

AN ABSTRACT OF THE DISSERTATION OF

Charles D. Knutson for the degree of Doctor of Philosophy in Computer Science
presented on January 9, 1998. Title: Pattern Systems and Methodologies for
Multiparadigm Analysis and Design.

Abstract approved: _____

Curtis R. Cook

In this research, we have captured, in pattern form, key elements of programming and design in four programming paradigms (imperative, object-oriented, functional and logical) as well as multiparadigm programming. These pattern sets have formed a foundation upon which we were able to build a deeper understanding of multiparadigm programming and design. For each paradigm, we identified sets of programming patterns. We then identified design patterns for those paradigms that already have design methodologies (imperative and object-oriented). For those that do not (functional and logical), we created design pattern sets that may yet play a seminal role in formal design methodologies for those paradigms. From the relationships between programming and design patterns, we were able to identify and record methodological patterns that provide generative mappings between programming patterns and design patterns. From the sets of programming patterns, we were able to derive a pattern set for multiparadigm programming. We were also able to perform a critical analysis of the multiparadigm programming language Leda using this pattern set. Finally, we were able to apply the methodological patterns to this multiparadigm programming pattern set to aid in our search for multiparadigm design patterns. We were also able to derive insight into multiparadigm design patterns by studying the pattern sets for each of the four paradigms studied. Armed with this rich pattern system, we then created and presented a new pattern-based methodology for multiparadigm design. Finally, we applied our methodology and our pattern sets to three common design problems. We found that this new methodology lent new insights into software design, and suggested the role that multiparadigm programming and design can play in many aspects of software creation.

© Copyright by Charles D. Knutson

January 9, 1998

All Rights Reserved

Pattern Systems and Methodologies for Multiparadigm Analysis and Design

by

Charles D. Knutson

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented January 9, 1998
Commencement June 1998

Doctor of Philosophy dissertation of Charles D. Knutson presented on January 9, 1998

Approved:

Major Professor, representing Computer Science

Head of Department of Computer Science

Dean of Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Charles D. Knutson, Author

ACKNOWLEDGEMENT

This work represents a collaborative effort between many people, whether directly associated with this research or not. First of all, I owe a tremendous debt of gratitude to my advisors, Dr. Curtis Cook and Dr. Tim Budd. They have spent many hours with me providing counsel, insight, and feedback as this work has progressed. They have become advisors and mentors to me in the fullest sense. Working with Dr. Cook and Dr. Budd as a researcher and as a student has provided many of the highlights of my doctoral experience at Oregon State University. To my other committee members, Dr. Paul Cull, Dr. Gregg Rothermel and Dr. Jim Lundy, I give my thanks and appreciation for their time, their support, and their encouragement.

I also wish to express my thanks to Dr. Evan Ivie and Dr. Scott Woodfield of Brigham Young University whose examples as educators and researchers planted some of the earliest seeds in my heart and mind more than a decade ago of what I could achieve in Computer Science. This research effort is, in some ways, a fruit of those early seeds which they helped nurture in me.

I wish to express my deep love and respect for my wife's parents, Dr. and Mrs. Don and Eleanore Kartchner. They have been a constant source of support, encouragement and inspiration to me throughout my education.

To my own parents, Allen and Marjorie Knutson, I owe some of the most profound debts that can be incurred in this life. Their support and encouragement have strengthened me in times of struggle. I love them with all my heart and give them the gift of knowing that I am finally done with school!

My deepest love and appreciation are to my wife, Alana, and our six beautiful children, Aubrey, Brooke, Brad, Christopher, Eric and Sarah. Alana has supported me in my studies almost our entire marriage, and has struggled and suffered in ways that only she and I understand. My children have suffered from my absence during most of their young years. To my wife and children I can finally say, "Dad's coming home!"

Finally, I wish to express my thanks to a loving Heavenly Father who has more than once lifted our family up, dusted us off, inspired us, strengthened us, guided us, and grew us into something none of us were when we started.

TABLE OF CONTENTS

	<u>Page</u>
1. Introduction	1
1.1 Problem Statement	1
1.1.1 Programming Paradigms	2
1.1.2 Imperative Programming	2
1.1.3 Object-Oriented Programming	3
1.1.4 Functional Programming	3
1.1.5 Logical Programming	4
1.1.6 Multiparadigm Programming	5
1.1.7 Programming Languages Evolve Over Time	5
1.1.8 Programming Languages Rise with Design Methodologies	7
1.1.9 Multiparadigm Languages and Design	8
1.2 Language and Design Foundations	9
1.3 Design Patterns	10
1.4 Design Methodology	12
1.5 Case Studies	14
1.6 Summary	14
2. Imperative Programming Patterns	16
3. Imperative Analysis and Design Patterns	38
4. Object-Oriented Programming Patterns	59
5. Object-Oriented Analysis and Design Patterns	73
6. Functional Programming Patterns	93
7. Functional Analysis and Design Patterns	112
8. Logical Programming Patterns	126
9. Logical Analysis and Design Patterns	141
10. Methodological Patterns	150

TABLE OF CONTENTS, Continued

	<u>Page</u>
11. Multiparadigm Programming Patterns	163
11.1 Introduction	163
11.2 Adoptive versus Hybrid Combination.....	163
11.3 Programmatic Equivalence Classes and Inherent Commonality.....	164
11.4 “Multiple Paradigm” versus “Multiparadigm”	165
11.5 An Analysis of Leda Using Multiparadigm Programming Patterns.....	196
12. Multiparadigm Analysis and Design Patterns	200
12.1 Introduction	200
12.2 Creating a Pattern Set for Multiparadigm Design	201
12.3 Results of Top-to-Bottom Evaluation	202
12.4 Results of Left-to-Right Evaluation	203
13. Applying Multiparadigm Analysis and Design Patterns	218
13.1 Introduction	218
13.2 Patterns and Design Methodologies	218
13.3 A Pattern Based Methodology for Multiparadigm Design.....	223
13.4 Case Study: Multiparadigm Compiler Design	225
13.4.1 Determine Overall System Architecture	228
13.4.2 Make Specific Design Decisions.....	232
13.4.3 Multiparadigm Compiler Design Summary	244
13.5 Case Study: Multiparadigm Database Design.....	246
13.5.1 Determine Overall System Architecture	248
13.5.2 Make Specific Design Decisions: Overall System Architecture	249
13.5.3 Multiparadigm Database Overall Design Summary.....	253
13.5.4 Determine Database Engine System Architecture.....	254

TABLE OF CONTENTS, Continued

	<u>Page</u>
13.5.5 Make Specific Database Engine Design Decisions.....	256
13.5.6 Multiparadigm Database Engine Design Summary	260
13.6 Case Study: Multiparadigm Network Protocol Design.....	262
13.6.1 Determine Overall System Architecture	262
13.6.2 Make Specific Design Decisions.....	265
13.6.3 Multiparadigm Network Protocol Design Summary.....	269
14. Conclusions	271
14.1 Summary	271
14.2 Contributions.....	272
14.3 Conclusion.....	273
14.4 Future Research.....	274
14.4.1 Programmer Cognition	274
14.4.2 Patterns	275
14.4.3 Patterns as Pedagogy	276
14.4.4 Multiparadigm Programming	276
14.4.5 Software Design	277
References	279

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
2.1	Hierarchy of imperative programming patterns.	16
2.2	<i>Function</i> in Pascal.	19
2.3	<i>Procedure</i> in Pascal.	20
2.4	<i>Function</i> in C.	20
2.5	<i>Function</i> used procedurally in C.	20
2.6	<i>Data Definitions</i> in Pascal.	24
2.7	<i>Data Definitions</i> in C.	25
2.8	<i>Data Declarations</i> in Pascal.	26
2.9	<i>Data Declarations</i> in C.	26
2.10	IF / THEN / ELSE in Pascal.	28
2.11	Switch (CASE) in C.	29
2.12	WHILE in Pascal.	29
2.13	REPEAT in Pascal.	29
2.14	FOR in C.	30
2.15	GOTO in Pascal.	30
2.16	BREAK in C.	30
2.17	Assignment in Pascal.	32
2.18	Dynamic data creation and deletion in C.	32
2.19	Unary increment operators in C.	33
2.20	Multiplicative operators in Pascal.	34
2.21	Additive operators in Pascal.	34
2.22	Relational operators in Pascal.	34
2.23	Equality operators in C.	35
2.24	Shift operators in C.	35
2.25	Bitwise operators in C.	35
2.26	Common I/O in C.	37
3.1	Hierarchy of imperative analysis and design patterns.	39
3.2	Steps in making a pizza.	41
3.3	<i>Data Flow Diagram</i> for baking a cake.	44
3.4	An input flow.	46
3.5	An output Flow.	46
3.6	A dialog flow.	47
3.7	A diverging data flow.	47
3.8	A DFD with control flows and control processes.	49
3.9	<i>Data Dictionary</i> for a keyed telephone number.	52
3.10	Example Process <i>Specification</i> using Structured English.	54
3.11	A typical <i>Entity/Relation Diagram</i>	56
3.12	A typical <i>State Transition Diagram</i>	58
4.1	Hierarchy of object-oriented programming patterns.	60
4.2	<i>Inheritance</i> using OSA notation.	67
4.3	An example of <i>Composition</i> using OSA notation.	68
4.4	<i>Message Passing</i> in C++.	70

LIST OF FIGURES, Continued

<u>Figure</u>		<u>Page</u>
4.5	An example in C that uses ad hoc <i>Polymorphism</i>	72
4.6	Parametric <i>Polymorphism</i> in Leda.	72
5.1	Hierarchy of object-oriented analysis and design patterns.	74
5.2	An example of a <i>Static Declarative Model</i> involving multiple inheritance.....	77
5.3	An example of generalization-specialization.	79
5.4	An example of aggregation.	80
5.5	An example of <i>General Constraints</i>	81
5.6	An example of a state net.	83
5.7	<i>Timing Constraint</i> applied to the duration of a state.	86
5.8	Interaction diagram with an intermediate repository.....	87
5.9	Modeling intraobject concurrency using multiple subsequent states.	90
6.1	Hierarchy of Functional Programming Patterns.	94
6.2	Definition and use of a LISP <i>Function</i> to increment by three.....	97
6.3	An APL <i>Function</i> to perform factorials.	97
6.4	A Haskell <i>Function</i> to calculate the area of a circle.....	98
6.5	A <i>Higher Order Function</i> in Haskell.	99
6.6	The map function in Haskell.	100
6.7	The filter function in Haskell.....	101
6.8	The fold function in Haskell.	101
6.9	Explicit typing of functions in Haskell.....	103
6.10	The declaration and initialization of a data structure in LISP.	104
6.11	Creating <i>Lists</i> in LISP.	105
6.12	Creating and manipulating <i>Lists</i> in Haskell.	105
6.13	The <i>Polymorphic Type</i> length function in Haskell.	107
6.14	An infinite series defined in Haskell.	108
6.15	<i>Input/Output</i> using streams in Haskell.	110
6.16	<i>Input/Output</i> using monads in Haskell.....	111
7.1	Hierarchy of functional analysis and design patterns.....	114
7.2	Pipes and Filters used in a UNIX command-line environment.....	123
8.1	Hierarchy of Logical Programming Patterns.....	127
8.2	Known <i>Facts</i> in a Prolog program.	130
8.3	Known <i>Facts</i> in an SQL program.	130
8.4	<i>Inference Rules</i> in Prolog.	132
8.5	Use of Inference Rules via the WHERE clause in SQL.	132
8.6	A Prolog program with <i>Queries</i> for sibling relationships.	134
8.7	A complex SQL <i>Query</i>	134
8.8	Building complex rules using <i>Relations</i> in Prolog.....	136
8.9	<i>Relations</i> in SQL.	136
8.10	The use of <i>Relations</i> in Leda.	137
8.11	Input and output variables to a relation in Prolog.	138
8.12	The use of cuts in Prolog.....	140
9.1	Hierarchy of Logical Analysis and Design Patterns.....	142

LIST OF FIGURES, Continued

<u>Figure</u>	<u>Page</u>
10.1 Hierarchy of Methodological Patterns.....	152
11.1 Hierarchy of Multiparadigm Programming Patterns.	167
11.2 Problem solution with data, operative units and access points.	174
11.3 An example of <i>Adoptive Combination</i>	176
11.4 Example of <i>Multiple Paradigm Programs in a Single System</i>	178
12.1 Organization of pattern sets created for this research.	200
12.2 Hierarchy of Multiparadigm Analysis and Design Patterns.	204
12.3 Multiparadigm programming patterns by level of abstraction.	205
12.4 Architectural patterns for multiparadigm programming.	207
12.5 Design patterns for multiparadigm programming.	208
12.6 Multiparadigm programming patterns by level of abstraction.	209
12.7 Analysis and design patterns for <i>Multiparadigm Program Architecture</i>	211
12.8 Analysis and design patterns for <i>Multiparadigm Data Design</i>	213
12.9 Analysis and design patterns for <i>Multiparadigm Operative Unit Design</i>	214
12.10 Design patterns for <i>Multiparadigm Access Point Design</i>	216
12.11 Design patterns for <i>Guiding Principles of Multiparadigm Design</i>	217
13.1 A multiparadigm design methodology.	224
13.2 Phases of a compiler.....	226
13.3 A Multiparadigm Compiler Model.	227
13.4 A high-level view of a client-server database system.	250
13.5 One view of a Database Engine.	254
13.6 A tuple composed of data and actions.....	255
13.7 A relation composed of tuples.....	255
13.8 A Database composed of relations.	256
13.9 The IrDA Protocol Stack.....	263
13.10 Two IrDA applications communicating.	264

LIST OF TABLES

<u>Table</u>		<u>Page</u>
10.1	Example of a comparison between programming and design pattern sets.....	151
10.2	Comparison of object-oriented programming and design pattern sets.....	154
11.1	Adoptive Combinations for <i>Multiple Paradigm Data Declaration</i>	179
11.2	Adoptive Combinations for <i>Multiple Paradigm Operative Unit</i>	182
11.3	Adoptive Combinations for <i>Multiple Paradigm Access Point</i>	186
11.4	Hybrid Combinations for <i>Multiparadigm Data Declarations</i>	191
11.5	Hybrid Combinations for <i>Multiparadigm Operative Units</i>	193
11.6	Hybrid Combinations for <i>Multiparadigm Access Points</i>	195
11.7	Analysis of the Multiparadigm Programming Language Leda.....	197
13.1	Paradigmatic summary of multiparadigm compiler designs.....	245
13.2	Paradigmatic summary of multiparadigm database system design.....	254
13.3	Paradigmatic summary of multiparadigm database engine design.....	261
13.4	Paradigmatic summary of multiparadigm design of network protocol.....	270

LIST OF PATTERNS

<u>Pattern</u>	<u>Page</u>
2.1 <i>Imperative Program</i>	17
2.2 <i>Procedure</i>	18
2.3 <i>Data Statement</i>	21
2.4 <i>Data Type</i>	22
2.5 <i>Data Definition</i>	23
2.6 <i>Data Declaration</i>	25
2.7 <i>Executable Statement</i>	26
2.8 <i>Control Flow Statement</i>	27
2.9 <i>Data Manipulation Statement</i>	31
2.10 <i>Expression</i>	32
2.11 <i>Input/Output Statement</i>	36
3.1 <i>Stepwise Refinement</i>	39
3.2 <i>Structured Analysis and Design</i>	41
3.3 <i>Data Flow Diagram</i>	43
3.4 <i>Data Flow</i>	44
3.5 <i>Control Flow</i>	48
3.6 <i>Data Dictionary</i>	50
3.7 <i>Process Specification</i>	52
3.8 <i>Entity/Relation Diagram</i>	55
3.9 <i>State Transition Diagram</i>	56
4.1 <i>Object-Oriented Program</i>	60
4.2 <i>Object Class</i>	61
4.3 <i>Data</i>	62
4.4 <i>Method</i>	63
4.5 <i>Forms of Reuse</i>	64
4.6 <i>Inheritance</i>	65
4.7 <i>Composition</i>	67
4.8 <i>Points of Interaction</i>	69
4.9 <i>Message Passing</i>	69
4.10 <i>Polymorphism</i>	71
5.1 <i>Object-Oriented Analysis and Design</i>	74
5.2 <i>Object Relationship Model</i>	76
5.3 <i>Objects</i>	77
5.4 <i>Relationships</i>	78
5.5 <i>Constraints</i>	80
5.6 <i>Object Behavior Model</i>	82
5.7 <i>States</i>	82
5.8 <i>Triggers and Transitions</i>	84
5.9 <i>Actions</i>	85
5.10 <i>Timing Constraints</i>	85
5.11 <i>Object Interaction Model</i>	86
5.12 <i>Object Distribution</i>	87

LIST OF PATTERNS, Continued

<u>Pattern</u>	<u>Page</u>
5.13 <i>Object Concurrency</i>	88
5.14 <i>Real-Time Objects</i>	90
6.1 <i>Functional Program</i>	94
6.2 <i>Function</i>	96
6.3 <i>Simple Function</i>	97
6.4 <i>Higher Order Function</i>	98
6.5 <i>Mapping, Filtering and Folding</i>	99
6.6 <i>Data Type and Declaration</i>	102
6.7 <i>List</i>	104
6.8 <i>Polymorphic Type</i>	106
6.9 <i>Infinite Type</i>	107
6.1 <i>Expression</i>	108
6.1 <i>Input/Output</i>	109
7.1 <i>Functional Analysis and Design</i>	114
7.2 <i>General Design Approaches</i>	115
7.3 <i>Adapting Similar Approaches</i>	115
7.4 <i>Stepwise Refinement</i>	116
7.5 <i>Functional Data Design</i>	117
7.6 <i>Recursive Refinement</i>	118
7.7 <i>Type Refinement</i>	119
7.8 <i>Functional Design Perspectives</i>	121
7.9 <i>Pipes and Filters</i>	122
7.1 <i>Functional Composition</i>	124
8.1 <i>Logical Program</i>	128
8.2 <i>Facts</i>	129
8.3 <i>Rules of Inference</i>	130
8.4 <i>Queries</i>	132
8.5 <i>Relations</i>	135
8.6 <i>Unification Function</i>	137
8.7 <i>Control Elements</i>	139
9.1 <i>Logical Analysis and Design</i>	142
9.2 <i>Data Refinement</i>	143
9.3 <i>Fact Refinement</i>	145
9.4 <i>Rule Refinement</i>	146
9.5 <i>Query Refinement</i>	146
9.6 <i>Think Non-Deterministically</i>	147
9.7 <i>Control Design</i>	149
10.1 <i>Expect Isomorphism</i>	152
10.2 <i>Expect Non-Isomorphism</i>	154
10.3 <i>Sources of Non-Isomorphism</i>	155
10.4 <i>High-Level Programming Patterns</i>	156
10.5 <i>Environmental Forces</i>	157

LIST OF PATTERNS, Continued

<u>Pattern</u>	<u>Page</u>
10.6 <i>Destinations of Non-Isomorphism</i>	158
10.7 <i>New Language Features</i>	158
10.8 <i>Stretching the Paradigm</i>	160
10.9 <i>System-Specific Language Elements</i>	161
11.1 <i>Multiparadigm Program</i>	167
11.2 <i>Inherent Commonality</i>	168
11.3 <i>Program</i>	169
11.4 <i>Data</i>	170
11.5 <i>Operative Unit</i>	171
11.6 <i>Access Point</i>	172
11.7 <i>Paradigmatic Interchangeability</i>	173
11.8 <i>Adoptive Combination</i>	174
11.9 <i>Multiple Paradigm Programs in a Single System</i>	176
11.10 <i>Multiple Paradigm Data Declaration</i>	178
11.11 <i>Multiple Paradigm Operative Unit</i>	181
11.12 <i>Multiple Paradigm Access Point</i>	184
11.13 <i>Hybrid Combination</i>	188
11.14 <i>Multiparadigm Data Declaration</i>	189
11.15 <i>Multiparadigm Operative Unit</i>	192
11.16 <i>Multiparadigm Access Point</i>	194
12.1 <i>Multiparadigm Design Levels</i>	204
12.2 <i>Multiparadigm Architecture</i>	206
12.3 <i>Multiparadigm Design</i>	207
12.4 <i>Multiparadigm Idiom</i>	208
12.5 <i>Multiparadigm Program Architecture</i>	209
12.6 <i>Multiparadigm Data Design</i>	212
12.7 <i>Multiparadigm Operative Unit Design</i>	213
12.8 <i>Multiparadigm Access Point Design</i>	215
12.9 <i>Guiding Principles of Multiparadigm Design</i>	216

DEDICATION

This dissertation is lovingly dedicated to my sister,

Cindy Lee Knutson (January 7, 1964 – January 18, 1994)

who didn't get to see me finish but who would have
definitely made a scene at commencement.

