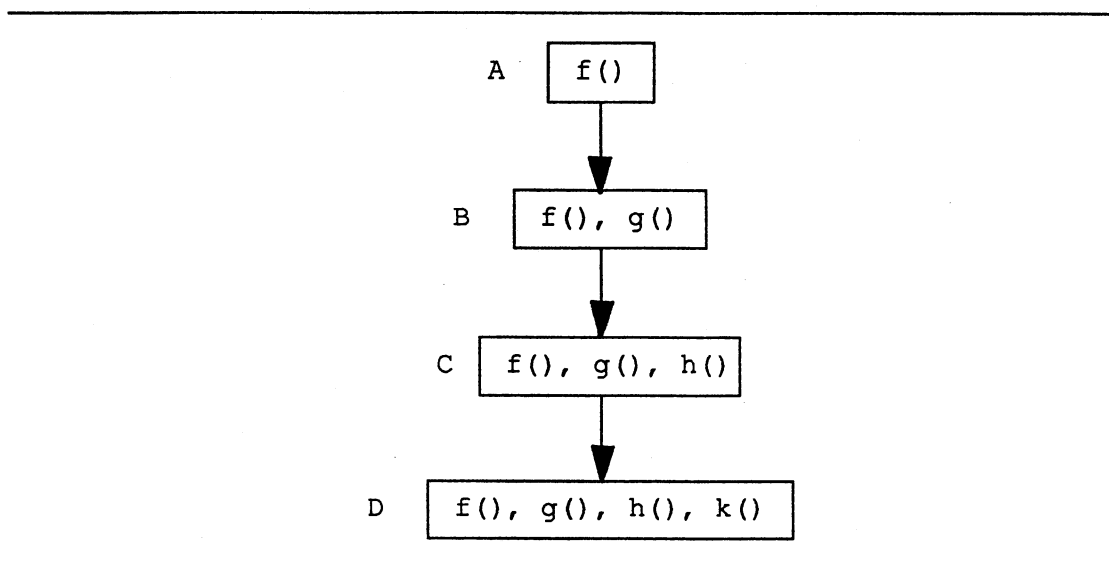


Figure 7.15: Inheritance hierarchy among some classes

In Figure 7.15, the multi-implementation slots are $f()$, $g()$, and $h()$. Each slot is associated with several implementations in different classes. Conceptually, we view each slot as an aggregate consisting of a handler and a set of procedures. Handlers perform the selection process of a procedure and procedures are the implementations in different classes. In this representation we use the notation $A:f()$ to indicate the implementation of the slot $f()$ in class A . The aggregate representation of the multi-implementation slots in Figure 7.15 is given in Figure 7.16.

Figure 7.16 represents individual aggregates for the slots $f()$, $g()$, and $h()$. To represent the I-inheritance relationship among the classes of Figure 7.15, we relate

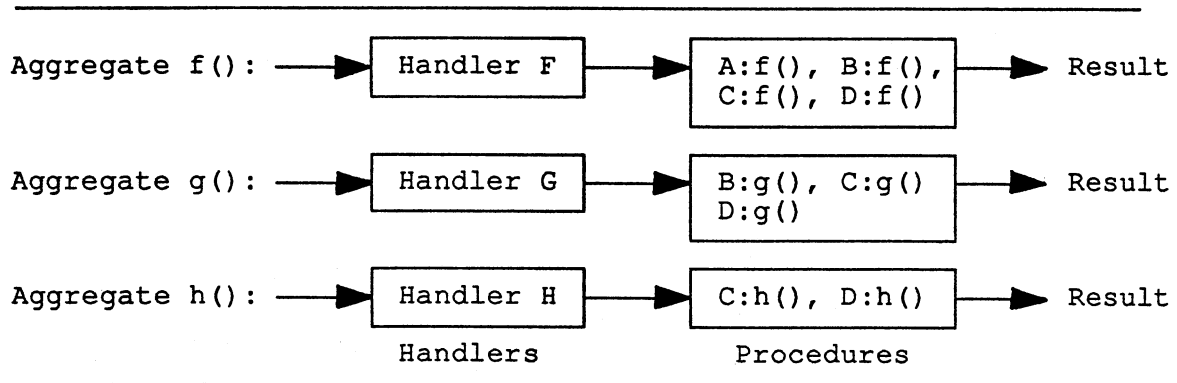


Figure 7.16: Aggregate representation of the slots f(), g(), and h() in Figure 7.15

handlers to classes as follows.

Class B uses Handler F
 Class C uses Handlers F and G
 Class D uses Handlers F, G, and H

A class uses one or more handlers when the class includes slots that have several implementations in ancestor classes. For example, class A uses only the handler F because only slot f() has an implementation in class A. On the other hand, class D uses three handlers since the slots f(), g(), and h() in class D have several implementations in ancestor classes. For notational convenience, let us assume that

HB is the set of handlers used by class B (i.e., {F}),
 HC is the set of handlers used by class C (i.e., {F,G}), and
 HD is the set of handlers used by class D (i.e., {F,G,H}).

Note that handler sets HB, HC, and HD select specific implementations of each slot at each level of the inheritance hierarchy. For example, the handler set HB and HC select specific implementations of the slot f() at the level of classes B and C. For class B, the handler set HB selects from A:f() and B:f(); while for class C, the handler set HC selects from A:f(), B:f(), and C:f(). Figure 7.16 can be modified to reflect the relationship among these handlers. The modification is illustrated in Figure 7.17.

In Figure 7.17, because of the inheritance relationship between classes, each set of procedures includes the contents of the set of procedures associated with the previous

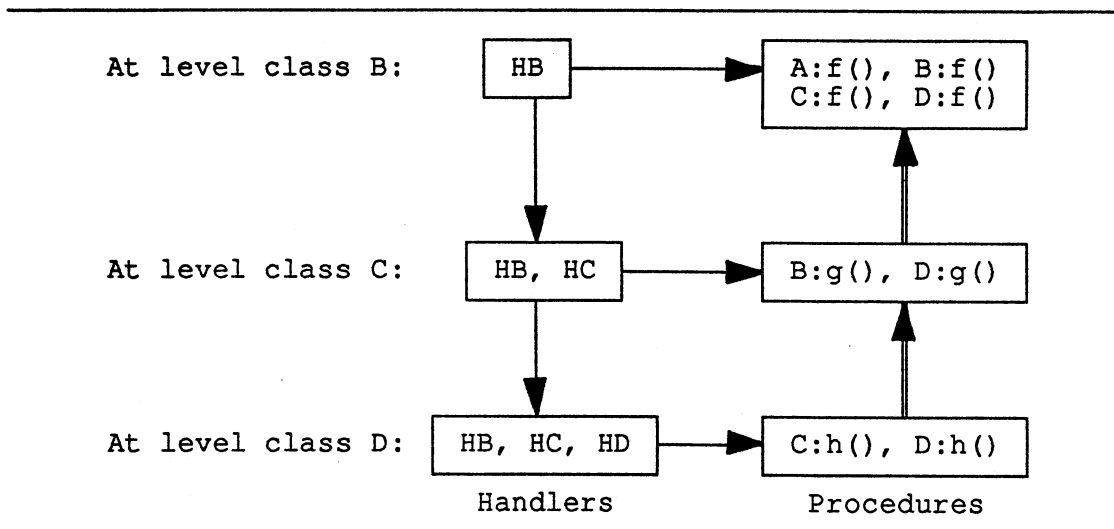


Figure 7.17: Relationships among handlers of aggregates in Figure 7.16

level. We denote this inclusion using feedback double arrows between sets of procedures. Therefore, an implementation of a slot may belong to more than one aggregate and a handler of an aggregate may be shared among several aggregates as well. The inter-relationship among the aggregates and I-inheritance is depicted in Figure 7.18.

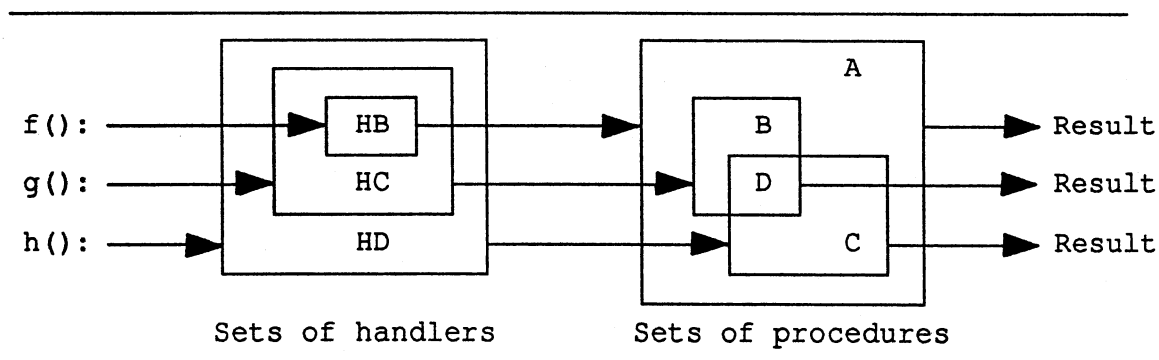


Figure 7.18: Inter-relationship between aggregates and I-inheritance

In Figure 7.18, the inclusion relationship among handlers represents the inheritance relationship illustrated in Figure 7.15. Handlers of HD include handlers of HB and HC because class D inherits from the classes B and C that use these handlers. No handler is

contained in HB because class B inherits from class A that does not use handlers. In the set of procedures of a slot we use the class names to indicate the locations of implementations of that slot.

To exhibit the relationship between the current and the following sections, we need to explain the correspondence between the concept and implementation of I-inheritance. At the concept level, the selection of an implementation is achieved by the handlers of an aggregate; while at the implementation level, the handlers are represented by a run-time data structure and associated with a selection algorithm (see the following section). Procedures are the slot's implementations distributed over different classes. The mapping between the concept and implementation levels is shown in Figure 7.19.

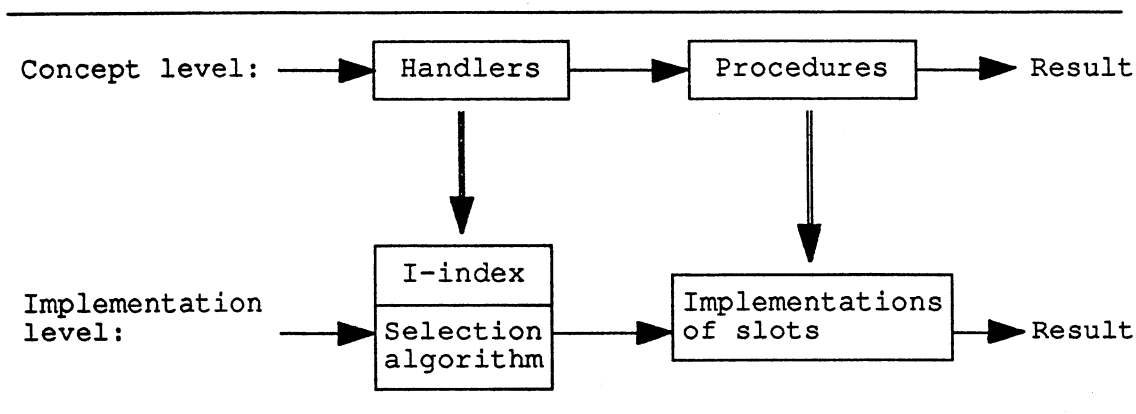


Figure 7.19: Mapping between the concept and implementation of I-inheritance

In the following section we present an implementation scheme which will accomplish the objectives set forth by the concept of I-inheritance.

