

A MULTI-LEVEL PROGRAMMING PARADIGM  
AND ITS SUPPORT ENVIRONMENT  
FOR SOFTWARE REUSABILITY

By

ZHIYU ZHANG

Bachelor of Science

Xi'an Institute of Technology

Xi'an, China

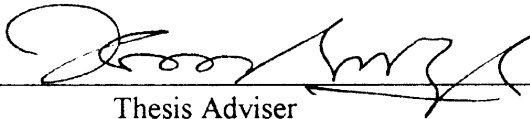
1983

Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in Partial Fulfillment of  
the Requirements of  
the Degree of  
MASTER OF SCIENCE  
July, 1993

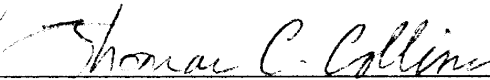
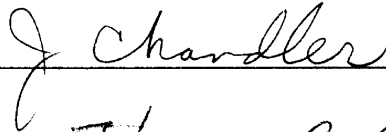
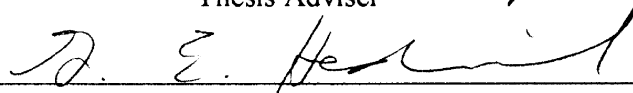
OKLAHOMA STATE UNIVERSITY

A MULTI-LEVEL PROGRAMMING PARADIGM  
AND ITS SUPPORT ENVIRONMENT  
FOR SOFTWARE REUSABILITY

Thesis Approved:



Thesis Adviser



Dean of Graduate College

## PREFACE

Software reusability has long been realized as a potential means to overcome the software crisis — the problem of building large, reliable software systems in a controlled, cost-effective way. Though a variety of efforts have been devoted to software reusability, software reuse still is not a common practice. The mismatch of software artifacts and human cognitive capability as well as the lack of standard data interchange formats, architectural support and reusable designs are the main obstacles to the software reusability. This thesis proposes a programming paradigm and its support environment for software reusability. It aims at simplifying the cognitive process and providing architectural support for software reusability. The proposed paradigm is a combination of both functional programming (FP) and object-oriented programming (OOP), which consists of three levels; i.e., the functional level, the class level and the object level. Prominently, each of the three levels exists in its own autonomy and has its own paradigm. Thus, programming complexity is reduced by the fact that programmers can concentrate on just one concept at a time. The reusability is realized in a fashion that programming is done by "wiring" existing components at different levels by means of the tools provided. Since the multi-level paradigm is a combination of both functional and object-oriented programming paradigms, the paradigm provides its users with the advantage of both functional and object-oriented programming.

## ACKNOWLEDGMENTS

I wish to express my sincere gratitude to Dr. K. M. George, my principal adviser, for his constant inspiration and encouragement throughout my graduate study. None of this would have been possible without his consistent advice and generous aid.

I also wish to express my sincere gratitude to Dr. G. E. Hedrick for his guidance and support during my graduate study. I would also like to thank him for inviting me as a visiting scholar to the Computer Science Department at Oklahoma State University in 1987, when I started research on FP languages. Thanks go to Dr. J. P. Chandler for serving on my committee. Both his kind advice on my academic affairs and thoughtful arrangement for my work have been invaluable for my graduate study.

I am grateful for the help I have gotten from many individuals. In particular, I wish to extend my gratitude to Mrs. Sylvia Duncan and Mr. Aaron Duncan for their kind support and friendly association throughout the time I have stayed in Stillwater.

My elder sister, Quichuang Zhang, deserves my deep gratitude for being a constant source of support in my life. Being in a different country, I am truly indebted to my parents, Delu Zhang and Shuiling Xu, for their worries and concerns during the time I have been far away from them. This thesis is dedicated to them without reservation.

Last, but not least, my appreciation goes to my lovely wife Ronie for her behind-the-scene support and tolerance. It is her devotion to the family matters that allowed me plenty of time to work on this thesis.

## TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION .....	1
1.1 Prologue .....	1
1.2 Promising Techniques.....	2
1.2.1. Functional Programming.....	2
1.2.2.. Object-Oriented Programming.....	3
1.3 The Theme of the Thesis.....	4
1.4 Thesis Organization .....	4
II. FUNCTIONAL PROGRAMMING (FP).....	5
2.1 Introduction.....	5
2.2 Object.....	5
2.3 Application.....	6
2.4 Primitive Functions.....	6
2.5 Functional forms .....	8
2.6 Function definition.....	8
2.7 Programming in FP .....	9
2.8 The Algebra of FP Programs.....	11
2.9 Promises and Difficulties.....	12
III. OBJECT-ORIENTED PROGRAMMING (OOP).....	13
3.1 Introduction.....	13
3.2 Object View of the World.....	14
3.3 Classes.....	15
3.4 Inheritance.....	15
3.5 Object Modeling .....	16
3.6 Compared with Conventional Programming .....	18
IV. A MULTI-LEVEL PROGARMMING PARADIGM .....	22
4.1 Introduction.....	22
4.2 Functional Level .....	23
4.3 Class level .....	24
4.3.1 Class Specification.....	24

Chapter	Page
4.3.2 Binding Mechanism .....	26
4.3.3 More Examples .....	28
4.4 Object Level.....	30
4.5 Related Work .....	32
4.6 Conclusion .....	34
V. A SUPPORTING ENVIRONMENT.....	35
5.1 Rationale .....	35
5.2 A Reuse Process Model .....	36
5.3 Support Required in Reuse Process .....	38
5.4 User Interface .....	40
5.4.1 Object Manager.....	41
5.4.2 Class Manager.....	42
5.4.3 Function Manager .....	43
5.5 Summary .....	44
VI. LANGUAGE DEFINITIONS .....	46
6.1 Introduction.....	46
6.2 Lexical Elements.....	47
6.2.1 Character Set.....	47
6.2.2 Lexical Elements.....	48
6.2.3 Reserved Words .....	48
6.3 Class Declaration .....	49
6.3.1 Domain Declaration.....	49
6.3.2 Signature Declaration.....	50
6.3.3 Body Declaration .....	50
6.4 Object Module Declaration.....	51
6.4.1 Object Instantiation.....	51
6.4.2 Object Access.....	52
6.4.3 Statement.....	52
6.4.4 Procedure Declaration.....	54
6.4.5 Object Module Declaration.....	54
6.5 Summary .....	55
VII. SUMMARY, CONCLUSIONS AND FUTURE WORK.....	56
7.1 Summary .....	56
7.2 Contributions and Conclusion.....	56
7.3 Future Work .....	57
REFERENCES.....	59

## LIST OF FIGURES

Figures	Page
1. A Software Reuse Process Model.....	38
2. Interface to the Object Level.....	40
3. Interface to the Class Level.....	42
4. Interface to the Function Level .....	44

## CHAPTER I

### INTRODUCTION

#### 1.1 Prologue

Software reusability long has been realized as a potential means to overcome the software crisis, the problem of building large, reliable software systems in a controlled, cost-effective way. Early in 1968, McIroy [McI68] proposed a reuse library approach to promoting software reusability at the NATO Software Engineering Conference [NR68] which is considered the birth of software engineering. Since then, a variety of efforts have been devoted to software reusability [Kru92, Jon84]. Unfortunately, software reuse has failed to realize its promise. Jones in [Jon84] reported that less than 15 percent of the code written in 1983 is unique, novel, and specific to individual applications and the remaining 85 percent are common and generic to all the applications. If the 85 percent had been developed once and used for all applications, the software crisis would have been solved. Jones predicated in [Jon84], by the year 2000, the percentage of new applications may be only 10-15 percent. Thus, exploiting software reusability has a number of pay-offs. Obviously, it reduces the cost, increases the reliability, and facilitates the maintenance of a software system [Che84, BP90, GD89]. In light of this, the computer science community has renewed its interests in finding approaches to the software reusability [Kru92]. Simply stated, software reuse is to construct a software system using existing software artifacts, rather than to code line by line from scratch. This seemingly simple idea has proven very difficult to realize in software development. Krueger in [Kru92] attributes the difficulty to a mismatch of software artifacts with



human cognitive capability. For a reuse technique to be effective, it must simplify the cognitive process. Jones in [Jon84] attributes the obstacles for creating reusable modules to the lack of standard data interchange formats, architectural support and reusable designs. Although least researched, the architecture for software reusability is fundamental since any effective reuse requires a software architectural starting point, rather than simply collecting together random modules and trying to link them together [Ken83].

## 1.2 Promising Techniques

In searching for a solution to the problems encountered in the software engineering, functional programming and object-oriented programming have been two main streams. Functional programming treats programming in mathematical domains, while object-oriented programming offers a new way to program in the large. They address the problems in the same area, yet from different perspectives. Both can date back to the early days of computer science. However, the culmination of the research efforts is a very recent phenomenon. The interests in functional programming have resulted in variety of functional languages and architectures [Veg84]. Current interests can be found in [HEA91] by a joint effort for a standard functional programming language. As structured programming won its popularity in 1960s, the object-oriented programming makes its way to the stage today.

### 1.2.1. Functional Programming

Functional programming can be characterized as writing an expression whose value solely depends on the values of its sub expressions, if any. For example, the value of the expression  $x + y$  is the sum of the values of  $x$  and  $y$ . The style of programming implies that there are no side effect as in conventional programming. In the absence of

side effects, an expression has the same value whenever it is evaluated. Thus, an expression can be replaced with its functional equivalents any time when needed without causing any unpredictable effects. This style of programming is superior to conventional programming for the following reasons as far as software engineering is concerned: most prominently, functional programming has a mathematical basis, thus, programs are mathematically tractable; furthermore, functional programs are applicative rather than imperative, thus it takes less efforts to write the programs since the specification of a problem is almost the solution to the problem; moreover, functional programs are shorter than their conventional counterparts, therefore, are easier to understand and to maintain.

### 1.2.2. Object-Oriented Programming

Object-oriented programming is not a coding technique, rather it is a new way to approach the problems to be solved on computers. In object-oriented programming, a problem is envisioned as a collection of cooperative objects. Every object is independent from other objects. Yet, they can communicate with each other by sending messages. This simple view has a number of implications: Firstly, it provides a natural way to decompose a large system into manageable modules. The way program is organized is a metaphor of the problem being solved. Thus, the solution is more intuitive than in conventional programming; Secondly, programming complexity is reduced since each object is an encapsulated unit. An object is seen by the outside via its interface. Its inside details are hidden from outside. Thus, any change in the implementation of an object does not affect its use by the other objects. The ripple effect in the conventional programming is limited to the object; Thirdly, object-oriented programming introduces a systematic methodology into software reusability. Software systems are organized into a class hierarchy. A class can inherit from other classes. Thus, defining a class does not start from scratch, rather, starts from existing classes. "The effect is to put reusability squarely in the mainstream of the software development process" [Cox86].

### 1.3 Theme of the Thesis

This thesis is concerned with architectural support for promoting software reusability, and eventually promoting programming productivity. The author approaches this problem by proposing a multi-level programming paradigm, in which functional programming and object-oriented programming are harmoniously unified in a three level hierarchy. Reusability is realized by "wiring" the existing components into new components through different mechanisms provided at different levels in the hierarchy. Although the research follows the general attempt to unify different paradigms to a single paradigm, the approach to the unification is novel and the final paradigm exists in hierarchy, rather than in a opaque wide spectrum language. The research emphasizes on the ease of programming in the final paradigm since the eventual purpose is to promote programming productivity. Thus, different paradigms are put at different levels and each level is maintained conceptually autonomous. When one programs in this paradigm, he/she concentrates on just one concept at a time. The author believes that such a paradigm is amenable to human cognitive capability. To further ease programming in the paradigm, a programming environment to support the paradigm is also proposed.