PREFACE

“Languages differ not only in how they build their sentences but also in how they break down nature to secure the elements to put in those sentences....Language and our thought grooves are inextricably interrelated, are, in a sense, one and the same.”

— Edward Sapir, Linguist [1921]

“The forms of a person’s thoughts are controlled by inexorable laws of pattern of which he is unconscious. These patterns are the unperceived intricate systematizations of his own language....And every language is a vast pattern-system, different from others, in which are culturally ordained the forms and categories by which the personality not only communicates, but also analyzes nature, notices or neglects types of relationship and phenomena, channels his reasoning, and builds the house of his consciousness.”

— Benjamin Whorf, Linguist [1956]

“Every society which is alive and whole, will have its own unique and distinct pattern language.”

— Christopher Alexander, Architect [1979]

Pattern Systems and Methodologies for Multiparadigm Analysis and Design

1.0 Introduction

1.1 Problem Statement

Programming languages are steadily evolving to include elements from different programming paradigms. The clearest example of this evolution is the imperative language C, which spawned the object-oriented language C++, which in turn evolved into Visual C++, a programming environment that integrates the C++ language with elements of visual programming. Popular programming languages (like C++) and paradigms (like object-orientation) need design methodologies that guide engineers to create effective solutions within a certain paradigmatic or linguistic mind set. The evolution of programming languages is moving increasingly toward multiparadigm languages like Leda [Budd 1995a], but no design methodologies exist for multiparadigm languages. This dissertation presents a multiparadigm design methodology based on patterns. Design patterns have been used in object-oriented program design as a method for analysis as well as design representation. Patterns can be applied to any aspect of design in any technical field, from cities to software, and so provide the necessary generality needed in a multiparadigm design methodology. This section lays the groundwork for this problem statement, and supports the need for this research.

The remainder of this introduction is as follows. Section 1.2 examines principles that form a foundation on which to build a discussion of multiparadigm design. Section 1.3 introduces pattern languages of program design, migrating design pattern principles from an object-oriented perspective to a multiparadigm perspective. Section 1.4 explores principles of multiparadigm design, and introduces the concept of a pattern-based design methodology. Section 1.5 discusses the need for results that validate some of the claims made by this multiparadigm design methodology. Section 1.6 provides a brief summary of this introduction. The remainder of this section provides additional useful background information.

1.1.1 Programming Paradigms

Four paradigms have dominated programming since its inception: imperative, object-oriented, functional, and logical.¹ Each of these paradigms encapsulates a particular view of programming and problem solving, and colors the world of the programmer who sees through the eyes of the paradigm. As Sapir [1921] pointed out, “Languages differ not only in how they build their sentences but also in how they break down nature to secure the elements to put in those sentences.” Not only are the syntactic things we are able to do impacted by the languages in which we program, but the actual task of breaking down a problem into a solution is strongly influenced by the choice of paradigm (or even the particular language within a given programming paradigm). This implies that there may be elegant solutions to problems hidden from our view because the paradigm in which we think does not accommodate that view. By examining these multiparadigm patterns, unique design aspects will be more readily visible, and these patterns will lend themselves to more elegant design, even in situations where the language in which the solution is coded is not (strictly speaking) a multiparadigm language.

To lay some foundation, the following sections discuss these four programming paradigms, and summarize their views of programming and problem solving.

1.1.2 Imperative Programming

The imperative programming paradigm is the one that most clearly emulates the von Neumann architecture, with sequenced execution of instructions, and changes to values stored in memory locations. It is the Betty Crocker approach to programming, with “recipe cards” that take the computer through the necessary steps of combining the right things in the right way in the right bowls. Not surprisingly, it is the paradigm that has dominated programming through most of the history of computing. Imperative design typically involves some form of stepwise refinement (usually top-down, but including bottom-up, side-to-side, and other incarnations), by which a problem is broken down into

¹ For an excellent taxonomy of programming paradigms, see [Ambler 1992].

the necessary steps required to solve it. Data structures are designed, and procedures and functions operate on them. The most popular imperative languages have been FORTRAN [ANSI 1966], COBOL [ANSI 1974], BASIC, Pascal [Wirth 1971a] [Koffman 1981] [Schneider 1981], and C [Kernighan 1978]. Other imperative languages include Forth [Winfield 1983] [McCabe 1983] and Icon [Griswold 1990].

1.1.3 Object-Oriented Programming

The object-oriented programming paradigm views systems as groups of interacting objects [Curtis 1995] [Fisher 1995] [Budd 1997]. Objects communicate by sending and receiving messages, which are requests for action bundled with whatever arguments necessary to complete the task. Each object has its own memory, which may consist of other objects. This view of a system as being composed of objects forces a different kind of problem decomposition than the imperative approach. Rather than refining the problem into smaller steps, we think of refining the system into smaller collaborating subsystems or objects. In addition, the definitions of objects are stored in classes, which have behaviors that can be shared through inheritance and composition. Object-oriented design is often centered around the design of class hierarchies (where object classes are declared) and instance hierarchies (where objects are instantiated). This approach largely subsumes the imperative paradigm, since the methods contained in classes typically operate in an imperative fashion on data encapsulated within the class. The most popular object-oriented languages have been Smalltalk [Ingalls 1978] [Budd 1987] [Kay 1993], C++ [Stroustrup 1991] [Eckel 1995], and (most recently) Java [Arnold 1996] [Flanagan 1996]. Other object-oriented languages include Modula-II [Wirth 1985] [King 1988], Ada [Whitaker 1996] [Dale 1994], Objective-C [Cox 1986] [Pinson 1991], and Object Pascal [Tesler 1985].

1.1.4 Functional Programming

The functional programming paradigm is built on the notion of functional composition [Henderson 1980] [Glaser 1984] [Backus 1978] [Peyton-Jones 1987] [Bird 1988] [Hudak 1989] [Hughes 1989] [MacLennan 1990] [Wadler 1992]. This is a view of

computation in which the ultimate solution is seen as the composition of all the functions that operate upon some value. For example, in an imperative view, if x is a variable with the value 5, and it needs two things done to it, we would perhaps view it in the following way:

$$x := 5; x := f(x); x := g(x);$$

At this point x holds our solution. But in a functional approach, we might represent the solution as: $g(f(x=5))$, where x becomes a handle for the value 5, and the functions $f()$ and $g()$ are applied to it in turn. Functional programming eschews the side-effects inherent in imperative programming. Values, once created, are not variables, but entities that cannot be changed (there is no concept of changing values at memory locations). Reuse is facilitated through the creation of functions which can, in turn, be applied to other functions. In this way, entire systems can be built up through functional composition. These functions are “first-class” data values, meaning that functions can be assigned to identifiers, passed as arguments, or returned as the result of executing other functions. The most popular functional languages have been APL [Iverson 1962] [Polivka 1975], LISP [Wilensky 1984], Scheme [Steele 1978] and ML [Wikstrom 1987] [Milner 1990] [Paulson 1991]. Other functional programming languages include Haskell [Hudak 1992] [Davie 1992] [Thompson 1996], and Hope [Bailey 1990].

1.1.5 Logical Programming

The logical programming paradigm is based upon propositional calculus which was originally used for formalized theorem proving [Hogger 1984] [Lloyd 1984]. In this system, there are three key parts: a set of axioms (or facts), a set of rules of inference, and a question or query. This approach is declarative, or nonprocedural. That is, the programmer provides a series of assertions of facts, a collection of rules of inference, and a query. The programmer need not specify how the query is to be answered using the information provided. Instead, an underlying searching mechanism works behind the scene to determine whether the facts can be deduced from the given information. The search mechanism typically uses a depth-first search strategy with backtracking to search the problem space for the answer to a query. The first and most popular logical language

is PROLOG [Clocksin 1981] [Cohen 1985] [Sterling 1986] [Ross 1989] [Colmerauer 1993]. Another language that can be loosely termed a logical programming language is the database query language SQL [Chamberlain 1976] [Celko 1995]. Also associated are constraint logic programming languages such as CLP [Cohen 1990].

1.1.6 Multiparadigm Programming

In a loose sense, any combination of programming paradigms can be viewed as “multiparadigm.” For example, logic programming with “cuts” to control infinite backtracking can be viewed as multiparadigm, combining declarative logic with control flow constraints. Similarly, functional programming with monads [Wadler 1990] can be viewed as multiparadigm, combining functional programming with input and output, the nature of which requires side effects.

The most common implementations of multiparadigm programming involve the linking together of separately compilable object modules, which interact through some common procedure call mechanism. This provides some of the benefit of multiparadigm programming, but does not exploit the advantages of hybrid solutions, where the paradigmatic elements are tightly coupled. Languages like Leda [Budd 1995a] attempt to bring these elements together in a single language, providing a synthesis of programming concepts. Interesting design patterns can be found in the hybrid solutions that are made possible by multiparadigm languages.

The following two premises help establish the need for a multiparadigm design methodology. First, programming languages evolve over time, incorporating aspects of non-native programming paradigms as part of their evolution. Second, programming languages gain popularity in lockstep with accompanying design methodologies.

1.1.7 Programming Languages Evolve Over Time

Language evolution, in this case, refers not only to changes within a single language, but changes in the nature of languages that emerge from a single paradigm. The most obvious case in point are the imperative languages, which have historically been used as the general purpose languages of choice. Looking at most recent history, Pascal

gained popularity as a teaching tool in the 80's, and some success as a commercial language, but was limited in its capabilities and viewed as lacking power and control. C emerged to gain popularity as a more serious developer tool largely because of the great freedom given to developers to manipulate structures directly, particularly pointers. This facilitated a number of functional programming techniques within C, including the manipulation of structures containing functions, and the returning of functions (or pointers to functions) as the result of a function call². The extensible nature of C via libraries of functions created even greater evolutionary freedom, and the language eventually became the lingua franca of the development community, and later, of the educational community.

Object-oriented programming gained popularity first through “pure” object-oriented languages, such as Smalltalk. Pascal and C were eventually refitted with object-oriented clothing, yielding languages such as Object Pascal, Objective-C and C++, among others. These languages embraced object-oriented capabilities, while retaining their imperative roots.³

As graphical user interfaces (GUIs) became more pervasive, it was natural for these languages to evolve further, to take advantage of this new way of seeing and manipulating things. The most notable languages to adopt graphic elements have been Visual BASIC⁴ [Microsoft 1993a] and Visual C++ [Microsoft 1993b]. While not strictly visual programming languages, these programming environments allow the developer to

² One of the fundamental principles of functional programming is that functions are first class citizens, and can be used wherever any other language element is used. The ability of C to pass and return function pointers is not functional programming in the strictest sense, but certainly pushes the borders in that direction in a way that Pascal never did.

³ Object-oriented purists would argue (like functional purists in the previous footnote) that these languages compromise important object-oriented features because of their evolutionary growth, which includes holding on to imperative characteristics inherited from their parent languages. Still, languages like C++ have become imminently popular in large part because of the evolutionary nature of their change, pushing the borders toward object-orientation.

⁴ So important has been this visual environment revolution that serious developers who wouldn't otherwise be caught dead programming in BASIC have embraced Visual BASIC because of the tools available for building the GUI interface pieces that were so extremely tedious and error-prone in C and C++.

create visually those objects that exist visually in the final application, such as windows, dialog boxes, etc., without having to write the code to create or modify them.

There is a natural evolution in programming languages to be more paradigmatically inclusive over time. We believe this trend will continue, and that languages will appear increasingly multiparadigmatic in their nature. There may come a point when the next programming revolution (some time after Java?) will be a language, like Leda, that incorporates aspects from several paradigms in a seamless and powerful way.

1.1.8 Programming Languages Rise with Design Methodologies

It is difficult (if not impossible) to draw a cause and effect relationship between the popularity of languages and the existence of design methodologies, but it seems apparent (via a casual glance at the computer shelves at the bookstore) that the most popular programming languages are accompanied by the most books on program analysis and design within that paradigm and specific to that language. The inherent features of a language will contribute somewhat to its popularity, but at some point a reasonable design methodology must arise to show developers how to make the best use of the language capabilities, and to take the language to its next level of popularity. There seems to exist a cyclic relationship between these two factors.

When a language gains enough popularity, market forces dictate that design books based on the language will sell enough copies to make a profit, so authors rush in to provide. Users and developers, when investigating a new programming language will sometimes be swayed to a particular language because of its apparent popularity (indicated in part by the plethora of books about it) and because of the availability of help (a reasonable design methodology) in using the language effectively. The capstone of every viable general purpose programming language (or set of related languages) is a design methodology. The most popular imperative design methodologies have been the Yourdon [1989] and Jackson [1983] approaches to structured analysis and design. Imperative design and analysis is further explored in [Wirth 1971b] [Alagic 1978] [Birrell 1985] [Bishop 1986] [Liskov 1986] [Hughes 1987] [Bailey 1989] [Budgen 1994]. The

most popular object-oriented design methodologies have been the Object Modeling Technique [Rumbaugh 1991], Booch [1994a], Coad-Yourdon [Coad 1991], Shlaer-Mellor [Shlaer 1988] [Shlaer 1992], and OSA [Embley 1992]. Object-oriented design and analysis is further explored in [Halbert 1987] [Wirfs-Brock 1990a] [Wirfs-Brock 1990b] [Graham 1994] [Tepfenhart 1997].

1.1.9 Multiparadigm Languages and Design

General purpose programming languages naturally evolve and become more multiparadigmatic over time. But in addition to this evolutionary change, there is growing interest in multiparadigm programming languages (as a revolutionary change), and a modest but expanding body of research on the subject.⁵ The most common and fundamental research efforts deal with integrating two (sometimes three) programming paradigms into a single language solution, often to solve part of a specific problem domain [Hailpern 1986] [Zave 1989] [Placer 1991b]. For example, research efforts have combined object-oriented and functional [Keene 1989] [Gabriel 1991] [Kuhne 1994]; imperative and logical [Budd 1991] [Delrieux 1991]; imperative and functional [Peyton-Jones 1993]; object-oriented and logical [Brogi 1994]; imperative and object-oriented [Zettel 1995]; and logical with constraint [Loia 1993] [Xu 1995]. Specific efforts have been made in the domain of database design [Catarci 1996] [Embley 1996]. More elaborate research efforts attempt to combine up to four programming paradigms into general purpose languages that can be used to solve problems in a wide variety of domains [Wells 1989] [Placer 1991a] [Budd 1995a] [Ng 1995] [Callegarin 1996]. Research has explored the potential effectiveness of multiparadigm programming languages for solving complex problems [Justice 1994] [Budd 1995b] [Knutson 1997a]. Efforts have also focused on the use of multiparadigm programming languages as pedagogical tools [Placer 1993] [Hartel 1995] [Robertson 1995] [Brilliant 1996] [Leska 1996].

⁵ Our literature search shows no applicable references prior to 1983, and an average of two references per year from 1983 to 1988. From 1989 to 1995, that average climbs to around 10. The word “multiparadigm” only appeared as a subject in popular computer science indices in the last six years.

Given the general trend of programming languages to evolve in multiparadigm ways, programming in the future will increasingly require an understanding of different programming paradigms, and programming languages will increasingly accommodate those perspectives. The capstone of these multiparadigm languages will be multiparadigm design methodologies.

Multiparadigm design has not been fully exploited or studied, but some work exists that clearly points to the need for further research [Winograd 1979] [Zave 1983] [Luker 1989] [NASA 1993] [Agarwal 1995] [Lopez 1996]. This dissertation will propose a multiparadigm design methodology that uses design patterns as a method for analysis as well as a mechanism for design representation.

1.2 Language and Design Foundations

This dissertation will explore the principles underlying programming languages, paradigms, and design methodologies. Each of these areas of study is made up of essential elements that can be captured in patterns. Manipulation of these essential elements is foundational to designing within these paradigms, and is critical to the development of a single multiparadigm design methodology that combines elements from disparate paradigms.

Through a study of language elements, paradigmatic patterns will suggest mechanisms for modeling and representing design concepts specific to particular paradigms. A survey of design methodologies will suggest mechanisms that have been used to represent programs in various paradigms and languages.

We will establish foundations for our study of multiparadigm design by first completing the following:

- Survey and analysis of programming language elements and paradigmatic patterns, suggesting ways to represent those ideas from a design perspective. These programming pattern sets comprise Chapters 2 (imperative), 4 (object-oriented), 6 (functional) and 8 (logical).
- Survey and analysis of design methodologies for each programming paradigm. This may require creating or enhancing methodologies for paradigms that are

extremely weak on design methodologies (such as functional and logical). These design analysis pattern sets comprise Chapters 3 (imperative), 5 (object-oriented), 7 (functional) and 9 (logical).

- Survey and analysis of multiparadigm languages (with a focus on Leda), identifying the programming elements that are represented, as well as those not represented. This pattern set is presented in Chapter 11.
- Analysis of Leda relative to the comprehensive list of programming and paradigm elements. This analysis is contained in Chapter 11.

1.3 Design Patterns

A pattern language for architectural design was originally proposed almost 20 years ago by Christopher Alexander in *The Timeless Way of Building* [1979], and then expanded upon in *A Pattern Language* [1977]. These volumes encapsulate essential architectural elements into patterns, the collective fabric of patterns forming a pattern language. As proposed by Alexander, patterns capture elements of cities and buildings, and provide a mechanism for documenting the key principles that underlie the use of the patterns. Patterns also describe their interactions with other patterns, and provide samples of pattern instantiations. But the pattern form is not in any way tied specifically to architecture. Individuals in the object-oriented design community have recently seen the usefulness of patterns for software.

The use of patterns as a software design tool began in 1991 with Erich Gamma's doctoral thesis [Gamma 1991].⁶ Gamma saw that the pattern concept could be extended as an analysis and documentation tool for object-oriented software design, and that pattern languages could exist for programs. In the six years since Gamma's original work a relatively large body of research on software design patterns has been published, almost exclusively focused on patterns for object-oriented design. Key among these recent works has been *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (now referred to as

⁶ Gamma is typically given credit for first drawing parallels to Alexander's work, but others were also working on similar concepts at the time. For a fascinating first-hand history of how the gang of four came together, see [Coplien 1996a].

the gang of four) [Gamma 1994], and the first attempt at a pattern-based object-oriented design methodology by [Pree 1994] [Pree 1997]. Other significant and notable contributors to this growing field have been [Beck 1994a] [Beck 1994b], [Coad 1992a] [Coad 1992b] [Coad 1995], [Gamma 1993], [Gabriel 1996], [Johnson 1992] [Johnson 1994], Lea [1994], [Coplien 1995a] [Coplien 1997], [Schmidt 1995] [Schmidt 1996], and [Vlissides 1997]. There is also an annual conference, Pattern Languages of Programming (PLoP), dedicated to programming patterns, and currently in its fourth year [Coplien 1995b] [Vlissides 1996] [Martin 1997].

Programming patterns capture some of the aspects of literate programming [Knuth 1984] with object-oriented design, yielding a sort of “literate design,” in which the designer is free to philosophize and elaborate about the uses of certain design elements, and to discuss its potential interactions with other elements. The pattern approach to design is not limited to the object-oriented perspective, even though nearly all the work on patterns of software design has come from that community. For example, patterns could be used to capture essential elements of imperative languages and structural design. The same can be said of any of the other programming paradigms in use. This makes the pattern approach an ideal foundation for investigating multiparadigm design. The same method can be used to design and document patterns from each of the programming paradigms, as well as the complex multiparadigm patterns that involve two or more programming elements from disparate paradigms blending in some way. Patterns can be used to capture the essential elements of each of the programming languages and paradigms investigated in section 1.2, and then can be used to describe the higher-level combinations of these elements, and the frameworks in which language elements operate together.

The use of pattern sets as a literary form is somewhat different from traditional forms of publication. The main focus of the pattern set is to present its patterns. These patterns should, in theory, stand alone (or together as a pattern set or pattern language) and should need very little other supportive material. Therefore, patterns are typically presented with little extraneous text. In traditional design pattern publications, a pattern

set is introduced briefly, and then patterns are presented—there is no conclusion or summary following the patterns.