7.9 An Implementation Scheme for Implementation Inheritance

A slot may have several implementations in different classes. Our approach in this scheme is to keep track of all possible implementations of a slot and provide access path(s) to each implementation. The retrieval of a specific implementation depends on the

message. To retrieve an implementation, we need:

- 1) Information in the message indicating the desired implementation. For example, full-name qualification allows the system to choose the invoked implementation.
- 2) Information about different implementations of a slot in ancestor classes. In the implementation scheme presented in this section, this information includes names of ancestor classes that provide different implementations, and pointers to the implementations themselves. This information is represented by a run-time data structure called **Implementation-Index** (I-index).

An I-index is a two-dimensional array of entries containing addresses of the different implementations of a slot. Rows are indexed by slot names and columns are indexed by the class names where implementations are provided. An entry contains a value (address of a procedure's code) if the slot is associated with an implementation in the corresponding class. Otherwise, the entry is undefined. An undefined entry indicates that the slot is inherited by the corresponding class, but is not re-implemented. For example, consider the class USE_ANY illustrated in Figure 7.20.

```

Class USE_ANY child of POLAR_POINT
  Cbegin
    slots: create_point(type,type) : void;
            ...      -- other slots

    implement: slot create_point(type,type)
      begin
        -- Create a point using the implementations
        -- of the slot setpoint in ancestor classes.
      end;
      ...      -- other implementations
  Cend.

```

Figure 7.20: Class USE_ANY using implementations of the slot setpoint

In Figure 7.20, class USE_ANY inherits from the class POLAR_POINT shown in Figure 7.12. The slot create_point creates a point using the implementations of the slot

setpoint provided in ancestor classes of the class USE_ANY. Therefore, instances of the class USE_ANY can use any of these implementations. The I-index associated with the class USE_ANY is given in Figure 7.21. The I-index is a 2x2 array. The entries

I-index[offset, CART_POINT] and
I-index[offset, POLAR_POINT]

indicate that the slot offset has only one implementation in the class CART_POINT. The entry I-index[offset, POLAR_POINT] is undefined. That is, class POLAR_POINT inherits (not re-implements) the slot offset.

	cart_point	polar_point
setpoint	pointer to slot setpoint (integer, integer)	pointer to slot setpoint (real, real)
offset	pointer to slot offset (integer, integer)	undefined

Figure 7.21: An I-index example

7.9.1 Organization of the I-index

Slot names marking rows of the I-index of a class are the names of slots associated with implementations in ancestor classes. The order of rows (slots) depends on the appearance of their implementations in ancestor classes. The class names marking columns of the I-index of a class are the names of ancestor classes of that class. Order of columns (class names) depends on an ordering process applied to the inheritance graph. This process produces an ordered list (called a C-list) of all classes of the graph. An algorithm to construct a C-list is given below.

Algorithm Class-List

Input : Inheritance graph C.

Output: Ordered C-list.

Method: step 1: r_level=1; hgt=height(C); C-list=[];

```

step 2: Call list_roots(r_level, hgt, C-list);
step 3: Return C-list.

```

The procedure `list_roots` is defined recursively as follows:

```

list_roots(r_level, hgt, C-list) =
[
  C-list=C-list||[root nodes]r_level||
    [list_roots(r_level+1, hgt, C-list)] if r_level ≤ hgt
  C-list=[] if r_level > hgt
]

```

where the variable `hgt` is the height of the graph, `r_level` is the root level of the current graph, and `C-list` is the ordered list of all classes of the graph `C`. The procedure `list_roots` recursively lists child classes from left to right following the listing of their superclass(es). Figure 7.22 illustrates the application of the algorithm `Class_List` on a given class hierarchy. The Figure is self-explanatory.

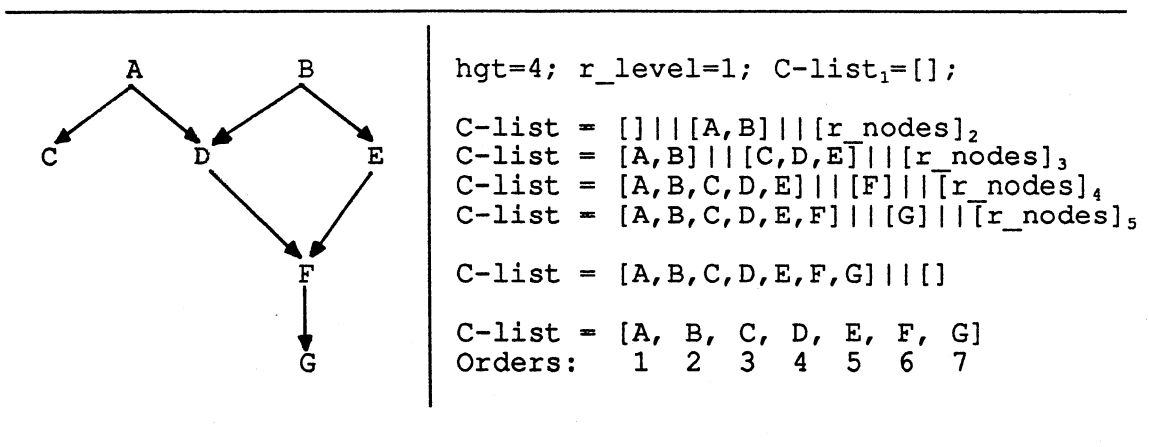


Figure 7.22: The C-list of a multiple inheritance hierarchy

7.9.2 Size of the I-index

The size of an I-index of a subclass depends on the number of inherited slots associated with new implementations, and the number of the ancestor classes that introduce different implementations for the inherited slots. If there are n inherited slots

in a subclass, the maximum size of the I-index is "n * the number of all ancestor classes". This case implies that the every ancestor class introduces a new implementation for at least one inherited slot re-implemented in the subclass. The minimum size of the I-index is "1 * m" where $m \geq 1$ is number of ancestor classes. This case implies that at most m ancestor classes provide implementations for at most one inherited slot re-implemented in the subclass. For example, consider the single inheritance hierarchy in Figure 7.23.

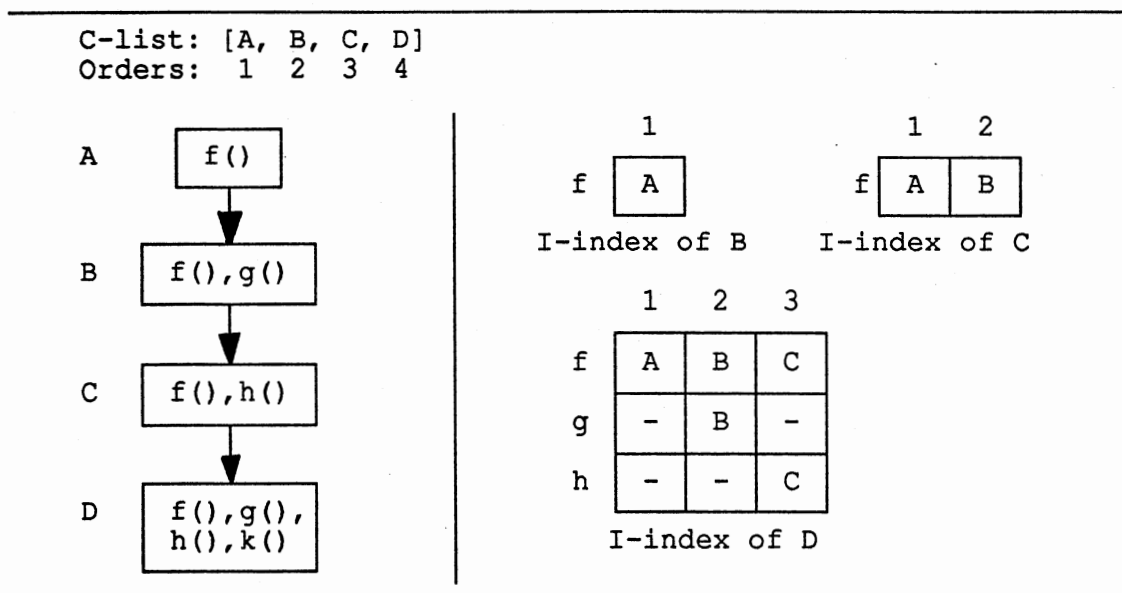


Figure 7.23: Representation of I-indices of classes

In Figure 7.23, each class re-implements at least one inherited slot. The I-index of class B is of the minimum size since class B inherits and re-implements only slot f(). Note that the I-index of class B is of the maximum size as well. The I-index of class C is of the minimum size. Class C inherits the slots f() and g(), but only f() is re-implemented by the class C. Therefore, instances of the class C can use two previous implementations for the slot f().

The I-index of class C would be of the maximum size (2x2) if the inherited slot g() has been re-implemented in class C. The I-index of class D is of the maximum size.

Each re-implemented slot in the class D has an implementation in at least one ancestor class. The size of an I-index is not affected by the type of the inheritance hierarchy (single or multiple).

7.9.3 Construction of the I-index

A subclass may add new slots and re-implement inherited slots as well. A class that re-implements inherited slots is associated with an I-index that keeps track of previous implementations. A class that does not re-implement inherited slots need not be associated with an I-index since no new implementations are introduced by the class. In the following subsection, we illustrate the concept of I-index construction using single and multiple inheritance hierarchies. Hereafter, in stead of using the class names to label columns of the I-index, we will use their orders in the C-list. Moreover, we use the class names in entries of the I-index to show the classes that provide previous implementations for the corresponding slots.

7.9.3.1 Single Inheritance. Consider the single inheritance hierarchy illustrated in Figure 7.24. Class B inherits the slots f() and g() from the class A, and introduces the new slot h(). Class B does not re-implement any of the inherited slots. Therefore, the classes A and B are not associated with I-indices since the class A originally defines all of its slots, and the class B provides no new implementations for inherited slots. On the other hand, the classes C and D are associated with I-indices since they provide new implementation for inherited slots.

