

**Separating Data from Instructions:
Investigating a New Programming Paradigm**

by
Yasushi Kambayashi

ISBN: 1-58112-168-7

DISSERTATION.COM



USA • 2002

Separating Data from Instructions: Investigating a New Programming Paradigm

Copyright © 2002 Yasushi Kambayashi
All rights reserved.

Dissertation.com
USA • 2002

ISBN: 1-58112-168-7
www.Dissertation.com/library/1121687a.htm

A Dissertation
entitled
Separating Data from Instructions:
Investigating a New Programming Paradigm

by
Yasushi Kambayashi

As partial fulfillment of the requirements for the
Doctor of Philosophy Degree in Engineering

Advisor: Dr. Henry F. Ledgard

Graduate School

The University of Toledo
May 2002

An Abstract of

Separating Data from Instructions:
Investigating a New Programming Paradigm

Yasushi Kambayashi

Submitted in partial fulfillment
of the requirements for the
Doctor of Philosophy Degree in Engineering

The University of Toledo
May 2002

We have experienced different theories of software construction paradigms in the last few decades; such as “structured programming” in the 1970’s and “object-oriented programming” in the 1980’s. The object-oriented paradigm is considered a standard for many software development activities, from the analysis phase to various support phases. There is little quantitative research, however, regarding the question whether object-oriented programming improves productivity. Many assume that object-oriented programming is more productive than traditional structured programming. This assumption lacks concrete, empirical data that support such belief.

This dissertation identifies problems in the current object-oriented programming practice, and then presents an alternative paradigm to help overcome these problems. This paradigm separates the declaration of data structures from program executable instructions. We call this paradigm *the separation principle*. We first tried to understand what this paradigm means in practice. We developed example programs in a variety of application areas. We found that the separation principle is a viable paradigm for practical program construction. In order to demonstrate the validity of

this paradigm, we have conducted both theoretical and empirical studies. The theoretical study consists of complexity measurements. The empirical study constitutes human understanding measurement; its purpose is to show statistical significance. The results give evidence of the effectiveness of the separation principle for practical software construction.

Acknowledgments

The author would like to express his sincere thanks to all who made this dissertation possible.

It is with special appreciation that I acknowledge my advisor, Professor Henry Francis Ledgard for his insight, his encouragement, and his enthusiasm for this project. Without the direction of Professor Ledgard, this work could have never been completed.

I wish to express my gratitude to other members of the Examination Committee, Professor Mansoor Alam, Professor Moshin M. Jamali, Professor Jeffrey D. Johnson, Professor Hilda M. Standley, and Mr. Glenn Jacobson, President of Unique Systems, for their kind advice, deep insight and encouragement.

I wish to thank Mr. William C. Cave, originator of VSE, who provided many deep observations on the software field.

I extend many thanks to Professor Warren Harrison, from Portland State University, for his help with the program complexity measurement.

To Professor Stephen R. Schach, from Vanderbilt University, I extend gratitude for his permission to use his program for the experiments.

I wish to thank all the tutors in the Writing Center who meticulously polished my writing.

I would like to thank the Department of Electrical Engineering and Computer Science for the support to my graduate study.

It is with deep appreciation that I acknowledge students in the department of Electrical Engineering and Computer Science who participated in experiments, especially my colleagues, Shantanu Kalkarni, for his encouragement in numerous ways, as well as Mike Orra and Dave Hamden, for their help in experiments and productive discussions.

Last, but not least, I would like to express my deep appreciation to my parents, Tsutomu and Kikuko Kambayashi, and my fiancé, Yukari, for their encouragement and patience.