The patterns presented in this work follow this form, but are shorter than typical patterns. Their relative brevity is intended as a compromise between full patterns (which may take up to dozens of pages) and thumbnail sketches (which typically consist of several sentences). By presenting patterns in one or two pages, we give enough information to enable the reader to understand the pattern, but we acknowledge that these patterns are, per force, somewhat incomplete. Each of Chapters 2 through 12 contains a pattern set preceded by a brief introduction. In keeping with the spirit of patterns, none of these chapters is graced with a summary or conclusion. In addition, each of the patterns in this work is typographically distinct from the rest of the text. All pattern names are italicized within the pattern in which they are introduced, or where otherwise appropriate. All of the patterns presented here follow a similar organizational template consisting of a pattern name, aliases, motivation, discussion, examples, and related patterns. Fuller patterns typically include other sections, most notably context and forces. For a superb discussion of the pattern form, see [Mezsaros 1996].

We will use design patterns in our effort to analyze multiparadigm analysis and design in the following ways:

- Exploring the patterns that emerge from an analysis of the relationships between programming and design patterns. This pattern set comprises Chapter 10.
- Exploring multiparadigm patterns of design, influencing the work done in section 1.2, with particular emphasis on patterns of thought that emerge relative to problem solving in each paradigm. This effort contributes to Chapter 12.
- Creating a set of multiparadigm design patterns that can be used as examples in applying the proposed multiparadigm design methodology. This pattern set is included in Chapter 12.

1.4 Design Methodology

The production of a pattern-based multiparadigm design methodology will necessarily build on the analysis of language elements, paradigm elements and design

methodologies proposed in section 1.2 and the multiparadigm pattern language proposed in section 1.3. But creating such a methodology involves more than simply picking and choosing among design elements and patterns and combining them in interesting ways.

A design methodology involves decisions that progressively take a problem solution from vague and high-level to specific and low level. Some design decisions leave options open, while other decisions limit future options. One of the challenges with multiparadigm design is that elements from different paradigms sometimes directly conflict. That is, a design decision at one level may severely limit the options available at lower levels. An adequate multiparadigm design methodology must deal with these levels of compatibility.

Another complicating factor concerns the merging of paradigmatic elements. We can represent the design process as a decision tree, with decision points at each node. At any decision point, an appropriate solution may involve any one of the patterns from any of the supported paradigms. However, a solution might also require hybrid elements that combine two or more multiparadigm attributes into a single pattern. The potential complexity of those decisions will be driven by the number of elements that can be potentially combined from among the paradigms, and may involve from two to four paradigm elements in a single pattern. The proposed multiparadigm design methodology must address the complexity inherent in multiparadigm design decisions, and present reasonable methods for achieving good design despite the complexity.

Our effort to create a pattern-based methodology for multiparadigm design will include the following:

- Creating an effective model for representing the interactions between disparate paradigmatic elements in both loosely coupled (adoptive) and tightly coupled (hybrid) ways. This effort is included in Chapter 11.
- Synthesizing aspects of applicable design methodologies into a framework for the proposed multiparadigm design methodology. This effort is included in Chapter 12.
- Proposing a unified pattern-based multiparadigm design methodology. This methodology is presented in Chapter 13.

1.5 Case Studies

Once the proposed multiparadigm design methodology has been created, this dissertation should produce some demonstrable evidence of its merit. It is not our expectation that the proposed methodology will necessarily be easier to use, faster to design with, or easier to comprehend than structural or object-oriented design methodologies. Rather, its intention is to make available a methodology for thinking, designing, and rendering effective programs in a multiparadigm fashion. Its primary goal is to make accessible another school of thought that currently lies largely unexploited, and in doing so suggest new solutions to existing problems.

We will provide three case studies in which the proposed design methodology is employed, showing in particular those aspects of the design solution that could not have been achieved without a multiparadigm design methodology. The case studies will include compiler and relational database construction. Preliminary studies in these two areas have been done in [Justice 1994] and [Knutson 1997a].

We will demonstrate the viability of our methodology in the following way:

- Identifying three real-world design problems for case studies using the proposed multiparadigm design methodology. The examples we have chosen are the design of a compiler, a database, and a network protocol. These case studies are presented in Chapter 13.

1.6 Summary

Many of the great breakthroughs in computer science (and in almost any science, for that matter) have occurred when individuals conceptualized typical problems in atypical ways. When a new problem demands new solutions, the thought process itself may be the key deliverable. As an example, when C programmers begin writing object-oriented C++ programs, it is generally harder, slower, and more incomprehensible than writing C programs. But there are problems that lend themselves to object-orientation, and so imperative programmers endure the learning curve to obtain the thought process, and it later becomes natural to them. When C++ programmers begin working with visual environments like Microsoft Visual C++, a similar learning curve is introduced as they

come to understand the tools and the environment. But the environment allows them to create visual elements in a visual fashion, instead of painstakingly by hand, as in the past. So the learning curve is the price of admission, and is paid by many programmers for the benefit of a new way of thinking about problems. Multiparadigm languages are becoming more of a reality, and an effective multiparadigm design methodology is quickly becoming a necessity.

2.0 Imperative Programming Patterns

Imperative programming has been the perennial default paradigm since the inception of modern computing. It is what most people mean when they say, “computer programming.” Although this is changing with the advent of languages from new paradigms, the foundations of computation are so firmly rooted in the imperative paradigm (and the von Neumann architecture) that it remains an important perspective to explore in any thorough examination of programming languages.

In this chapter, we have attempted to capture the essential elements of imperative programming in pattern form. This pattern set will be used as a building block to analyze aspects of imperative design, and to explore multiparadigm programming and design. We do not expect this pattern set to be exhaustive in revealing everything that could be said about imperative programming (although it could provide an excellent foundation on which to build such an analysis).

This taxonomy of imperative programming elements was culled from a thorough analysis of programming elements in selected imperative languages including Pascal, C, and assembly. The essential programmatic elements were identified, and are listed in the pattern hierarchy below (Figure 2.1). Each of these elements is discussed in greater detail in the patterns that follow.

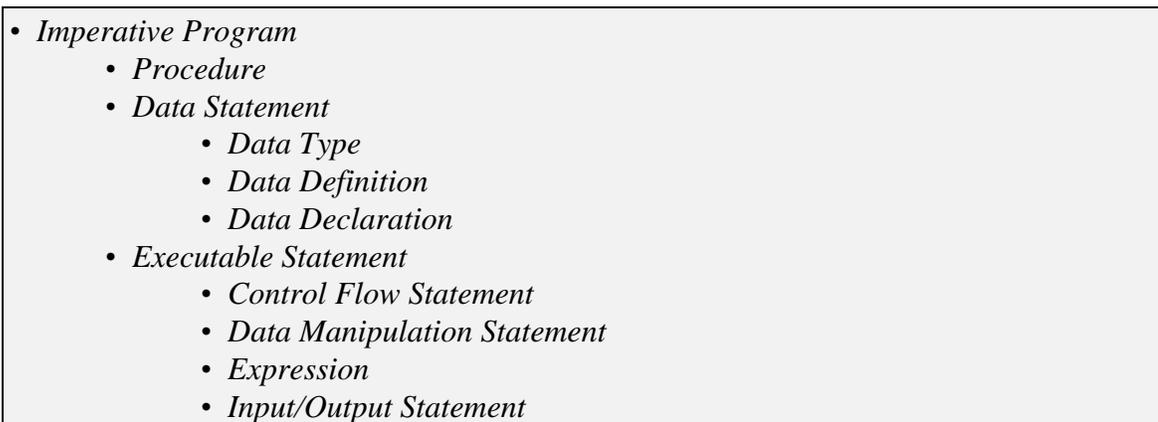


Figure 2.1. Hierarchy of imperative programming patterns.

Pattern 2.1 Imperative Program

Aliases: *Procedural Program, von Neumann Program*

Motivation:

Imperative Programming is closely tied to the von Neumann architecture, in which a computer is conceptualized as consisting of a number of memory locations, into which data and code can be placed. An *Imperative Program* is this collection of data and code in the memory locations of a von Neumann machine.

Discussion:

The code portion of a program is executed through a fetch-decode-execute cycle in which the main processor examines a single instruction at a time (indexed by a program counter), decodes it, and executes it. Control passes to the next sequential instruction by default through an incrementing of the program counter. An instruction may alter the main processor's program counter, resulting in the fetching of an instruction out of sequence. The data portion of a program can be changed by the execution of a program's code. The result of a program can be viewed as the cumulative result of side effects through which program data is altered until it holds the final state or result of the program's execution.

A useful metaphor for *Imperative Programming* is to view it as the "Betty Crocker" approach to programming, with "recipe cards" that take the computer through the necessary sequential steps of combining the right things in the right way in the right order in the right bowls. Side effects are a natural part of Betty Crocker computing since ingredients, once mixed, become part of the whole, and cannot be separated. *Imperative Programming* uses memory locations as bowls and combines data within them in such a way that they can't be reliably separated back into their original conditions. The result of an *Imperative Program* is a program state consisting of a set of data locations, just as the result of a Betty Crocker recipe is a food dish resulting from the preparation of ingredients in accordance with the recipe.

Examples:

The most popular imperative programming languages have been Assembly, FORTRAN [ANSI 1966], COBOL [ANSI 1974], BASIC, Pascal [Wirth 1971], and C [Kernighan 1978].

Lower-level Patterns: *Procedure, Data Statement, Executable Statement*

Other Related Patterns: *Object-Oriented Program, Functional Program, Logical Program, Multiparadigm Program*

Pattern 2.2 Procedure

Aliases: *Module, Function, Subroutine*

Motivation:

Any imperative program can be viewed as a collection of one or more *Procedures*. The main thread of execution in any program can be viewed as the highest level *Procedure*, even when no explicit *Procedure* declaration, such as the function `main()` in C, is required. Modularization happens when a program is organized in separate executable chunks (referred to by different names in various imperative programming languages).

Discussion:

As long as programs remain extremely small, no extensive modularization is required for a programmer to read, understand and maintain a piece of code. But even at a relatively small size, imperative code becomes unwieldy without some mechanism for modularization. *Procedures* provide the normal way for imperative programs to break up programs, encourage reuse, and increase maintainability.

Procedures may be used to encapsulate both data and code through the declaration of local variables and through language-specific scoping rules. However, they do not enforce encapsulation of data and the accompanying hiding of information. This leaves imperative programmers free to access global variables (or to do other heinous things like

creating self-modifying code). The peril of unwanted side effects is one of the most serious problems with imperative programming, despite the amount of modularization and encapsulation provided by *Procedures*. Many programming paradigms that followed the imperative were focused on stricter enforcement of encapsulation and information hiding in the modularization of programs.

Almost all imperative languages support modularization, since maintaining large programs is almost impossible without some form of procedural breakdown. However, the various incarnations of *Procedures* are not necessarily synonymous. For example, procedures in Pascal involve control flow and parameter passing only, while functions return values and can be evaluated in expressions. C supports only functions, which can be used either as procedures that return values, or as members of expressions.

The use of *Procedures* in imperative programs encourages reuse through standard function libraries (such as those supported by C) and standard built-in procedures and functions.

Examples:

The examples in this section (Figures 2.2 through 2.5) all involve a simple *Procedure* that converts a Celsius temperature to the corresponding Fahrenheit temperature. Each shows a different approach to *Procedures* in C and Pascal.

```
(* DECLARING THE FUNCTION *)
FUNCTION F_TEMP (C_TEMP : REAL) : REAL;
  BEGIN
    F_TEMP := 1.8 * C_TEMP + 32.0;
  END

(* USING THE FUNCTION *)
FAREN := F_TEMP (X);
```

Figure 2.2. Function in Pascal.

```

(* DECLARING THE PROCEDURE *)
PROCEDURE TEMP_CONVERT (C_TEMP : REAL; VAR F_TEMP : REAL);
    F_TEMP := 1.8 * C_TEMP + 32.0;
BEGIN

(* USING THE PROCEDURE *)
TEMP_CONVERT (X, FAREN);

```

Figure 2.3. *Procedure* in Pascal.

```

/* Declaring the function */
float temp_convert(float c_temp)
{
    return (1.8 * c_temp + 32);
}

/* Using the function */
faren = temp_convert (x);

```

Figure 2.4. *Function* in C.

```

/* Declaring the function */
void temp_convert(float c_temp, float *f_temp)
{
    *f_temp = 1.8 * c_temp + 32;
    return;
}

/* Using the function */
temp_convert (x, &faren);

```

Figure 2.5. *Function* used procedurally in C.

Higher-level Patterns: *Imperative Program*

Other Related Patterns: *Data Statement, Executable Statement*

Pattern 2.3 Data Statement**Motivation:**

Since imperative programs manipulate (or operate on) data, an imperative programming language must have the ability to deal with predefined data types, define its own data types as needed, and instantiate these data types in variables that can be manipulated by program code. *Data Statements* are the programmatic elements that permit the programmer to define (*Data Definition*) and declare (*Data Declaration*) data. The user must be able to use predefined data types provided by the language as building blocks for user-defined data types (*Data Type*).

Discussion:

All imperative languages support a set of fundamental built-in data types. Most also allow the user to construct not only new variables, but new data types as well. These user-defined data types can, in turn, be used in other data statements.

There are two basic kinds of data statements: *Data Definitions* are programming elements that allow the programmer to create new data types. *Data Declarations* are programming elements that allow the programmer to instantiate variables to hold data of particular types.

Most imperative languages are strongly typed, so *Data Statements* play an important role in programming. This means that variables must be declared to be of a certain type, allowing the compiler to correctly manipulate data during execution. Some languages (like C) support some ad hoc polymorphism, in which compatible data elements of different types are allowed to interact. For example, if a short integer is added to a long integer, the result will be a long integer by default, since a long integer can fully represent a short integer, but not vice versa. If a long integer is copied into a short integer, the high half of the long integer will be ignored.

Higher-level Patterns: *Imperative Program*

Lower-level Patterns: *Data Type, Data Definition, Data Declaration*

Other Related Patterns: *Procedure, Executable Statement*

Pattern 2.4 Data Type

Motivation:

At a fundamental level, all data manipulated by an imperative program consists of bits and bytes. However, it is convenient from a design perspective to group and manipulate bytes in cohesive and conceptually consistent ways. There is also a need to group simple *Data Types* in ways that allow for larger, more complex data structures to be manipulated. To be of use, these *Data Types* must be instantiated within the context of a programming language (see *Data Statement*) but are dealt with in this pattern only at a conceptual level.

Discussion:

Mathematics lends to computing certain fundamental data concepts such as integers, real numbers, boolean values, positive and negative numbers, etc. These concepts must be implemented in some way for programming to be able to solve real problems.

There is general agreement at a conceptual level of what constitutes data, but instantiated *Data Types* vary in implementation depending on the machine for which they are implemented. For example, the integer data type in C cannot be guaranteed to be of a particular size or number of bytes, but is dependent on the machine for which a program is compiled and running.

The most common simple *Data Types* are integer, character, real and boolean. More sophisticated *Data Types* include variations on these types, such as signed and unsigned integers, short and long integers. Other types, such as floating point, may not be universally supported, sometimes requiring special hardware.

Certain complex *Data Types*, such as arrays and records (or structures) may form a framework in which other data may be utilized. For example, arrays of characters are

commonly referred to as strings, but arrays can also hold integers. Arrays can also be filled with arrays (yielding multi-dimensional arrays).

Slightly more complex are those *Data Types* that can be built from less sophisticated *Data Types*, but which carry a simple model with particular design ramifications. For example, arrays can be manipulated in such a way that they might be referred to as strings, stacks, or queues, even though the *Data Type* (strictly speaking) is still an array. The introduction of this higher-level model can aid design.

The most sophisticated *Data Types* require a more elaborate model to represent them, and typically depend on some kind of dynamic memory model and pointer to dynamic structures. Linked lists are one of the most common building blocks for this kind of *Data Type*, and can be used to construct various kinds of tree structures. Linked lists can also be used to represent less complex items such as sets, stacks and queues when dynamic data manipulation is required or preferable.

Each conceptual *Data Type* discussed in this pattern must be instantiated via some *Data Definition* and/or *Data Declaration* to be of use to a programmer, but the *Data Types* form an important conceptual foundation for imperative programming. They are one of the most important building blocks used to design systems. Once data structures are designed using these conceptual data types, design of the executable code can often proceed in a straightforward fashion.

Higher-level Patterns: *Data Statement*

Other Related Patterns: *Data Definition, Data Declaration*

Pattern 2.5 *Data Definition*

Motivation:

In any imperative programming system in which programmers can define types, there must be programming elements that allow those definitions. *Data Definitions* are programming statements that allow the conceptual construction of user-defined data types without instantiating these data types in variables and other structures.

Discussion:

Any imperative programming language used as a serious development tool includes user-defined structures, and hence will have some programming mechanism for *Data Definition*.

Imperative languages differ in their programmatic support for complex data types. While most imperative languages (such as Pascal and C) permit complex user-defined types to be created through lower-level building blocks, others (such as APL) provide native linguistic elements to handle complex data types.

The most common use of *Data Definition* is to create unique structures that provide the collection of information needed by a programmer for a particular task. These structures are sometimes created with dynamic links so that structures can be joined by various linked list strategies.

Examples:

The following two examples (Figures 2.6 and 2.7) show the use of *Data Definition* instructions in Pascal and C.

```
(* DEFINE NEW TYPES -- STRING OF CHARACTERS AND RECORD *)
TYPE
  STRING = ARRAY [1..10] OF CHAR;
  S_NODE =
    RECORD
      X : INTEGER;
      S : STRING;
    END;
```

Figure 2.6. *Data Definitions* in Pascal.

```

/* Define new types -- pointer to a character, and structure */
typedef char *STRING;
typedef struct s_node {
    int x;
    STRING s;
} NODE;

```

Figure 2.7. *Data Definitions* in C.

Higher-level Patterns: *Data Statement*

Other Related Patterns: *Data Type, Data Declaration*

Pattern 2.6 *Data Declaration*

Motivation:

Once data types exist and have been defined so that they are available programmatically (*Data Definition*), an imperative programming language must provide a mechanism whereby variables can be declared to be of a particular type, and space allocated accordingly. The result of a *Data Declaration* programming statement is that memory is allocated to the variable, referred to within the program by the variable name, and reserved to be of the defined type.

Discussion:

Allocation of variables gives rise to questions concerning access (commonly referred to as scoping rules). Global variables are those variables declared in such a way that they are visible to a large portion of the program. Local variables are those declared in such a way that they are only visible within a limited scope (typically the procedure or function in which the variables are declared). The mechanisms for managing these issues vary from one imperative programming language to the next.

There are several common examples of *Data Declarations*. Constants can be viewed either as data definitions or as data declaration, depending on the philosophical

perspective and the language implementation. Constants may be allocated space (although not necessarily) but cannot be modified.

Variables are the currency of imperative programming. They are the bowls in which ingredients are mixed within the Betty Crocker metaphor.

Examples:

The following two examples (Figures 2.8 and 2.9) show the use of *Data Declaration* instructions in Pascal and C.

```
(* DECLARE VARIABLES *)
VAR
  NODE : S_NODE;
  X    : INTEGER;
```

Figure 2.8. *Data Declarations* in Pascal.

```
/* Declare variables */
int x;
NODE node;
STRING s;
```

Figure 2.9. *Data Declarations* in C.

Higher-level Patterns: *Data Statement*

Other Related Patterns: *Data Type, Data Definition*

Pattern 2.7 *Executable Statement*

Motivation:

The essence of program execution in the imperative perspective is the fetching of an instruction, decoding it by the processor, executing it, and fetching the next instruction. The vast majority of all imperative program code consists of executable

statements. Because imperative programming models the von Neumann architecture so closely, the concept of executing statements is central to the paradigm, whereas in other programming paradigms, sequential execution of instructions may be irrelevant to programming.

Discussion:

While much of imperative design work involves the organization of procedures and data structures, the majority of programming work involves controlling and managing the execution of the program as it manipulates data.

There are four broad categories of *Executable Statements*. *Control Flow Statements* are those statements that modify the fetch-decode-execute cycle from taking the next sequential instruction. *Data Manipulation Statements* are those that use data in some way, whether it be creating, viewing, deleting, or altering data. *Expressions* are used to combine values, variables and functions in such a way that the composition of them all can viewed as a single resultant piece of data. *Input/Output Statements* facilitate interaction between users and the program.

Higher-level Patterns: *Imperative Program*

Lower-level Patterns: *Control Flow Statement, Data Manipulation Statement, Expression, Input/Output Statement*

Other Related Patterns: *Procedure, Data Statement*

Pattern 2.8 Control Flow Statement

Motivation:

The fetch-decode-execute cycle of von Neumann computing presumes as a default that the next instruction to fetch is the one that lies contiguously in memory after the current instruction. However, complex computations require more than simple linear execution to achieve appropriate results. Hence, there is a need for *Control Flow Statements* to facilitate modified control paths through a program.