In Figure 7.24, class C re-implements g() and introduces k(). The I-index of the class C is simply one-entry array since only slot g() has previous implementation in the ancestor class A. Therefore, instances of the class C have two implementations for the slot g() to choose from; one is provided in class A as indicated in the entry [g,1], and the second one is provided in the class C itself. Note that A (the value stored in the entry

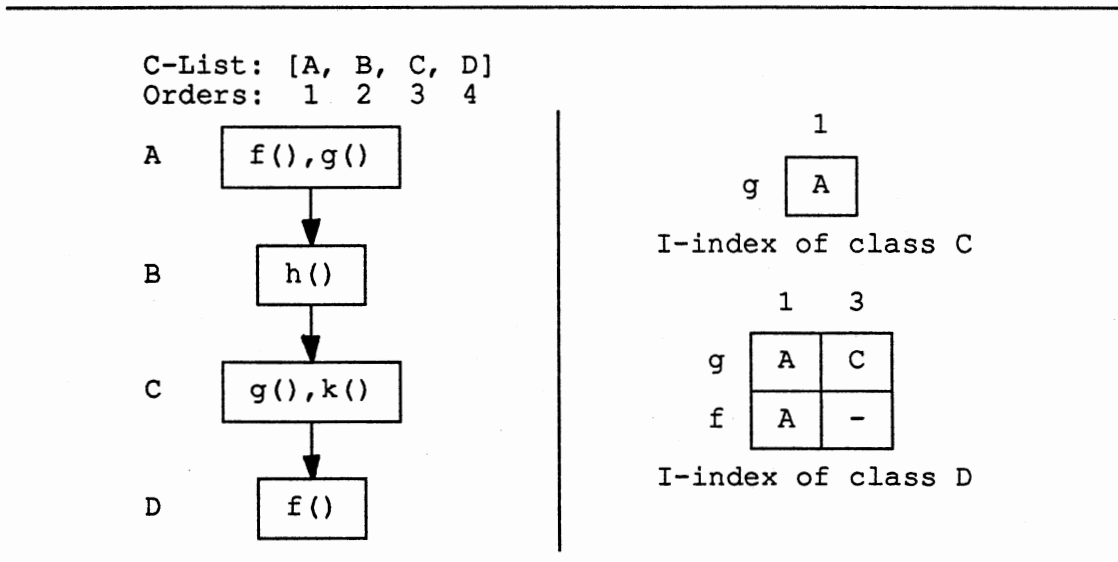


Figure 7.24: Illustration of an I-index

[g,1] of either I-index) is the address of the implementation provided in the class A. The slot k() is not included in the I-index since it has no previous implementations in ancestor classes, it may be included in the I-indices of descendants of the class C.

Class D re-implements the slot f(). Instances of the class D have different implementations for the slots f() and g() to choose from. The I-index of class D is a 2x2 array. It provides information about the implementations of the inherited slots f() and g(). Slot g() has implementations in the classes A, and C; and slot f() has implementations in the classes A and D. Note that the slot g() marks the first row of the I-index of the class D because slot g() has been re-implemented in class C before the slot f() re-implemented in class D.

The I-index of a subclass may be constructed by adding information to the I-index of its superclass. The newly provided information is about inherited slots that are re-implemented in the subclass. If the superclass has no I-index (i.e., each slot has a single implementation in ancestor classes), a new I-index is constructed for the subclass. In Figure 7.24, the I-index of class D is an extension of that of the class C. The additional information is about the new implementations of the slots f() and g() that are re-

implemented in the classes C and D. Instances of the class D have the choice to use implementations from the class A or C.

7.9.3.2 Multiple Inheritance. Consider the multiple inheritance hierarchy illustrated in Figure 7.25. Class C inherits and re-implements the slot $f()$ from the class A. Class D inherits and re-implements the slot $f()$ from the class A and the slot $g()$ from the class B. Class E inherits the slot $f()$ from both classes C and D. Suppose that class E re-implements the slot $f()$ inherited from the class C. Therefore, instances of the class E can use either $A:f()$, $C:f()$, $D:f()$, or $E:f()$. Class E also inherits and re-implements the slots $g()$ and $h()$ from the class D. Finally, class F inherits and re-implements the slot $f()$ from the class E. The only classes that do not require I-indices in Figure 7.25 are the classes A and B since they originally define their own slots. The I-indices of other classes in the Figure are given in Figure 7.26.

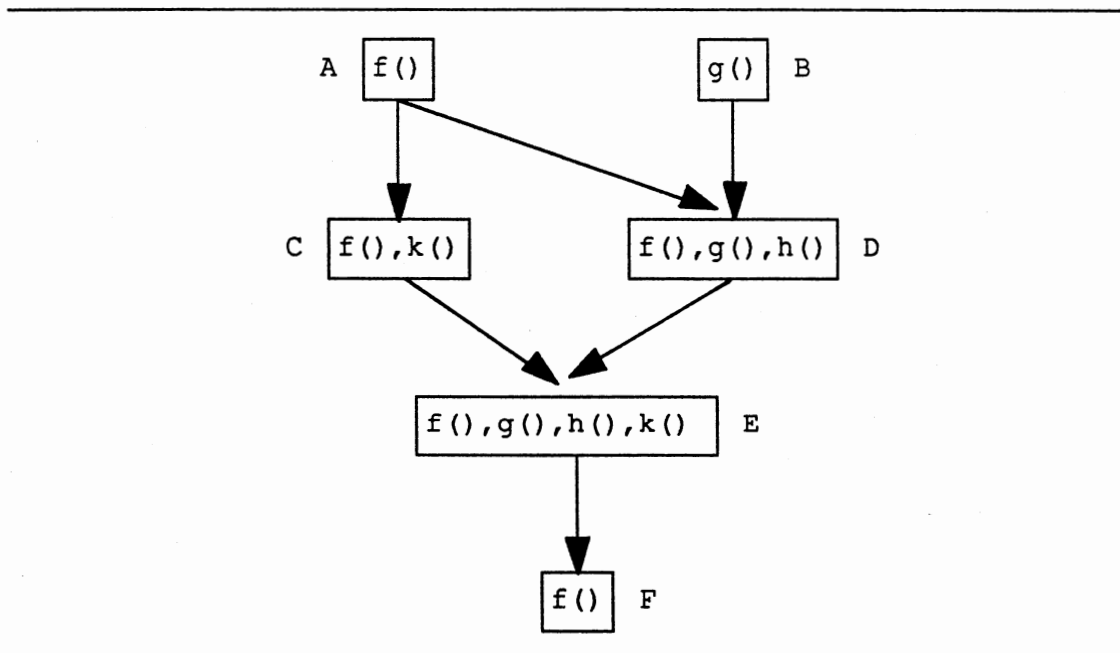


Figure 7.25: Multiple inheritance among classes

C-list: [A, B, C, D, E, F]
 Orders: 1 2 3 4 5 6

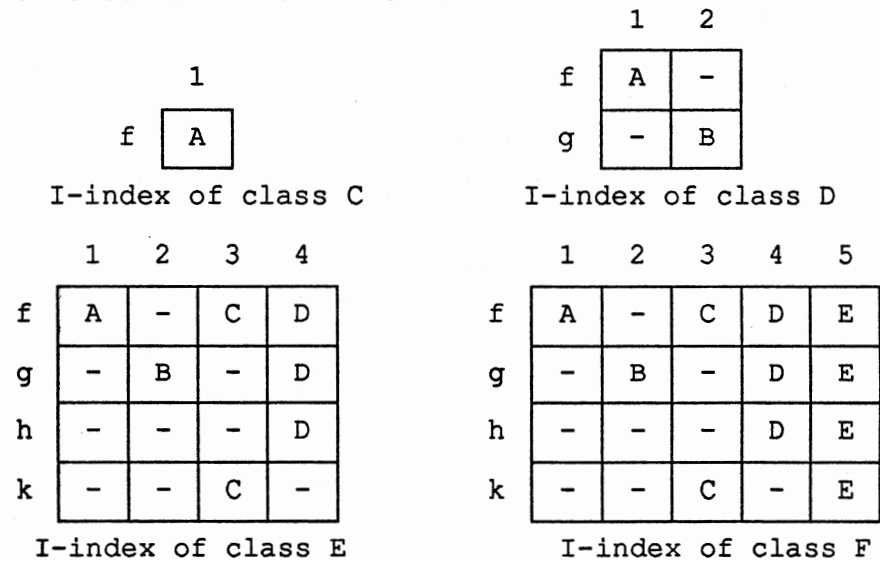


Figure 7.26: I-indices of the classes in Figure 7.25

When a class inherits from two or more superclasses, the order of the superclasses in the C-list determines the order of slots in the I-index of the inheriting class. Slots that are inherited and re-implemented from a superclass of lower order appear before slots inherited and re-implemented from superclasses of higher orders. For example, in the I-index of the class D in Figure 7.26, slot f() inherited from the class A appears in the first row; while slot g() inherited from the class B appears in the second row because the order of the class A is lower than the order of the class B in the C-list. The same argument applies to other I-indices in the figure. Like single inheritance, the contents of a new I-index includes the contents of previous I-indices. That is, the new I-index is a super set of the previous ones.

7.9.4 Optimization of the I-index

In sections 7.9.3 we noticed that the deeper we travel along the inheritance path, the faster the I-index array grows. We also associated each class with an I-index, and the

last I-index is a super set of the previous one(s).

The first thought of optimization is to allow classes that are associated with I-indices to use the last I-index, and eliminate all previous indices. For example, the I-index of the class F in Figure 7.26 includes all information provided in the I-indices of the class C, D, and E. Therefore, the classes C, D, and E can use the I-index of the class F.

When all classes use the same I-index, each class is concerned about partial information of that I-index. A class should not be able to use or refer to information about implementations provided in descendant classes. For example, when class D in Figure 7.26 uses the I-index of class F, class D should refer only to the four entries ([f,1], [f,2], [g,1], [g,2]) of that I-index. These entries are equivalent to its own I-index in the figure, and they provide information about the implementations provided in the ancestor class A and B of the class D in Figure 7.25.

This approach of optimization requires some modification in the selection algorithm above. Here, we need to know the order of the receiving object's class, and change the postcondition of the third step of the algorithm. The new algorithm is the same as before except for the steps 2 and 3. Here, we use the function `class (object_name)` that returns the class name of `object_name`. In the modified algorithm, the variable `receiver_name` is the name of the object O that received the message, and the variables `receiver_order` and `class_order` are the orders of the receiving object's and the discriminant classes respectively. The modified steps 2 and 3 of the previous version of the algorithm are given below.

```

step 2: object_name=message[1]; -- receiving object's name
        class_name=message[3]; -- discriminant field
step 3: obtain the order of the class of the object_name and
        class_name from the corresponding C-list as follows:
        receiver_name=class(object_name);
        precondition:
            class_name and receiver_name ∈ C-list;
        r=1; flag1=flag2=FALSE;
        class_order=receiver_order=0;
        while NOT (flag1 AND flag2) do
            begin
                if C-list[r]=class_name

```

```

then begin
    class_order=r; flag1=TRUE;
end;
else if C-list[r]=receiver_name
then begin
    receiver_order=r; flag2=TRUE;
end;
r=r+1;
end; -- while
postcondition: class_order ≤ receiver_order;

```

The modified algorithm guarantees that a class cannot refer to information about implementations in descendant classes. For a given class A, any class of a lower order than the order of A is either located at a higher level of the inheritance hierarchy or the same level of the class A. Classes at the same level have no inheritance relationship. Therefore, the postcondition

$$\text{class_order} < \text{receiver_order}$$

implies that the discriminant class is located at a higher level than the class of the receiving object. In the case where

$$\text{class_order} = \text{receiver_order}$$

the receiving object is an instance of the discriminant class, and this is equivalent to a message without discriminant (i.e., missing discriminant).

7.9.5 Selection Mechanism

The concepts of behavior slot and generalized message require a selection mechanism that determines the appropriate implementation of the invoked slot. The generalized message is developed to carry sufficient information that can be used in the selection process applied to implementations of slots. The concept of aggregate includes handlers that conceptually perform the selection process and activate the appropriate implementation of a slot. From the perspective of implementation reuse, identification and access mechanisms are incorporated into the message.

In the implementation scheme, the I-indices correspond to handlers. Entries of an

I-index include addresses of (pointers to) implementations of slots in different classes. Here, we define the selection mechanism as an algorithm that takes a generalized message as an input, applies a conversion process to the message components, and determines the address of the appropriate implementation code (if exists). In section 7.6, the generalized message was described in terms of the receiving object O, the slot S, and the discriminant D. In general, the mapping of a message to a specific implementation depends on the specification of the discriminant. In this section we examine a selection mechanism when the discriminant D is a class name.