Contents

Abstract	ii
Acknowledgments	iv
Contents	v
List of Figures	ix
List of Tables	xiv
1 Introduction	1
2 Programming Paradigms	5
2.1 Assembly Programming	6
2.2 Imperative Programming	7
2.3 Structured Programming and Functions	10
2.4 Encapsulation and Ada Packages	12
2.5 Object-Oriented Programming	16
3 Problems with the Object-Oriented Paradigm	18

3.1	OO is Unnatural for Practical Software	18
3.2	Object-Oriented Programs Are Difficult to Reuse	20
3.3	OO Concepts Are Difficult to Learn	22
3.4	OO Lacks Objective Studies on Productivity	24
4	The Separation Principle	26
4.1	Basic Idea	26
4.2	Cave's Solution	27
4.2.1	The VSE Language: Separating Data from Instructions	27
4.2.2	Another Example: Selection Sort	31
4.3	Simplicity	32
4.3.1	Traditional Scope Rules	34
4.3.2	Simplicity by Using the Separation Principle	37
4.4	Software Design and the Separation Principle	39
4.4.1	Scalability	39
4.5	Application of the Separation Principle: How It Is Applied	42
4.5.1	Application to the Data Flow Analysis	42
4.5.2	Application to Object-Oriented Design	46
4.6	Application of the Separation Principle: Suitable Problems	49
4.7	Three Dimensions of this Thesis	51
5	Examples	53
5.1	Sequential File Processing	57
5.2	Restaurant Ordering	60

5.3	Simple Encryption	62
5.4	Producer/Consumer	77
5.5	Weather Polling	78
5.6	Mathematical Objects	81
5.7	Airport Simulation Example	82
5.8	Painting Gallery Management	89
6	Direct Measurement of Complexity	92
6.1	Cyclomatic Complexity	93
6.2	Halstead Complexity Measures	94
6.3	Maintainability Index	97
6.4	Function Point Analysis	99
6.5	Applying Complexity Measures to the Examples	100
6.6	Analysis of the Results	101
6.7	Assembly Codes Produced from the Examples	110
7	Measuring Understandability	116
7.1	Measuring Understandability through Experiments	117
7.2	The Experiment	119
7.3	Subjective Reactions	126
7.4	Analysis of the Results	129
7.5	Additional Experiment	130
8	Software Reuse and Design Patterns	136

9 Conclusion and Future Works	141
A Separation Principle Version of Airport Simulation Program	145
B OOP Version of Airport Simulation Program	151
C Separation Principle Version of Painting Gallery Management Program	158
D OOP Version of Painting Gallery Management Program	173
Bibliography	190

List of Figures

2-1	Conceptual view of a stored program computer. It consists of a memory unit and a CPU. A CPU further comprises a control unit, arithmetic unit and I/O unit.	6
2-2	An example program in SPARC assembly language	8
2-3	Sequence control commands.	9
2-4	A Program Compute the Factorial of x	10
2-5	If-then-else Construct	11
2-6	Case Construct	11
2-7	Loop Construct	11
2-8	If-then-else Construct Diagram.	12
2-9	Case Construct Diagram	13
2-10	Loop Construct Diagram	13
2-11	Visible Part of the Stack Package. The user of the stack package sees this part.	14
2-12	Hidden Part of the Stack Package. The user of the stack package cannot see this implementation part.	15

4-1	Conceptual Diagram of a Program Using the Separation Principle . .	27
4-2	Data Module of Calculating Average Program Written in VSE	29
4-3	Instruction Module of Calculating Average Program Written in VSE .	30
4-4	Design Diagram for Calculating Average Program	30
4-5	Data Module of Selection Sort Program Written in VSE	32
4-6	Main Instruction Module of Selection Sort Program Written in VSE .	33
4-7	Auxiliary Instruction Module of Selection Sort Program Written in VSE	33
4-8	Design Diagram for Selection Sort Program	34
4-9	Data/Instruction Modules as Reusable Software Components	41
4-10	Data Flow Diagram	43
4-11	Data/Instruction Module Extracted from the DFD	43
4-12	Data Flow Diagram for the Airport Simulation	44
4-13	Structure Chart for the Airport Simulation	44
4-14	Collaboration Diagram for the Painting Gallery Management	48
4-15	Class Diagram for the Painting Object	50
5-1	C++ Selection Sort Program Written Using Structured Programming	55
5-2	C++ Selection Sort Program Written Using Object-Oriented Style . .	56
5-3	C++ Selection Sort Program Written Using the Separation Principle	58
5-4	Design Diagram for Selection Sort Program	59
5-5	C++ Selection Sort Program Written Using Object-Oriented Style . .	61
5-6	Sequential File Processing Program Written Using Object-Oriented Style (cont.)	62