### 1.4 Thesis Organization

This thesis is organized in the following way: Chapter II and Chapter III provide reviews of functional programming (Backus' FP) and object-oriented programming (OOP) respectively. Both strong and weak points of the two programming paradigms are discussed. Chapter IV proposes a new paradigm, a multi-level programming paradigm. Chapter V outlines an environment to support the proposed programming paradigm. Chapter VI formally defines the language to be used in the programming environment. Finally, Chapter VII summarizes and concludes the thesis.

## CHAPTER II

### FUNCTIONAL PROGRAMMING (FP)

#### 2.1 Introduction

In his Turing Award lecture in [Bac78], Backus proposed an alternative to the conventional style of programming. His approach emphasizes programming at the functional level. One of the advantages of this functional approach is that it allows one to reason about his program using the algebra of programs. An FP system consists of the following five parts:

- 1) A set of objects.
- 2) A set of primitive functions mapping objects to objects.
- 3) A set of functional forms used to build new functions from existing functions.
- 4) A set of function definitions associating names with functions.
- 5) An application operator, denoted by  $:$ , applying functions to objects.

This chapter describes an FP system briefly. Following a definition of an FP, the style of programming in FP is introduced. At the end of the chapter, both strong and weak points of FP are discussed.

#### 2.2 Object

An object is either an atom or a sequence of objects or undefined (represented by "bottom"  $\perp$ ). Atoms in an FP system include numbers, strings and Boolean constants. A sequence is a list of the form  $\langle x_1, \dots, x_n \rangle$ , whose elements  $x_i$  are objects. Examples of objects are followings: 23, 32.7, String,  $\langle 1, 2, 3 \rangle$ ,  $\langle \langle 1, 2, 3 \rangle, \text{String} \rangle$ , T, F,  $\perp$ . Objects are

the only kind of data type in an FP system. All objects are bottom preserving, i.e.,  $\langle x_1, \dots, x_n \rangle = \perp$  if for some  $i$ ,  $x_i = \perp$ .

### 2.3 Application

If  $f$  is a function in an FP system and  $x$  is an object in the system, then an application is  $f:x$ , which means the object produced by applying the function  $f$  to the object  $x$ .  $f$  is the operator for the application and  $x$  is the operand. An example of application is  $+:<1,2>$ , which produces 3.

### 2.4 Primitive Functions

Primitive functions are the basic operators provided by an FP system. They map objects to objects by means of applications. All primitive (as well as user-defined) functions are  $\perp$  preserving, i.e.,  $f: \perp = \perp$ . An expression of the form  $p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n; e_{n+1}$  in an FP system is the conditional structure form, which means "if  $p_1$  then  $e_1$  else if  $p_2$  then  $e_2$  ...else  $e_{n+1}$ ". A set of FP primitive functions typically includes functions for arithmetic calculation, predicates and functions for sequence composition and decomposition. The following are some examples of primitive functions:

Arithmetic functions

$+:x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y+z; \perp$

Similarly  $-$ ,  $*$ ,  $/$  can be defined. Thus,  $+:<2, 3> = 5$ ,  $+:<2> = \perp$ ,  $+:<2, A> = \perp$ .

Logical functions

$\text{and}:x \equiv x = \langle T, T \rangle \rightarrow T; x = \langle T, F \rangle \rightarrow F; x = \langle F, T \rangle \rightarrow F; \perp$

Logical function *or* can similarly be defined. The function *not* is defined as the following:

$\text{not}:x \equiv x = T \rightarrow F; x = F \rightarrow T; \perp$

Relational functions

$>:x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \ \& \ x > y \rightarrow T; x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \ \& \ y \leq z \rightarrow F; \perp$

Similarly the functions  $<, >=, <=$  for numerical comparison can be defined.

Equational function

$eq:x \equiv x = \langle y, z \rangle \ \& \ y = z \rightarrow T; x = \langle y, z \rangle \ \& \ y \neq z \rightarrow F; \perp$

This function returns true if its operand is a pair of identical objects, false if its operand is a pair of non-identical objects and bottom  $\perp$  otherwise. e.g.,  $eq:\langle \langle 3, B \rangle, \langle 3, B \rangle \rangle = T$ ,  $eq:\langle 3, B \rangle = F$ ,  $eq:\langle 1, 2, 3 \rangle = \perp$ .

Null function

$null:x \equiv x = \langle \rangle \rightarrow T; x \neq \langle \rangle \rightarrow F; \perp$

The null function tests emptiness of its operand of a sequence structure. Thus, by the definition,  $null:\langle \rangle = T$ ,  $null:\langle 1, 2, A \rangle = F$ ,  $null:2 = F$ ,  $null:\perp = \perp$ .

Selectors

$1:x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow x_1; \perp$

For any positive number  $s$

$s:x \equiv x = \langle x_1, \dots, x_n \rangle \ \& \ n \leq s \rightarrow x_s; \perp$

Thus,  $3:\langle A, B, C, D \rangle$  produces  $C$  and  $2:\langle 1, \langle 2, 3 \rangle \rangle$  produces  $\langle 2, 3 \rangle$  and  $3:\langle 2 \rangle$  produces  $\perp$ .

Identity function

$id:x \equiv x$

Thus,  $id:x = x$  for all  $x$  in an FP system.

List functions

$length:x \equiv x = \langle x_1, \dots, x_n \rangle \rightarrow n; x = \langle \rangle \rightarrow 0; \perp$

$length$  returns the number of the elements in its sequence structured operand. Thus,  $length:\langle 1, 2, 3, \langle 1, 2 \rangle \rangle = 4$ ,  $length:\langle \rangle = 0$ , and  $length:3 = \perp$ .

$tlr:x \equiv x = \langle x_1 \rangle \rightarrow \langle \rangle; x = \langle x_1, \dots, x_{n-1} \rangle \ \& \ n \geq 2 \rightarrow \langle x_2, \dots, x_n \rangle; \perp$

$tlr$  returns its operand of a sequence structure with the first element eliminated, e.g.,

$tlr:\langle 1, 2, B \rangle = \langle 2, B \rangle$ .

$apndl:x \equiv x = \langle y, \langle \rangle \rangle \rightarrow \langle y \rangle; x = \langle y, \langle z_1, \dots, z_n \rangle \rangle \rightarrow \langle y, z_1, \dots, z_n \rangle; \perp$

Append to the left, when applied to a pair of objects, produces a sequence whose first element is the first element of the paired operand and whose tail is the second element of the paired operand. For example,  $\text{apndl}:\langle 1, \langle 2, 3 \rangle \rangle = \langle 1, 2, 3 \rangle$  and  $\text{apndl}:\langle \langle 1, 2 \rangle, \diamond \rangle = \langle \langle 1, 2 \rangle \rangle$ .

### Iota Function

$\text{iota}:x \equiv x=0 \rightarrow \diamond; x=\text{integer number} \ \& \ x \geq 1 \rightarrow \langle 1, \dots, x \rangle; \perp$

When applied to a positive integer number, *iota* produces a sequence of consecutive integers starting from 1. It produces the empty sequence when applied to 0. It produces  $\perp$  when applied to negative integer or non-integer operand. For example,  $\text{iota}:3 = \langle 1, 2, 3 \rangle$ ,  $\text{iota}:1 = \langle 1 \rangle$  and  $\text{iota}:0 = \diamond$ .

## 2.5 Functional forms

Functional forms are the means by which a programmer can build new functions out of existing functions. The following are some examples of functional forms.

Composition  $(f \circ g):x \equiv f:(g:x)$

Construction  $[f_1, \dots, f_n]:x \equiv \langle f_1:x, \dots, f_n:x \rangle$

Condition  $(p \rightarrow f:g):x \equiv (p:x)=T \rightarrow f:x; (p:x)=F \rightarrow g:x; \perp$

Constant  $\bar{x}:y \equiv y=\perp \rightarrow \perp; x$

Insert Left  $/f:x \equiv x=\langle x_1 \rangle \rightarrow x_1; x=\langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow f:\langle x_1, /f:\langle x_1, \dots, x_n \rangle \rangle; \perp$

Apply to all  $\alpha f:x \equiv x=\diamond \rightarrow \diamond; x=\langle x_1, \dots, x_n \rangle \rightarrow \langle f:x_1, \dots, f:x_n \rangle; \perp$

## 2.6 Function definition

Function definition is the mechanism provided in an FP system to associate a name with a function being defined. It has a form as the following:

$\text{Def } l \equiv r$

where *l* is a name and *r* is the function body being defined.

The following are some examples of function definitions.

Def push  $\equiv$  eq  $\circ$  [length,  $\bar{2}$ ]  $\rightarrow$  apndl  $\circ$  [1, 2];  $\bar{\perp}$

Def pop  $\equiv$  not  $\circ$  null  $\rightarrow$  tlr; null  $\rightarrow \overline{NULL}$ ;  $\bar{\perp}$

Def top  $\equiv$  not  $\circ$  null  $\rightarrow$  1; null  $\rightarrow \overline{NULL}$ ;  $\bar{\perp}$

Def poptop  $\equiv$  not  $\circ$  null  $\rightarrow$  [1, tlr]; null  $\rightarrow [\overline{NULL}, \overline{NULL}]$ ;  $\bar{\perp}$

Def Pythagoras  $\equiv$  sqrt  $\circ$  add  $\circ$  ( $\alpha*$ )  $\circ$  ( $\alpha$ [id, id])

## 2.7 Programming in FP

This section examines some problems and corresponding solutions in FP style by means of examples.

Example 1: Consider the problem of calculating the length of a sequence as defined in Section 2.3. Instead of thinking of the function length as a primitive, think of implementing it as a user defined function.

There are many choices for a solution to the problem. One simple solution would be as the following:

Def length  $\equiv$  null  $\rightarrow \bar{0}$ ;  $+\circ[\bar{1}, \text{length}\circ\text{tlr}]$

For example, if the function is applied to <A,B>, the program is evaluated through the following steps:

length:<A,B>  
 $\Rightarrow$  null  $\rightarrow \bar{0}$ ;  $+\circ[\bar{1}, \text{length}\circ\text{tlr}]$  : <A,B>  
 $\Rightarrow +\circ[\bar{1}, \text{length}\circ\text{tlr}]$  : <A,B>  
 $\Rightarrow +$  : <1, length:<B>>  
 $\Rightarrow +$  : <1, null  $\rightarrow \bar{0}$ ;  $+\circ[\bar{1}, \text{length}\circ\text{tlr}]$ :<B>>  
 $\Rightarrow +$  : <1,  $+\circ[\bar{1}, \text{length}\circ\text{tlr}]$ :<B>>  
 $\Rightarrow +$ : <1,  $+$  : <1, length:<>>>  
 $\Rightarrow +$  : <1,  $+$  : <1, 0>>>



$$\Rightarrow + : \langle 1, 1 \rangle$$

$$\Rightarrow 2$$

As argued in [Wil82b], this solution is an indirect description of the program, i.e., an equation that the program has to satisfy. A better solution, which gives direct description of the same problem, is as the following:

$$\text{Def length} \equiv \text{null} \rightarrow \bar{0}; /+ \circ \alpha \bar{1}$$

This function, when applied to  $\langle A, B \rangle$ , is evaluated through the following steps:

$$\text{length} : \langle A, B \rangle$$

$$\Rightarrow \text{null} \rightarrow \bar{0}; /+ \circ \alpha \bar{1} : \langle A, B \rangle$$

$$\Rightarrow /+ : \langle \bar{1} : A, \bar{1} : B \rangle$$

$$\Rightarrow /+ : \langle 1, 1 \rangle$$

$$\Rightarrow + : \langle 1, /+ : \langle 1 \rangle \rangle$$

$$\Rightarrow + : \langle 1, 1 \rangle$$

$$\Rightarrow 2$$

Comparing this solution with the first solution, one can find that the second solution is more efficient than the first one. It simply maps each element in a sequence to 1 and then adds them up. This is the style that the FP encourages [Wil84b].