When an object receives a message, a search algorithm is used to look for the recent implementation of the invoked method. The search starts at the class of the receiving object and continues up along the inheritance path until an implementation is found or an error message is returned. In the proposed model, we assume that such an algorithm exists and we refer to it by the name **Search**. Algorithm Search is invoked by the selection algorithm **Select** described below. In the following algorithm, we assume that the message is a record structure with three fields: object name O, slot name S, and discriminant D.

Algorithm Select

Input : A generalized message (O, S, D) .

Output: Execution of the appropriate implementation or an error message.

Method:

```

step 1: input a generalized message;
step 2: class_name=message.D; -- the discriminant field
step 3: obtain the order of the class_name from the
        corresponding C-list as follows:
        precondition: class_name ∈ C-list;
        r=1;
        while C-list[r] <> class_name do r=r+1;
        order=r;
        postcondition: order ≤ length(C-list);
step 4: slot_name=message.S; -- the slot name field
step 5: address=I-index[slot_name,r];
        if address="undefined"
        then call algorithm Search;
        else activate the procedure pointed to by address;

```

The conversion process uses orders of the C-list elements and the slot names as

indices to entries of the I-index. The response to a message is the execution of the invoked implementation or the response provided by algorithm Search.

For illustration, consider the class hierarchy given in Figure 7.25. Suppose that *d* is an instance of the class D, and *d* receives the following messages

```
message1(d, f(), A) and message2(d, g(), A)
```

The C-list and class orders of the class hierarchy in Figure 7.25 are:

```
C-list: [A, B, C, D, E, F]
orders: 1 2 3 4, 5, 6
```

Figure 7.27 illustrates the selection process applied to these messages. The figure is self explanatory.

```

step 1: message1(d, f(), A);
step 2: class_name=A;
step 3: order[A]=1;
step 4: slot_name=f();
step 5: address=I-index[f(), 1]=A; -- I-index of the class D
        Activate that implementation;
step 1: message2(d, g(), A);
step 2: class_name=A;
step 3: order[A]=1;
step 4: slot_name=g();
step 5: address=I-index[g(), 1]=undefined;
        call algorithm search;
step 5: STOP.

```

Figure 7.27: Application of algorithm Select

In Figure 7.27, the result of step 5 in processing message1 is the address of the implementation of the slot *f()* provided in the class A. This implementation will be activated as a response to message1. On the other hand, the result of step 5 in processing message2 indicates that class A does not provide an implementation for the slot *g()*. Therefore, algorithm Search is invoked to look for the recent implementation of the slot *g()* in an ancestor class, which is class B in this case. If no implementation exists, an error message is issued by algorithm search in this regard as a response to message2. Note that class D uses its I-index to determine the addresses of the invoked methods.

7.10 Discussion

In this section we discuss the impact of I-inheritance on the issue of encapsulation, and the relationship between the concept of multi-methods defined in CLOS and the concept of generalized message introduced in section 7.6.

7.10.1 Implementation Inheritance and Encapsulation

Alan Snyder [Snyder 86a,87] has indicated that inheritance in most OOPLs compromises encapsulation by exposing the implementation details to inheriting classes. He has outlined a set of requirements for full support of encapsulation with inheritance. These requirements and their benefits are given below.

- 1) Providing different external interfaces for objects of the class and inheriting classes. Such interfaces allow the designer to re-implement the class methods with out affecting inheriting classes. New implementations are compatible with the external interface provided for inheriting classes. This requirement implies that classes are encapsulated, and methods are accessed by inheriting classes only through the defined external interface.
- 2) Preventing direct access to the instance variables inherited from ancestor classes. Direct access to the instance variables of ancestor classes limits the designer's freedom to change, rename, or remove an instance variable without affecting its inheriting classes. Therefore, direct access to the instance variables compromises encapsulation. Snyder has indicated that using methods to access the inherited instance variables preserves the benefits of encapsulation and prevents direct access by descendant classes.
- 3) Hiding the use of inheritance by not making it part of the external interface. When the use of inheritance becomes part of the external interface and is visible to inheriting classes, then changing the use of inheritance among classes (inheritance hierarchy) may affect the inheriting classes and require changes in descendant classes. This issue affects the designer's ability to safely change the inheritance hierarchy without changing the

implementation details of classes.

It should be noted that I-inheritance is an extension of specification inheritance. I-inheritance is based on the identification and access concept of reuse. It allows users to reuse as much as possible of implementations provided in ancestor classes. The objective is to maximize the reuse of code segments by providing a minimum knowledge about the previous implementation(s) in ancestor classes. The impact of I-inheritance on encapsulation in the context of the above requirements is discussed below.

In the specification inheritance, users need not understand how methods are implemented since each method is associated with only one implementation. I-inheritance encounters the previous implementations of a method as alternatives for users to choose from. In order to make such selection, users of I-inheritance should know the availability of these implementations. Such knowledge is provided in the external interface of the class.

In I-inheritance, users need not know the implementation details, such as the specific implementation details of an algorithm, since the different specifications of previous implementations imply the functionality of each implementation. When all previous implementations of a method have the same specification, language constructs such as the "." operator in C++ and **call_method** construct in `CommonObjects` can be used to determine the invoked implementation. Therefore, I-inheritance has little impact on encapsulation and information hiding since the external interface provides enough knowledge to select a previous implementation of a method.

The issue of accessing inherited instance variables is not affected by I-inheritance since I-inheritance is an extension of the underlying specification inheritance. The use of methods to access inherited instance variables can also be used in the context of I-inheritance. Using methods preserves encapsulation and prevents direct access to the instance variables. Excluding the instance variables from the external interface preserves encapsulation by allowing the class designer to change, rename, or remove an instance

variable inherited from an ancestor class without affecting inheriting classes.

Since I-inheritance implies direct access to methods defined in the ancestor classes, this feature of I-inheritance violates the third requirement above by exposing the use of inheritance, and hence partially violates encapsulation. I-inheritance compromises the visibility of inheritance in favor of reusability of implementations. It allows users to avoid re-writing implementations that are already provided in the ancestor classes, and therefore, it contributes to the reduction of code and coding efforts.

7.10.2 Multi-Methods and Generalized Messages

In this subsection we contrast the notion of generalized message introduced in section 7.6 and the notion of multi-method used by CLOS. Both notions provide the language with a mechanism for invoking the appropriate method from among a set of available methods. The generalized message follows the *message passing* approach; while the multi-method follows the *generic function* approach.

In the message passing approach, the invoked method is determined by the class of the receiving object to which the message is sent. In some cases, the receiving object may have the knowledge about several methods of the same name. Hence, more information may be required by the receiving object in order to determine the invoked method. Discriminant is an abstraction of such information.

A **multi-method** in CLOS is a method that specializes more than one parameter. That is, its applicability depends on the types (classes) of two or more arguments. However, a multi-method is a method name associated with two or more different implementations in different classes (i.e., two or more methods of the same name). In the generic function approach, the invoked method is determined by the type of the arguments to which the method is applied.

Although the two approaches are intended to accomplish the same purpose, they

are conceptually different and the generalized message is more general in the abstract sense. In the message passing approach, the receiving object has been given the responsibility to determine and choose the invoked method. We view this approach as a distributed approach in terms of responsibility. On the other hand, the generic approach adopted by CLOS uses a generic dispatch process that chooses the appropriate invoked method. We view this approach as a centralized approach of responsibility. Therefore, both approaches are distinctly different.

In the generic function approach, the type of arguments is the fixed discriminant information that the dispatch process uses in order to determine the invoked method. The discriminant information of a generalized message can be any information (including the type of arguments) that provides sufficient information to the receiving object to determine the invoked method. Therefore, the generalized message approach can handle more general conditions than the multi-method approach.

7.11 Summary

As more complex software systems are being built, the significance of software reuse is further emphasized by users and researchers. Object-oriented programming provides support for code reuse through inheritance. The inheritance models currently used by OOPLs can be characterized as "Specification Inheritance". That is, classes inherit the specification of methods from ancestor classes. Even if several implementations are available to a method, the semantics of specification inheritance restrict the object to using the most recent implementation of an inherited method. Other implementations are in general inaccessible to that object.

In this work a scheme has been developed a scheme to provide objects with the capability to reuse previous implementations of inherited methods, and to choose any implementation provided by any ancestor class. We characterize this approach as

"Implementation Inheritance" (I-inheritance). Even though it is possible to manipulate features of some languages in order to access previous implementations of inherited methods, so far there is no such model available in the literature.

The concept of I-inheritance and the proposed implementation scheme are based on the ideas of slot, aggregate, and generalized message. We use the notion of behavior slot to represent a method that have several implementations in different classes. A behavior slot has a specification and implementation. We considered its specification as a constant, and implementation as a variable. The presented model allows multiple implementations to be associated to a slot. To associate the set of implementations of a slot with its specification, we use the concept of aggregate. Here, an aggregate is a collection of implementations associated with a set of handlers that perform the selection of an implementation requested by a message.

When invoking an implementation of a slot, the selection process requires information to identify the invoked implementation. Such information is provided by the requesting object, and is carried with the message. A message in current languages does not provide such information since an object has no choice for an implementation but the most recent one. Therefore, we introduce the notion of a generalized message to accommodate the selection of any implementation of a slot. A generalized message provides a discriminant (information) sent by the requesting object to the receiving object. The discriminant allows the requesting object to specify the requested implementation, and the receiving object to correctly select the appropriate implementation.

The suggested implementation scheme is based on a data structure called "I-index". The I-index of a class contains information about all slots' implementations provided by ancestor classes. A retrieval algorithm is presented for a special case. Finally, the impact of I-inheritance on encapsulation is discussed, and the notion of generalized message and multi-method are contrasted.

CHAPTER VIII

SUMMARY, CONCLUSIONS, AND FUTURE WORK

A key distinguishing feature of OOP is the inheritance mechanism and the purpose that it is designed to serve. Inheritance is an important program design concept that promotes and facilitates code reusability and extensibility. This dissertation examines the inheritance models adopted by current OOPs, and proposes new models that facilitate code reuse and reliable extension of existing software components in the development of software systems. The proposed models are based on an extensive study and analysis of the inheritance models provided in some of the most common OOPs.

The first part of this dissertation (Chapters 2, 3, and 4) provides an extensive literature review, and examines different features of inheritance in a number of well-known languages. A classification (taxonomy) based on the main characteristics of inheritance is presented as a binary tree. Nodes in the tree represent sets of characteristics, and edge are annotated by the selected languages. An inheritance model (or, equivalently, the language designed around it) at the leaf level inherits the characteristics represented by all nodes along the path from that leaf node to the root node. The classification approach provides a framework for identifying new inheritance models in the space of inheritance models allowable within the confines of the taxonomy. It also helps determine the similarities and differences among inheritance models based on their location on the tree.

The first part of the dissertation also explores the approaches to reusability and extensibility (along with some other related issues such as visibility, information hiding,

external interfaces, and the visibility of inheritance) in C++ and Eiffel. The strengths and weaknesses of each language from code reuse perspective are also highlighted.

The second part (Chapters 5, 6, and 7) describes newly proposed inheritance models that are based on some of the discussion of the first part. Three inheritance models are proposed. They are designed to overcome three major problems found in the current models.