5-7	Sequential File Processing Program Written Using the Separation Principle	63
5-8	Design Diagram for Sequential File Processing Program	64
5-9	Design Diagram for Restaurant Ordering Program	64
5-10	Restaurant Ordering Program Written Using Object-Oriented Style	65
5-11	Restaurant Ordering Program Written Using Object-Oriented Style (cont.)	66
5-12	Restaurant Ordering Program Written Using the Separation Principle	67
5-13	Restaurant Ordering Program Written Using the Separation Principle (cont.)	68
5-14	Sample User Interaction for Encryption Program	69
5-15	Contents of the Output File Caused by the User Interaction in Figure 5-15	70
5-16	Simple Encryption Program Written Using Object-Oriented Style	71
5-17	Simple Encryption Program Written Using Object-Oriented Style (cont.)	72
5-18	Simple Encryption Program Written Using Object-Oriented Style (cont.)	73
5-19	Simple Encryption Program Written Using the Separation Principle	74
5-20	Simple Encryption Program Written Using the Separation Principle (cont.)	75
5-21	Design Diagram for Encryption Program	77
5-22	Design Diagram of Producer/Consumer Program	78
5-23	Producer/Consumer Program Written Using the Separation Principle	79

5-24	Producer/Consumer Program Written Using the Separation Principle (cont.)	80
5-25	Producer/Consumer Program Written Using the Separation Principle (cont.)	81
5-26	Design Diagram for Weather Polling Program	82
5-27	Weather Polling Program Written Using the Separation Principle . .	83
5-28	Weather Polling Program Written Using the Separation Principle (cont.)	84
5-29	Rational Number Abstraction Using OOP	85
5-30	Rational Number Abstraction Using OOP (cont.)	86
5-31	Rational Number Abstraction Using OOP (cont.)	87
5-32	Design Diagram for Airport Simulation Program	89
5-33	Design Diagram for Painting Gallery Management Program	91
6-1	Software Science Effort. These values suggest how much effort is re- quired to produce the program.	109
6-2	Cyclomatic Complexities. These values suggest how structurally com- plex the program is.	109
6-3	Number of "call" instructions by non-optimized compiler. The number of call instructions affects most in program efficiency.	114
6-4	Number of "call" instructions by optimized compiler. The number of call instructions affects most in program efficiency.	115
7-1	The Question Sheet for Program Comprehension	121

7-2	Scores of Each Subject. Thirteen out of twenty-six subjects got higher scores with the separation principle version.	124
7-3	The Question Sheet for Subjective Reactions	127
7-4	Results of Subjective Questions	127
7-5	The Question Sheet for Program Comprehension	132
7-6	Time Spent for the Experiment	134
8-1	Conceptual View of an Application that Uses a Design Pattern. Pattern is implemented by a set of classes.	139
8-2	Conceptual View of an Application that Uses a Design Pattern Using the Separate Principle. A pattern is divided into data pattern and instruction pattern.	140

List of Tables

4.1	Instruction and Data Modules for the Airport Simulation	45
6.1	Summary of Complexity Report on the Airport Simulation Program Written Using the Separation Principle	102
6.2	Summary of Complexity Report on the Airport Simulation Program Written Using OOP	103
6.3	Summary of Complexity Report on the Painting Gallery Management Program Written Using the Separation Principle	104
6.4	Summary of Complexity Report on the Painting Gallery Management Program Written Using OOP	105
6.5	Assembly Code Size and the Number of Branch/Call Instructions . . .	112
6.6	Optimized Assembly Code Size and the Number of Branch/Call In- structions	113
7.1	Raw Data from the First Experiment	122
7.2	Raw Data from the Second Experiment	123
7.3	Number of Correct Answers: First Experiment	124
7.4	Number of Correct Answers: Second Experiment	125

7.5	Number of Correct Answers of Paired Test (Combination of the First and the Second)	125
7.6	Time Spent for the Second Experiment: (unit: minute)	126
7.7	Raw Data from the Subjective Questions	128
7.8	Summary of Subjective Questions	128
7.9	Raw Data from the Sep. Prin. Group	131
7.10	Raw Data from the OOP Group	133
7.11	Time Spent for Test (unit: minute)	134

Chapter 1

Introduction

In the development of any technology, there is always a tendency to lose sight of the basic problems that stimulated its introduction. Technologies in construction software are no exception. The essential parts of software construction technologies are programming languages and programming paradigms.