Discussion:

There are three common mechanisms provided by programming languages to enable the programmer to control the flow of execution: conditional statements, looping constructs, and jump statements. These are discussed briefly below, with examples from Pascal and C.

Compound statements are groups of statements physically grouped together (through the use of key words like BEGIN and END, or other symbols, such as { and }) that can form an executable subset without breaking the piece into a separate module. These compound statements are often used as groups of statements that execute or fail to execute, depending upon tested conditions.

Conditional statements are those that allow the testing for a condition (typically the evaluation of some expression), and then one or more paths pursued, depending on the outcome of the test. The two most common conditional statements follow the IF/THEN/ELSE model (in which an expression is evaluated during the IF, with the THEN compound statement taken if true, and the ELSE compound statement taken if false), or the CASE model (in which evaluation is made at the beginning and the appropriate CASE within the set of possibilities is taken.)

Examples:

The following examples (Figures 2.10 and 2.11) show IF/THEN/ELSE and CASE statements in Pascal and C.

```
IF X > 0 THEN
    WRITELN(X, ' IS POSITIVE');
ELSE IF X < 0
    WRITELN(X, ' IS NEGATIVE');
ELSE
    WRITELN(X, ' IS ZERO');
```

Figure 2.10. IF/THEN/ELSE in Pascal.

```

switch (x) {
    case '0':
        printf("x is zero\n");
        break;
    case '1':
        printf("x is one\n");
        break;
    default:
        printf("x is not a valid number\n");
        break;
}

```

Figure 2.11. Switch (CASE) in C.

Discussion:

There are a few common looping constructs, including WHILE, REPEAT, and FOR. These models are common to most imperative programming languages.

Examples:

The following examples (Figures 2.12 through 2.14 show various looping constructs in Pascal and C.

```

I := 0;
WHILE (I < 10) DO
BEGIN
    WRITELN('I = ', I);
    I := I + 1;
END

```

Figure 2.12. WHILE in Pascal.

```

I := 0;
REPEAT
    WRITELN('I = ', I);
    I := I + 1;
UNTIL (COUNT > 9)

```

Figure 2.13. REPEAT in Pascal.

```

for (i = 0; i < 10; i++) {
    printf("i = %d\n", i);
}

```

Figure 2.14. FOR in C.

Discussion:

Jump statements are a throwback to assembly programming, but still serve useful purposes in isolated circumstances. GOTO statements are absolute jumps to a particular label. BREAK statements allow the program to exit a loop prematurely (depending on a condition). CONTINUE statements perform similarly to BREAK statements, but serve to skip the rest of a compound statement and resume the looping.

No discussion of imperative control flow can be complete without at least mentioning the debate generated over the use of GOTO statements in high-level programming languages [Dijkstra 1968] [Knuth 1974] [Wulf 1972].

Examples:

The following examples (Figures 2.15 and 2.16 illustrate jump statements in Pascal and C.

```

I := 0;
1: WRITELN('I = ', I);
  I := I + 1;
  IF I < 9 THEN GOTO 1

```

Figure 2.15. GOTO in Pascal.

```

for (i = 0; ;i++) {
    printf("i = %d\n", I);
    if (i > 9)
        break;
}

```

Figure 2.16. BREAK in C.

Higher-level Patterns: *Executable Statement*

Other Related Patterns: *Data Manipulation Statement, Expression, Input/Output Statement*

Pattern 2.9 Data Manipulation Statement

Motivation:

From a strictly computational perspective, side effects constitute the sum total of what imperative programs have to offer. Imperative programming languages must have mechanisms for causing data values to be created, deleted or modified. *Data Manipulation Statements* facilitate the manipulation of data in various ways.

Discussion:

Most imperative languages (such as Pascal and C) provide primitive programmatic elements to manipulate only the most fundamental data elements. In these languages, manipulation of more complex structures must be programmatically crafted using *Data Manipulation Statements* and other language elements (such as control flow statements and expressions) as building blocks.

The most common form of data manipulation is assignment. This is the way in which variables obtain new values. These values are typically the result of expressions (whether simple or complex), and thus may hold the results of some other data manipulations. Another important form of *Data Manipulation Statement* involves the management and manipulation of dynamic data structures. This can include creating, initializing, and manipulating dynamically allocated variables and structures.

Many *Data Manipulation Statements* are destructive in that they permanently alter the contents of memory locations. The potential negative effects of these changes must be managed effectively when programming imperatively. However, not all *Data Manipulation Statements* are destructive. For example, a statement may reference a variable without changing it.

Examples:

The following two examples (Figure 2.17 and 2.18) show *Data Manipulation Statements* in Pascal and C.

```
X := 5;
```

Figure 2.17. Assignment in Pascal.

```
MYTYPE *p;
p = alloc(sizeof(MYTYPE));
free(p);
```

Figure 2.18. Dynamic data creation and deletion in C.

Higher-level Patterns: *Executable Statement*

Other Related Patterns: *Control Flow Statement, Expression, Input/Output Statement*

Pattern 2.10 Expression

Motivation:

It is very common for imperative programming to require a fair amount of computation, even in programs for which the problem domain is not intensely mathematical. This is in large part because imperative control mechanisms and data structures often have foundations that require mathematical manipulation (for example, incrementing an index to traverse an array). *Expressions* are groups of operators and operands where the operands can be constant values, variables, or other nested expressions.

Discussion:

Most imperative languages (such as Pascal and C) provide only primitive programmatic elements with which to build *Expressions*. These are typically dependent upon data manipulation statements and control flow statements.

There are several broad classes of operators, discussed below. The result of an expression can be assigned to a variable, used as a value in another *Expression*, passed as a parameter to a function call, or used as the evaluated criteria for a conditional statement.

The primitive operators that occur within *Expressions* can be grouped into the following classes: Unary, Multiplicative (including modulo operations), Additive, Relational/Equality, and Shift/Bitwise.

Unary operators involve only one operand. Not all imperative programming languages support the same set of unary operations. For example, most languages support some form of NOT, but most don't support unary incrementation (such as C).

Example:

Figure 2.19 shows unary operators in C.

```
x++;  
++i;
```

Figure 2.19. Unary increment operators in C.

Discussion:

Multiplicative operators involve two operands and perform either multiplication or division operations.

Example:

Figure 2.20 shows multiplicative operators in Pascal.

```
x := ( z * 5 ) / 13;
```

Figure 2.20. Multiplicative operators in Pascal.

Discussion:

Additive operators involve two operands and perform either addition or subtraction.

Example:

Figure 2.21 shows additive operators in Pascal.

```
x := z + 5 - y;
```

Figure 2.21. Additive operators in Pascal.

Discussion:

Relational and equality operators typically yield a boolean result. They are often used within conditional statements to determine which of several possible control paths to follow.

Example:

Figures 2.22 and 2.23 show relational and equality operators in Pascal and C.

```
IF (X < Y) THEN  
    WRITELN("X IS THE SMALLEST NUMBER");
```

Figure 2.22. Relational operators in Pascal.

```
if (x == y)
    printf("x and y are the same\n");
```

Figure 2.23. Equality operators in C.

Discussion:

Shift operators function only on integral data types (some form of integer—long, short, etc.). One of the operands is the set of bytes to be shifted, and the other operand is the number of bits to shift. The direction of the shift is inherent in the operator. Bitwise operators also deal with integral data types, but allow manipulation of individual bits through AND, OR, NOT, and XOR operations.

Example:

The following examples (Figures 2.24 and 2.25) show shift and bitwise operators in C.

```
x<<2; /* Shift x left 2 bytes */
y>>2; /* Shift y right 2 bytes */
```

Figure 2.24. Shift operators in C.

```
x = y & 1; /* AND operation to mask off all but lowest bit */
z = y | 1; /* OR operation to set lowest bit to 1 */
```

Figure 2.25. Bitwise operators in C.

Higher-level Patterns: *Executable Statement*

Other Related Patterns: *Control Flow Statement, Data Manipulation Statement, Input/Output Statement*

Pattern 2.11 *Input/Output Statement***Motivation:**

Not all programs require intensive *Input/Output*, but most programs require input from the user, and any program that fails to demonstrate its results is not worth much. Imperative programming languages universally accommodate input from the user (via keyboard and mouse) and output (typically via monitor). Other forms of I/O include reading from and writing to external storage (hard disks and other non-volatile storage media) and printing (to plotters, printers, or other hard copy devices). External devices such as modems, network cards, infrared ports, sound systems, microphones and video are becoming increasingly important forms of I/O that have to be dealt with in imperative languages.

Discussion:

The nature of I/O from system to system can vary so greatly that most imperative programming languages do not dictate particular forms of I/O within the languages themselves, but reserve those aspects for function libraries that are specific to the target system or environment. Given the increasingly divergent and innovative forms of input and output, it is easy to see the wisdom in reserving I/O functions of a language to system-dependent libraries, instead of building such dependencies into a language.

The nature of I/O has changed over the course of computing history. Early computing was not I/O-intensive at all, since programs typically ran in batch mode with very little (if any) input from users. Computation has now become a largely interactive activity with the user very involved and interacting with programs (typically with a keyboard and mouse, but with other devices becoming more important).

The most common forms of I/O involve reading and writing to files, reading from the keyboard, and writing to the screen. Imperative languages typically handle these activities in consistent, well-known ways.

Examples:

The following example (Figure 2.26) shows *Input/Output* in C.

```
c = getch();      /* Read a character from the keyboard */  
printf("%c\n", c); /* Echo the character back to the screen */
```

Figure 2.26. Common I/O in C.

Higher-level Patterns: *Executable Statement*

Other Related Patterns: *Control Flow Statement, Data Manipulation Statement, Expression*

3.0 Imperative Analysis and Design Patterns

The earliest programmers were primarily interested in the capabilities of their particular languages, and in the creation of rather small programs to perform specific tasks. The earliest courses in computer programming taught language syntax, and not design methodology. But as programming efforts began to grow larger, some form of design approach was needed. One of the earliest attempts to bring structure and rigor to design was the use of flow charts [Bohm 1966]. The introduction of a somewhat standard way to represent program control flow structure quickly led to additional discussions about appropriate forms of structure. One of the most famous debates from the earliest days of program design was the GOTO debate launched by Dijkstra, with his letter to the editor of *Communications of the ACM* entitled, “Go To Statement Considered Harmful” [Dijkstra 1968]. In that same year, Donald Knuth declared computer programming to be an “art” requiring guidelines for its effective channeling [Knuth 1968]. Shortly thereafter, Niklaus Wirth, the father of Pascal, introduced the concept of stepwise refinement as a guiding methodology for imperative design [Wirth 1971b]. Elements of this approach evolved, were formalized and popularized by Ed Yourdon as Structured Design [Constantine 1979] and Structured Analysis [Yourdon 1989]. During this evolution of design strategies, other approaches were proposed, including object-oriented design [Parnas 1972], and data-driven design [Jackson 1975] [Warnier 1977] [Jackson 1983]. But stepwise refinement held the majority of mind share in the program design community until object-oriented design, coupled with object-oriented languages, began to rise in popularity a decade later.

Given this history, stepwise refinement must be considered as a flagship methodology for imperative design. In addition, Yourdon’s work became a de facto standard for the formal representation of imperative design. We have therefore chosen to draw heavily from these two domains to create this pattern set for imperative analysis and design. In the patterns that follow, we outline key elements of structured analysis and design and discuss these principles in greater detail. These patterns in this set are listed in Figure 3.1.

- *Stepwise Refinement*
- *Structured Analysis and Design*
 - *Data Flow Diagram*
 - *Data Flow*
 - *Control Flow*
 - *Data Dictionary*
 - *Process Specification*
 - *Entity/Relation Diagram*
 - *State Transition Diagram*

Figure 3.1. Hierarchy of imperative analysis and design patterns.

Pattern 3.1 *Stepwise Refinement*

Aliases: *Top-Down Design*

Motivation:

Almost all imperative design approaches involve some form of problem decomposition in which a large problem is broken down into smaller subparts until some acceptable atomic unit is discovered and designed. This general approach to imperative design is known as *Stepwise Refinement* (or *Top-Down Design*) and is not limited to programming. Any plan or problem solution that can be described in terms of steps can be broken down using some form of *Stepwise Refinement*.

Discussion:

Stepwise Refinement represents one of the earliest efforts to bring structure and order to program design. *Top-Down Design* is but one of many forms of *Stepwise Refinement*, but is so commonly used as to be essentially synonymous. *Top-Down Design* begins with a view to the overall goals of a program—*what* needs to be accomplished, rather than *how* it needs to be done. A very broad solution can be described at this point in terms of course grained steps, which constitute the highest level of the design solution. This process continues with a series of *Stepwise Refinements*, in which each generalized high-level task is decomposed into smaller subtasks. The process repeats itself for each of

these subtasks until the tasks defined can be encapsulated into a single function or process.

Stepwise Refinement naturally leads to modular programming, since each subtask identified during the design process can be implemented as a separate module, unit, function, or procedure.

Examples:

The following example illustrates *Stepwise Refinement* with a non-technical example that is very easy to grasp and whose applicability to programming should be apparent. This example was borrowed from [Mandell 1985].

The task at hand is the creation of a pizza. Upon examination of the task, we determine that there are four distinct steps that constitute the creation of an effective pizza: 1) preheat oven; 2) prepare dough; 3) put sauce and toppings on pizza; 4) cook pizza. This organization is not the only initial refinement that would work, but is sufficient as an example.

Each one of these steps could be further refined, but we will look particularly at step 2, preparing the dough. We discover upon study and examination, that making dough is composed of seven distinct steps: 1) read recipe; 2) get ingredients ready; 3) measure ingredients; 4) mix ingredients; 5) let dough rise; 6) grease pan; 7) spread dough in pan. Again, this break down of preparing the dough could have resulted in more or fewer steps and not necessarily this organization.

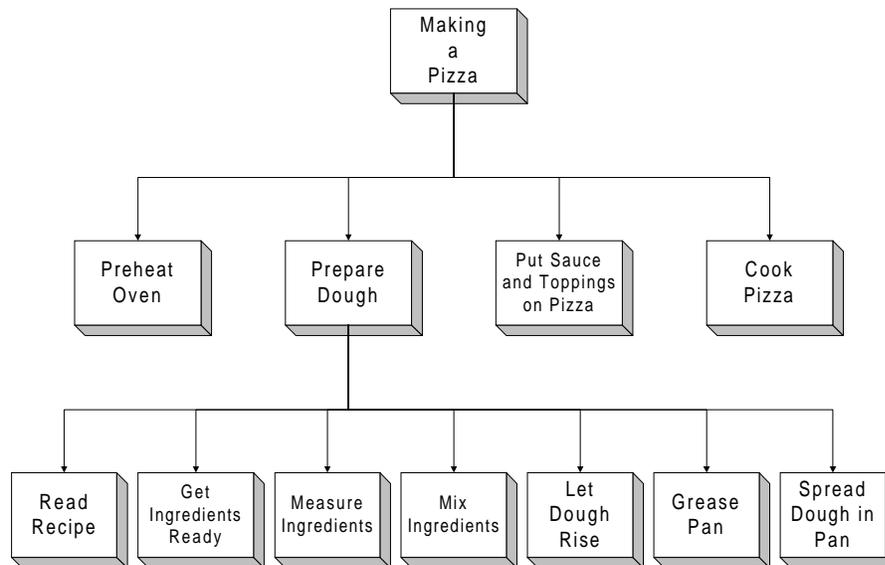


Figure 3.2. Steps in making a pizza.

Note that this refinement ultimately results in a sequential set of steps. There are also other concepts that might enter into our design. For example, since there are multiple ingredients, the process of getting ingredients ready, measuring them, and adding them to the mixture, could be done in an iterative fashion for each ingredient. It could also be done sequentially if we explicitly identified these three steps for each ingredient. Despite the fact that we have imposed some structure, and some modularity, our example still views programming as the task of accepting some input to produce some output. It gives us a better handle with which to grasp imperative program design, but does not immediately lead us to broader plains.

Other Related Patterns: *Structured Analysis and Design, Imperative Program*

Pattern 3.2 Structured Analysis and Design

Motivation:

Structured Analysis is the classic model of imperative design, championed by Ed Yourdon [Yourdon 1989] and so sometimes referred to as Yourdon Structured Systems

Analysis or just the Yourdon Method. Structured Analysis is not a single cohesive methodology for analysis and design, but rather a conglomeration of approaches that evolved over more than twenty years without a consistent theoretical foundation. As such, it is an interesting model of imperative design, and embodies essentially all key aspects of design pertinent to imperative programming.

Discussion:

Structured Analysis grew out of the mid-1970's time frame in which essentially all programming was imperative. Despite the fact that it evolved over a twenty year period, Structured Analysis remains quite limited in its ability to model modern operating system concepts like real-time systems, multi-tasking and multi-threaded environments (including inter-object and intra-object concurrency), and distributed programming environments.

Yourdon's approach has survived in part because it was willing to adapt and absorb other approaches. This leads to some conceptual confusion, when similar concepts are represented differently in different sub-models. For example, objects in an entity-relation diagram correspond to stores in a data flow diagram. In a unified model, this concept would not be divided in this way, but would be represented in a unified way.

This lack of conceptual foundation is both a blessing and curse. The lack of a foundation has allowed *Structured Analysis and Design* to adapt over years and absorb useful models from other areas of design and analysis, which has contributed in part to its popularity and longevity. However, the lack of a consistent foundation lends some confusion and representational inconsistency when the model is extended and stretched to modern environments.

It should be noted that *Systems Analysis* models computer behavior, not real world behavior. It does not attempt to bridge the gap between the real world of the end-user and the programming world of the software engineer. Its purpose is to provide a mechanism for designing and documenting imperative programs.

Lower-Level Patterns: *Data Flow Diagram, Data Dictionary, Process Specification, Entity/Relation Diagram, State Transition Diagram*

Other Related Patterns: *Top-Down Design, Imperative Program*

Pattern 3.3 Data Flow Diagram

Aliases: *DFD, Bubble Chart, Bubble Diagram, Process Model, Work Flow Diagram, Function Model*

Motivation:

One perspective views an imperative program as a black box that transforms inputs into outputs. This black box representing the entire program can be viewed as being composed of smaller black boxes each converting inputs to outputs. *Data Flow Diagrams* capture these black boxes as bubbles that share information (data entering as inputs and leaving as outputs).

Discussion:

Data Flow Diagrams can adequately describe functional programs in an imperative fashion, but seriously lack the ability to capture more modern models of computer usage (highly interactive, event-driven, concurrent, or distributed systems).

DFDs capture two kinds of flow: data flow and control flow. A *DFD* consisting entirely of data flow information can be used to represent functional programs as the composition of black boxes comprising the program. The control flow portions of data flow diagrams were initially introduced to enable the modeling of real-time systems, and lend themselves to other more recent kinds of programming environments, but still tend to fall short.

DFDs provide a convenient overview of the major functional components of a system, but do not provide any detail about the components. However, multiple *DFDs* can be arranged hierarchically, suggesting composition of larger functions.

The *Data Flow Diagram* is a graphical model, but may be augmented with detailed textual information as needed.

Example:

In the following example, we examine the creation of a cake with data (or cake ingredients) moving into a function (Bake Cake), and output emerging (Cake). This example was borrowed from [Yourdon 1989].

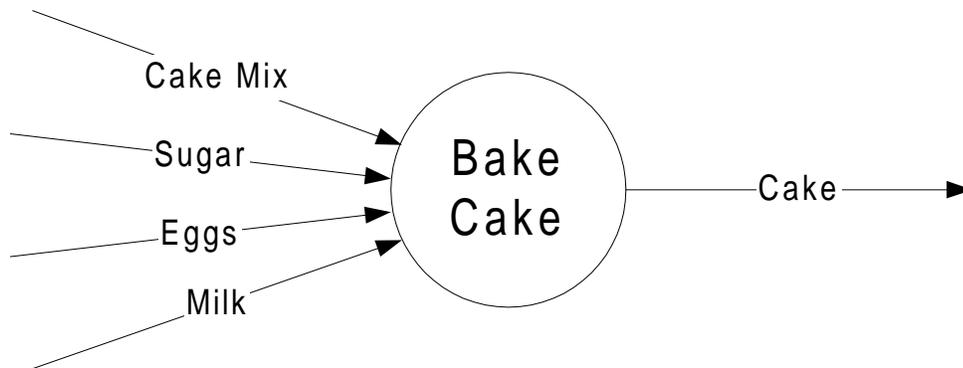


Figure 3.3. *Data Flow Diagram* for baking a cake.