1) Upon analyzing the current models, their advantages were gleaned and a new model was proposed by defining a Two-faceted object-based Inheritance Model (TIM). This is the first inheritance model, which consists of two orthogonal sets of objects. It provides single and multiple inheritance based on the message passing paradigm. It also provides semantics for object creation and deletion. TIM provides full support for encapsulation and other related issues including information hiding, access techniques, subtyping, and the visibility of inheritance. TIM is compared with the existing models in terms of the inheritance features provided by these models.

2) Generally, a preferred representation of a problem is one that provided by a model that reflects the "natural" structure of the problem. The object-oriented paradigm is generally touted as paradigm that provides a better correspondence between a problem and its representation. Even so, many real-life situations still cannot be well reflected in the object-oriented paradigm. The strict hierarchical inheritance model does not provide a satisfactory representation for situations where the dependency among objects is bi-directional.

The second inheritance model, A feedback inheritance model, is proposed to relax some of the constraints of the hierarchical inheritance model and provide control over dependency among related classes. These relaxations furnish the tools to discourage users from attempting tricky and costly approaches using the hierarchical model to control the dependency among classes. Such attempts may result in inefficient and expensive software. The proposed model allows a superclass and its subclass(es) to exchange

attributes. In general, feedback inheritance avoids replicating functions among classes, increases reusability, eases software maintenance, and facilitates the sharing of functions in a distributed environment.

Additionally, the feedback inheritance model maintaining consistency with issues such as information hiding, access and visibility, and encapsulation. The notions of synthesized attributes, synthesized interfaces, and clans are introduced as part of the definition of the model. The notion of clan relaxes the message passing technique and potentially increases the probability of answering a message as well as the use of attributes among classes.

3) As more complex systems are being built, the significance of software reuse is further emphasized by researchers and users. OOP provides support for code reuse through inheritance. The inheritance models currently used by OOPLs can be characterized as "Specification Inheritance". That is, classes inherit the specification of the methods from ancestor classes. Even if several implementations are available to a method, the semantics of specification inheritance restricts an object to using the most recent implementation of an inherited method, and other implementations are inaccessible to that object.

The third inheritance model, Implementation inheritance (I-inheritance), has been developed to provide objects with the capability to reuse previous implementations of the inherited methods and to choose any implementation provided by any ancestor class. Even though it is possible to manipulate some of the features of some OOPLs to access previous implementations of inherited methods, to date there is no such model available in the literature.

The concept of I-inheritance and its proposed implementation scheme are based on the notions of slots, aggregates, and generalized messages. The notion of a behavior slot is used to represent a method that has several implementations in different classes. A behavior slot has a specification and an implementation. The specification is considered

as a constant and the implementation as a variable. The model allows multiple implementations to be associated with a slot. To associate a set of implementations of a slot with its specification, the concept of an aggregate is used in the definition of the model.

Since an object in current OOPs has no choice for an implementation except for the most recent one, the notion of a generalized message is introduced to accommodate the selection of any implementation of a slot. A generalized message provides a discriminant (information) sent by a requesting object to a receiving object. The discriminant allows the requesting object to specify the requested implementation and the receiving object to select the appropriate implementation correctly.

A pervasive theme of this dissertation is that all proposed models promote source code reuse and facilitate the development of software components in the context of OOP. This theme is essential for both centralized and distributed programming environments. Besides the refinements to each individual model suggested in [Al-Haddad 90 a,b] [Al-Haddad 91 b,c] [Al-Haddad 92b], this dissertation suggests some viable issues for future work. Such issues include the development of formal/mathematical models, the implementation and evaluation of the proposed models, and the incorporation of these model to the concurrent OOP and distributed object-oriented environments.

BIBLIOGRAPHY

[Ada 83] Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A 1983. United States Department of Defense (American National Standards Institute, Inc.), February 1983.

[Ada 79] Preliminary ADA Reference Manual and Rationale for the Design of the ADA Programming Language. ACM SIGPLAN Notices, Vol. 14, No. 6, (parts A and B), June 1979.

[Agha 86a] G. Agha. Actors: A Model for Concurrent Computations in Distributed Systems. The MIT Press, 1986.

[Agha 86b] G. Agha. "An Overview of Actor Languages." OOP Workshop (ACM SIGPLAN Notices (October 1986)), June 1986, pp. 58-67.

[Agha 87] G. Agha and C. Hewitt. "Concurrent Programming Using Actors." Research Directions in Object-Oriented Programming. Edited by B. Shriver and P. Wegner, [MIT Press Series in Computer Systems], The MIT Press, Cambridge, Massachusetts, 1987, pp. 37-53.

[Agrawala 91] R. Agrawala and A. Arvind. "Static Type Checking of Multi-Methods." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Phoenix, Arizona, November 1991, pp. 113-128.

[Algol 68] Proceedings of the 1975 International Conference on ALGOL 68. Edited by G.E. Hedrick, Oklahoma State University, June 10-12, 1975.

[Al-Haddad 92a] H.M. Al-Haddad, K.M. George, and M.H. Samadzadeh. "Multiple Representation of Abstract Data Types and Reuse of Realizations." Proceedings of the ACM Symposium on Applied Computing, Kansas City, Missouri, March 1992, To appear.

[Al-Haddad 92b] H.M. Al-Haddad, K.M. George, and M.H. Samadzadeh. "A Feedback Inheritance Model." The Journal of Systems and Software, To be published Spring 1992.

[Al-Haddad 91a] H. Al-Haddad, K.M. George, and M.H. Samadzadeh. "Multiple Representations of Abstract Data Types (Extended Abstract)." Proceedings of the ACM/IEEE Symposium on Applied Computing, Kansas City, Missouri, April 1991, p. 403.

[Al-Haddad 91b] H. Al-Haddad, K.M. George, and M.H. Samadzadeh. "A Taxonomy of Object-Oriented Programming Languages." Proceedings of the 1st Golden West International Conference on Intelligent Systems, Reno, Nevada, June 1991, pp. 169-174.

[Al-Haddad 91c] H. Al-Haddad, K.M. George, and M.H. Samadzadeh. "Approaches to Reusability in C++ and Eiffel." The Journal of Object-Oriented Programming, Vol. 4, No. 5, September 1991, pp. 34-45.

[Al-Haddad 90a] H. Al-Haddad, K.M. George, and M.H. Samadzadeh. "Description of a New Approach to Object Inheritance." Proceedings of the ACM/IEEE Symposium on Applied Computing, Fayetteville, Arkansas, April 1990, pp. 289-296.

[Al-Haddad 90b] H. Al-Haddad, K.M. George, and M.H. Samadzadeh. "TIM: A Unified Approach to Object Inheritance." Proceedings of the 7th International Conference on Systems Engineering, Las Vegas, Nevada, July 1990, pp. 780-787.

[Alagic 89] S. Alagic. Object-Oriented Database Programming. Springer-Verlag, 1989.

[Alws 85] K. H. Alws and I. Schapeler. "Experience with Object-Oriented Programming." Proceedings of the International Joint Conference on Theory and Practice of Software Development, Vol. 2, Berlin, Germany, March 1985, pp. 435-452.

[America 85] P. America. "Design Issues in Parallel Object-Oriented Programming." Proceedings of the Second International Conference on Parallel Computing, Berlin, Germany, December 1985, pp. 325-330.

[America 86] P. America. "Object-Oriented Programming: A Theoretician's Introduction." Bull European Association Theoretical Computing Society (Austria), No. 29, June 1986, pp. 69-84.

[America 87] P. America. "Inheritance and Subtyping in Object-Oriented Programming." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 234-242.

[Anderson 75] G. Anderson and E. Jensen. "Computer Interconnection Structure: Taxonomy, Characteristics, and Examples." ACM Computing Surveys, Vol. 7, No. 4, December 1975, pp. 197-213.

[AT&T 85] UNIX System V, AT&T C++ Translator, Release Notes #307-175, AT&T, 1985.

[AT&T 86] AT&T C++ Translator Release 1.2, Addendum to Release Notes #307-005, AT&T, 1986.

[AT&T 89a] UNIX System V, AT&T C++ Language System Release 2.0. Selected Readings #307-144, AT&T, 1989.

[AT&T 89b] UNIX System V, AT&T C++ Language System Release 2.0. Product Reference Manual #307-146, AT&T, 1989.

[Baldassari 88] M. Baldassari, V. Berti, and G. Bruno. "Object-Oriented Conceptual Programming Based on PROT Nets." Proceedings of the International Conference on Computer Languages, Miami, Florida, October 1988, pp. 226-233.

[Bancibon 86] F. Bancibon. "A Logic Programming/Object-Oriented Cocktail." ACM SIGPLAN Notices, Vol. 15, No. 3, September 1986, pp. 11-21.

[Berlin 90] L. Berlin. "When Objects Collide: Experience with Reusing Multiple Class Hierarchies." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications OOPSLA '87, Ottawa, Canada, October 1990, pp. 181-190.

[Berson 87] S. Berson, E. Silva, and R. Muntz. "Object-Oriented Methodology for the Specification of Markov Models." Technical Report CSD-870030, Computer Science Department, UCLA. July 1987, 27 pages.

[Bezivin 87] J. Bezivin. "Some Experience in Object-Oriented Simulation". Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 394-405.

[Biggerstaff 89] T. Biggerstaff and A. Perlis. Software Reusability: Applications and Experience. Volumes I & II, ACM Press and Addison-Wesley, Reading, Massachusetts, 1989.

[Black 86] A. Black, N. Hutchinson, E. Jul, and H. Levy. "Object Structure in Emerald System." ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 78-86.

[Blaschek 89] G. Blaschek, G. Pomberger, and A. Stritzinger. "A Comparison of Object-Oriented Programming Languages." The International Journal of Structured Programming, Vol. 10, No. 4, April 1989, pp. 187-197.

[Bobrow 86] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. "CommonLoops: Merging Lisp and Object-Oriented Programming." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), Portland, Oregon, September 1986, pp. 17-29.

[Bobrow 88] D. Bobrow, D. Moon, L. DeMichiel, R. Gabriel, S. Keene, and G. Kiczales. "CommonLisp Object System Specification X3J13." ACM SIGPLAN Notices, Vol. 23, Special issue on CommonLisp, September 1988.

[Booch 86] G. Booch. "Object-Oriented Development." IEEE Transaction on Software Engineering, Vol. SE-12, No. 2, February 86, pp. 211-221.

[Booch 91] G. Booch. Object-Oriented Design with Applications. The Benjamin/Cumming Publishing Company, Reading, Massachusetts, 1991.

[Borgida 86] A. Borgida. "Exceptions in Object-Oriented Languages." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 107-119.

[Borning 86a] A. Borning and D. Ingalls. "Multiple Inheritance in Smalltalk-80." Proceedings of the Fall Joint Computer Conference (FJCC '86), Dallas, Texas, November 1986, pp. 234-240.

[Borning 86b] A. Borning. "Classes Versus Prototypes in Object-Oriented Languages." Proceedings of the Fall Joint Computer Conference (FJCC '86), Dallas, Texas, November 1986, pp. 36-40.

[Bracha 90] G. Bracha and W. Cook. "Mixin-Based Inheritance." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 303-311.

[Briot 89] J. Briot and P. Cointe. "Programming with Explicit Metaclasses in Smalltalk-80." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '89), New Orleans, Louisiana, October 1989, pp. 419-431.