Example 2: Factorial function is defined as follows:

$$\text{Def eq0} \equiv \text{eq} \circ [\text{id}, \bar{0}]$$

$$\text{Def sub1} \equiv - \circ [\text{id}, \bar{1}]$$

$$\text{Def fact} \equiv \text{eq0} \rightarrow \bar{1}; * \circ [\text{id}, \text{fact} \circ \text{sub1}]$$

One defines the factorial function from the existing functions, i.e., eq0 and sub1. Of course the factorial function could be defined as follows :

$$\text{Def fact} \equiv \text{eq} \circ [\text{id}, \bar{0}] \rightarrow ; * \circ [\text{id}, \text{fact} \circ - \circ [\text{id}, \bar{1}]]$$

This functionally is as same as the first definition. However, the first is better than the second one since the first one is more readable. A non-recursive version of factorial can be defined via iota function as the following:

Def fact  $\equiv / * \circ \text{iota}$

From this example one can see that one can write very compact yet readable programs in FP in a very efficient manner. For more examples of FP programs and the FP style of programming, the reader is referred to [Bac78, HK87, Wil82b].

## 2.8 The Algebra of FP Programs

One of the advantages that FP provides is the algebra associated with FP programs. One can reason about his/her program using the algebra without rendering any other means such as computational induction or fixed point arguments [Wil84b]. Instead, one just uses a collection of algebraic identities to show the equality of the functions computed by different programs.

The algebraic system of the FP programs can be divided into three categories, i.e., algebraic laws, derived theorems, and expansion theorems. The algebraic laws serve as the axioms in the algebra. They are the identities derived directly from the definitions of primitive functions and combining forms in a particular FP system. For example, the law

$$[f, g] \circ h \equiv [f \circ h, g \circ h]$$

asserts that, for any given programs  $f, g, h$ , the composition on the left is identical to the construction on the right. This law is derived from the definitions of the combining form composition and construction. A proof of this law can be found in [Bac78].

Based on the laws like the one in the above, a lot of theorems can be derived, which constitute the second category of the algebra. These derived theorems can be used to reason about an FP Program, for example, to prove a program correct.

The last category of the algebra is the expansion theorems. Using these theorems, one can get a non-recursive version of the programs that are defined as recursive

programs. There are several expansion theorems available in the literature. The reader is referred to [Bac78, GH88, Wil82a, Wil82b] for a more detailed account of the theory.

## 2.9 Promises and Difficulties

FP has a number of advantages over conventional programming language. It is superior to conventional programming for the following reasons as far as software engineering is concerned:

- 1) Functional programming has a mathematical basis, and since it has no side effect, the program correctness is mathematically provable. It is possible to use the associated algebra of FP programs to prove program correctness automatically.
- 2) FP is more expressive than the conventional programming languages. It is applicative rather than imperative, thus it takes less effort to write programs since a problem specification is almost the solution to the problem itself.
- 3) Programs written in FP are shorter than in their conventional counterparts, therefore, are easier to understand and maintain.

However, FP poses some difficulties in designing temporal effects. For example, to simulate a vending machine, FP faces the difficulties in remembering the temporally updateable state of the machine. Such a problem, intuitively, is best suitable for object-oriented programming (OOP) to be described in the next chapter. Another difficulty of FP is its abilities to handle I/O. Up till now, there is no systematic approach to the problem. Finally, FP programs are very inefficient on the current von Neumann machines. Large mutable data structures in the FP is the major source of the inefficiency. Interestingly, all the weak points of FP are well handled in the OOP as will be seen in the next chapter.

## CHAPTER III

### OBJECT-ORIENTED PROGRAMMING (OOP)

#### 3.1 Introduction

The term "object" emerged in various areas in computer science in 1970's to refer different, yet related, notions [YT87]. Typical notions are the following:

- 1) abstract data type, where data and associated operations are clustered together into an object. Each object has its own private memory and functions, resulting modularity and information hiding [HN87].
- 2) a way to decompose a large system into a number of smaller manageable components. Each component is an object with its own data and operations. Similar components are programmed to share code or data by being put into a class, which serves as a template of the objects. Following this method of managing software system development, one can control complexity [Abb83].
- 3) a means to represent knowledge in artificial intelligence [RK83].
- 4) a way to manage protected resources in the operating systems.

Although, all these notions emerged in different areas to mean different things to different people, they serve a single purpose; i.e., to manage the complexity of a large software system by using objects to represent the modularly organized components.

In [Boo91], object-oriented programming is defined as the following:

Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of

which represents an instance of some class and whose classes are all members of a hierarchy of classes united via inheritance relationship.

Booch explains that the definition means three things that constitute object-oriented programming (OOP): firstly, OOP uses objects, rather than conventional algorithms, to model the real world problems; Secondly, the objects in problem domain do not only exist individually, but also exist in classes, each object being an instance of some class; Thirdly the classes are related with each other via an inheritance relationship. A similar statement about object-oriented programming in [Weg87] is as the following:

"object-oriented = objects + object classes + class inheritance"

If a program appears missing one of the three things in the definition, the program can not be said to be object-oriented [Boo91]. By this definition, one can classify programming as object-based if there are no classes involved, or programming with abstract data types (ADT) if there is no inheritance involved among classes.

This chapter present a brief overview of object-oriented programming.

### 3.2 Object View of the World

An object is a self-contained entity, either conceptual or physical entity, with its own state and a set of operations to affect the state. Alternatively, one can say that an object is an entity that has state, behavior and identity [Boo91]. The state of an object remembers the history of the object. It reflects the effects of the operations that have ever performed on the object. The state of an object also encompasses all the quantitative properties of the object [Boo91]. For example, if one thinks of a vending machine as an object, the state of the object reflects the number of goods available and the amount of money collected so far. On the other hand, the state also reflects that the goods and the number of goods that have ever been sold. In the object-oriented view, the world is a collection of cooperative objects. Thus, beside the properties, an object also has

behaviors visible to other objects as it exists with other objects. An object behaves by performing operations through which the object interacts with other objects. Objects communicate by sending each other messages. Upon receiving a message, an object reacts by invoking some operations on its own internal state.

An important property of an object is that it is self-contained and its state is hidden from other objects and can only be affected or accessed via its own set of operations. This property helps reduce programming complexity and increase program reliability. More discussions about this property is presented later in Section 3.5.

### 3.3 Classes

A class is a set of objects that share some common properties and behaviors [Boo91]. Looked at from outside, a class specifies an interface of some objects. Internally, a class specifies all the properties and behaviors of its own objects. Classes do not behave as objects but serve as templates of objects from which objects can be created. In practice of real programming, a class may just specify the interface of its objects. It may also contain a body that specifies a partial or a complete implementation of the operations in the class. Classes sometimes are defined only for inheritance purposes. In that case the classes are never instantiated but used to provide inheritable functionalities for defining new classes. These classes are called abstract classes [Weg87].

### 3.4 Inheritance

Inheritance is a mechanism for composing the interface of one or more inherited classes with the interface of the inheriting class [Weg87]. This suggests that when one invents a class, one does not have to start from scratch. Instead, if the invented class has some common property with the existing classes, one can simply let the newly defined class inherit the properties from the existing classes. The inheriting class has the state

representation of the inherited class as part of its own state representation as well as all the operations defined in the inherited class. The inheriting class is called a subclass of the inherited one. The inherited class is called a superclass of the inheriting class. Inheritance means four things [HN87]. Firstly, a subclass contains all the instance variables defined in its superclasses as its own instance variables. In addition, it can have its own instance variables; secondly, a subclass have all the visible operations defined in its superclasses as its own operations visible to the other classes. Besides, it can define new operations to complement the inherited ones; thirdly, a subclass can use the operations provided by its superclasses as defined by itself. Thus, the same copy of the code can be used for many different classes, eliminating code duplications; finally, concrete operations are derived by specializing generic operations defined in the superclasses.

Although important, the concept of inheritance is not central to object-oriented problem solving [Bud91]. It is mainly a means to save programming effort. The benefits of using it are significant.

### 3.5 Object Modeling

A real world system typically consists of a number of cooperative objects. A typical example of such a system is a factory, where there are a number of different workshops or departments, each being an independent unit yet cooperative with the others. Every department has its own offices, staff and facilities and functions by its own rules. Every department communicate with other departments by sending messages, for example, a memorandum, a design graph, or a sample product. A department may further consist of many offices, each of which may still consists of people and facilities, and so forth. Each item; i.e., facilities, people, offices, in such a system is a self-contained

identity having its properties and behaviors. Items can communicate with one another by predetermined rules without intruding anyone's privacy.

An object in object-oriented programming is a metaphor for an object in the real world system just like the one above. Thus, to model a real world object, no other means can be more direct than the metaphor corresponding to the object itself. Object-oriented programming provides a direct mapping from a problem space to its solution space. This direct mapping gives one a useful hint for system decomposition. Conventionally, when one is faced with a complex system, he/she tends to decompose the system into different functional units. To do that, there is a lot of mental work involved. With object modeling, one decomposes the system by identifying the objects existing in the system. Objects that share common properties are put to one class. A class that has the properties common to other existing classes inherits properties from them. To model a factory by object modeling, for example, one organizes his/her program into a set of cooperative objects, each of which represents a real object in the factory. Since the organization or execution of each department in the factory is similar to the other, one can invent a class for all the departments, through which all the departments can share their commonalities. Similarly, office, facilities and people can also be organized as classes. Classes can share their commonalities through class inheritance. For example, one can define a class for department heads. Since department heads are also people, instead of specifying people attributes in a "heads" class, one inherits the attributes from a "people" class. Following object modeling, one not only gets a complex system naturally decomposed into modularly organized programs, but also saves a lot of programming effort since classes and their inheritance avoid code duplications. At the same time, the program tends to be more amenable to future change, reuse and maintenance since objects are self-contained entities.

Hierarchy has been realized as an essential property of problem complexity. To deal with the complexity, the modeling mechanism needs the power to organize things in



a hierarchy. A study of the complexity in [Boo91] shows that a complex system often consists of interrelated subsystems which contain their own subsystems, and so on, until some low level of primitive components is reached. This is a major facilitating factor for one to handle such complex systems. Past experience with complex system also shows that building a system from the existing working yet simple ones is more rapid than from scratch. All these properties of a complex system are not well exploited in the conventional modeling. OOP, however, provides a systematic framework to model a complex system through abstraction, encapsulation and inheritance.