Note that this *Data Flow Diagram* is not concerned with the mechanism by which Bake Cake is achieved. This is determined further along in the refining process, and may result in nested *DFDs*.

Higher-Level Patterns: *Structured Analysis and Design*

Lower-Level Patterns: *Data Flow, Control Flow*

Other Related Patterns: *Data Dictionary, Process Specification, Entity/Relation Diagram, State Transition Diagram*

Pattern 3.4 Data Flow

Motivation:

The most essential and original motivation for data flow diagrams is to illustrate the movement of data between functions. This movement is referred to as a *Data Flow*, and is illustrated by the *Data Flow* portion of a *DFD*.

Discussion:

The *Data Flow* portion of a DFD is composed of several pieces that play a part in representing the flow of data between functions. The following components of *Data Flows* are described below: Data Store, Flow, Data Packet, Process, and Terminator.

A Data Store shows collections (or aggregates) of data that the system must remember for some period of time. When the system designers and programmers finish building the system, the Stores will typically exist as files or databases. Data Stores are also used to model a collection of data packets at rest.

Flows show the connections between the processes (or system functions). They represent the information that the processes require as input and/or the information they generate as output. They also describe the movement of chunks, or packets of information from one part of the system to another (data in motion). The chunks may conceptually be anything. Flow can be either to or from a data store (or both). Flow from a store is typically seen as a read or an access to information in the store. Flow to a store is typically seen as a write, an update, or possibly a delete. Flows connected to a store can only carry packets of information that the store is capable of holding.

Data Packets represent data moving from one object to another.

Processes represent the various individual functions that the system carries out. Processes transform inputs to outputs.

Terminators show the external entities with which the system communicates. Terminators are typically individuals, groups of people, external computer systems, or external organizations. They are always outside of the system being modeled. Flows represent the interface between the system being modeled and the real world. Neither the systems analyst nor the systems designer is in a position to change the contents of a terminator or the way the terminator works since it lies outside of the system being designed.

Examples:

The following figures show different kinds of Data Flows. All examples are borrowed from [Yourdon 1989]. In the following example (Figure 3.4), the DFD represents a single input (Phone Number) to a process or procedure (Validate Phone Number).

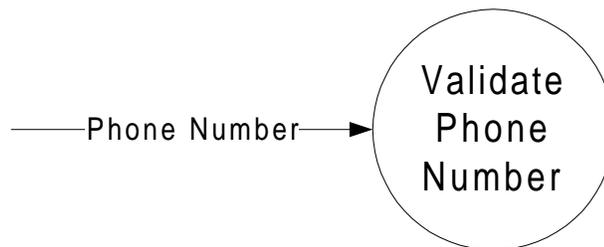


Figure 3.4. An input flow.

In the following example (Figure 3.5), the DFD represents a single output (Invoice) from a process or procedure (Generate Invoice Report).

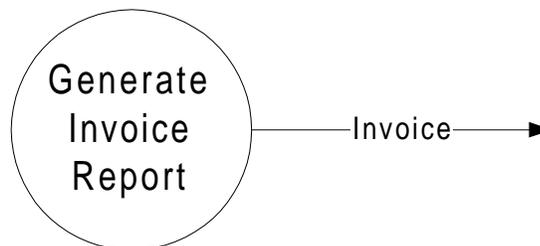


Figure 3.5. An output Flow

In the following example (Figure 3.6), the DFD represents a dialog flow in which an input (Order Status Inquiry) is given to the process or procedure (Determine Order Status) and an output (Order Status Response) is generated.

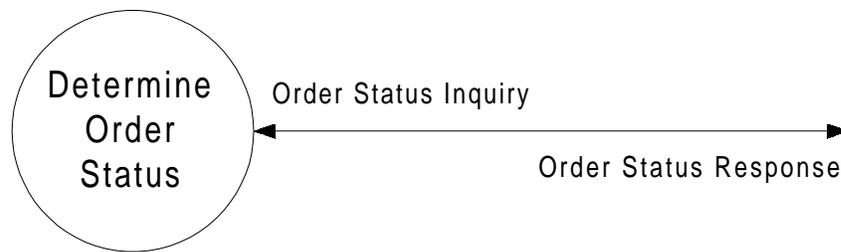


Figure 3.6. A dialog flow

In the following example (Figure 3.7), the DFD represents a diverging flow in which a single data flow (Customer Address) is split into different inputs to different processes or procedures.

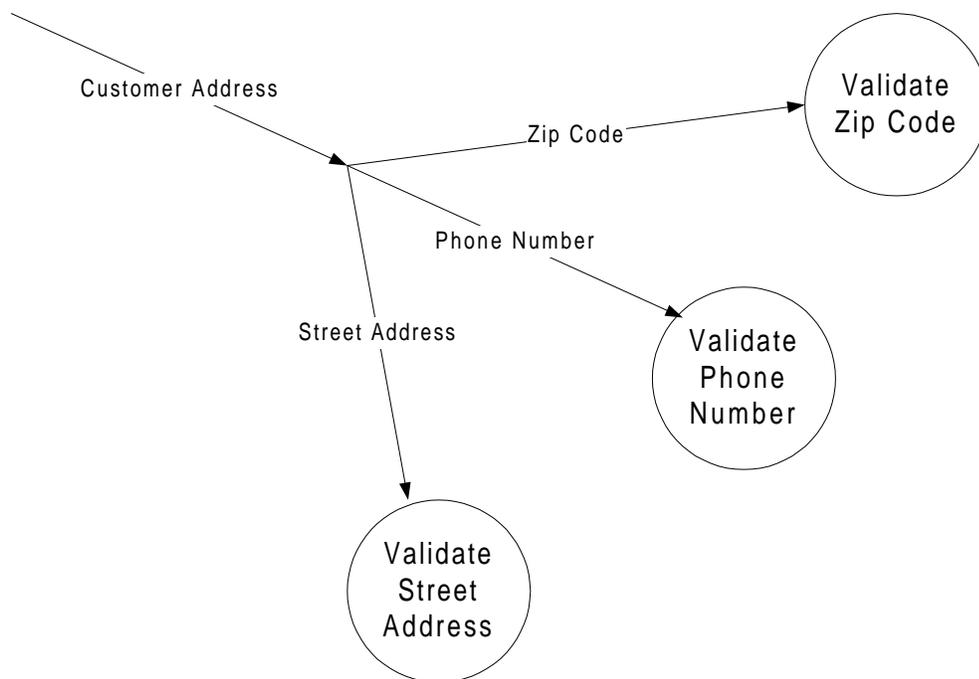


Figure 3.7. A diverging data flow.

Higher-level Patterns: *Data Flow Diagram*

Other Related Patterns: *Control Flow*

Pattern 3.5 Control Flow**Motivation:**

There are two separate motivating factors for modeling *Control Flow*. Flow charts were one of the earliest imperative design tools, and primarily modeled the *Control Flow* of programs, while representing actions and decisions performed along the way. Within the constraints of structured analysis, flow charts are largely dealt with as a part of process specification, since they describe the internal behavior of an individual process. Therefore, a fuller discussion of flow charts is contained in the *Process Specification* pattern.

The motivation for *Control Flow* as a separate category of programming to be modeled within structured analysis grew ostensibly out of the need to support real-time systems, but could as well have been motivated by any number of modern programming environments that must be handled by programmers, such as highly interactive, event-driven, concurrent, or distributed systems. *Control Flows* attempt to model the interaction between processes independent of the flow of data used in computations. In this way, the *Control Flow* extensions to data flow diagrams represent less of a functional view, since the system is no longer strictly modeled as the composition of functions to achieve a single black box.

Discussion:

The *Control Flow* portion of a DFD is composed of several pieces that play a part in representing the flow of control information between functions. The following components of *Control Flows* are described below: Control Store, Control Flow, Signal Packet, and Control Process. Notice the similarities between these components and those used for data flow.

Control Stores are used to store signals.

Control Flow consists of a signal or interrupt, but does not carry value-bearing data. In other words, it is outside the domain of inputs and outputs to functions.

Signals are the currency of exchange for *Control Flow*.

A Control Process is a bubble whose only job is to coordinate and synchronize the activities of other bubbles in the DFD. Inputs and outputs to Control Processes consist only of *Control Flows*. Typically there is only one control process in a DFD. A Control Process has a state-transition diagram.

Example:

In the example below (Figure 3.8) processes exist to deal with satellite and radar data, but require some external synchronization in order to fulfill some real-time requirements. The Control Processes and Control Flows are represented by dashed lines. This example was borrowed from [Yourdon 1989].

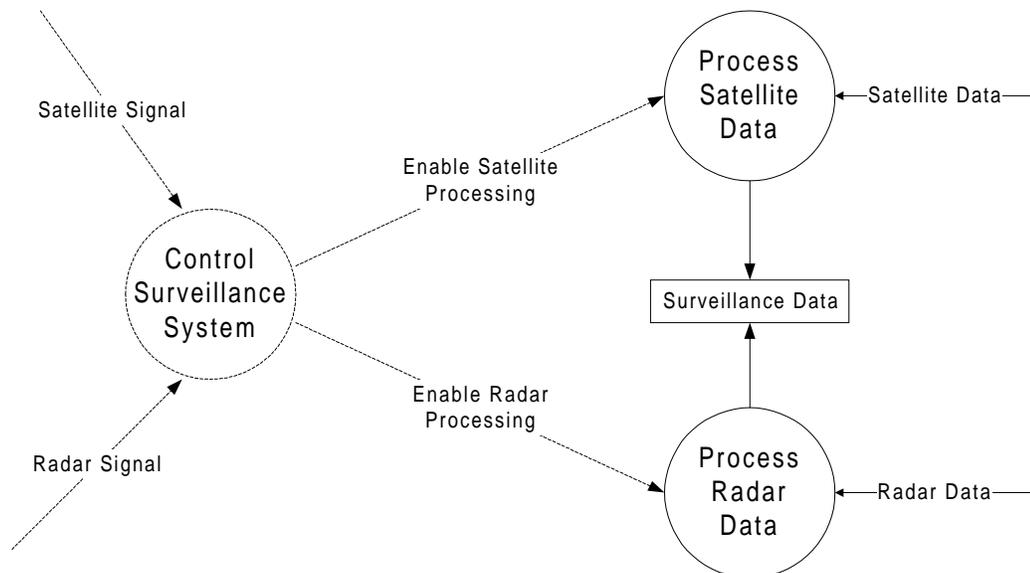


Figure 3.8. A DFD with control flows and control processes.

Higher-level Patterns: *Data Flow Diagram*

Other Related Patterns: *Data Flow*

Pattern 3.6 Data Dictionary

Motivation:

Imperative programming involves the manipulation of data that resides in memory. In fact, we can often view the output of an imperative program as the cumulative side effects wreaked upon the data elements of a given program. A *Data Dictionary* is used to store information about data used by an imperative program. If a data flow diagram illustrates the transformation of data by processes, the *Data Dictionary* shows the details of just what information is being transformed.

Discussion:

A *Data Dictionary* is part of a DFD that shows the detailed description of the information that is being transformed. It can be viewed as an organized listing of all the data elements that are pertinent to the system. This listing must include precise and rigorous definitions so that there is a consistent understanding of inputs, outputs, composition of stores, and intermediate calculations.

A *Data Dictionary* does the following things:

- Describes the meaning of flows and stores shown in data flow diagrams.
- Describes the composition of data packets moving along the flows.
- Describes the composition of data stores.
- Specifies relevant values and units of elementary chunks of information in the data flows and data stores.
- Describes the relationships between stores that are highlighted in an entity/relation diagram.

Data Dictionaries use a consistent formal notation for describing data elements.

Notation includes the following symbols with their meanings:

- = is composed of
- + and
- () optional (may be present or absent)

- {} iteration
- [] select one of several alternative choices
- ** comment
- @ identifier (key field) for a store
- | Separates alternative choices in the [] construct

Each element in a *Data Dictionary* has a definition. Each definition consists of the following elements:

- The meaning of a data element within the context of the user's application, usually provided as a comment using **.
- The composition of the data element, if it is composed of meaningful elementary components.
- The values that the data element can take on, if it is an elementary data element that cannot be decomposed any further. Values should have units, range, accuracy or precision.

Elements in a *Data Dictionary* can be either elementary or composite data elements. Elementary data elements are those for which there is no meaningful decomposition in the context of the user's environment. They are part of the data dictionary, once they have been identified. Composite data elements are those for which there is a meaningful decomposition.

The following principles guide the usage of a *Data Dictionary*.

- Every flow on the DFD should be defined in the *Data Dictionary*.
- All components of composite data should be defined
- No data element should be defined more than once.
- Correct notation should be used for all definitions.
- All data elements should be referenced in DFDs, entity/relation diagrams, or state transition diagrams.

Example:

The following example (Figure 3.9) shows an example *Data Dictionary* for a keyed phone number. This example is borrowed from [Pressman 1987].

keyed phone number	= [local extension outside number 0]
local extension	= [2001 2002 ... 2999 conference set]
outside number	= 9 + [local number long distance number]
local number	= prefix + access number
long distance number	= (0) + area code + local number
conference set	= {# + local extension + #(#)} ₂ ⁶

Figure 3.9. *Data Dictionary* for a keyed telephone number.

Higher-level Patterns: *Structured Analysis and Design*

Other Related Patterns: *Data Flow Diagram, Process Specification, Entity/Relation Diagram, State Transition Diagram*

Pattern 3.7 Process Specification

Alias: *Minispec*

Motivation:

Imperative programming involves the mapping of inputs to outputs through some kind of functional transformation. Whereas data flow diagrams illustrate data moving between processes as inputs and outputs, *Process Specifications* are textual models that describe how the information is transformed within each process or bubble of a DFD.

Discussion:

Process Specifications are sometimes referred to as *Minispecs* (miniature specifications). They describe what's happening inside each bottom-level, primitive bubble in a *DFD*, defining what must be done in order to transform inputs into outputs.

Process Specifications may use any mechanism for representation, but there are several criteria:

- It must be expressed in a form that is unambiguous and verifiable by the user and systems analyst.
- It must be expressed in a form that can be effectively communicated to the various audiences involved.
- Any tool used should not impose (or imply) arbitrary design and implementation decisions.

Process Specifications typically take one of the following forms: Structured English, Pre/Post Conditions, and Decision Tables. These are described below.

Structured English is the form favored by most systems analysts. It consists of English with elements of formal structure. Although Structured English is a subset of the English language, there are major restrictions on the kind of sentences that can be used and the manner in which sentences can be put together. It seeks to strike a balance between the precision of formal programming languages and the informality and readability of the English language.

Pre/Post Conditions are a convenient way of describing the function that must be carried out by a process, without having to describe the algorithm or procedure that will be used within the function. Preconditions describe all the things (if any) that must be true before a process begins operating. Postconditions describe what must be true when a process has finished doing its job.

Decision Tables are used when neither Structured English and Pre/Post Conditions are appropriate. It is particularly suitable when a process must produce some output or take some actions based on complex decisions.

The following forms for Process Specification can be used but are less desirable and less common: Nassi-Shneiderman Diagrams, Charts or Graphs, Flow Charts, and Narrative English. These are described below.

Nassi-Shneiderman Diagrams are a type of structured flowcharting technique. They can be viewed as a form of Structured English with boxes drawn around them.

Charts or Graphs are only used when the specification is easily represented by a graph or a chart. For example, if a program presents information in a chart, a chart can be used to represent the chart displayed to the user.

Flow Charts show sequential logic of procedures. They should only be used to describe detailed logic, and should use a subset of the Structured English constructs.

Narrative English is typically not recommended. In fact, Yourdon states that he hates narrative English because it is notoriously ambiguous.

Examples:

The following example (Figure 3.10) shows an example *Process Specification* using Structured English for a process that totals order records. This example is borrowed from [Yourdon 1989].

```

daily-total = 0
DO WHILE there are more orders in ORDERS with Invoice-date = today's date
    READ next order in ORDERS with Invoice-date = today's date
    DISPLAY to Accounting invoice-number, customer-name, total-amount
    daily-total = daily-total + total-amount
ENDDO
DISPLAY to Accounting daily-total
  
```

Figure 3.10. Example *Process Specification* using Structured English.

Higher-level Patterns: *Structured Analysis and Design*

Other Related Patterns: *Data Flow Diagram, Data Dictionary, Entity/Relation Diagram, State Transition Diagram*

Pattern 3.8 Entity/Relation Diagram

Aliases: *ERD, E-R Diagram*

Motivation:

Data flow diagrams show the flow of data between processes, and data dictionaries describe data in detail, but it is also important to identify the relationships that exist between different data elements or types. *Entity/Relation Diagrams* are used to demonstrate those relationships in a graphical way.

Discussion:

Entity/Relation Diagrams are graphical models of data relationships. *ERDs* can be augmented by detailed textual information where needed for clarity. They are typically used to model the stored data of a system at a high level of abstraction. The relationships between data are independent of the processing performed on them.

ERDs consist of the following elements: Object Types, Relationships, Associative Object Type Indicators, and Supertype/Subtype Indicators. Each of these is described below.

Object Types represent a collection, or set of objects (or things) in the real world whose members play a role in the system being developed, can be identified uniquely, and can be described by one or more facts or attributes.

Relationships represent a set of connections, or associations, between the object types connected by arrows to the relationship. There can be more than one relationship between two objects. There can also be multiple relationship between multiple objects. Relationships can occur between different instances of the same object type.

Associative Object Type Indicators are a special notation in E-R Diagrams, representing something that functions as an object and a relationship. It represents a relationship about which we wish to maintain some information.

Supertype/Subtype Object Types consist of an object type and one or more subcategories, connected by a relationship.

Example:

In the example below (Figure 3.11) shows a typical *Entity/Relation Diagram*. This example was borrowed from [Yourdon 1989].

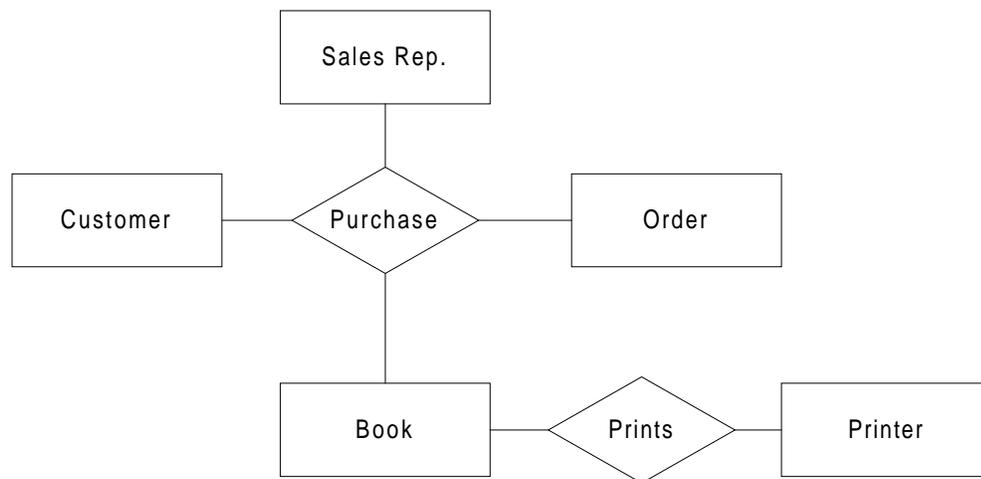


Figure 3.11. A typical *Entity/Relation Diagram*.

Higher-level Patterns: *Structured Analysis and Design*

Other Related Patterns: *Data Flow Diagram, Data Dictionary, Process Specification, State Transition Diagram*

Pattern 3.9 State Transition Diagram

Motivation:

As systems evolved from the imperative batch model, timing became a programmatic issue that had to be modeled in some way. *State Transition Diagrams* are an attempt to focus on and model the time-dependent behavior of a system.

Discussion:

State Transition Diagrams consist of the following elements: State (Initial State, Prior State, Successor State, Final State, High-Level State), Action, Condition, and State Change. Each of these is described below.

States represent periods of time during which the system exhibits some observable behavior. A State can consist of waiting for something in the external environment to occur. It may also consist of waiting for a current activity in the environment to change to some other activity. A system will be in a given State for a finite period of time. Within State Transition Diagrams are various kinds of States, including Initial States, Final States, and Successor States.

State Changes show transitions from one State to another.

Conditions are associated with each State Change, and are the events or circumstances that caused the change of State.

Actions are the activities that take place as part of the change of State. There may be zero or more actions associated with a transition. Actions are perceived to occur instantaneously.