[Bruce 86] K. Bruce and P. Wegner. "An Object-Oriented Algebraic Model of the Subtypes in Object-Oriented Languages." ACM SIGPLAN Notices, Vol. 21, No. 11, October 1986, pp. 163-172.

[Budd 85] T. Budd. A Little Smalltalk User Manual. Department of Computer Science, The University of Arizona. Tucson, Arizona, 1985.

[Budd 91] T. Budd. An Introduction to Object-Oriented Programming. Addison-Wesley, Reading, Massachusetts, 1991.

[Cannon 82] H. Cannon. Flavors : A Non-Hierarchical Approach to Object-Oriented Programming. Symbolics, Inc., 1982.

[Cardelli 84] L. Cardelli. "A Semantics of Multiple Inheritance." Proceedings of the Semantics of Data Types Conference, Sophia Antipolis, France, June 1984, pp. 51-67.

[Cardelli 86] L. Cardelli and P. Wegner. "On Understanding Types, Data Abstractions, and Polymorphism." ACM Computing Surveys, August 1986, pp. 29-36.

[Carre 90] B. Carre and J. Geib. "The Point of View Notion for Multiple Inheritance." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 312-321.

[Casais 88] E. Casais. "An Object-Oriented System Implementing KNO's." Proceedings of the Conference on Office Information Systems, Palo Alto, California, March 1988, pp. 282-290.

[Chambers 90] C. Chambers and D. Ungar. "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs." Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, New York, New York, October 1990, pp. 150-163.

[Chambers 91] C. Chambers and D. Ungar. "Making Pure Object-Oriented Languages Practical." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Phoenix, Arizona, November 1991, pp. 1-15.

[Chien 90a] A. Chien and W. Dally. "Experience with Concurrent Aggregates: Implementation and Programming." Proceedings of the Fifth Distributed Memory Conference, Charleston, South Carolina, 1990, pp. 1040-1049.

[Chien 90b] A. Chien and W. Dally. "Concurrent Aggregates (CA)." ACM SIGPLAN Notices, Vol. 25, No. 3, March 1990, pp.177-186.

[Christian 86] K. Christian. A Guide to Modula-2. Springer-Verlag, 1986.

[Codani 87] J. Codani. "Microprogramming in Object-Oriented Style: An Experience with Lisp Co-processor." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 181-186.

[Cointe 87] P. Cointe. "Metaclasses are First Class: The ObjVlisp Model." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 156-167.

[Cook 86] P. Cook. "Languages and Object-Oriented Programming." Software Engineering Journal, Vol. 1, No. 2, March 1986, pp. 73-80.

[Cook 89] P. Cook and J. Palsberg. "A Denotational Semantics of Inheritance and Its Correctness." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '89), New Orleans, Louisiana, October 1989, pp. 433-443.

[Cox 86a] B. Cox and B. Hunt. "Objects, Icons, and Software-ICs." BYTE, Vol. 11, No. 8, August 1986, pp. 161-176.

[Cox 86b] B. Cox. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley, Reading, Massachusetts, 1986.

[Cox 88] I. Cox. "C++ Language Support for Guaranteed Initialization, Safe Termination, and Error Recovery in Robotics." IEEE International Conference on Robotics and Automation, 1988, pp. 641-643.

[Danforth 88] S. Danforth and C. Tomlison. "Type Theories and Object-Oriented Programming." ACM Computing Surveys, Vol. 20, No. 1, March 1988, pp. 29-72.

[Detlefs 87] D. Detlefs, M. Herlthy, and J. Wing. "Inheritance of Synchronization and Recovery Properties in Avalon/C++." Proceedings of the 20th Annual Hawaii International Conference on System Sciences (HICSS '90), Vol. I, January 1987, pp. 416-423.

[Dewhurst 87] S. Dewhurst. "Object Representation of Scope During Translation (C++)." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 79-86.

[Ditlefs 88] D. Ditlefs M. Herlihy, and J. Wing. "Inheritance of Synchronization and Recovery Properties in Avalon/C++." IEEE Computer, December 1988, pp. 57-69.

[Dony 90] C. Dony "Exception Handling and Object-Oriented Programming: Towards a Synthesis." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 322-330.

[Duff 86] C. Duff. "Designing an Efficient Language." BYTE, Vol. 11, No. 8, August 1986, pp. 211-224.

[Edelson 87] D. Edelson. "How Objective Mechanisms Facilitate the Development of Large Software Systems in Three Programming Languages." ACM SIGPLAN Notices, Vol. 22, No. 9, September 1987, pp. 54-63.

[Faust 90] J. Faust and H. Levy. "The Performance of an Object-Oriented Threads Package." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 278-288.

[Franta 78] W. Franta. "SIMULA Language Summary." ACM SIGPLAN Notices, Vol. 13, No. 8, August 1978, pp. 243-244.

[Franz 87] M. Franz. "Succeeding C [C++ Language]." PC Tech Journal, Vol. 15, No. 9, September 1987, pp. 109-183.

[Freeman 83] P. Freeman. "Reusable Software Engineering: Concepts and Research Directions." Proceedings of the Workshop on Programming, New Port, Rhode Island, September 1983, pp. 2-16.

[Gabriel 89] R. Gabriel. "The CommonLisp Object System." AI Expert, March 1989, pp. 54-65.

[Gannon 77] J. Gannon. "An Experimental Evaluation of Data Type Conversions." Communications of the ACM, Vol. 20, No. 8, August 1977, pp. 584-595.

[Gehani 88] N. Gehani and D. Roome. "Concurrent C++: Concurrent Programming with Class(es)." Software Practice and Experience, Vol. 18, No. 12, December 1988, pp. 1157-1177.

[Geoffry 88] S. Geoffry. "Object-Oriented Programming Explained." Journal of Systems Management, Vol. 39, No. 7, March 1988, pp. 13-19.

- [Geonardi 87] L. Geonardi and P. Mello. "Combining Logic and Object-Oriented Programming Language Paradigms." Proceedings of the 20th Annual Hawaii International Conference on System Sciences (HICSS '20), Vol. I, January 1987, pp. 379-385.
- [Ghelli 91] G. Ghelli. "A Static Type System for Message Passing." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 91), Phoenix, Arizona, November 1991, pp. 129-143.
- [Gittins 86] M. Gittins. "The Role of Object-Oriented Programming in Knowledge Engineering." Proceedings of the International Conference on Knowledge Based Systems (KBS 86), London, England, July 1986, pp. 249-260.
- [Goguen 86a] J. Goguen. "Reusable Interconnecting Software Components." IEEE Computer, Vol. 19, No. 2, February 1986, pp. 16-28.
- [Goguen 86b] J. Goguen and J. Meseguer. "Extension and Foundation of Object-Oriented Programming." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 153-162.
- [Gold 91] E. Gold and M. Rosson. "An Instance-Centered Environment for Smalltalk." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Phoenix, Arizona, November 1991, pp.62-74.
- [Goldberg 83] A. Goldberg and D. Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, Massachusetts, 1983.
- [Goldberg 89] A. Goldberg and D. Robson. Smalltalk-80: The Language. Addison-Wesley, Reading, Massachusetts, 1989.
- [Grogon 89a] P. Grogon. Design Criteria for a Simple Object-Oriented Language. OOP89-5, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1989.
- [Grogon 89b] P. Grogon and A. Bennett. A Theory for Object-Oriented Languages (Extended Abstract). OOP-89-1, Department of Computer Science, Concordia University, Montreal, Quebec, Canada, 1989.
- [Guimaraes 91] N. Guimaraes. "Building Generic User Interface Tools: An Experience with Multiple Inheritance." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Phoenix, Arizona, November 1991, pp. 89-96.
- [Haarslev 90] V. Haarslev. "A Framework for Visualizing Object-Oriented Systems." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Ottawa, Canada, October 1990, pp. 237-244.
- [Habert 90] S. Habert and L. Mosseri. "COOL: Kernel Support for Object-Oriented Environment." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Ottawa, Canada, October 1990, pp. 169-177

[Hailpern 87] B. Hailpern and V. Nguyen. "A Model for Object Based Inheritance." Research Directions in Object-Oriented Programming. Edited by B. Shriver and P. Wegner, [MIT Press Series in Computer Systems], The MIT Press, Cambridge, Massachusetts, 1987, pp.147-164.

[Halbert 87] D. Halbert, P. O'Brien. "Using Types and Inheritance in Object-Oriented Languages." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 20-31.

[Hendler 86] J. Hendler. "Enhancement for Multiple Inheritance." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 98-106.

[Hudson 87] S. Hudson and R. King. "Object-Oriented Database Support for Software Environment." ACM SIGMOD Records, Vol. 16, No. 3, December 1987, pp. 491-503.

[Hur 87] J. Hur and K. Chon. "Overview of a Parallel Object-Oriented Language CLIX." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 315-323.

[Ishikawa 90] Y. Ishikawa, H. Tokuda, and C. Mercer. "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 289-302.

[Jacky 86] J. Jacky and I. Kalet. "Object-Oriented Approach to A Large Scientific Applications." ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986, pp. 368-376.

[Kaehler 86] T. Kaehler and D. Patterson. "A Small Test of Smalltalk." BYTE, Vol. 11, No. 8, August 1986, pp. 148-159.

[Kahn 87] K. Kahn, E. Tribble, M. Miller, and D. Bobrow. "Vulcan: Logical Concurrent Objects." Research Directions in Object-Oriented Programming, Edited by B. Shriver and P. Wegner, [MIT Press Series in Computer Systems], The MIT Press, Cambridge, Massachusetts, 1987, pp. 75-112.

[Kaiser 87a] G. Kaiser and D. Gorlen. "MELDing Data Flow and Object-Oriented Programming." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 254-265.

[Kaiser 87b] G. Kaiser. "A Low-Based Approach to Object-Oriented Programming." ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 482-493.

[Kaiser 90] G. Kaiser and B. Hailpern. "An Object Model for Shared Data." Proceedings of the International Conference on Computer Languages, Ottawa, Canada, October 1990, pp. 136-144.

[Keene 89] S. Keene. Object-Oriented Programming in CommonLisp. Addison-Wesley, Reading, Massachusetts, 1989.

[Kempf 87] J. Kempf, W. Harris, and A. Snyder. "Experience with CommonLoops." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 214-226.

[Kempf 88] J. Kempf, A. Paepcke, B. Beach, J. Moham, B. Mahbob, and A. Snyder. "Language Level Persistence for an Object-Oriented Application Programming Platform." Proceedings of the Twenty First Annual Hawaii International Conference on System Sciences (HICSS '21), Vol. II, January 1988, pp. 424-433.

[Kernighan 78] B. Kernighan. The C Programming Language. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kerr 86] R. Kerr. "Object-Based Programming: A Foundation for Reliable Software." Proceedings of the Fourteenth Simula User's Conference, Stockholm, Sweden, August 1986, pp. 159-165.

[Kerr 87] P. Kerr and D. Percival. "Use of Object-Oriented Programming in Time Series Analysis Systems." ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 1-10.

[Kirkerud 89] B. Kirkerud. Object-Oriented Programming with Simula. Addison-Wesley, Reading, Massachusetts, 1989.

[Klint 86] P. Klint. "Modularization and Reusability in Current Object-Oriented Languages." Proceedings of the CERN School of Computing, Geneva, Switzerland, 1986, pp. 65-77.