### 3.6 Compared with Conventional Programming

The object paradigm of programming is very different from the conventional paradigm of programming. In conventional programming, people are forced to think in terms of variables and assignments. Computation takes place by wondering through memory cells, fetching values, transforming them in some manner and writing back to memory. Such a model does little to help people how to solve a problem using computers, and of course it is not the way people usually solve a problem [Bud91]. In contrast, in object-oriented model, computation takes place by sending messages among a set of objects. People never think in terms of variables and assignments, which are not directly relevant to the solution being sought. Instead, people solve a problem by making a metaphor of the problem itself on computers. Programming is done by creating a community of well-behaved objects, each being responsible for the community by sending messages to or reacting on receiving messages from others. One obvious advantage of such a programming paradigm is that people are strongly inspired by their intuition, common sense and every day experience in solving problems when they approach to the problems in terms of behaviors and responsibilities of objects [Bud91].

The conventional paradigm, however, does not exploit people's intuition in solving problems, which is abundant and valueless in every average programmer's background.

Software systems are inherently complex. The complexity does not come from the size of the problem, since size can be attacked by partitioning the problem into many pieces. It comes from the way one approaches problems. The conventional approach leads to solution in which every part is highly dependent on the rest of the system. This high interconnectedness is the unique feature of conventional programming technique [Bud91]. To handle complexity, abstraction - the ability to encapsulate unnecessary details from being seen is the major means. Objects provide a natural abstraction which is superior to any conventional means; i.e., subroutines, modules and abstract data types. Subroutines make it possible for tasks that are repeatedly executed to be kept in one place and executed many times in the same program. By using subroutines, one saves a lot of program space since the same piece of code is reused rather than duplicated. Subroutines also make it possible for information hiding. One can define a set of subroutines, while others can use them without being concerned with details of implementation. However, subroutines do not solve all the problem. When used to simulate a stack, for example, they are not so effective. They tend either to expose the variable representing the stack or to be dedicated to just one stack. Modules serve as an information-hiding mechanism, making public names known outside of modules and hiding rest of them in the modules. They face the same problems as subroutines since only one copy of the global data is allowed. Abstract data types are much capable than subroutines and modules in the sense that users can use them to have many instances of a user defined type. Abstract data types are user defined types, which can be used as types pre-defined in the system. A user defines a data structure and a set of operations with it and package them into one module. Later, he/she or other users can use the package to declare a variable. The information hiding is realized by the fact that users of the abstract data types know only the interface of the abstract types, not the details of implementation, and they can use the

types as system pre-defined ones. Thus, abstract data types extend the modules by allowing duplicate copies of data to exist in a program. They are almost like objects. However, object-oriented programming has added to abstract data types important innovations in code sharing and reusability [Bud91], which have been realized as a critical issue in software engineering. One of the innovations is the message passing. Superficially, a message passing looks like a procedure call. However, message passing is more powerful than a procedure call because the same message can be interpreted differently by different objects. With this feature, overloading of names and reusing software can be realized. Inheritance is another innovation beyond abstract data types. It allows different data types to share the same piece of code, resulting in reduced efforts of programming and compact size of software. The other innovation is that object-oriented programming allows a generic function associated with a data type to be specialized according to particular cases, resulting in polymorphism. In short, object mechanism provide a systematic way to deal with abstraction, which is much more effective than the means in conventional programming.

Software reusability has been a dream of software engineers for decades. It is still a dream today. However, the object-oriented programming provides a hope to make the dream come true. As Meyer in [Mey87] points: "The answers lie in object-oriented design." As pointed out above, conventional programming leads to highly interconnected components of software. It is very difficult to use a component in one project for another project, since the component is dependent on other components. In object-oriented programming, a component ( class ) is a self-contained unit and interconnections of components in a system is not so high as in conventional programming. Therefore, the components in object-oriented programming are more amenable to reusability.

In summary, OOP is advantageous over conventional programming. Noticeably, OOP has the following advantages:

- 1) OOP modeling provides a natural approach to starting a complex system. It better handles the inherent complexity of a system. In addition, the OOP modeling provides an evolutionary approach to problem solving, by which one builds a system from existing working components.
- 2) through inheritance from and reuse of the existing classes, OOP produces more compact systems than conventional programming. It also saves programming efforts, hence increases programming productivity.
- 3) a system designed in OOP is naturally prepared for future changes. The effect of a change is not likely to cause unpredictably ripple effect as in conventional programming since abstraction and encapsulation prevent it from happening. Thus, the program reliability is enhanced.

## CHAPTER IV

### A MULTI-LEVEL PROGRAMMING PARADIGM

#### 4.1 Introduction

This chapter proposes a hierarchical approach to blending two prevalent programming paradigms, functional programming (FP) and object-oriented programming (OOP), into one paradigm. The resulting paradigm exists hierarchically in three levels. The first level is the functional programming level at which a programmer works in terms of functions. At the moment Backus' FP is adopted as the language to effect this level. The functional level serves as an implementation level for the whole paradigm. Above the functional level is the class level where a programmer defines classes using functions defined at the level below. A class is defined by grouping functions into an abstract entity using a binding mechanism provided at that level. Defining a class can be thought of as a process of reusing functions existing at the functional level. Classes at this level create a foundation based on which an object-oriented programming paradigm exists. With classes defined at the class level, a programmer above this level can program in terms of classes without any knowledge of the existence of the functional level. This is what is envisioned at the object level, which is an object-oriented programming paradigm. A programmer at the object level uses the classes defined at the class level to define objects. To effect message passing among objects, some control mechanisms are provided at the object level. Although the resulting paradigm is a combination of the existing paradigms, the combination approach is different from general approach in that the different paradigms exist at different levels both hierarchically and autonomously.

With such a paradigm, programmers can concentrate on only one concept at a time, thus greatly reducing the conceptual complexity. Programming is further eased by the fact that each level is an abstraction of the level below it. The motivation for blending FP and OOP is to take advantage of features of both FP and OOP for reusability. The advantage of FP is preserved at the functional level and that of OOP is preserved at the class/object level. Reuse of existing programming entities is effected both at the functional level, where new functions are built by combining existing functions, and at the class/object level, where programs evolve by reuse of or inheritance from existing functions/classes. FP has a lot of advantages over conventional programming [Bac78]. However, FP is not history sensitive, which makes it hard to program in FP in many real world cases. Blending FP with OOP makes a compensation for what FP lacks. In complement, FP provides a theoretical foundation for building provably correct components intended for reusability in the OOP paradigm. The rest of this chapter explains each level in detail with the help of examples.

## 4.2 Functional Level

Existing at this level is a working language FP. The paradigm to be described in this chapter does not impose any *a priori* limitations on the FP system. Programming at this level is purely functional. One can take all the advantage that an FP system has whatsoever. If a programmer wishes, he/she can use FP to do all his/her work without venturing into anything else. However, if a programmer must build abstract modules in object-oriented approach yet still use the functions he/she has defined at this level, the programmer can do this by grouping functions into a class using the binding mechanism provided at the class level above FP. The only burden for a functional programmer is probably that he/she has to put the functions he has defined into a reuse library for later reuse.

### 4.3 Class level

What a programmer can "see" at the class level is nothing but the reuse library containing the functions defined at the function level, including the primitive functions in the original FP. Armed with a binding mechanism (defined in section 4.3.2), a programmer at this level defines classes for object-oriented programming.

A class definition consists of two parts; i.e., a signature and a body. The body part of a class could be omitted in some cases. The signature of a class specifies the external properties of the class, while the body specifies the implementation of the class. In order to avoid introducing formalism, examples are used to illustrate the definitions of classes. One important concept at this level is the domain associated with instance variables and class methods. A domain is a constraint imposed on an item being specified. It specifies a set of legal FP objects to that the item belongs. Since a domain is associated with FP objects, it is not available at the highest level where FP is transparent. Thus, it is only usable at this level. A domain associated with an instance, for example,  $a: \{A, B, C\}$  specifies that the object  $a$  can only have object  $A$ ,  $B$  and  $C$  as its legal values. For brevity, one may define a domain like the following:

domain  $U \equiv \{0, 1, \dots\}$ ;

then, use  $U$  to define an instance such as

$a: U$ ;

Furthermore, the sets *Boolean*, *String*, *Real*, *Unsigned*, *Integer* represents the domains with conventional interpretations.

#### 4.3.1 Class Specification

A class specification specifies the external properties of a class being defined. It has the following form:

```
signature ClassName { InstanceVar : Domain; ...
                        Form MethodName (Domain, ...) : Domain; ...
                    }
```

The *signature* is a key word indicating a class definition. After the class name and inside of the pair of braces are specifications of instance variables and methods. Associated with each instance variable is a domain as explained in the beginning of this section. A method specification specifies the form (infix or prefix), the name, the number of parameters and the domains of parameters of the method. The following explains this with the help of examples.

#### Example 1 Boolean class

```
signature Bool {
    b : Boolean;
    infix And (Boolean, Boolean) : Boolean;
    infix Or (Boolean, Boolean) : Boolean;
    Not (Boolean) : Boolean;
    infix = (Boolean, Boolean);
}
```

The *signature* of the class Bool specifies an instance variable b with the domain of Boolean, which represents FP objects T and F, and other four operations on the class, where *And*, *Or* and *=* are infix operators and *Not* is an ordinary function. The domain specification associated with the instance in the class specifies the legal range of FP objects the instance can hold.

#### Example 2 Stack class

```
signature Stack {
    l : < Integer* >;
    Push(Integer); PopOff( ); PopOnto(Integer);
    Top( ) : Integer; Empty( ) : Bool;
```



}

Note that the stack is represented by an FP object of a sequence of integers. This also means that *l* has a sequence of integers as its domain.

If a class is to extend an existing class, the class can be defined using inheritance specification. For example, to define a queue class, one can extend stack class as explained by example 3.

Example 3 Queue class

signature Queue inherit Stack {

BackPush(Integer);

BackPopOff( );

BackPopOnto( Integer);

BackTop( ) : Integer;

}

Since the class *Queue* is defined by extending the class *stack*, it has the new methods as well as the methods inherited from class *Stack*.

#### 4.3.2 Binding Mechanism

To implement the class methods using FP functions defined at the functional level, this section introduces a binding mechanism. The binding mechanism links the class level with the functional level in a very unusual way. A binding is a two-way mapping specification between the functional level and the class level. It has a general form as follows:

MethodName  $\equiv$  FPfunc: InParaPattern: OutParaPattern

where *FPfunc* is an FP function, *InParaPattern* and *OutParaPattern* are patterns of objects. *InParaPattern* specifies how to assemble instance variables as input parameter of the FP function, while *OutParaPattern* specifies how to disassemble the object returned from FP function to put back in the instance variables of a class.

The binding mechanism is used to implement the bodies of classes.

#### Example 4 Implementation of Class Bool

```
body Bool {
    And (a, b) ≡ and: <a, b>: and;
    Or (a, b) ≡ or: <a, b>: or;
    Not (a) ≡ not: a: not;
    = (a, b) ≡ id: b: a;
}
```

The *body* of the class Bool specifies the implementations of the methods specified in the *signature* of the class. This is where links with FP functions occur. The implementation of a method is specified by binding input parameters and instance variables to FP functions and binding returning result back to output parameters and instance variables. For example, The following definition

And (a, b) ≡ and: <a, b>: and;

specifies that input parameters *a* and *b* are bound to a pattern of <a, b> as an input parameter to the FP function *and*. Whenever the method is invoked, the FP function *and* is executed with a parameter of the form <a, b>. On exit, the method binds the result from the FP function to *And*. = is the assignment operator defined for the class *Bool*. It binds its right side operand to the identity function and the return value to its left side operand.