State Transition Diagrams can be used to represent the process specification for a process in a *DFD*. The conditions in the *State Transition Diagram* correspond to the control flows on a *DFD*, and the actions on the *State Transition Diagram* correspond to outgoing control flows on the *DFD*.

Example:

The example below (Figure 3.12) shows a typical *State Transition Diagram* for a telephone answering machine. This example was borrowed from [Yourdon 1989].

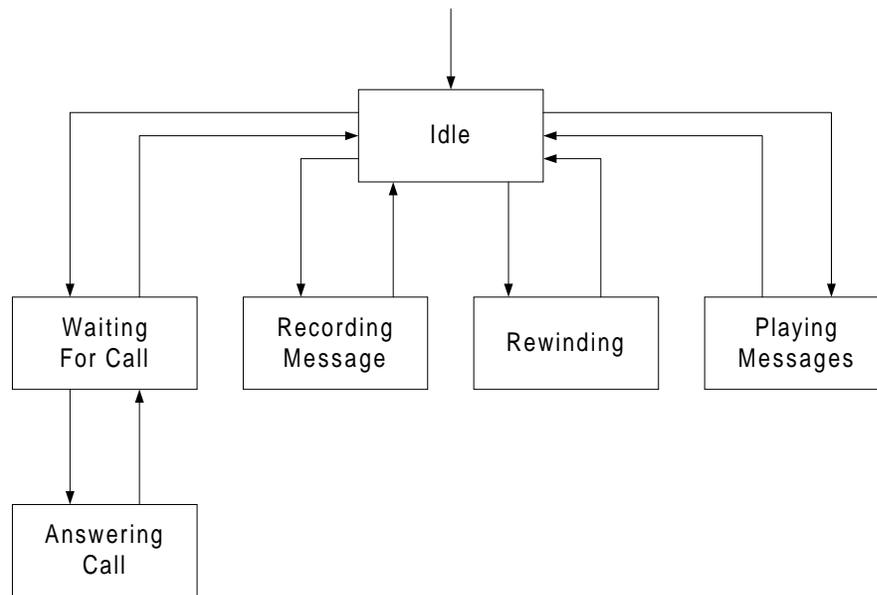


Figure 3.12. A typical *State Transition Diagram*.

Higher-level Patterns: *Structured Analysis and Design*

Other Related Patterns: *Data Flow Diagram, Data Dictionary, Process Specification, Entity/Relation Diagram*

4.0 Object-Oriented Programming Patterns

Some of the earliest efforts to improve imperative programming attempted to eliminate side effects and introduce more natural modularity. Some of the earliest work in this area was done by Liskov and Zilles [1974] and involved the use of abstract data types (ADTs). Conceptually, an ADT can be viewed as a tuple consisting of data objects and the operations that can be performed on them. In imperative design, data types and declarations are used to represent items in the real world that need to be modeled, while procedures and instructions operate on the data. In the ADT model, items in the real world are modeled as “objects” and consist of both data and the operations that can be performed on them. In this way, data and procedures are encapsulated, and information is carefully contained within the context in which it is used. Parnas [1972] dealt with similar principles of modularity, and used the term *information hiding* to describe the concept that information should be hidden from other parts of the program in order to eliminate unwanted side effects. The principles that were evolving into object-oriented design were initially applied to imperative programming, since there were no object-oriented programming languages at the time.

The earliest example of a programming language supporting objects is Simula [Nygaard 1981], which was an extension of the imperative programming language Algol 60. The grandfather of modern object-oriented languages is probably Smalltalk [Kay 1993] [Budd 1987]. Object-oriented languages can be viewed in two broad groups: those that are “pure” object-oriented, and those that are adaptations from imperative languages. Among the “pure” object-oriented languages, the most prominent are Smalltalk and Eiffel [Meyer 1985]. Among those languages that are adaptations from imperative languages are Ada [Whitaker 1996], Modula-II [Wirth 1985], Objective-C [Cox 1986], Object Pascal [Tesler 1985], and C++ [Stroustrup 1991].

In this chapter, we have attempted to capture the essential elements of object-oriented programming in pattern form. This pattern set will be used as a vehicle to analyze aspects of object-oriented design, and to explore multiparadigm programming and design. We do not expect this pattern set to be exhaustive in revealing everything that

could be said about object-oriented programming (although it could provide an excellent foundation on which to build such an analysis).

This taxonomy of object-oriented programming elements was culled from an analysis of programming elements in selected object-oriented languages including C++, Ada, Java, and Smalltalk. The essential programmatic elements were identified, and are listed in the pattern hierarchy below (Figure 4.1). Each of these elements is discussed in greater detail in the patterns that follow.

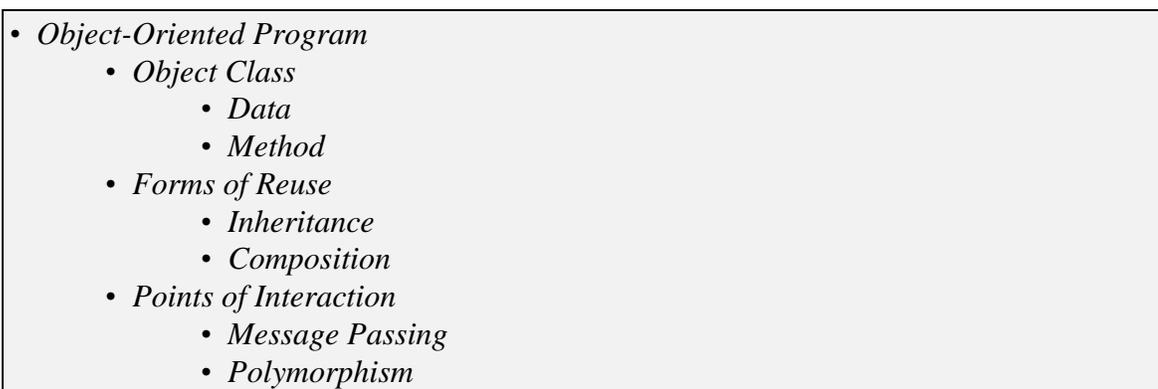


Figure 4.1. Hierarchy of object-oriented programming patterns.

Pattern 4.1 *Object-Oriented Program*

Motivation:

Despite the traditional popularity of imperative programming, there have always been serious limitations to designing large systems in an imperative fashion. The most serious problems stem from the difficulty in organizing a large imperative program in a modular fashion, and the natural tendency for imperative programs to generate undesirable side effects. Neither of these is particularly surprising, given the nature of imperative programming, which includes sequential execution and side effects as the operative mindset. Object-oriented programming is the result of an effort to create high-quality software through programmatic and design elements that eliminate unwanted side effects and cumbersome sequential execution while encouraging reuse.

Discussion:

Improvements in systematic imperative design has made it easier to build appropriately modular imperative code, but has still not completely solved the problem. Further need for modularity and encapsulation has aided the evolution and adoption of object-oriented programming.

Object-oriented programming has evolved over time to include three fundamental principles or programmatic components: Objects, Classes, and Inheritance. It should be noted that there are other aspects that are important to object-oriented programming, but these three are the most fundamentally distinguishing aspects that set object-oriented programming apart from imperative programming. Each takes its place in the evolutionary history of object-oriented programming, and is discussed in its respective patterns in this section. There has been debate over the years as to what constitutes object-oriented programming, but there seems to be a general consensus that these three aspects are essential to a full definition of object-orientation.

Booch [1994a] defines object-oriented programming as “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 relationships.”

Lower-level Patterns: *Object Class, Forms of Reuse, Points of Interaction*

Other Related Patterns: *Imperative Program, Functional Program, Logical Program, Multiparadigm Program*

Pattern 4.2 <i>Object Class</i>

Aliases: *Package, Cluster, Module*

Motivation:

Abstract data types [Liskov 1974] and information hiding [Parnas 1972] express principles of encapsulation that relate to the declaration of data as well as the creation of

the functions that operate on that data. The encapsulation of data and functionality (methods) in object-oriented programming is done with *Object Classes*.

Discussion:

Object Classes provide the mechanism through which an object-oriented program captures data and the functions that operate on them. These functions are referred to as methods. Data and methods are discussed in separate patterns. *Object Classes* are conceptual notions, and provide a form of data and method definition that can later be instantiated through a declaration of some kind. Because of this, *Object Classes* can be manipulated in abstract ways to create other classes through subclassing, including inheritance and composition. These patterns are also included in this pattern set.

Examples:

All object-oriented languages have the notion of *Object Classes*, although it takes different forms and is referred to by different names. Most object-oriented languages use the term *Class*. However, Ada refers to these elements as *Packages* [Whitaker 1996], Modula II refers to them as *Modules* [Wirth 1985], and CLU refers to them as *Clusters*. For each of these languages the *Object Class* concept is implemented in slightly different ways.

Higher-level Patterns: *Object-Oriented Program*

Lower-level Patterns: *Data, Method*

Other Related Patterns: *Forms of Reuse, Points of Interaction*

Pattern 4.3 Data

Motivation:

With roots in the imperative perspective, object-oriented programming has a view that involves the manipulation of data stored in locations. The problem of side-effects is acceptable from a strictly theoretical perspective, but out-of-control side-effects are not

acceptable. These are the concerns that led to information hiding and encapsulation. The information that is being hidden and encapsulated is, of course, *Data*.

Discussion:

Every object class encapsulates data definitions in the same sense as is used in the imperative perspective. The difference here is that *Data* is bundled together with the functions, or methods, that will be used to modify the data. Apart from this bundling, the essence of data definitions and data declarations within an object class is not significantly different from an imperative approach.

Higher-level Patterns: *Object Class*

Other Related Patterns: *Method*

Pattern 4.4 Method

Aliases: *Function, Procedure*

Motivation:

A class definition includes data and the means to affect this data without inflicting side effects on the rest of the system. These *Functions*, or *Procedures*, are commonly referred to as *Methods* within the object-oriented perspective.

Discussion:

The typical *Method* looks very much like an imperative procedure. The difference lies in the strict scoping that determines the data upon which the *Method* can act. Also important is the concept that other objects in the system are not permitted access to the data except through the appropriate *Method*.

Methods are associated with data within a certain object class. *Methods* can be used to build other object classes (through composition) or can be reused and borrowed through inheritance.

Methods are conceptual entities when defined as part of a class, but become living agents when a class becomes instantiated.

Specialized *Methods* include constructors and destructors. Constructors are *Methods* that perform initialization appropriate for a given object class. This might include creating and initializing a dynamic data structure, such as a linked list. Destructors are *Methods* that perform de-initialization appropriate for a given object class. Following the same example, this might include tearing down a linked list and freeing its memory.

Higher-level Patterns: *Object Class*

Other Related Patterns: *Data*

Pattern 4.5 *Forms of Reuse*

Motivation:

One of the early questions raised in the field of software engineering involved the reuse of ideas and knowledge as well as software components [Prieto-Diaz 1987].⁷ Some progress was made using imperative programming languages (such as standard function libraries). However, object-oriented programming languages encourage reuse on a grander scale by reusing not just function libraries, but entire object class definitions.

Discussion:

One of the early factors limiting reuse in imperative programming was a perceived low quality of most software modules, and a general distrust on the part of engineers toward software not developed in house (often referred to as NIH syndrome—“Not Invented Here”). The use of function libraries helped to standardize certain functions and raise confidence in reusable modules in a general way. This in part helped pave the way for acceptance of object-oriented subclassing as a form of reuse. The subclassing of

⁷ For an interesting and informative discussion of the historical context for the rise of reuse, see [Jones 1979].

object classes naturally involves a greater amount of reuse than function libraries, since the data structures are reused as well as the methods.

In imperative programming it is common to construct data structures out of other structures. For example, a list of structures can be created that is independent of the structure to be used. Structures themselves are constructed of smaller units, including other structures. But when structures are contained within object classes, reuse has to be done at the level of the object class, since access to the structures is available only within the confines of the methods provided by the object class.

Two primary forms of subclassing are common in object-oriented programming languages: *Inheritance* and *Composition*. These are described in separate patterns.

Higher-level Patterns: *Object-Oriented Program*

Lower-level Patterns: *Inheritance, Composition*

Other Related Patterns: *Object Class, Points of Interaction*

Pattern 4.6 *Inheritance*

Aliases: *Is-A*

Motivation:

Once an object class has been created and contains a full type definition of the object, including data and methods, it would seem convenient to attempt to reuse it without having to explicitly restate all the things already in it. One form of reuse is known as *Inheritance*, and involves the declaring of new object classes using existing object classes as a foundation. Specifically, *Inheritance* involves taking the declaration of an existing class and adding to it without modifying the existing class, so that the original class is preserved and reused in its entirety.

Discussion:

Inheritance is a natural form of reuse for object classes, borrowing concepts of reuse from imperative programming such as the construction of data structures from simpler structures, and the reuse of functions through libraries.

Inheritance has an innate problem (referred to as the yo-yo problem) that stems from the reuse of object classes in a hierarchy. To understand a class that inherits data or methods from another object class, one must move up the hierarchy until that material is defined. This process of moving up and down the hierarchy tree is the yo-yo.

Inheritance is really the premier approach to subclassing. It permits a designer to leave a class largely undisturbed, and use its features without having to fully understand what is inside. This is sometimes referred to as specialization. Specialization happens, for example, in the medical world. An individual attends medical school to become a general practitioner. If he wants to continue in school, he can specialize and become a pediatrician. Doing so adds to the standard medical training, and results in greater abilities, even as the area of focus is being narrowed. This approach is also typically referred to as using inheritance for extension. If a general practitioner chooses to restrict his practice to some domain that is narrower than his medical degree qualifies him for, but broad enough to make a living, we say that he is using inheritance for restriction. In practice, much inheritance partakes of both extension and restriction.

Examples:

The following example (Figure 4.2) shows inheritance using OSA notation [Embley 1992]. In this example, there are attributes that pertain to Person (such as name and address) that are common to all employees and customers, since they are also people. This example is borrowed from [Embley 1992].

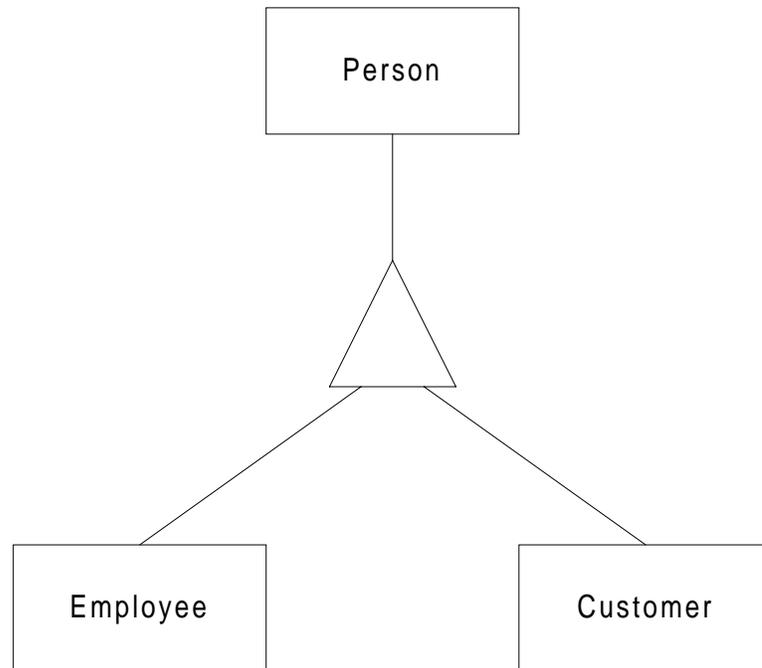


Figure 4.2. *Inheritance* using OSA notation.

For additional programmatic examples of inheritance in various object-oriented programming languages, see [Budd 1997].

Higher-level Patterns: *Forms of Reuse*

Other Related Patterns: *Composition*

Pattern 4.7 *Composition*

Aliases: *Has-A*

Motivation:

Another approach to reuse, related to inheritance, involves taking pieces of existing classes, but taking only parts, and not the entire class. This approach is known as *Composition*. It requires a more explicit understanding of the elements within individual classes, but eliminates the yo-yo problem that sometimes occurs with inheritance.

Discussion:

Composition is achieved by creating a class and then borrowing methods and data from other classes by explicitly declaring them (as opposed to inheriting the entire class, with all its data and methods). Even object classes that don't borrow from other classes use *Composition*, since they explicitly import from built-in types.

Composition relieves the yo-yo problem by explicitly declaring those methods and data that will be included in the subclass, eliminating the need to look up the hierarchy tree to find information.

Examples:

The following example (Figure 4.3) shows *Composition* using OSA notation. In this example, the object *Person* is composed of elements (*Name* and *Address*) taken from other object classes. The numbers ("1" and "1:*") are participation constraints that indicate that a *Person* has only one *Name* and one *Address*, but that *Names* and *Addresses* may each pertain to one or more *Persons*. This example is borrowed from [Embley 1992].

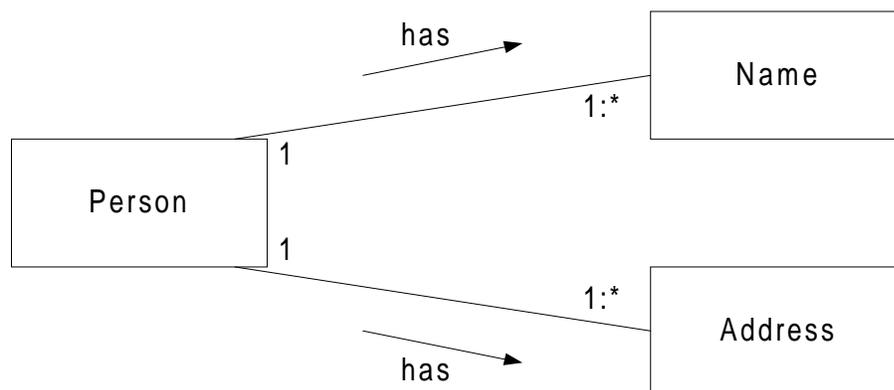


Figure 4.3. An example of *Composition* using OSA notation

For additional programmatic examples of composition in various object-oriented programming languages, see [Budd 1997].

Higher-level Patterns: *Forms of Reuse*

Other Related Patterns: *Inheritance*

Pattern 4.8 *Points of Interaction*

Motivation:

Once objects have been defined and created with static relationships and internal behaviors, a system doesn't tend to do much until the objects begin interacting. There must exist mechanisms through which various objects can interact dynamically. We refer to these as *Points of Interaction*.

Discussion:

In imperative programming, a *Point of Interaction* is a function or procedure call through which sequential control can be channeled, or through which instructions can be grouped into subroutines.

The conceptual autonomy of objects yields a much more flexible model of interaction, yielding concepts such as *Message Passing* and *Polymorphism*, which are described in this pattern set.

Higher-level Patterns: *Object-Oriented Program*

Lower-level Patterns: *Message Passing, Polymorphism*

Other Related Patterns: *Object Class, Forms of Reuse*

Pattern 4.9 *Message Passing*

Motivation:

Conceptually, objects live and operate as independent agents. In traditional imperative perspectives information is passed via parameters to functions (or via global variables). But in object-oriented programming there is no equivalent concept, since the function call presumes an imperative perspective. Instead, in object-oriented

programming objects interact by sharing information in the form of messages. This exchange of information is referred to as *Message Passing*.

Discussion:

In some ways *Message Passing* is a conceptual, design-level concept that in practice may look remarkably like a function call when used. This is especially true in languages like C++ that are firmly rooted in the imperative heritage. Still, the concept is important, and worthy of distinction, if only for the value that it lends to design and the mindset required for object-oriented programming.

Key to the idea of message passing is the concept that how an object deals with a particular message is not important to the sender, only that the object deals with it appropriately and returns some information in the form of a message. This approach is different from the imperative functional call approach, which presupposes a sequential interaction between caller and function. In an imperative program, the behavior of a program is viewed sequentially, with callers waiting for procedures to return before continuing.

The *Message Passing* concept can be used between distributed objects independent of the paradigm within which the distributed objects are written, creating an object-based system.

Examples:

In the following C++ example (Figure 4.4) the method `member_function` in the object class `object` is sent a message consisting of `arglist`. This example is borrowed from [Eckel 1995].

```
object.member_function(arglist);
```

Figure 4.4. *Message Passing* in C++.

For additional programmatic examples of message passing in various object-oriented programming languages, see [Budd 1997].