[Knapp 87] V. Knapp. "The Smalltalk Simulation Environment." Proceedings of the 1987 Winter Simulation Conference. Atlanta, Georgia, December 1987, pp. 146-151.

[Kreczmar 89] A. Kreczmar. "On Inheritance in Object-Oriented Programming." Advanced Programming Methodologies, Academic Press, 1989.

[Kristensen 87] B. Kristensen, O. Madsen, P. Pedersen, and K. Nugaard. "The BETA Programming Language." Research Directions in Object-Oriented Programming, Edited by B. Shriver and P. Wegner, [MIT Press Series in Computer Systems], The MIT Press, Cambridge, Massachusetts, 1987, pp. 7-47.

[Kukula 87] J. Kukula and S. Gupta. "Object-Oriented Programming with Speculative Parallelism for Parallel Processing." Proceedings of the 1987 IEEE International Conference on Computer Design: VLSI in Computers and Processors, Rey Brook, New York, October 1987, pp. 596-600.

[Laff 85] M. Laff and B. Hailpern. "SW2- An Object-Based Programming Environment." ACM SIGPLAN Notices, Vol. 20, No. 7, July 1985, pp. 1-11.

[Law 87] C. Law. "Smalltalk: Effectively Exploiting a New Technology." Proceedings of the International Technology: Emerging Opportunities and Programming. The Second Pan-Pacific Computer Conference, Singapore, Malaysia, August 1987, pp. 296-303.

[Lewis 91] J. Lewis, S. Henry, D. Kafura, and R. Schulman. "An Empirical Study of the Object-Oriented Paradigm and Software Reuse." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Phoenix, Arizona, November 1991, pp. 184-196.

[Lieberman 86] H. Lieberman. "Using Prototypical Objects to Implement Shared Behaviors in Object-Oriented Systems." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, September 1986.

[Love 86] T. Love. "The Concept of Object -Oriented Programming." Feedback 86: DSSD User's Conference. Overland Park, Kansas, October 1986, pp. 1-13.

[Loumis 87] M. Loumis, A. Shah, and J. Runbough. " An Object Modeling Technique for Conceptual Design." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 325-335.

[Lucas 89] P. Lucas. "Multiple Inheritance and Exceptions in Frame Systems." Technical Report CS-R8931, Center for Mathematics and Computer Science, August 1989.

[MacLennan 85] B. J. MacLennan. "A Simple Software Environment Based on Objects and Relations." ACM SIGPLAN Notices, Vol. 20, No. 7, July 1985, pp. 199-207.

[Madsen 86] O. Madsen. "Block Structure and Object-Oriented Languages." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 133-142.

[Madsen 87] O. Madsen and C. Nygaard. "An Object-Oriented Metaprogramming System." Proceedings of the 20th Annual Hawaii International Conference on System Sciences (HICSS '20), Vol. II, January 1987, pp. 406-415.

[Madsen 89] O. Madsen. "Virtual Classes: A Powerful Mechanism in Object-Oriented Programming." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '89), New Orleans, Louisiana, October 1989, pp. 397-406.

[Madsen 90] O. Madsen, B. Magmusson, and B. Pederson. "Strong Typing of Object-Oriented Languages Revisited." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 140-150.

[Marcke 88] K. van Marcke. "Towards Explicit Inheritance Schemes." Proceedings of the Twenty First Annual Hawaii International Conference on System Sciences (HICSS '21), Vol. II, January 1988, pp. 386-395.

- [McDonald 90] J. McDonald and W. Stuetzle. "Painting Multiple Views of Complex Objects." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 145-157
- [Meng 86] B. Meng. "Programming Gets Object-Oriented." Digital Design., Vol. 16, No. 10, September 1986, pp. 28-31.
- [Merrow 87] T. Merrow and J. Laursen. "A pragmatic System for Shared Persistent Objects." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 103-110.
- [Meulen 87] P. Meulen. "INSIST: Interactive Simulation in Smalltalk." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 366-376.
- [Meyer 87] B. Meyer. "Eiffel: Programming for Reusability and Extendibility." Proceedings of the Fifteenth SIMULA Conference, St. Helier, England, September 1987, pp. 109-118.
- [Meyer 88] B. Meyer. Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
- [Meyer 89] B. Meyer. "From Structured Programming to Object-Oriented Programming: The Road to Eiffel." The International Journal of Structured Programming, Vol. 10, No. 1, Springer Verlag, 1989, pp. 19-39.
- [Micallef 89] J. Micallef. "Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages." Journal of Object-Oriented Programming, April/May 1989, pp. 12-34.
- [Miranda 87] E. Miranda. "BrouHaHa- a Portable Smalltalk Interpreter." ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 345-365.
- [Moon 85] D. Moon and S. Keene. "Flavors: Object-Oriented Programming on Symbolic Computers." CommonLisp Conference, Boston, Massachusetts, December 1985, pp. 1-18.
- [Moon 86] D. Moon. "Object-Oriented Programming with Flavors." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), Portland, Oregon, September 1986, pp. 1-8.
- [Nierstrasz 86] O. Nierstrasz. "What Is the 'Object' in Object-Oriented Languages." Proceedings of the CERN School on Computing, Geneva, Switzerland, 1986, pp. 43-53.
- [Nierstrasz 89] O. Nierstrasz. "A Survey of Object-Oriented Concepts." Object-Oriented Concepts, Databases, and Applications, Edited by W. Kim and F. Lochovsky, Addison-Wesley, Reading, Massachusetts, 1989.
- [Nygaard 78] K. Nygaard and O. Dahl. "The Development of the SIMULA Language." ACM SIGPLAN Notices, Vol 13, No. 8, August 1978, pp. 245-272.

- [Nygaard 86] K. Nygaard. "Basic Concepts in Object-Oriented Programming." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 128-132.
- [Oldford 86] R. Oldford and S. Peters. "Object-Oriented Data Representation of Statistical Data Analysis." COMSTAT: Proceedings in Computational Statistics, Seventh Symposium, Rome, Italy, 1986, pp. 301-306.
- [OOP Workshop 85] "Object-Oriented Workshop." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1985.
- [OOPSLA '86] "Object-Oriented Programming Systems, Languages, and Applications", ACM SIGPLAN Notices, Vol. 21, No. 11, Portland, Oregon, September 1986.
- [OOPSLA '87] "Object-Oriented Programming Systems, Languages, and Applications", ACM SIGPLAN Notices, Vol. 22, No. 12, Orlando, Florida, October 1987.
- [OOPSLA '88] "Object-Oriented Programming Systems, Languages, and Applications", ACM SIGPLAN Notices, Vol. 23, No. 11, San Diego, California, September 1988.
- [OOPSLA '89] "Object-Oriented Programming Systems, Languages, and Applications", ACM SIGPLAN Notices, Vol. 24, No. 12, New Orleans, Louisiana, October 1989.
- [OOPSLA '90] "Object-Oriented Programming Systems, Languages, and Applications", ACM SIGPLAN Notices, Vol. 25, No. 12, Ottawa, Canada, October 1990.
- [OOPSLA '91] "Object-Oriented Programming Systems, Languages, and Applications", ACM SIGPLAN Notices, Vol. 26, No. 13, Phoenix, Arizona, November 1991.
- [Oucournan 87] R. Oucournan. "On Some Algorithms for Multiple Inheritance in Object-Oriented Programming." Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87), Paris, France, June 1987, pp. 291-300.
- [Paepcke 90] A. Paepcke. "PCLOS: Stress Testing CLOS Experiencing the Metaobject Protocol." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 194-200.
- [Palsberg 90] J. Palsberg and M. Schwartzbach. "Type Substitution for Object-Oriented Programming." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '90), Ottawa, Canada, October 1990, pp. 151-160.
- [Palsberg 91] J. Palsberg and M. Schwartzbach. "Object-Oriented Type Inference." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91), Phoenix, Arizona, November 1991, pp. 146-161.
- [Parnas 76] D. Parnas, J. Shore, and D. Weiss. "Abstract Types Defined as Classes of Variables." Proceedings of the ACM Conference on Data: Abstractions, Definitions, and Structure, Salt Lake City, Utah, 1976, pp. 149-154.

[Pascoe 86] G. Pascoe. "Elements of Object-Oriented Programming: Smalltalk Environment Concepts." BYTE, Vol. 11, No. 8, August 1986, pp. 139-144.

[Pedersen 89] C. Pedersen. "Extending Ordinary Inheritance Schemes to Include Generalization." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '89), New Orleans, Louisiana, October 1989, pp. 407-417.

[Phaal 86] P. Phaal. "A Smalltalk Environment for Computer-Aided Control System Design." Proceedings of the Third Symposium IEEE Control System Society on Computer-Aided Control System Design, Arlington, Virginia, September 1986, pp. 19-24.

[Pinson 88] L. Pinson and R. Wiener. An Introduction to Object-Oriented Programming and Smalltalk. Addison-Wesley, Reading, Massachusetts, 1988.

[Pohl 89] I. Pohl. C++ for C Programmers. The benjamin/Cumming Publishing Company, Reading, Massachusetts, 198.

[Pope 87] S. Pope, A Goldberg, and L. Deutsch. "Object-Oriented Approach to Software Lifecycle Using Smalltalk-80 System as a CASE Toolkit." Proceedings of the Fall Joint Computer Conference (FJCC '87), Dallas, Texas, October 1987, pp. 13-20.

[Pugh 87] J. Pugh, W. Lanlonde, and P. Thomas. "Introducing Object-Oriented Programming into the Computer Science Curriculum." ACM SIGCSE, Vol. 19, No. 1, February 1987, pp. 98-102.

[Pyle 86] I. Pyle. "Objects in Ada and Its Environment." Proceedings of the CERN School of Computing, Geneva, Switerland, 1986, pp. 44-64.

[Rosentien 86] L. Rosentien and S. Wallace. "Object-Oriented Programming for Mackintosh Applications." Proceedings of the Fall Joint Computer Conference (FJCC '86), Dallas, Texas, November 1986, pp. 31-35.

[Row 87] L. Row and C. Williams. "An Object-Oriented Database Design for Integrated Circuit Fabrication." Proceedings of the International Conference on Data and Knowledge Systems for Manufacturing and Engineering, Harlford, Connecticut, October 1987, pp.42-56.

[Sborn 86] S. Sborn. "Shared Object Hierarchy." Proceedings of the Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986, pp. 231.

[Schaffert 86] G. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. "An Introduction to Trellis/Owl." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), Portland, Oregon, September 1986, pp. 9-16.

[Schmucker 86] T. Schmucker. "Object-Oriented Languages for the Mackintosh." BYTE, Vol. 11, No. 8, August 1986, pp. 177-185.

[Seidewitz 87] E. Seidewitz. "Object-Oriented Programming in Smalltalk and Ada." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 202-213.

[Sethi 89] R. Sethi. Programming Languages: Concepts and Constructions. Addison-Wesley, Reading, Massachusetts, 1989.

[Snyder 86a] A. Snyder. "Encapsulation and Inheritance in Object-Oriented Languages." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), Portland, Oregon, September 1986, pp. 38-45.

[Snyder 86b] A. Snyder. "CommonObjects: An Overview." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 19-28.

[Snyder 87] A. Snyder. "Inheritance and the Development of Encapsulated Software Components." Proceedings of the 20th Annual Hawaii International Conference on System Sciences (HICSS '20), Vol. I, January 1987, pp. 227-237.