#### Example 5 Implementation of Class Stack

```
body Stack {
    Push (in) ≡ push: <in, l>: l;
    PopOff( ) ≡ pop: l: l ;
    PopOnto(out) ≡ pop: l: <out, l> ;
    Top( ) ≡ top: l: Top ;
    Empty( ) ≡ null: l: Empty ;
```

}

The stack example shows that a class can be defined instantly by using functions defined at the functional level.

#### 4.3.3 More Examples

Example 6. Unsigned integer

```
signature Unsigned {
  U: {0,1, ...};
  infix + (Unsigned, Unsigned) : Unsigned;
  infix - (Unsigned, Unsigned) : Unsigned;
  infix * (Unsigned, Unsigned) : Unsigned;
  infix / (Unsigned, Unsigned) : Unsigned;
  infix > (Unsigned, Unsigned) : Bool;
  infix < (Unsigned, Unsigned) : Bool;
  infix >= (Unsigned, Unsigned) : Bool;
  infix <= (Unsigned, Unsigned) : Bool;
  infix == (Unsigned, Unsigned) : Bool;
  infix = (Unsigned, Unsigned);
}
```

Note that *Unsigned* can be replaced by *U* as *Bool* has been replaced by *Boolean* in the example 1. They represent the same domain. However, they are conceptually different. *Unsigned*, which is a class being defined at the class level, is defined by *U*, which is from FP. The body of the class *Unsigned* can be defined as the following:

```
body Unsigned {
  + (a, b) ≡ +: <a, b>: + ;
  - (a, b) ≡ -: <a, b>: - ;
  * (a, b) ≡ *: <a, b>: * ;
```

```

/ (a, b) ≡ /: <a, b>: / ;
> (a, b) ≡ >: <a, b>: > ;
< (a, b) ≡ <: <a, b>: < ;
>= (a, b) ≡ >=: <a, b>: >= ;
<= (a, b) ≡ <=: <a, b>: <= ;
== (a, b) ≡ eq: <a, b>: == ;
= (a, b) ≡ id: b: a ;
}

```

Since the domain unsigned integer  $\{0, 1, \dots\}$  is not available at the higher level (object level), defining an unsigned integer at that level is not possible without introducing this class. Similarly, the integer type can be introduced in the class level.

In the FP, there does not exist any concept of type. However, through class level one can introduce the concept. By doing this one can impose type checking at the highest level of programming, deriving the benefits of types in the conventional programming languages. Since each FP function in the binding is constrained by domains associated with the methods, FP functions can be specialized for better performance.

Example 7 Simulated input class

```

signature Input {
    ReadTo(< Integer* >, Integer);
    Empty(< Integer* >) : Bool;
}

body Input {
    ReadTo(a, b) ≡ pop: a: <b, a> ;
    Empty(a) ≡ null: a: Empty ;
}

```

Functions defined at the functional level are exposed fully to the class level so that they can be grouped in the way they are needed. The same function, *poptop* for example, can be used in more than one class and in different ways.

#### Example 8 Integer sequence class

signature IntegerSequence {is: < Integer\* >; }

This defines a class of integer sequence. It has no body part. As will be seen in the following section, this class is used at the object level.

### 4.4 Object Level

This level offers an object-oriented programming paradigm. Although the level is based on the FP, it conceptually has nothing to do with FP from programmers' perspective. A programmer at this level thinks in terms of classes and conventional procedures or functions. FP at this level is transparent to users. The basic facilities this level offers are the mechanisms a user uses to instantiate objects from the classes defined at the class level and mechanisms to control message passing among the instantiated objects. There are no predefined data types or statements at this level. The data types (or classes) used at this level including even the conventional integer type are solely the types defined at the class level. Without the class level or with nothing defined at class level, users can not write any runnable programs. To write a program, a programmer starts with browsing class library established at the class level. If some classes satisfy his/her needs, he/she can then instantiate objects using the classes and accomplish his/her purpose by passing messages to the objects. For example, using *Unsigned* class defined at the class level, one can write the following program:

#### Example 9 Factorial function

Fact( p: Unsigned) : Unsigned

{

```

i, fact: Unsigned;
if p < 2 then return 1;
fact = 1;
for( i = 2; i <= p; i = (i + 1) )
    fact = ( fact * i );
return fact;
}

```

Note that the program looks similar to a conventional Pascal or C program. However, there is a fundamental difference between the program and a conventional program. All the types and operations such as  $<$ ,  $=$  are all user-defined at the class level and their implementations are based on the underlying FP system. *for*, *if* and *return* are control abstractions to control message passing.

The *for* control frame has the following form:

```
for ( initial; condition; update ) body;
```

*initial*, *condition*, *update* and *body* are all based on message passing. The *for* frame is executed in the following way: *initial* part is executed; Then *condition* part, which is a function, is executed. If the message passed back from executing the function contains the first member in the domain associated with the corresponding class, the *for* body is executed; After that, the update part is executed; Then the control goes back to executing *condition* part and repeats the above steps until the *condition* yields a value other than the first member in the domain of that class. Each of the three parts can be omitted as in the following fragment of program:

```

i, fact : Unsigned;

for ( i = 2; i < p; ) { fact = ( fact * i ); i = ( i + 1 ); }

```

where the *update* part is empty.

It is worth noticing that the *condition* part is not a conventional Boolean expression. The meaning of it is user-defined.

### Example 10 Push down automaton

Pda (Stream: IntegerSequence): Bool

```
{
    x: Integer;
    stck: Stack;
    inpt: Input;
    for ( ; Not ( Empty (stream) ) ; )
    {
        ReadTo ( stream, x );
        if x == Top ( ) then PopOff ( );
        else Push ( x );
    }
    if Empty ( ) then return T ;
    else return F;
}
```

Note again that all the operations and types used to declare variables come from class level. Here it is assumed that the compiler can resolve name conflicts. Otherwise, Empty( ), for example, should be written as stck.Empty( ) as to distinguish itself from inpt.Empty(stream).

## 4.5 Related Work

George originated the multilevel paradigm exploited in this chapter [Geo90], where the programming process is conceived analogously as a process wherein a computer architect designs a computer system at three levels: namely, the gate, register and processor levels. Thus, a programmer correspondingly can design a program in the multi-level paradigm at the functional, class and object levels. Although the original

paradigm also exists in three levels, the effecting concepts at each level are different from what is described in this chapter. This research follows the same direction. However, it enhances the original in the following major ways: different levels with different concepts of programming exist autonomously; a binding mechanism is introduced at the class level so that a programmer can group FP functions to form a class efficiently; a class is introduced with states so that encapsulation is realized. In [Geo90], functional programming language FP is adopted from [Bac78]. Its advantages over conventional languages are well argued in the same paper. Prominently, the FP has associated algebra by which a programmer can reason about his programs and build provably correct programs. The algebra of the FP programs is further exploited in [Bac81, Wil82a, Wil82b]. As the Boolean algebra provides theoretical basis for the gate level logic, the algebra of FP programs provides the theoretical basis for function level programming [Goe90]. Work towards extending FP as a general language can be found in [Bac86, DM84]. The work described in this chapter does not intend to extend but incorporate FP into a new multiple-paradigm, where FP exists autonomously. Using FP in the new paradigm is strongly motivated by its ability allowing one to reason about his/her programs and to build new programs from existing ones. Blending functional and object-oriented ideas into one single paradigm can also be found in [GM87]. The approach to the blending described in this chapter is different from [GM87] in that different ideas exist autonomously at different levels, rather than in a opaque wide spectrum language. Maintaining conceptual autonomy in a programming paradigm helps reduce the human cognitive distance [Kru92] and hence the programming complexity [Boo91]. Issues of reusability of software are addressed in [Jon84, Ker84, Mat84]. The paradigm proposed in this paper satisfies the criteria for reusability architecture [Goe90]. The argument for realizing reusability by evolutionary approach can be found in [Che84]. The success of hierarchical approach is highly praised in [Ker84]. The paradigm being described in this chapter supports all the approaches within a single paradigm.



## 4.6 Conclusion

This chapter proposes an approach to blend systematically two programming paradigms: functional programming and object-oriented programming, into one. The motivation is to use FP to build provably correct program modules and then to reuse them to construct higher level programming entities. One can take advantage of both FP and OOP in a single paradigm. The blended paradigm consists of three levels; The first level is the functional level where FP is used as a working language; The second level is the class level at which a user can define the building blocks (classes) intended for object-oriented programming; Sitting at the outermost is the object level where classes defined at the class level are instantiated to concrete objects. Message passing among the objects is conducted by the control mechanisms provided at this level. A binding mechanism is proposed as to effect the links between the functional level and the class level. A prominent feature of the resultant paradigm is that each level is an abstraction of the level below it yet has its own conceptual autonomy. In the process of programming, users concentrate on just one concept at a time. This reduces the programming complexity. To further ease programming in the proposed paradigm, a programming environment based on the idea presented in this chapter is proposed in the next chapter.

## CHAPTER V

### A SUPPORTING ENVIRONMENT

#### 5.1 Rationale

The overall goal of the research outlined in this thesis is to promote software reusability and, eventually, to promote programming productivity. While one exploits the reusability to gain productivity, one may introduce some overheads, which would overshadow the overall gain in productivity. This conjecture is not just an after thought. Instead, it is a real problem in the practice software reuse. Imagine that an electrical engineer who has piles of old components of thousands kinds is going to wire a board for missile control. Not only his design is complex, but so is his environment also. Finding a usable component in piles of components would take one as much time as designing the component. A reuse programmer faces the same problem which is even more serious than what an electrical engineer faces if the reuse artifacts are not kept in an appropriate order. In order to avoid overwhelming reuse programmers with the reuse components, this chapter proposes a supporting environment to assist programmers to get relevant information about reusable components and to help him/her locate a reuse artifact.

Unlike the other design process, software development from start to the end is mainly an intellectual process. The amount of intellectual efforts a developer puts to carry the development from one stage to the next in the process is defined as cognitive distance [Kru92]. Exploiting reusability is to shorten the distance in the process. The only goal of the environment is to help programmers to reduce the cognitive distance in

the software development process. An environment, as part of a reuse approach, must reduce the cognitive distance [Kru92].

## 5.2 A Reuse Process Model

Simply stated, software reuse is a process of constructing new software from existing ones. This process is different from conventional programming process in that the design does not start from scratch. In conventional processes, a designer does not have any usable components to use. He/she has to start from code level. This process involves many detailed design issues, hence is very time consuming. Moreover, conventional process repeats the same efforts over and over again. There is no sharing of code among projects. Every project has its own specific code, even though the some of the code has been encountered in some other projects before. Reuse intends to change the software practice by fabricating software components. Just as in electrical engineering, where engineers design circuits using the off-the-shelf integrated circuits (IC), software engineers build software-ICs [Cox86]. When the needs arise, instead of writing code from scratch, one can build software using the off-the-shelf components. Thus, repeating writing old stuff can be avoided and software design cycle can be minimized.

As the other design processes, reuse process involves several phases or activities. Typically, it involves the following:

- 1) Initial design: when a programmer has a problem to solve, he/she needs to understand the problem first. He/she has to know the requirements for the final product. Based on his understanding of the problem and his/her knowledge of programming, he/she may come up with an initial design. The initial design need not to be complete to the extent all the details are exposed. In fact, it may be just some sketchy ideas about the program structure, or functionality partitions, or relations among conceived constituents of the program. Since a programmer is supposed to program with off-the-shelf components,

he/she should try to stay at a higher level than he/she would with conventional programming languages. If one tends to think of all the details at the initial step, he/she may lose the advantages of higher level components available on the shelf. A novice programmer needs training to know to reuse design and to know what components are available in the repository. An experienced programmer may, at the very initial step, think in terms of usable components and his/her initial design is more concrete towards a feasible product.