Higher-level Patterns: *Points of Interaction*

Other Related Patterns: *Polymorphism*

Pattern 4.10 *Polymorphism*

Motivation:

In any system in which objects exist as instantiations of classes (with methods to operate on encapsulated data) there are situations in which a single message may have different meaning to different objects. *Polymorphism* refers to the ability of different objects to handle the same message in different ways. It can also be viewed as the ability of a single object to handle different messages or messages of different types.

Discussion:

As an illustration, assume that stepping on something is a message, and the brake and gas pedals are different objects. In this situation, *Polymorphism* deals with the ability of these objects to interpret a single message in different ways. When given the same message, the brake pedal causes the car to slow down while the gas pedal causes the car to speed up.

Similarly, programmatic objects need to be able to respond differently to different messages. For example, an object that prints other objects needs to be able to respond correctly no matter what type of object is given. It should print integers as well as strings.

Parametric *Polymorphism* refers to the capability of an object to accept messages of multiple types so long as the type is passed along as a parameter.

Ad hoc *Polymorphism* refers to the situation in which an object will accept messages of multiple types without the need to explicitly state the type. The most obvious example of this is when two values are added together. A plus sign will add values of any types without having to explicitly state “Add Integers”. In fact, in this case the ad hoc

Polymorphism extends to doing automatic type casting so that values are converted to compatible types before being added where possible. This also involves overloading of an operator, such that the behavior of the operator will vary, depending on the type of the object being operated upon.

Examples:

In the following example (Figure 4.5) , the “+” has been overloaded to deal correctly with objects of all compatible types.

```
int    x;  
char   y;  
float  z;  
z := x + y;
```

Figure 4.5. An example of ad hoc *Polymorphism*.

The following (Figure 4.6) is an example in Leda that uses parametric polymorphism to determine the kind of elements contained in the `List` structure.

```
var  
  a : List[integer];
```

Figure 4.6. Parametric *Polymorphism* in Leda.

Higher-level Patterns: *Points of Interaction*

Other Related Patterns: *Message Passing*

5.0 Object-Oriented Analysis and Design Patterns

Object-oriented analysis and design is the current lingua franca of the software design community. The following statement by Booch [1994a] is a fitting introduction:

Let there be no doubt that object-oriented analysis and design is fundamentally different than traditional structured design approaches: it requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture. These differences arise from the fact that structured design methods build upon structured programming, whereas object-oriented design builds upon object-oriented programming. Unfortunately, object-oriented programming means different things to different people. As Rentsch [1982] correctly predicted, “My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.” Rentsch’s predictions still apply to the 1990s. [Booch 1994]

Object-oriented analysis and design grew out of structured design, and was aided considerably by the rise of object-oriented programming languages such as Smalltalk and C++. The following summary of object-oriented design provides an effective characterization of object-oriented design:

No longer is it necessary for the system designer to map the problem domain into predefined data and control structures present in the implementation language. Instead, the designer may create his or her own abstract data types and functional abstractions and map the real-world domain into these programmer-created abstractions. This mapping, incidentally, may be much more natural because of the virtually unlimited range of abstract types that can be invented by the designer. Furthermore, software design becomes decoupled from the representational details of the data objects used in the system. These representational details may be changed many times without any fallout effects being induced in the overall software system. [Wiener 1984]

Current methodologies are typically concerned with three attributes of objects: 1) their static description; 2) their internal behavior; and 3) their interaction with other objects. Dealing with these three aspects of objects tends to encompass the bulk of issues related to object-oriented design. Other issues, such as concurrency and distribution are

also dealt with in various object-oriented methodologies. The most popular object-oriented design methodologies have been the Object Modeling Technique [Rumbaugh 1991], Booch [1994a], Coad-Yourdon [Coad 1991], Shlaer-Mellor [Shlaer 1988] [Shlaer 1992], and OSA [Embley 1992]. The analysis and design patterns in this pattern set were mined from a careful survey and analysis of these design methodologies that was previously carried out by the author [Knutson 1997c] [Knutson 1997d] [Knutson 1997e]. This pattern set attempts to capture the most important elements of design and analysis in these popular methodologies. Throughout this pattern set, examples are in OSA notation and are borrowed from [Embley 1992].

- *Object-Oriented Analysis and Design*
 - *Object Relationship Model*
 - *Objects*
 - *Relationships*
 - *Constraints*
 - *Object Behavior Model*
 - *States*
 - *Triggers and Transitions*
 - *Actions*
 - *Timing Constraints*
 - *Object Interaction Model*
 - *Object Distribution*
 - *Object Concurrency*
 - *Real-Time Objects*

Figure 5.1. Hierarchy of object-oriented analysis and design patterns.

Pattern 5.1 *Object-Oriented Analysis and Design*

Motivation:

The purpose for analysis and design is to take a high-level view of a problem that needs to be solved and bring it down to the level that it can actually be solved through a programmatic mechanism. Within the object-oriented perspective the design process must interpret potential solutions in terms of objects and the relationships between them.

Object-oriented analysis and design seeks to model the world as a set of interacting objects.

Discussion:

Object-oriented analysis seeks to model the world as it is, rather than modeling a computer (as in the imperative perspective). As such, it is viewed as a much more generically applicable methodology for all kinds of software design. Conceptually there is no reason that one can't examine a problem using object-oriented analysis and then apply imperative design to create a solution. But in practice there is a great deal of information obtained through object-oriented analysis that directly influences design. Similarly, the design phase will often yield descriptions of classes that can then be implemented in a rather straightforward fashion in an object-oriented programming language. There is therefore a strong correlation between object-oriented programming elements and object-oriented design elements.

There are three broad views of systems that need to be explored during analysis and design: declarative (*Object Relationship Model*), behavioral (*Object Behavior Model*), and interactive (*Object Interaction Model*).

Booch [1994a] defines object-oriented design as “a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.” He defines object-oriented analysis as “a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.” The design and analysis process involves viewing the world from an object-oriented perspective, and translating this view into solutions that can be solved through object-oriented programming.

Examples:

The most popular object-oriented design methodologies have been Object Modeling Technique [Rumbaugh 1991], Booch [1994a], Coad-Yourdon [1991], Shlaer-Mellor [1988] [1992], and OSA [Embley 1992].

Lower-level Patterns: *Object Relationship Model, Object Behavior Model, Object Interaction Model, Object Distribution, Object Concurrency, Real-Time Objects*

Pattern 5.2 *Object Relationship Model*

Aliases: *Static Declarative Model*

Motivation:

Within the object-oriented perspective, classes can be declared as sets of data and methods that operate on them. Once classes have been designed, they can be combined in various ways through subclassing to create other classes. All of these elements are part of a class declaration, or a *Static Declarative Model* of a system.

Discussion:

A *Static Declarative Model* will include descriptions of objects and the relationships between them. In order to create an object-oriented solution to a problem, you have to be able to visualize the problem in terms of objects interacting. The *Object Relationship Model* attempts to capture a view of the system in terms of objects related in a static declarative fashion, irrespective of individual object behavior or the interactions between live objects.

Examples:

The following example (Figure 5.2) shows a simple *Static Declarative Model* in OSA notation. This example is borrowed from [Embley 1992].

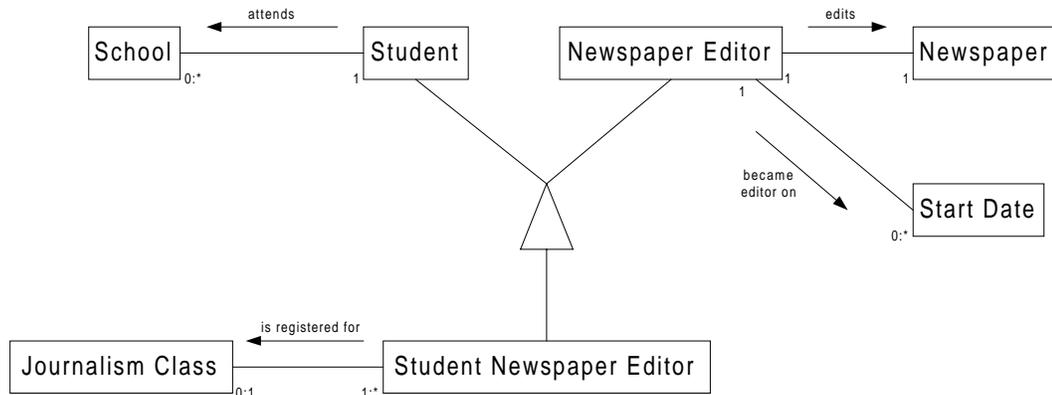


Figure 5.2. An example of a *Static Declarative Model* involving multiple inheritance.

Higher-level Patterns: *Object-Oriented Analysis and Design*

Lower-level Patterns: *Objects, Relationships, Constraints*

Other Related Patterns: *Object Behavior Model, Object Interaction Model, Object Distribution, Object Concurrency, Real-Time Objects*

Pattern 5.3 *Objects*

Motivation:

In object-oriented analysis and design, system behavior is viewed in terms of the interactions between individual *Objects*. One of the keys in designing a system from an object-oriented perspective is to determine what the classes and *Objects* in the system are going to be.

Discussion:

According to Booch [1994a], “The purpose of identifying classes and objects is to establish the boundaries of the problem at hand. Additionally, this activity is the first step in devising an object-oriented decomposition of the system under development.”

This means that we take a given problem, and seek to view it in terms of *Objects*. We conceptualize the static view of *Objects* by describing classes, containing data and

methods. We conceptualize the dynamic of view of the system by describing individual *Objects* with behavior and interactions with each other (responses to messages).

Objects may be used to represent tangible things, roles, events, interactions, people, places, things, organizations, concepts, events, structure, and anything else that doesn't fall neatly into these categories.

Higher-level Patterns: *Object Relationship Model*

Other Related Patterns: *Relationships, Constraints*

Pattern 5.4 Relationships

Motivation:

Objects alone are not only uninteresting, but fail to provide meaningful mechanisms for building complex systems. *Relationships* between objects can be established as a way of sharing and reusing characteristics and aspects of objects.

Discussion:

The most common situation in which relationships are sought is during the object-oriented decomposition of a system. By analyzing relationships that exist in nature, and mapping this to objects we can more accurately model the world. There are many ways to view the relationships between objects, but the following five are important enough to specifically mention and describe.

1. Generalization (Is a). This relationship corresponds to inheritance in object-oriented programming. It can also be viewed as a superset object class. As an example, human is a generalization of man, woman or child. The concept of "human" encapsulates certain key concepts in a general way that applies to any and all subsets.

2. Specialization (Is a). This is the other side of generalization. These are the subsets. Man, woman and child are all specializations of human.

3. Aggregation (Is Part of). This relationship corresponds roughly to composition in object-oriented programming. A wheel is part of a car.

4. Composition (Is Made of). This is the view of aggregation from the other side. A car is made of wheels, chassis, etc.

5. Association (Is Member of). This view is used when a set of objects that are not otherwise related via inheritance ought to be viewed as a single object. For example, a particular group of customers can be targeted for some promotion.

All of these relationships can be viewed in one-to-one, one-to-many, many-to-one, and many-to-many forms, lending both complexity in the representation as well as power to capture the real world.

Examples:

The following example (Figure 5.3) shows a very simple case of generalization and specialization in OSA notation.

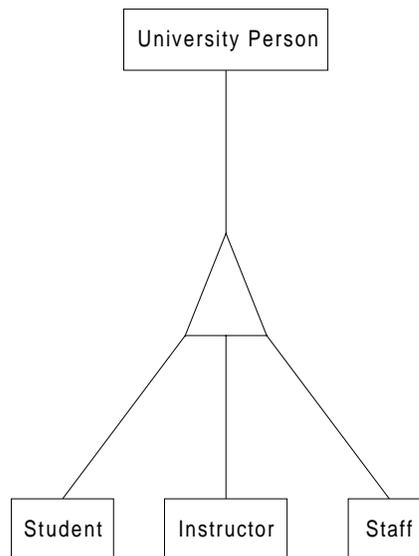


Figure 5.3. An example of generalization-specialization.

The following example (Figure 5.4) shows aggregation in OSA notation. The numbers associated with Body, Engine, and Wheel are cardinality constraints, which are explained in the next pattern (*Constraints*).

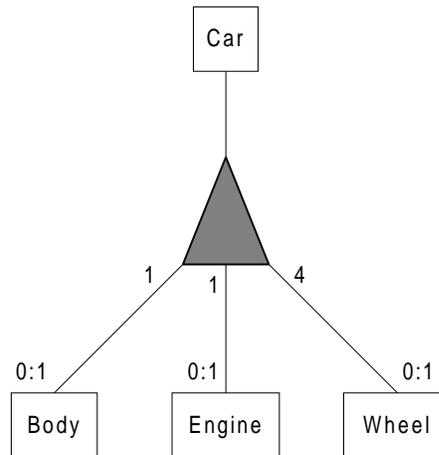


Figure 5.4. An example of aggregation.

Higher-level Patterns: *Object Relationship Model*

Other Related Patterns: *Objects, Constraints*

Pattern 5.5 Constraints

Motivation:

When designing object classes and dealing with the relationships between them, *Constraints* are necessary to draw tight enough bounds around the relationships so that the resultant design is meaningful. Representations of object-oriented analysis and design typically include various forms of *Constraints*.

Discussion:

There are typically limitations to the relationships between objects in any system. For example, a class might be limited to 20 available slots, or a message queue might have a maximum window size of seven packets in its buffers. To create an effective design, this clarifying information is critical to express, otherwise the description of the object will not be adequate.

There are several different kinds of relationship constraints, including General Constraints, Membership Constraints, and Cardinality Constraints. These are briefly described below.

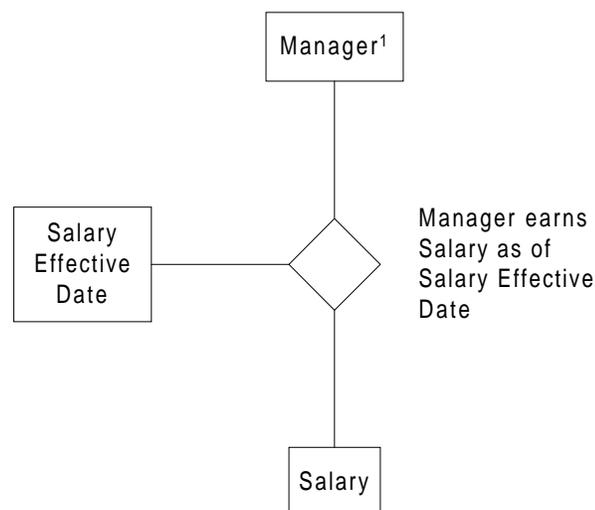
General Constraints are statements that restrict the membership of one or more object classes or relationship sets.

Membership Constraints are additional constraints that clarify the membership relationships defined elsewhere.

Cardinality Constraints place actual numerical limits on participation in a particular relationship class. Figure 5.4 in the previous pattern provides an example that uses Cardinality Constraints.

Examples:

The following example (Figure 5.5) shows a relationship that includes *General Constraints*. This example uses OSA notation. This example is borrowed from [Embley 1992].



¹A Manager's Salary never decreases.

Figure 5.5. An example of *General Constraints*.

Higher-level Patterns: *Object Relationship Model*

Other Related Patterns: *Objects, Relationships*

Pattern 5.6 Object Behavior Model

Motivation:

Objects not only exist, they do things. Each object has behavior. The *Object Behavior Model* describes the internal behavior for each object in the system.

Discussion:

One way of viewing the *Object Behavior Model* is to see the behavior as a kind of job description for an object—it describes what an object must do within a system. The mechanisms for representing an object’s behavior vary from one methodology to another. But commonly a behavioral description will include states (see *States*), transitions between states and the triggers that cause the transitions (see *Triggers and Transitions*), *Actions*, and other constraints (including timing constraints).

The *Object Behavior Model* has strong roots in the process descriptions of structured analysis and design.

Higher-level Patterns: *Object-Oriented Analysis and Design*

Lower-level Patterns: *States, Triggers and Transitions, Actions, Timing Constraints*

Other Related Patterns: *Object Relationship Model, Object Interaction Model, Object Distribution, Object Concurrency, Real-Time Objects*

Pattern 5.7 States

Motivation:

In order to model an object’s behavior we have to recognize that objects do different things at different times under different circumstances. We can conveniently use *States* to represent this behavior.

Discussion:

While *States* can effectively model behavior of object-oriented and imperative programs, it is not sufficient to model functional programs, since the concept of *States* is inconsistent with the nature of functional programming.

The *State* of an object represents its status, phase, situation, or activity. Once we view the behavior of an object as a series of transitions between states under various circumstances, we can apply elements of state theory to the task of design and analysis.

The various *States* of an object can be collected into state diagrams or state nets that help describe the behavior of the object.

Examples:

The following example (Figure 5.6) shows a partial state net for a robot object in OSA notation. This example is borrowed from [Embley 1992].

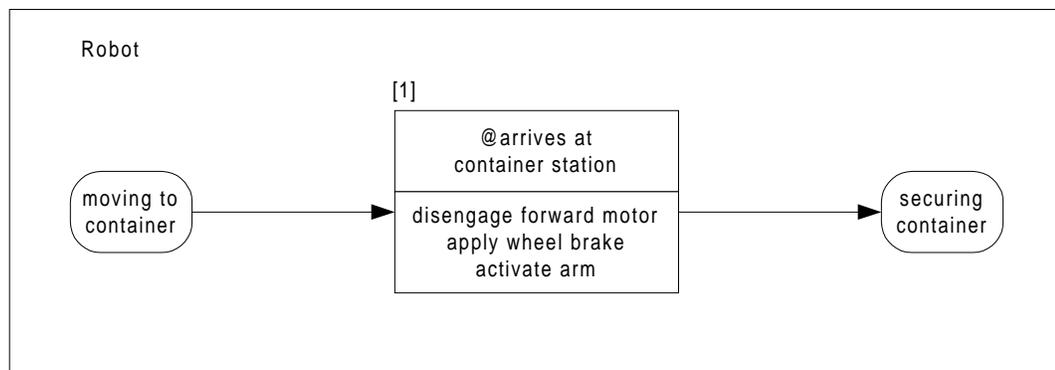


Figure 5.6. An example of a state net.

Higher-level Patterns: *Object Behavior Model*

Other Related Patterns: *Triggers and Transitions, Actions, Timing Constraints*

Pattern 5.8 *Triggers and Transitions*

Motivation:

Objects can exist in several different states. There are, naturally, *Transitions* between these states, and *Triggers* that induce these *Transitions*. To fully describe the behavior of an object, it is necessary to also describe these *Triggers* and *Transitions*.

Discussion:

Transitions can be viewed either as events that take no time (instantaneous movement from one state to another) or as a separate transitional states that take some time to pass through. For example, commuting in the morning could be seen as a *Transition* from home to work that takes some time. If the *Transition* itself is not important to the description of the system, this abstraction works fine. But one could just as easily view the commute itself as state that is entered into at home, and exited from at work.

Triggers are the stimuli that cause *Transitions*. In object-oriented systems this may commonly take the form of a message received that causes an object to change states. It could also be viewed as some external interrupt to the system that causes a *Transition*. Many systems could be modeled to include some interrupt from outside the system (keyboard input by a user, hardware signals from some instrument, etc.).

Examples:

The example in Figure 5.6 illustrates a simple state net in which a robot in the moving state receives a *Trigger* (arrives at the station) causing a *Transition* to a different state (securing the container).

Higher-level Patterns: *Object Behavior Model*

Other Related Patterns: *States, Actions, Timing Constraints*

Pattern 5.9 *Actions*

Motivation:

As objects move through states, they perform *Actions*. It is important to capture this information within the state nets that describe the behavior of a system.

Discussion:

Actions describe what an object does when it is in a particular state. The *Actions* may be triggers on other objects that cause them to change state. An *Action* may also include something that changes the state of the object itself. The kinds of *Actions* possible are unlimited.

Higher-level Patterns: *Object Behavior Model*

Other Related Patterns: *States, Triggers and Transitions, Timing Constraints*

Pattern 5.10 *Timing Constraints*

Motivation:

There are situations in which the timing of certain events can be critical to the correct operation of a system. When time is an important element of behavior, it may be necessary to add *Timing Constraints* to state nets to specify timing requirements. *Timing Constraints* may be applied to triggers, actions, states, and state-transitions paths.

Discussion:

Timing Constraints are a relatively new phenomenon, rising with the advent of interactive systems and environments.

In some systems *Timing Constraints* are imposed by the designer. In other systems, timing constraints are inherent in the description itself. In either case, *Timing Constraints* can be applied to any part of a state net.