[Sobell 89] M. Sobell. A Practical Guide to the UNIX System. The Benjamin/Cummings Publishing Company, 1989.

[Steele 84] G. Steele. CommonLisp - The Language. Digital Equipment Corp., 1984.

[Steele 90] G. Steele. CommonLisp - The Language. Digital Press, 1990.

[Stefik 86] M. Stefik and D. Bobrow. "Object-Oriented Programming: Themes and Variations." The IA Magazine, March 1986, pp. 40-62.

[Stein 87] L. Stein. "Delegation is Inheritance." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '88), Orlando, Florida, October 1987, pp. 138-146.

[Stein 88] J. Stein. "Object-Oriented Programming and Databases." Dr. Dobb's Journal on Software Tools, Vol. 13, No. 3, March 1988, pp. 18-34.

[Strom 86] R. Strom. "A Comparison of the Object-Oriented and Process Paradigms." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 88-96.

[Stroustrup 83] B. Stroustrup. "Adding Classes to the C Language: An Exercise in Language Evolution." Software-Practice and Experience, Vol. 13, 1983, pp. 139-161.

[Stroustrup 84] B. Stroustrup. "Data Abstraction in C." Technical Journal, AT&T Bell Labs, Vol. 63, No. 8, 1984, pp. 1701-1732.

[Stroustrup 86a] B. Stroustrup. The C++ Programming Language. Addison-Wesley, Reading, Massachusetts, 1986.

[Stroustrup 86b] B. Stroustrup. "An Overview of C++." ACM SIGPLAN Notices, Vol. 21, No. 10, 1986, pp. 7-18.

[Stroustrup 86c] B. Stroustrup. "What Is Object-Oriented Programming?" Proceedings of the Fourteenth SIMULA User's Conference, Stockholm, Sweden, August 1986, pp. 69-84.

[Stroustrup 87a] B. Stroustrup. "The Evolution of C++: 1985 to 1987." Proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico, November 1987, pp. 1-21.

[Stroustrup 87b] B. Stroustrup. "Possible Directions for C++." Proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico, November 1987, pp. 399-416.

[Stroustrup 88a] B. Stroustrup. "Design Issues in C++." ACM SIGPLAN Notices, Vol. 23, No. 1, January 1988, pp. 57-64.

[Stroustrup 88b] B. Stroustrup. "Type-Safe Linkage for C++." The Journal of USENIX Association (Computing Systems), Vol 1, No. 4, University of California Press, 1988, pp. 371-403.

[Stroustrup 89a] B. Stroustrup. "Multiple Inheritance for C++." UNIX System V, AT&T C++ Language System (Release 2.0), Selected Readings, Select Code 307-144. 1989.

[Stroustrup 89b] B. Stroustrup. "Parameterized Types for C++." The Journal of USENIX Association (Computing Systems), Vol 2, No. 1, University of California Press, 1989, pp. 55-85.

[Stroustrup 91] B. Stroustrup. The C++ Programming Language. Second Edition, Addison-Wesley, Reading, Massachusetts, 1991.

[Tarumi 85] H. Tarumi, K. Agusa, and Y. Ohno. "Acquaintance/Instance Variable Model for Object-Oriented Programming." Proceedings of the IEEE Computer Society's Nineteenth International Computer Software and Application Conference (COMSAC '85), Chicago, Illinois, October 1985, pp. 69-73.

[Tello 87a] E. Tello. "Object-Oriented SCOOPS." Dr. Dobb's Journal of Software Tools, Vol. 12, No. 6. June 1987, pp. 112-115.

[Tello 87a] E. R. Tello. "Object-Oriented Programming (Artificial Intelligence)." Dr. Dobb's Journal on Software Tools, Vol. 12, No. 11, November 1987, pp. 130-136.

[Tesler 86] L. Tesler. "Programming Experience." BYTE, Vol. 11, No. 8, August 1986, pp. 73-80.

[Tosten 88] R. Tosten. "Data Security in an Object-Oriented Environment Such as Smalltalk." Proceedings of the International Conference on Computer Languages, Miami, Florida, October 1988, pp. 234-241.

[Touretzky 86] D. Touretzky. The Mathematics of Inheritance Systems. Morgan Kaufmann Publishing Company, 1986.

[Tracz 88a] W. Tracz. "Software Reuse Myths." ACM SIGSOFT: Software Engineering Notes, Vol. 13, No. 1, January 1988, pp. 17-21.

[Tracz 88b] W. Tracz. "Software Reuse Maxims." ACM SIGSOFT: Software Engineering Notes, Vol. 13, No. 4, October 1988, pp. 28-31.

[Tsichritzis 87] E. Tsichritzis et al. "KNO's: Knowledge Acquisition, Dissemination, and Manipulation Objects." ACM Transactions on Office Information Systems, Vol. 5, No. 1, 1987, pp. 96-112.

[Ungar 87] D. Ungar and R. Smith. "Self: The Power of Simplicity." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 227-242.

[Van Wijngaarden 75] A. Van Wijngaarden, B.J. Mailloux, J.E. Peck, H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G. Meertens, and R.G. Fisker. Revised report on the algorithmic language Algol 68, Acta Informatic, Vol. 5, pts. 1-3, 1975 pp. 1-236 (Also reproduced in SIGPLAN Notices, Vol. 12, No. 5, May 1977, pp; 1-79.

[Verity 87] J. Verity. "The Object-Oriented Revolution [Object-Oriented Programming Trends]." Datamation, Vol. 33, No. 9, May 1987, pp. 73-78.

[Vines 89] D. Vines and T. King. "Experience in Building a Prototype Object-Oriented Framework in Ada." Proceedings of the Eleventh Annual International Computer Software and Application Conference (COMSAC '89), Tokyo, Japan, October 1989, pp. 642-648.

[Wasserman 90] A. Wasserman, P. Pircher, and R. Muller. "The Object-Oriented Structured Design Notation for Software Design Representation." IEEE Computer, Vol. 23, No. 3, March 1990, pp. 50-63.

[Watanabe 88] T. Watanabe and A. Yanezawa. "Reflection in an Object-Oriented Concurrent Languages." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '88), San Diego, California, September 1988, pp. 306-315.

[Wayne 89] W. Wayne. "A Practical Comparison for Two Object-Oriented Languages." IEEE Software, September 1989, pp. 61-67.

[Wechsler 88] H. Wechsler and D. Rine. "Object-Oriented Programming and Its Relevance to Designing Intelligent Software." Proceedings of the International Conference on Computer Languages, Miami, Florida, October 1988, pp. 242-248.

[Wegmann 86] A. Wegmann. "Object-Oriented Programming Using Modula-2." Journal of Pascal, Ada, and Modula-2, Vol. 1, No. 1, January 1986, pp. 42-43.

[Wegner 83] P. Wegner. "Varieties of Reusability." Proceedings of the Workshop on Reusability Programming, Newport, Rhode Island, September 1983. pp. 57-63.

[Wegner 86a] P. Wegner. "Classification in Object-Oriented Systems." ACM SIGPLAN Notices, Vol. 21, No. 10, October 86, pp. 173-183.

[Wegner 86b] P. Wegner. "Introduction to the Special Issue of the SIGPLAN Notices on the Object-Oriented Programming Workshop." ACM SIGPLAN Notices, Vol. 21, No. 10, October 1986, pp. 1-6.

[Wegner 87] P. Wegner. "Dimensions of Object-Based Language Design." ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 168-182.

[Wegner 90] P. Wegner. "Concepts and Paradigms of Object-Oriented Programming." ACM SIGPLAN OOPS Messenger, Vol. 1, No. 1, August 1990, pp. 7-87.

[Wiegand 87] J. Wiegand. "Object-Oriented Code Optimizer and Generator." Technical Report, University of Illinois, Urban-Champaign, July 1987.

[Wirfs-Brock 87] A. Wirfs-Brock and B. Wilkerson. "An Overview of Modular Smalltalk." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp.123-134.

[Wolf 87] W. Wolf. "Better Controllers Through Object-Oriented Hardware Design." Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors. Rey Brook, New York, October 1987, pp. 22-26.

[Wolf 89] W. Wolf. "A Practical Comparison of Two Object-Oriented Languages." IEEE Software, September 1989, pp. 61-68.

[Zdonik 86] S. Zdonik. "Why Properties are Object-Oriented Refinement of 'is-a'." Proceedings of the Fall Joint Computer Conference (FJCC '89), Dallas, Texas, November 1986, pp. 41-47.

[Yan 86] J. Yan and E. Schlumberger. "Identifying Depositional Environment Structure: An Expert System Approach Using Object-Oriented Programming and Model-Driven Verification." Proceedings of the Second International Expert Systems Conference, London, England, September 1986, pp. 441-449.

[Yokote 86] Y. Yokote and M. Tokoro. "Design and Implementation of Smalltalk." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), Portland, Oregon, September 1986, pp. 331-340.

[Yokote 87a] Y. Yokote et al. "Experience and Evaluation of Concurrent Smalltalk." Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87), Orlando, Florida, October 1987, pp. 406-415.

[Yokote 87b] Y. Yokote and M. Tokoro. "Concurrent Programming in Concurrent Smalltalk." Research Directions in Object-Oriented Programming, Edited by B. Shriver and P. Wegner, [MIT Press Series in Computer Systems], The MIT Press, Cambridge, Massachusetts, 1987, pp. 129-158.

[Zdonik 86a] S. Zdonik. "Why Properties are Object-Oriented Refinement of 'is-a'." Proceedings of the Fall Joint Computer Conference, Dallas, Texas, November 1986, pp. 41-47.

[Zdonik 86b] S. Zdonik. "Version Management in an Object-Oriented Database." Proceedings of the International workshop on Databases, Trondheim, Norway, June 1986, pp. 405-422.

[Zeigler 87] B. Zeigler. "Hierarchical, Modular Discrete-Event Modeling in an Object-Oriented Environment." Simulation, Vol. 49, No. 5, November 1987, pp. 219-230.

[Zhong 88] Y. Zhong, S. Lshizuka, and R. Enavi. "Integrating Abstract Data Types with Object-Oriented Programming by Specification-Based Approaches." Proceedings of the International Conference on Computer Languages (OOPSLA 88), Miami, Florida, October 1988, pp. 202-209.

[Zygmunt 87] A. Zygmunt. "Object-Oriented Programming and CACSD." Proceedings of the IEEE Annual Conference: Engineering Focuses on Excellence, Reno, Nevada, June 1987, pp. 648-652.

VITA 

Hisham M. Al-Haddad

Candidate for the Degree of

Doctor of Philosophy

Thesis: NEW INHERITANCE MODELS THAT FACILITATE SOURCE CODE REUSE IN OBJECT-ORIENTED PROGRAMMING

Major Field: Computer Science

Biographical:

Personal Data: Born in Irbid, Jordan, January 1964, the son of Turkieh and Mustafa Al-Haddad.

Education: Graduated from Irbid Senior High School, Irbid, Jordan, July, 1982; received the Bachelor of Science degree in Computer Science from Yarmouk University, Irbid, Jordan, June, 1986; received the Master of Science degree from Northrop University, Los Angeles, California, June, 1988; completed the requirements for the Doctor of Philosophy degree at Oklahoma State University, July, 1992.

Professional Experience: Teaching Assistant, Computer Science Department, Oklahoma State University, August, 1989, to May, 1992.