Computer scientists have been making an extensive effort to reduce complexity in software. Actually, one can argue that the history of programming languages is a series of trials to find the Holy Grail of clear and understandable ways to construct complex applications.

We have experienced different theories of software construction paradigms in the last few decades, such as “structured programming” in the 1970’s and “object-oriented programming” in the 1980’s. Although object-oriented programming dominates current software construction, there are several subtle problems with the object-oriented programming paradigm.

In this dissertation, we identify problems in the current object-oriented program-

ming practice, and then present an alternative programming paradigm to help overcome these problems. The programming paradigm we investigate here is *separating data from instructions*. This paradigm stems from VSE (Visual Software Environment), a product of Prediction Systems, Inc. [Cave 95] [Prediction 00].

Our thesis is that this programming paradigm, separating data from instructions, is a viable and simple software construction paradigm that is applicable to the traditional structured programming languages. In this dissertation, we present the following four major works:

1. We explore what the separation principle actually means in the context of conventional structured programming languages.
2. Using this paradigm, we develop a number of example programs in various application areas.
3. We measure the effectiveness of this paradigm with traditional complexity metrics.
4. We conduct experiments on this paradigm's program understandability.

This dissertation represents the first attempt to apply this paradigm to a general purpose programming language such as C++.

Many sources indicate that about eighty percent of all software cost goes to support activities [Boehm 81][Berry 92][Yourdon 96]. It is difficult to find up-to-date figures for the relative effort devoted to the different types of software developing activities. Most recent sources indicate at least about two-thirds of total software

cost were devoted to support activities [Bell 00][Schach 02]. The most recent and most comprehensive survey was conducted at Hewlett-Packard in 1992 [Coleman et al. 94]. The survey revealed that between sixty to eighty percent of the research and development personnel were involved in support activities. There is no report that this situation has improved today.

Support activities include enhancement and removal of residual faults, as well as adapting to new environments. Some research suggests that this removal of residual faults, sometimes called corrective maintenance, is the most expensive part of all maintenance activities [Arnold 83]. Therefore, the most urgent concern on a programming paradigm is to improve the productivity of support. The programming paradigm, the separation of data from instructions, should improve the productivity of the support effort by providing better understandability of programs.

In order to explore what the separation principle paradigm means, we developed a series of programs in a variety of fields, including a simple sort program, a business application, a few file I/O programs and two concurrent programs. In addition to these small examples, we developed larger programs. One is for simulation and the other is for business application. The major goal was to see how to apply this paradigm in a conventional language setting. Through developing these example programs, we found that the separation principle is a viable paradigm for practical program construction.

In order to demonstrate the validity of this paradigm, we have conducted both theoretical and empirical studies. The theoretical study consists of complexity measurements. There are several complexity measures. We employed a metrics tool,

namely UX-METRICS, that measures the cyclomatic complexity and the Halstead measures, as well as lines-of-code metrics. The results show that programs written with our proposed paradigm, the separation principle, are simpler than programs written with the object-oriented programming paradigm.

The empirical study constitutes human understanding measurement, and the purpose is to show statistical significance. Human subjects are employed to investigate program understandability by questionnaires. We measured program understandability by counting the number of correct answers to questions about programs. The test was conducted in a classroom setting for upper-level undergraduate students. The results show that students grasp the overall structure of programs written with the programming paradigm that we are proposing more accurately and in less time than those programs written with the object-oriented paradigm. Moreover, subjective questions revealed that the majority of the students preferred the program written with the programming paradigm *separating data from instructions*.

Chapter 2

Programming Paradigms

We begin with a conceptual view of a stored program computer as shown in Figure 2-1. This has been the model of all computers developed in the last half century. Data and their instructions are together stored in the memory. Control unit takes data or instructions one by one by using the accumulator and registers; arithmetic unit interprets the instructions just taken and manipulates the data according to the instruction. In some occasions, I/O unit participates in this collaboration.