2) Selection: with the conceived components in mind, one who plans to construct a software from existing components has to locate the components he/she conceives in the initial design. Once having located a plausibly reusable component, the designer must understand the components and its usage. After a careful study, he/she may pick up the component and adjust his/her initial design accordingly in order to use the components. He/she may have many choices to choose from for a conceived component. Thus, he/she must compare the components with one another in order to make an appropriate choice. In case of having no reusable components for certain conceived components, one has to take other measures.

3) Specialization: when an existing component is chosen, the usual case is that a chosen component may not directly be usable for the problem at hand. Thus, the component may need to be specialized or customized accordingly before being put into use. One important thing that a designer must do is that he/she must estimate the efforts that are needed for the specialization. If the specialization takes too much efforts, it may not be worthy of specializing it at all. The designer also has to consider the usability of the specialized components. If the specialized components do not comply to the requirements of the final products, the designer has to render other means.

4) Integration: when the components conceived in the initial design have been effected by the off-the-shelf components and are specialized according to the problem at hand or even written from scratch using an appropriate language, then the designer must put all

the components into a working program. He/she may simply link all the components into an executable component if all the components are directly usable, or invoke some tools such as compilers and transformers to make an executable product.

The activities discussed above in the reuse process are not independent, rather, they depend on one another. A designer cannot finish this process in one run. Instead, he/she may go back and forth many times since later activities constantly affect earlier designs. The process can be sketched as in Figure 1.

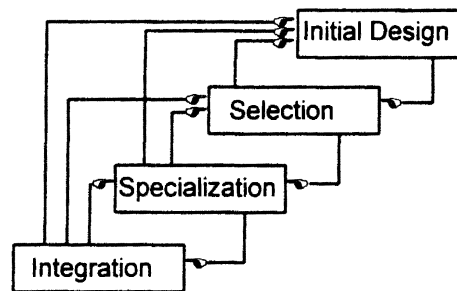


Figure 1. A Software Reuse Process Model

### 5.3 Support Required in Reuse Process

As stated in the Section 5.1, the goal of the environment is to reduce the efforts needed to carry software development from one stage to the next. Since the reuse process involves four phases as described in the last section, the environment supports the process phase by phase.

It is encouraged that the initial design is conducted in an object-oriented approach. That is, a problem is decomposed into components by the object-oriented process. Thus, the designer uses the object-oriented ideas to analyze his/her problems and his/her objective is to decompose the whole problem into manageable pieces. Each piece should have a clear interface and a functionality definition. Moreover, the relation with other

pieces must be made clear in terms of message passing. To support this phase, the environment must provide a means for a designer to define objects and to describe relations among the defined objects. When a designer needs to look at an object, the environment can help the designer to retrieve the object. The environment must also be able to provide a forest view of the problem decomposition by means of graphic capabilities. The grid mechanism in [Oss89] can be used to describe the interrelationship between objects.

With thousands of reuse artifacts, a designer easily can be lost in the reuse process. Thus, the environment must provide a catalog of reuse artifacts to assist the designer to locate a component. A paramount means for the selection process is a classification scheme by which all the components can be correctly classified and located. A set of functions such as searching, inserting, deleting, etc. are strongly desired. A reuse designer must be able to find the desired components very quickly. In addition, he/she is also allowed to design his own components and be able to keep in the inventory for later uses.

The specialization of a reusable component is effected by the languages in our multi-level paradigm. For example, if a designer finds a reusable function in the functional level, he/she can customized it using the FP language provided in the function level. Since our programming paradigm is a multi-level paradigm, reuse takes place at different levels. The languages provided at different levels in the multi-level programming paradigm are contrived to help designers to accomplish customizing reuse artifacts at different levels. To let a designer use all the languages efficiently, language specific editors are highly desired. One important thing that needs emphasizing is that designers must be liberated from syntax of the languages. To achieve this, graphic programming can be solicited. If the textual editing cannot fully be replaced by graphic means, the textual interaction must be suppressed to a minimal extent. Experience tells that experiments are intensively conducted to test the feasibility of a design in the early

stage of software development. Therefore, the environment must allow developers to conduct the experiment with short turn around time. Thus, it makes sense to arm developers with interpreters at hand.

Integration is the final phase in the reuse process. It is to make all the components linked into an executable product. To support this phase, the environment must provide a set of tools such as linker, compiler, or transformer to help make all components concrete. Interpreters are also acceptable if efficiency is not a critical issue in the final products. To save some efforts, the implementation is encouraged to provide a set of transformers to transform all representations into C/C++ programs. The C/C++ compiler is then used to make a final link of all the components.

#### 5.4 User Interface

Since the conceived programming paradigm is a three level paradigm and users may switch back and forth between different levels, the user interface must facilitate users to go from one level to the other. Each level is supported through an independent window. All three levels overlap but are active as in Figure 2. To switch to other levels, users simply activate the desired window by means of pointing devices.

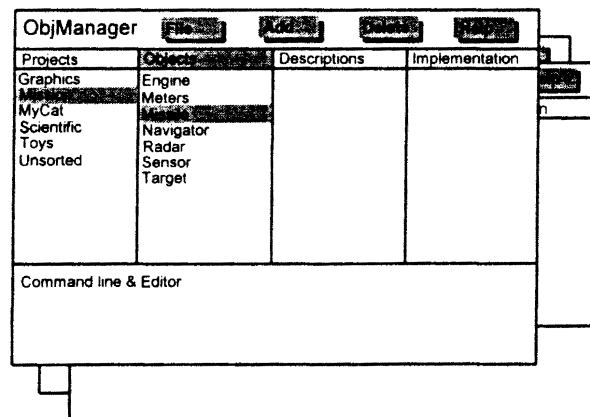


Figure 2. Interface to the Object Level

### 5.4.1 Object Manager

The outermost level is the level where the initial design takes place. The environment is going to provide users an interface facilitating initial design. The interface looks like the one in Figure 2. It is effected by a window consisting of several sub-windows, each of which is responsible for certain functionality. Users can switch back and forth among the sub-windows. Specifically, the window has following sub-windows:

- 1) button area showing available commands for the current active sub-window. the available command at a time depend on the active window beneath the area.
- 2) project window showing the projects being conducted in the system. When this window is active, users can choose the project to work on. Users can access all the information about the project via the commands in the command area. Users are not forced to have just one entry for one project in the object window. In the case of a large project, many entries can be used to refer to different parts of the project.
- 3) object widow showing all the objects in a specific project. Once users have chosen the project to work on, this widow shows all the objects in the project. Users can examine, search for, define and delete objects at will via the buttons available in the command area.
- 4) description window. This window shows the textual document of a specific object.
- 5) implementation widow. It gives the implementation code of a specific object selected in the object window.
- 6) command line and editor window. This window is designed to interact with users via textual input/output.



### 5.4.2 Class Manager

The interface to the second level of the three level paradigm is called class manager. The interface, therefore, provides the function to manage the activities concerning the generic classes in the reuse system. The activities on the classes include defining, retrieving, comparing and deleting classes. Behind the interface is a cataloging system by which all the classes are classified to different categories. The classification is done by users. Like the object manager, the class manager also consists of six sub-windows as in Figure 3.

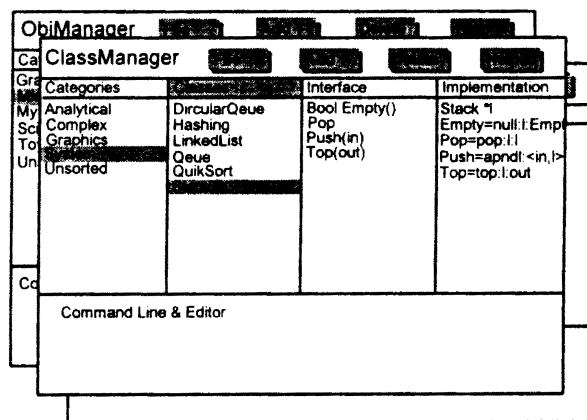


Figure 3. Interface to the Class Level

The six sub-windows are as the following:

- 1) command area as in object manager, showing available commands for the current active window underneath.
- 2) category window. Classes are classified into different categories. This window shows the available categories in the system. Users can create, delete, look at and search for a specific category.

- 3) class window showing all the classes in the current category selected in the category window. This is where the selection in the reuse process takes place. Users look for the reuse classes via the tools that are provided in this interface.
- 4) interface window. While users go through from one class to the next in the class window, the interface information is displayed in this window. Thus, users can take a brief look at what the class does without switching windows.
- 5) implementation window. It is basically the same as the interface window, but gives the implementation information about a class of current interest.
- 6) command line and editor window, which is similar to that in the object manager.

#### 5.4.3 Function Manager

This is an interface with the functional level, where users can program in FP language. Since users are encouraged to reuse the previously defined functions, the functions defined in the system need to be managed to facilitate the reuse process. To ease searching for a reusable function, like the object manager, the function manager also has a cataloging system by which all the functions are classified to different categories. Users are allowed to switch to different categories by either moving pointing devices or pressing arrow keys. The interface, as the other two interfaces introduced above, has six sub-windows. The overall organization of the interface is as in Figure 4. Six sub-windows are as the following:

- 1) command area as in object and class managers.
- 2) category window showing available categories in the function library. Users can add, delete, switch to categories via the commands in the command area once the category window is active.
- 3) function window showing all the functions under a specific category.
- 4) description window showing the concerning information of a given function.

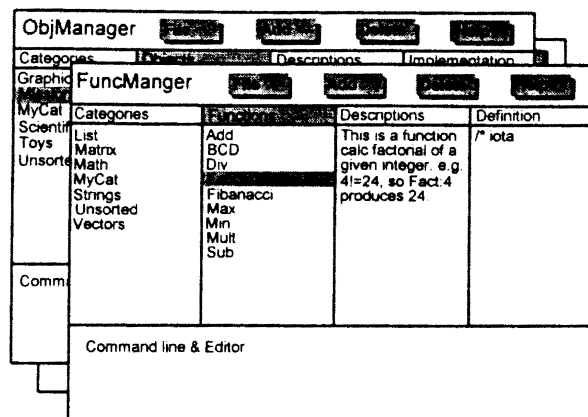


Figure 4. Interface to the Function Level

Once a function in the function window is highlighted, this window gives the descriptive information associated with the function.

5) definition window showing the definition of the function in FP language.

6) command line and editor. This window is basically as same as its cousins in the other two managers described before. Additionally, it provides an interpreter for FP language. Users can define and test FP functions using the interpreter.

## 5.5 Summary

This chapter proposes a supporting environment for the multi-level programming paradigm introduced in Chapter IV. The overall rationale for such an environment is to minimize the intellectual efforts involved in the reuse process. This rationale is mandatory as a part of this reuse technique. The reuse process introduced in this chapter involves four stages; i.e., initial design, selection, specialization and integration. The four stages are dependent upon one another. They affect one another as software development goes on. Thus, reuse designers often go through the four stages back and forth many times. The environment is conceived to provide support for all the stages in the software reuse process. The main effecting power of the environment is a set of tools interfaced

with each other by windows. Each level has an independent set of tools and interface as well as an independent conception of programming. Yet, three levels are integrated in such a way that users can switch back and forth to different levels at will.

## CHAPTER VI

### LANGUAGE DEFINITIONS

#### 6.1 Introduction

This chapter defines the language used at both the class and object levels. The FP language used at the functional level is not defined here deliberately. Omitting defining FP has two implications. One implication is the tolerance of the paradigm. It does not really matter what FP language is used in the paradigm since the class level interacts with the functional level via a function library. Any FP language, even non-Backus' FP, can be incorporated into the paradigm. The other implication is the environment portability. Since the class and object levels are abstract levels above FP, an implementor of the paradigm can transport the two levels from one machine to another without spending any efforts to the two levels.

The form of the language used at the class and object levels is described by means of a context-free syntax with context-sensitive requirements expressed by narrative rules. The context-free syntax of the language is described using a simple variation of Backus-Naur-Form (BNF). In particular,

(a) Lower-cased words, some containing embedded underlines, are used to denote syntactic categories, for example:

`method_invocation`

(b) Bold-faced words are used to denote reserved words, for example:

**signature**

(c) Square brackets enclose optional items, for example:

**return** [expression]

represents either **return** or **return** expression.

(d) Braces unless in a pair of quotation marks, in which case it represents itself, enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the following two rules are equivalent:

`compound_statement ::= '{' statement ; {statement;} '}'`

`compound_statement ::= '{' statement ; '}' | '{' compound_statement statement ; '}'`

(e) A vertical bar | separates alternative items unless it occurs in a pair of quotation marks as in (d), in which case, it represents itself.

(f) Quotation marks enclosing a single character represent the character in the language being defined as in (d) and (e). Thus the '{' represents { itself in the language being defined, and '"' represents '.

The language definition follows a bottom-up fashion, i.e., definitions of bottom entities like characters are given first, then identifiers, statements and programs are given with the constituents defined already.

## 6.2 Lexical Elements

### 6.2.1 Character Set

The only characters allowed in the text of a program are the graphical characters and format characters found in ASCII graphical symbol set. The ASCII character set are divided into the following categories:

`letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z`

`| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |`

`digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

`operator ::= + | - | * | / | % | < | = | > | & | ' | ^ | @`

`special_character ::= " | # | ' | ( | ) | , | . | : | ; | _ | ! | $ | ? | [ | \ | ] | ` | { | } | ~`

Note: except the characters that are visible, there are some characters in the ASCII character set that are not visible yet exist in the language being defined. These characters are called `space_character`, including space, horizontal tabulation, carriage return.

### 6.2.2 Lexical Elements

A program consists of a sequence of lexical elements, each of which being either an identifier, a separator, a delimiter, a literal, or a comment.

`identifier ::= letter { letter | digit | _ }`

`separator ::= space_character`

`delimiter ::= operator | ; | , | ' { ' } ' | ( | )`

`integer_literal ::= [ + | - ] digit { digit }`

`real_literal ::= [ + | - ] digit { digit } . digit { digit } [ [ + | - ] e digit { digit } }`

`string_literal ::= " { letter | digit | space_character | basic_operator | special_character } "`

`character_literal ::= ' letter | digit | space_character | basic_operator | special_character '`

`literal ::= integer_literal | real_literal | string_literal | character_literal`

`comment ::= // { letter | digit | space_character | basic_operator | special_character }`

Note:

If a string literal expects a " as part of the literal, the character " must be doubled in the literal to represent the character itself.

A comment starts with // and ends with a carriage return (new line). Thus, a comment can have all space characters but carriage return in the middle. A comment can appear anywhere between two lexical elements, but not within a lexical element.

### 6.2.3 Reserved Words

The following identifiers are the reserved words in the language being defined, thus, they should not be used as ordinary identifiers.

<b>body</b>	<b>domain</b>	<b>else</b>	<b>extern</b>	<b>for</b>
<b>if</b>	<b>infix</b>	<b>inherit</b>	<b>integer</b>	<b>main</b>
<b>prefix</b>	<b>real</b>	<b>return</b>	<b>signature</b>	<b>string</b>
<b>then</b>				

## 6.3 Class Declaration

A class declaration consists of two parts: one is the signature of the class, which specifies the interface of the class being declared; The other is the body of the class, which specifies the implementation of the class using FP function library. Class declaration is only conducted at the class level in the multi-level programming paradigm proposed in the Chapter IV. Every class has an entry in the class library, which is accessible from object level.

### 6.3.1 Domain Declaration

A domain declaration associates a set of objects with a name so that the name represents the set of objects. A domain declaration is only valid in the class in which the domain declaration appears.

```
domain_specification ::= integer | real | string | identifier | '{' { literal } '}'
                        | < [ domain_specification {, domain_specification } ] >
                        | < [ domain_specification [ * ] ] >
```

```
domain_declaration ::= domain identifier = domain_specification ;
```

Note: **integer**, **real** and **string** are predefined domains with conventional interpretations.

The identifier in the domain specification must be a name of a domain declaration that



following a domain specification indicates a domain of a sequence of objects in the domain specified by the domain specification.

### 6.3.2 Signature Declaration

A signature specifies the interface of a class, including the domains and names of all the instance variables in the class, methods and usage of the methods in the class. A method declaration specifies the interface of a particular method in a class, including the domain of the method, number, order, form and domain of the parameters of the method. The form of a method specifies if the method is an infix operator, a prefix operator, or an ordinary procedure.

form ::= **infix** | **prefix**

domain\_name ::= identifier | **integer** | **real** | **string**

instance\_variable\_declaration ::= identifier : domain\_specification;

method\_declaration ::= [ form ] identifier

( [ domain\_name { , domain\_name } ] ) : domain\_name;

signature\_declaration ::= **signature** identifier '{' { domain\_declaration }

{ instance\_variable\_declaration }

{ method\_declaration }

'}'

Note: the identifier in the domain\_name must be declared in the domain declaration in the signature of the same class.

### 6.3.3 Body Declaration

A body declaration of a class specifies the implementation of the class using the underlying FP functions. It consists of a set of binding specifications.

input\_pattern ::= identifier | < [ input\_pattern { , input\_pattern } ] >

output\_pattern ::= \* | identifier | < [ input\_pattern { , input\_pattern } ] >

```

binding_specification ::= [ identifier | operator ] ( [ identifier {, identifier} ] ) =
                        [ identifier | operator ] : input_pattern : output_pattern ;
body_declaration ::= body identifier '{'
                        { instance_variable_declaration }
                        { binding_specification }
                        '}'

```

The meaning of a binding specification is interpreted as the following:

- (a) values associated with all the identifiers in the `input_pattern` are collected in a structure of the `input_pattern`.
- (b) the structure is passed to the FP function named by the preceding identifier.
- (c) the application result, which has a structure of `output_pattern`, is extracted and assigned to the carriers according to the `output_pattern`.

Note: all the identifiers in the `input_pattern` and `output_pattern` of a method must be the instance variables declared in the same class, or the parameters passed to the method. The `output_pattern` can have as one element the identifier of the method with which the output pattern is specified. The `*` in the `output_pattern` matches anything structurally corresponding to it. Its appearance in a position indicates that the corresponding part of the application result is to be ignored.

#### 6.4 Object Module Declaration

An object module consists of a collection of object instantiations and a set of procedures that are used to control message passing among objects. Each module has an entry in the object library for project management. Object module declaration is only conducted in the object level in the multi-level programming paradigm proposed in Chapter IV.

### 6.4.1 Object Instantiation

An object instantiation creates an object of a given class. It allocates the memory needed for the object. If an object is instantiated other than within a procedure, it is accessible from all procedures in an application. If an object is instantiated within a procedure, it is only accessible from within that procedure.

`class_name ::= identifier`

`object_instantiation ::= identifier {, identifier } : class_name ;`

Note: identifiers proceeding `:` are the names of the instance variables of the class represented by the `class_name` following the `:`, which must be a valid class name in the class library.

### 6.4.2 Object Access

Once an instance of a class is created, the instance and its constituent structure can be accessed via the instance variables.

`object_name ::= identifier`

`instance_name ::= identifier`

`object_access ::= [ object_name . ] instance_name | object_name`

Note: the `object_name` in an object access must be a name of an object instantiated by an object instantiation prior to the point at which the object access appears. The `instance_name` following the period must be an instance variable in the class from which the object is instantiated.

### 6.4.3 Statement

A statement is a control abstraction used to manipulate message passing among objects. The most basic statement in the language being defined is method invocation, which is similar to a conventional procedure call.

```

method_name ::= identifier

expression ::= method_invocation | object_name | object_access | literal | ( expression )

method_invocation ::= [ object_name . ] method_name ( [ expression { , expression } ] )
                    | operator ( expression, expression )
                    | expression operator expression

if_control ::= if expression then statement [else statement]

initial ::= method_invocation

condition ::= method_invocation | literal

update ::= method_invocation

for_control ::= for ( [ initial ] ; [ condition ] ; [ update ] ) [ statement ]

return_control ::= return [ expression ] ;

simple_statement ::= method_invocation ;
                  | if_control
                  | for_control
                  | return_control

compound_statement ::= '{' { simple_statement } '}'

statement ::= simple_statement | compound_statement

```

Note: an object\_name must have a corresponding object instantiation prior to the point the object\_name is used. The object\_name in method invocation, which must be a valid method in a class, can be omitted provided that the method\_name is unique in the whole application. The if\_control has the following semantics: the expression is evaluated and the result is compared with the first element of the corresponding class. If the comparison yields an equality, control passes to the part following **then**. Otherwise control passes to the part following the **else** if there is an **else** part, or to the subsequent statement if there is no an **else** part. The if\_control in this language is different from conventional if statement. In this language, the interpretation of truth is dynamically dependent upon the class (domain) the truth is associated with. The for\_control has

control mechanism similar to conventional for statement except the interpretation of the condition part, which is as same as in the `if_control`. The `return_control` is used to terminate a procedure execution. If the procedure is declared to have a value to carry upon completion, an expression must exist after the **return**.

#### 6.4.4 Procedure Declaration

A procedure/function specifies a process of message passing. It contains a set of local objects used to manipulate message passing to be conducted in the procedure being declared. Message passing is conducted by means of statements in the procedure.

`procedure_head ::= identifier`

`( [ identifier : identifier { , identifier : identifier } ] ) [ : identifier ]`

`| main ( )`

`| main ( identifier : string )`

`procedure_body ::= '{' { object_instantiation } { statement } '}'`

`procedure_declaration ::= procedure_head procedure_body`

Note: if a procedure declaration contains a class name as its type, the procedure returns an object of that class to its invoker. In side of that procedure there must exist at least one return statement returning an object of the class expected in the procedure head declaration. Main procedure identified by **main** specifies a special procedure which intrigues the execution of the whole application. For one application, only one main procedure is allowed. Main procedure may have a string as its parameter. This parameter is obtained from execution environment and kept in the object supplied by the main procedure head.