The first programming paradigm came with assembly languages, which are essentially translations of machine code. Assembly languages helped people think in a more abstract way by constructing a layer of abstraction from pure machine instructions. Another layer of abstraction was provided by a set of high-level languages such as Fortran and Cobol. The first programming paradigm is called “imperative programming” paradigm.

As abstraction level increased, the programming languages became more abstract and started to evolve in their own way. New programming languages required new pro-

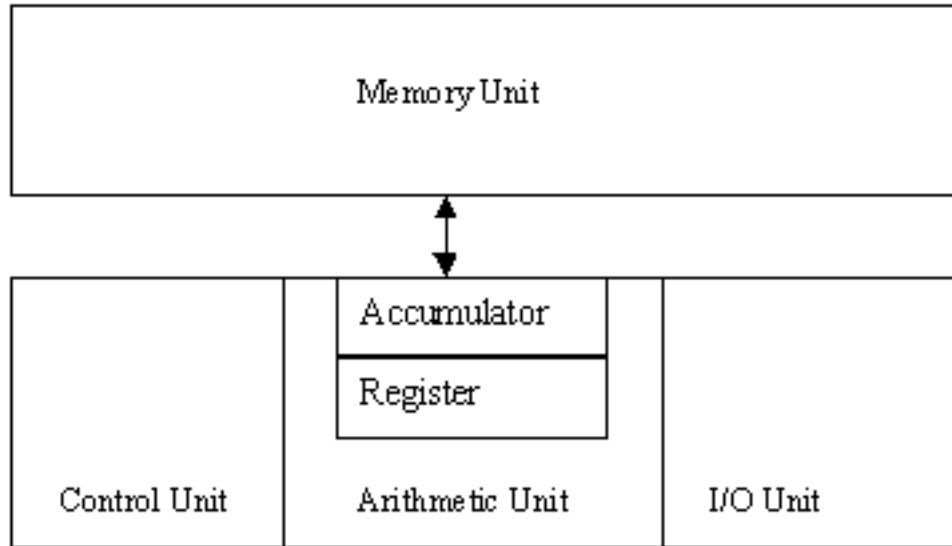


Figure 2-1: Conceptual view of a stored program computer. It consists of a memory unit and a CPU. A CPU further comprises a control unit, arithmetic unit and I/O unit.

programming paradigms. The first of such paradigms is “structured programming” and the second is “object-oriented programming.” There are other important programming paradigms such as “functional programming” and “logic programming.” They are outside the scope of our discussion. Therefore we concentrate on the paradigms built on the model of the stored program computer.

2.1 Assembly Programming

Assembly programming is a way to control the computer as a bare machine. The way stored program computers are modeled is based on the Turing machine. The Turing machine is one of the most fascinating creations of human imagination, and is the basis of the design of all modern computers. When Alan Turing developed the machine on paper, it consisted of memory (a tape) and instructions (a set of

transition functions). The reason he constructed a machine by using the two separate parts, a memory and a set of instructions, was that it was the most natural way for him [Hodges 00]. The state transition machine (Turing machine) is the model for assembly languages. The reason why people designed a computer language that is based on the state transition machine is that it is a natural paradigm for them.

Thus we can say the concept of assembly languages represents the first programming paradigm. This paradigm separates the segments of data from the segments of instructions. Figure 2-2 shows a program written in the SPARC assembly language [Paul 93][Taylor 01]. All the data are set in the top part of the program and that collection of data represents the state of the machine. All the instructions that manipulate the state are written in the bottom part of the program. We believe that this is the natural way in which people first think about the manipulation of the state transition machine, that is the computer.

The programming literature does not usually consider this way of program construction as a programming paradigm, but we would like to say that this is the first paradigm used when people started programming. This fact may imply that this way of constructing programs was most intuitive for people.

2.2 Imperative Programming

The imperative programming paradigm is a refinement of the assembly language paradigm, and a further abstraction from physical computers. The core of digital computers is the concept of a modifiable store, and is materialized as variables and