#### 6.4.5 Object Module Declaration

An object module specifies a compilation unit of objects. An object module declaration specifies a set of related objects and associated procedures/functions. All the

objects instantiated before procedure declarations are global objects, which are legally accessible from within other modules as well as the current module being declared. All the procedures declared in an object module are global procedures and accessible from every part of the application for which the procedures are defined. Objects or procedures declared using **extern** are the items defined in the other modules and intended to be used in the current module.

```
external_declaration ::= extern object_name { , object_name } : class_name ;
                        | extern procedure_name ( class_name { , class_name } ) : [ class_name ]
object_module_declaration ::= { external_declaration }
                             { object_instantiation } { procedure_declaration }
```

## 6.5 Summary

The language being defined in this chapter consists of two major parts. One is the class declaration used at the class level. The other is the object module declaration used at the object level. When using the two parts in the two different levels, one is assisted by the libraries existing at the two levels. The two parts are conceptually independent from each other, though they share some syntactic similarity. To simplify translation in the language implementation, all the items must be declared prior to their uses. An object instance variable or an object method access can be done without using object name only under the condition that the compiler can resolve name conflicts. Methods that bear names consisting of operator characters are called operators. There is no priority among the operators. Operations designated by operators in an expression are performed in the order they occur in the expression unless the order is changed by a pair of parenthesis, in which case, the operator in the innermost pair of parenthesis has the highest priority. The syntax of the language defined in this chapter is similar to the conventional language. However, the underlying principle is dramatically different from that in its conventional counterparts as explained in the concerning sections.

## CHAPTER VII

### SUMMARY, CONCLUSIONS AND FUTURE WORK

#### 7.1 Summary

The mismatch of software artifacts and human cognitive capability as well as the lack of standard data interchange format, architectural support and reusable designs are the main obstacles to software reusability. Neither functional programming nor object-oriented programming offers a perfect solution. As an alternative, this thesis proposes a multi-level programming paradigm and a support environment for software reusability, aiming at both simplifying cognitive process and providing architectural support for software reusability. The proposed paradigm is a combination of both functional programming and object-oriented programming paradigm, intending to provide its users with the advantage of both functional and object-oriented programming. The motivation is to enhance reusability facilities from OOP with mathematical foundations from FP for a disciplined approach to software reusability. In order to minimize the cognitive effort involved in using the proposed paradigm, the conceptual autonomy is observed in unifying the two different paradigms. In addition, a support environment is outlined as an integral part of the proposed paradigm to ease programming in this paradigm further.

#### 7.2 Contributions and Conclusions

The major contribution of this thesis is a multi-level programming paradigm and a programming support environment as a disciplined approach to software reusability. The multi-level programming paradigm consists of three levels; i.e., the functional level, the

class level and the object level. Prominently, each of the three levels exists in its own autonomy and has its own paradigm, thus programming complexity is reduced by the fact that programmers can concentrate on only one concept at a time.

Other contributions of this thesis include a systematic approach to the unification of functional programming with object-oriented programming. Being a bridge between functional programming and object-oriented programming, the binding mechanism introduced at the class level is the cornerstone of this approach. It hides all the details in the functional level from its users. Yet, it lets its users use functions defined at the functional level to define classes. The binding mechanism links a functional paradigm to an object-oriented paradigm without introducing much conceptual complexity.

Introduction of the notion *conceptual autonomy* is another contribution of this thesis. The notion emphasizes the autonomy of a concept a user concentrates at a time. Thus, it helps to reduce the cognitive complexity.

In addition, the thesis has also defined the languages to be used at all levels.

### 7.3 Future Work

The following issues need to be addressed in the future:

1) Evaluation: the work reported in this thesis has been claimed as a reusability technique both simplifying cognitive process in software development and providing architectural support for software reusability. The technique needs to be evaluated with respect to cognitive complexity and effectiveness of architectural support. This will need a prototype of the environment and users' involvement.

2) Comparison: the approach to unify functional paradigm with object-oriented programming paradigm outlined in this thesis is different from conventional approaches. This thesis takes a hierarchical approach, which has its obvious advantage over conventional ones. However, more comparison needs to be made between this approach



and other approaches with respect to the effects they have on users in the programming process.

3) Program algebra: the algebra of FP programs at the functional level can be directly used to reason about the functions defined at the functional level since it is inherent in FP. Extending the algebra to the class and object levels would be both interesting and challenging. Theory in the abstract data type needs to be extended and combined with that in FP for a systematic treatment of class correctness.

4) Implementation: the paradigm is conceived to be working with a programming support environment, which consists of tools and library management. Issues such as cataloging, user interface, library management, source code interpretation and compilation need to be detailed in the implementation of the environment.

5) Optimization: one of the drawbacks of functional programming is the low execution efficiency. This efficiency could be improved once put in the multi-level programming paradigm. Since an FP function used to effect a method in a class is constrained by some domain in the class, the function can be specialized using the domain information. Optimizations based on the domain information need to be developed.

## REFERENCES

- [Abb83] Abbott, R. Program design by informal English descriptions. Communication of the ACM Vol. 27. No. 11, November 1983.
- [Bac78] Backus, J. Can programming be liberated from von Neumann style? A functional style and its algebra of programs. CACM. Vol.21, No.8, Aug. 1978.
- [Bac81] Backus, J. The algebra of functional programs: functional level reasoning, linear equations, and expanded definitions. Lecture Notes in Computer Science, Vol. 107, Springer-Verlag, 1981.
- [BEA86] Bellia, M, et al. A two-level approach to logic plus functional programming integration. Lecture Notes in Computer Science, Vol. 117, Springer-Verlag, 1985.
- [Boo91] Booch, G. Object oriented design. The Benjamin/Cumming Publishing Company, Inc. 1991.
- [BP90] Bollinger, T. B. & Pfleeger, S. L. Economics of reuse: issues and alternatives. Information and Software Technology. vol. 32. No. 10. December 1990.
- [Bud91] Budd, T. An introduction to object-oriented programming. Addison-Wesley Publishing Company. 1991.
- [BWW86] Backus, J., Williams, J. H. & Wimmers, E. L. FL language manual. Technical Report, IBM Almaden Research Center, 1986.
- [Che84] Cheatham, Jr., T. E. Reusability through program transformations. IEEE Trans. Soft. Eng., Vol. SE-10, No.5, Sept. 1984.
- [Cox86] Cox, B. J. Object Oriented Programming: An Evolutionary Approach. Addison-wesley Publishing Company. 1986
- [DM84] Dosch, W. Moller, B. Busy and lazy FP with infinite objects. *in the Conference Record of the 1984 ACM Symposium on LISP and Functional programming*, 1984.

- [FG90] Frakes, W. B. & Gandel, P. B. Representing reusable software. *Information and Software Technology*. Vol. 32, No. 10. December 1990.
- [GD89] Gaffney, Jr, J. E. & Durek, T. A. Software reuse - key to enhanced productivity: some quantitative models. *Information and Software Technology*. Vol. 31. No. 5. June 1989.
- [Geo90] George, K. M. A multilevel programming paradigm. in *Proceedings of the Eighth Int'l Phoenix Conference on Computers and Communications*, (Mar. 23-25, 1990, Phoenix, Az.), pp.340-346.
- [Geo89] George, K. M. Objects and data structures in the FP paradigm. in *Proceedings of the Seventh Int'l Phoenix Conference on Computers and Communications*, (Mar. 1989, Phoenix, Az.)
- [GH88] George, K. M. & Hedrick, G. E. Expansions in the algebra of FP. *Proceedings of the ACM Computer Science Conference*. February 23-25, 1988.
- [GM87] Goguen, J. A. & Meseguer, J. Unifying functional, object-oriented and relational programming with logical semantics. In *Research Directions in Object-Oriented Programming*, The MIT press, 1987.
- [Hal87] Hall, P. A. V. Software components and reuse - getting more out of your code. *Information and Software Technology*. Vol. 29. No. 1. January/February 1987.
- [HEA91] Hudak, P. et al. Report on the Programming Language Haskell A Non-strict, Purely Functional Language. Department of computer Sciences, Yale University. 1991.
- [HK87] Harrison, P. & Khoshnevisan, H. An introduction to FP and the FP style of programming. In book *Functional programming languages, tools and architectures*. Eisenbach, S. Ed. Ellis Horward Ltd. 1987.
- [HN87] Hailpern, B., Nguyen, V. A model of object-based inheritance. In *Research Directions in Object-Oriented Programming*, The MIT press, 1987.
- [Jon84] Jones, T. C. Reusability in programming: a survey of the state of the art. *IEEE Trans. Soft. Eng.*, Vol. SE-10, No.5, Sept. 1984.
- [Ken83] Kendall, R. C. An architecture of reusability in programming. *ITT Programming*, Stratford, CT, May 1983.
- [Ker84] Kernighan, B. W. The Unix system and software reusability. *IEEE Trans. Soft. Eng.* Vol. SE-10, No.5, 1984.

- [Kru92] Krueger, C. W. Software reuse. ACM Computing Survey, Vol.24, No.2, June 1992.
- [Mat84] Matsumoto, Y. Some experiences in promoting reusable software: presentation in higher abstract level. IEEE Tans. Soft. Eng, Vol.. SE-10. No.5, Sept. 1984.
- [McI68] McIroy, M. D. Mass produced software components. In Software Engineering: Report on a Conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, Belgium, 1968.
- [Mey87] Meyer, B. Reusability: the case for object-oriented design. IEEE Software, March 1987.
- [NR68] Nauer, P. & Randell, B., Eds. Software engineering; Report on a Conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, Belgium, 1968.
- [Oss89] Ossher, H. A case study in structure specification: a grid description of Scribe. IEEE Transactions on Software Engineering. Vol. 15, No. 11, November 1989.
- [PA91] Purtilo, J. M. & Atlee, J. M. Module reuse by interface adaption. Software practice and Experience. Vol. 21. No. 6. June 1991.
- [RK83] Rich, E. & Knight, K. Artificial Intelligence. McGraw-Hill, Inc., 1983
- [RL89] Raj, R. K. & Levy, H. M. A compositional model for software reuse. the Computer Journal. Vol. 32. No. 4. 1989.
- [Veg84] Vegdahl, S. R. A survey of proposed architectures for the execution of functional languages. IEEE Transactions on Computers. Vol. C-33, No.12, December, 1984.
- [Weg87] Wegner, P. The object-oriented classification. In *Research Directions in Object-Oriented Programming*, The MIT press, 1987.
- [Wil82a] William, J. H. On the development of the algebra of functional programs. ACM TOPLAS, Vol.4, No.4, Oct. 1982, pp.733-757.
- [Wil82b] William, J. H. Notes on the FP style of functional programming. in *Functional Programming and its Application*, ed., J. Darlington, P. Henderson and D. A. Turner, Cambridge University Press, London, 1982.
- [YT87] Yonezawa, A. & Tokoro, M. Object-oriented concurrent programming: an introduction. in book *Object-Oriented concurrent Programming*. MIT, 1987.

VITA

Zhiyu Zhang

Candidate for the Degree of

Master of Science

Thesis: A MULTI-LEVEL PROGRAMMING PARADIGM AND ITS SUPPORT  
ENVIRONMENT FOR SOFTWARE REUSABILITY

Major Field: Computer Science

Biographical:

Personal data: Born in Changwu, Shaanxi, China, June 16, 1962, the son of  
Delu Zhang and Shuiling Xu.

Education: Received Bachelor of Engineering Degree in Electrical and Computer  
Engineering from Xi'an Institute of Technology in July, 1983; Obtained the  
Certificate of Excellence in Intensive English Program from Beijing Institute  
of Aeronautics and Astronautics in January, 1986; Completed requirements  
for the Master of Science Degree at Oklahoma State University in July, 1993.

Professional Experience: Teaching Assistant, Department of Computer Science,  
Oklahoma State University, August, 1991, to May, 1993; Software  
Engineer, Aeronautical Computing Technique Institute, Xi'an, China, July,  
1983, to August, 1991; Visiting Faculty, Department of Computer Science,  
Oklahoma State University, February, 1987, to February, 1988.