Examples:

In the following example (Figure 5.7) a *Timing Constraint* is applied to the duration of a state in OSA notation. This example is borrowed from [Embley 1992].

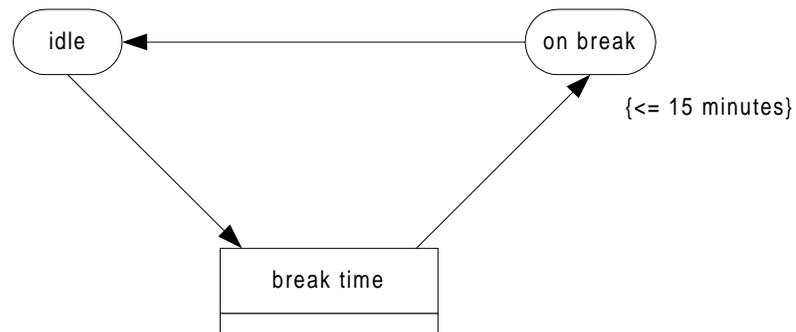


Figure 5.7. *Timing Constraint* applied to the duration of a state.

Higher-level Patterns: *Object Behavior Model*

Other Related Patterns: *States, Triggers and Transitions, Actions*

Pattern 5.11 <i>Object Interaction Model</i>

Motivation:

One of the most significant benefits of the object-oriented model is that objects can be designed somewhat independently, with individual behaviors. These objects can then interact together and handle messages that are exchanged. The *Object Interaction Model* helps describe these interactions.

Discussion:

Interactions between objects can take many forms. We can view these interactions from the perspective of two broad kinds of interactions—synchronous and asynchronous.

Synchronous interactions are those that occur in lockstep. Examples of this might include messages delivered from one object to another, while the sender waits for a response.

Asynchronous interactions do not occur in lockstep, and there are few if any constraints on when a sent message will be received by the intended object (barring timing constraints). This will typically involve some kind of intermediate store to contain the messages. The store may take the form of a queue or some other mechanism that can be written to by one object while being read by another object.

Examples:

The following example (Figure 5.8) shows an interaction diagram in OSA notation. In this example, there is an intermediate repository, so the interaction is asynchronous. This example is borrowed from [Embley 1992].

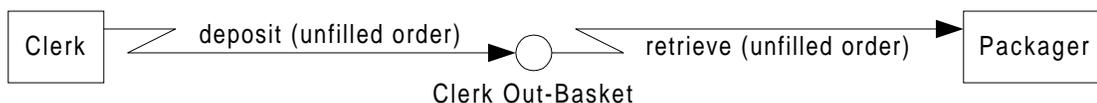


Figure 5.8. Interaction diagram with an intermediate repository.

Higher-level Patterns: *Object-Oriented Analysis and Design*

Other Related Patterns: *Object Relationship Model, Object Behavior Model, Object Distribution, Object Concurrency, Real-Time Objects*

Pattern 5.12 Object Distribution

Motivation:

The computing world in which we now live is a largely distributed one. When designing software, it is difficult to avoid some discussion of distribution, even when the application being developed is for a stand-alone system. The issues become more intense

when the application under design is a distributed application with pieces in different physical locations.

Discussion:

Object distribution is a relatively new phenomena, growing out of the proliferation of local-area networks (LANs), and wide-area networks (WANs). The advent of the World-Wide Web and the Java programming language has pushed object distribution even further into the forefront of issues that must be dealt with in object-oriented design.

A number of architectural patterns exist that deal with distribution, including Client-Server Computing, Object Brokers, Pipes and Filters, and Blackboard (see [Buschmann 1996] for discussions of these architectural patterns). These approaches encapsulate ways of organizing objects in a distributed fashion. For true distribution, some protocol object or layer must be involved in the design for data to move from one location to the next.

Higher-level Patterns: *Object-Oriented Analysis and Design*

Other Related Patterns: *Object Relationship Model, Object Behavior Model, Object Interaction Model, Object Concurrency, Real-Time Objects*

Pattern 5.13 Object Concurrency

Motivation:

This issue is somewhat related to object distribution, but in a different way. While object distribution involves concurrent objects running on different machines across some kind of network (or on different processors across some other communication mechanism), *Object Concurrency* involves the ability of multiple objects to run concurrently on the same system, or for a single object to have intraobject concurrency (for example, multi-threaded behavior).

Discussion:

Object Concurrency is a relatively new issue for computing, and has been influenced by the rise and proliferation of concurrent systems. Even the most popular operating system available today supports multi-tasking and multi-threading.

Two different kinds of concurrency need to be dealt with: interobject concurrency and intraobject concurrency.

Interobject concurrency describes the ability of multiple objects to live in a system concurrently. This capability is extremely system-sensitive, but most systems in operation today are multi-tasking systems. For examples the various Unix incarnations and the Microsoft desktop operating systems (Windows 3.11/95/NT) all permit multi-tasking, in which individual objects can be created and live independently. A classic example of these windowing systems involves an object owning a message queue, and not doing anything until the receipt of a message, at which time it comes alive, services the request, and goes back to sleep.

Intraobject concurrency is more difficult, both in terms of programming environment as well as analysis and design methodology. First of all, intraobject concurrency allows an object to be in more than one state at a time. For example, as a student object, an individual could conceivably be a master's student at one school waiting to defend his thesis, while simultaneously attending doctoral classes at another school. Real life accommodates such concurrency, but not all modeling methodologies offer the same accommodation. Another form of intraobject concurrency involves a single object doing two different things at the same time (for example walking and chewing gum). In reality we easily handle both the physical and conceptual task. But when modeling software the typical approach is for the designer to compose the person with a mouth and legs, so that the mouth can chew gum while the legs walk. This represents a sort of artificial break down of objects that is somewhat unnatural, but this is the common solution with most object-oriented analysis and design methodologies available today. In some respects it would be natural for the person to be described as both walking and chewing gum, since in our experience these are sometimes related, and not completely independent.

Examples:

The following example (Figure 5.9) shows intraobject concurrency in OSA notation. Not all object modeling techniques permit intraobject concurrency, but OSA permits the Copy Machine in this example to move into two separate states at the same time. This example is borrowed from [Embley 1992].

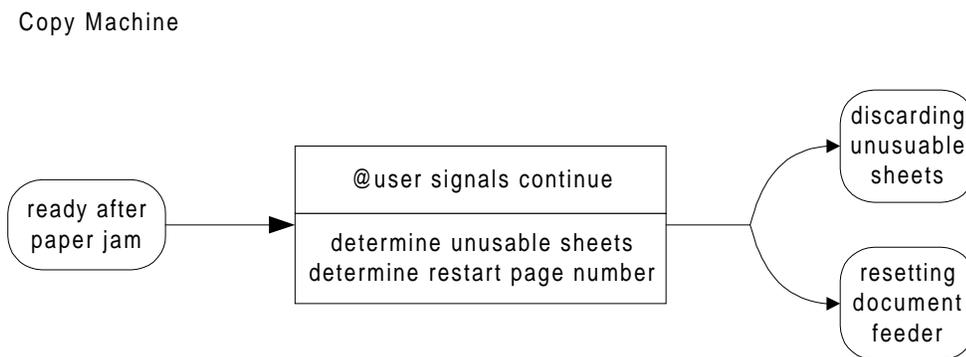


Figure 5.9. Modeling intraobject concurrency using multiple subsequent states.

Higher-level Patterns: *Object-Oriented Analysis and Design*

Other Related Patterns: *Object Relationship Model, Object Behavior Model, Object Interaction Model, Object Distribution, Real-Time Objects*

Pattern 5.14 Real-Time Objects

Motivation:

Some systems are so timing intensive that the entire architecture must accommodate the overall timing constraints in many ways. These systems are referred to as real-time systems, and objects that function within the timing-critical parts of these systems are *Real-Time Objects*.

Discussion:

Most embedded devices tend to have real-time needs. For example, digital cameras have to open and close shutters and process an image in a fixed amount of time. It is not acceptable in these systems to wait with the shutter open while the processor figures out what to do with the image.

Real-Time Objects are different from timing constraints in the following way. Timing constraints are used to describe timing issues, and can be used to help describe *Real-Time Objects*. But in addition, *Real-Time Objects* must deal with overall design principles of real-time programming, and so require the application of additional design methodologies specific to real-time systems.

Several design methodologies for real-time systems have been proposed [Gomaa 1993] [Kelly 1987]. These methodologies are not strictly object-oriented, but typically involve some decomposition of the system into interacting tasks, which lends itself well to object-oriented analysis and design. Key principles include the following:

Tasks should be separated in any of the following situations: 1) there is dependency on asynchronous I/O devices; 2) there are time critical activities; and 3) there are heavy computational requirements.

Tasks should be grouped together in any of the following situations: 1) there is functional cohesion; 2) there is temporal cohesion (time-locked behavior); 3) there is period execution (cyclic activities).

These principles guide the decomposition of a system into objects in such a way that the overall real-time requirements of the system can be more easily met. In some situations these criteria for grouping or separating tasks are non-intuitive outside of the real-time perspective. For example, real-time design will sometimes lead to grouping objects that have no commonality other than time-locked behavior. It might also lead to separating very related tasks simply because one has dependency on asynchronous I/O devices.

Higher-level Patterns: *Object-Oriented Analysis and Design*

Other Related Patterns: *Object Relationship Model, Object Behavior Model, Object Interaction Model, Object Distribution, Object Concurrency*

6.0 Functional Programming Patterns

In 1978 John Backus, in his Turing Award lecture, coined the phrase “von Neumann bottleneck” to describe the problem that exists when imperative programming languages mimic target hardware so closely that the language becomes a mere abstraction of the machine [Backus 1978]. The functional programming perspective (sometimes referred to as applicative) attempts to deal with data and functions in such a way that the dependencies on elements of the von Neumann architecture (such as memory locations for data and sequential execution for code) are eliminated.

Kamin defines functional (or applicative) programming in the following way:

A style of programming characterized by the (usually recursive) definition of functions over recursively defined data, avoiding iteration and side-effects. The more modern use of these terms extends this meaning by adding an emphasis on the use of higher-order functions. [Kamin 1990]

The earliest functional languages were LISP [McCarthy 1960] [McCarthy 1962] [Wilensky 1984], and APL [Iverson 1962] [Polivka 1975]. These languages share the characteristic of having elevated functions to the level of values, thus permitting a greater freedom for functional construction. LISP was the first language to employ some of the mathematical concepts of λ -calculus, therefore moving away from imperative programming and closer to theoretically pure functional composition. APL was the first language to provide access to “large values” or, in other words, to provide native language capabilities to manipulate complex data structures, typically based upon arrays.

Other functional programming languages include Scheme [Steele 1978], ML [Wikstrom 1987] [Milner 1990] [Paulson 1991], Haskell [Hudak 1992] [Davie 1992] [Thompson 1996], and Hope [Bailey 1990].

In this chapter, we have attempted to capture the essential elements of functional programming in pattern form. This pattern set will be used as a building block to analyze aspects of functional design, and to explore multiparadigm programming and design. We do not expect this pattern set to be exhaustive in revealing everything that could be said about functional programming (although it could provide an excellent foundation on which to build such an analysis).

This taxonomy of functional programming elements was culled from a thorough analysis of programming elements in selected functional languages including Lisp, APL, Scheme and Haskell. The essential programmatic elements were identified, and are listed in the pattern hierarchy below (Figure 6.1). Each of these elements is discussed in greater detail in the patterns that follow.

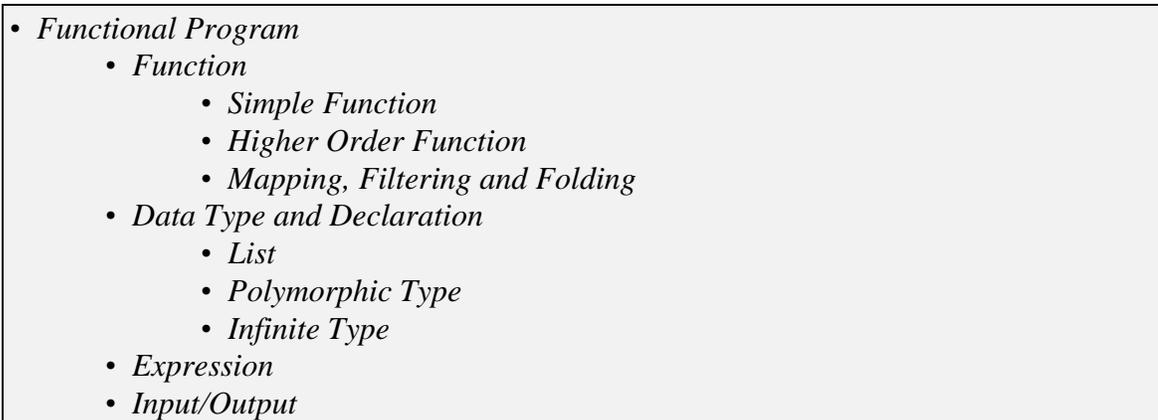


Figure 6.1. Hierarchy of Functional Programming Patterns.

Pattern 6.1 *Functional Program*

Alias: *Applicative Program*

Motivation:

The *Functional Programming* perspective attempts to represent programming in the same way mathematicians represent computation, that is, as the composition of functions to achieve some result. This eliminates dependencies on the von Neumann architecture and tends to eliminate side effects.

Discussion:

Mathematical computation does not inherently produce side effects, and yet imperative programming relies heavily on side effects—temporary results are held in memory, all manipulation happens on memory locations, and the ultimate answers to

programming questions are stored in memory locations. The *Functional Programming* perspectives eschews side effects, and *Functional Programming* languages provide a perspective on computation that does not include side effects (despite the fact that these programs run almost exclusively on von Neumann architectures, and are achieved through side effects underneath).

For mathematicians, *Functional Programming* is extremely natural and intuitive. It is easily adapted to the use of recursive functions, and so is a natural mechanism for representing problems that fit naturally in the domain of recursive function theory. As Davie puts it, “In the last chapter, we introduced two examples of recursive functions. We shall see many such examples in this chapter (and in the rest of the book: it is central to this style of applicative programming) and we hope that readers feel that this is an easy and natural way to program” [Davie 1992]. This statement is probably true as long as the readers in question are mathematicians comfortable with recursive function theory. For deeply entrenched imperative programmers, *Functional Programming* is challenging, since it removes computation from the computer architecture and deals with it in the mathematical world. Davie also states, “There is some hope that functional programming is a good medium for prototyping problems very fast” [Davie 1992]. Similarly, this is true where the individual is comfortable with the mathematical concepts embodied in *Functional Programming*. However, for imperative programmers, accustomed to thinking about the order in which problems are solved, it can be very difficult to let go of such details and lay out a declarative solution to a problem.

Functional Programming relies on several key principles, for which patterns are included in this section. Many data types are supported, but there is a heavy reliance on the list structure. Data declarations are important, but do not associate variable names with memory, since such side effects are not part of *Functional Programming*. Functions are first class citizens, and can be used anywhere where data can be used. The other patterns in this pattern set detail other key principles that set *Functional Programming* apart from other programming paradigms.

Lower-level Patterns: *Function, Data Type and Declaration, Expression, Input/Output*

Other Related Patterns: *Imperative Program, Object-Oriented Program, Logic Program*

Pattern 6.2 Function

Motivation:

What would “functional” programming be without *Functions*? The fundamental unit of functional programming is the *Function*, and the theoretical origins of *Functions* lie in recursive function theory. A *Function* is an object that returns a value (an output or result) when it is applied to its inputs. Inputs are also known as parameters or arguments.

Discussion:

One of the fundamental principles of functional programming is that *Functions* are first class values. That is, they can be created and used in any situation in which other programming elements (such as data structures) can be used. This naturally leads to the creation of expressions that are composed of *Functions*. These composite *Functions* can be combined indefinitely, leading to more complex higher-order functions. In this way, *Functions* are the reusable building blocks of functional programming.

The use of *Functions* leads to a certain kind of software reuse, since individual *Functions*, once created, can be applied to similar problems. The functional programmer has a tool set of sorts, comprised of useful (and reusable) *Functions*. In addition, polymorphic *Functions* can be applied to similar situations of various types, which further promotes reuse.

The basis for functional programming is recursive function theory, so the use of recursive *Functions* is very important to functional programming. Recursion (rather than iteration) is typically viewed as the acceptable mechanism for traversing data structures.

Examples:

The following are examples of *Functions* from several functional programming languages. Figure 6.2 shows the use of *Functions* in LISP. This example is borrowed from [Wilensky 1984].

```

->(defun addthree (x) (plus x 3))
->(addthree 11)
14

```

Figure 6.2. Definition and use of a LISP *Function* to increment by three.

The following example (Figure 6.3) shows the use of *Functions* in APL. This example is borrowed from [Kamin 1990].

```

▽ Z ← fac n
Z ← × / t n
▽

```

Figure 6.3. An APL *Function* to perform factorials.

Lower-level Patterns: *Simple Function, Higher Order Function, Mapping, Filtering and Folding*

Higher-level Patterns: *Functional Program*

Other Related Patterns: *Data Type and Declaration, Expression, Input/Output*

Pattern 6.3 *Simple Function*

Motivation:

Simple Functions are primarily composed of fundamental programming language elements, without heavy reliance upon other functions. We can view *Simple Functions* as being on the opposite side of the spectrum from higher-order functions.

Discussion:

In practice, it is difficult to draw a clean line and categorize all functions as either simple or higher-order. In fact, almost all functional programming consist to one degree or another of higher-order functions. However, it is important to point out that there are essential functions (*Simple Functions*) that comprise the fundamental building blocks of most functional programs.

Examples:

The following example (Figure 6.4) shows a *Simple Function* in Haskell. This example is borrowed from [Davie 1992].

```
>let area r = pi * r * r in
>area(5.0 + 0.7)?
102.070345315132
```

Figure 6.4. A Haskell *Function* to calculate the area of a circle.

Higher-level Patterns: *Function*

Other Related Patterns: *Higher Order Function, Mapping, Filtering and Folding*

Pattern 6.4 Higher Order Function

Alias: *Functional*

Motivation:

Functions are more than a fundamental building block for functional programming—functions are first class citizens. This means that they can be used in any situation where any other data type can be used. *Higher-Order Functions* accept functions as arguments and/or return functions as results.

Discussion:

Functions can be passed as arguments to other functions, combined (or composed) into other functions, or can be returned as the result of a function. This use of functions as first-class citizens is unique to functional programming, but is moving into other paradigms. The concept of setting up a function that does generic things, depending on the function that is passed in to it is now used by C++ in its Standard Template Library (STL). This lends itself to reuse patterns such as *Functional Backpack* [Knutson 1996a].

Examples:

The following example (Figure 6.5) shows the use of *Higher Order Functions* in Haskell. In this example, the function `iter` receives an integer and a function as parameters, and returns a complex function created through composing the function `f` with itself the number of times indicated. This example is borrowed from [Thompson 1996].

```
iter :: Int -> (t -> t) -> (t ->)
iter 0 f = id
iter n f = f >.> iter (n-1) f
```

Figure 6.5. A *Higher Order Function* in Haskell.

Higher-level Patterns: *Function*

Other Related Patterns: *Simple Function, Mapping, Filtering and Folding*

Pattern 6.5 *Mapping, Filtering and Folding*

Motivation:

Functional programming depends heavily on the list structure. Not surprisingly, various ways of manipulating lists have sprung up and have become somewhat standard approaches to doing it. *Maps*, *Filters*, and *Folds* are the most common methods, and include various ways of manipulating lists.

Discussion:

These three concepts have become somewhat fundamental to list manipulation, and are very powerful tools. These are not, strictly speaking, always programming language elements. But the use of these functions has become as important to functional programming as `printf` or other libraries are to imperative languages like C.

Map applies a function to every element of a list. This is very similar to the *Functional Backpack* pattern we described in [Knutson 1996a].

Example:

The following example (Figure 6.6) shows the declaration and usage of the `map` function in Haskell. This example is borrowed from [Thompson 1996].

```
>map :: (t -> u) -> [t] -> [u]
>map f []      = []
>map f (a:x) = f a : map f x

>map times2 [2,3,4]
[4,6,8]
>map isEven [2,3,4]
[True,False,True]
```

Figure 6.6. The `map` function in Haskell.

Discussion:

Filter picks out of a list those elements that have a particular desired property.

Example:

The following example (Figure 6.7) shows the declaration and usage of the `filter` function in Haskell. This example is borrowed from [Thompson 1996].

```

>filter :: (t -> Bool) -> [t] -> [t]

>filter p [] = []
>filter p (a:x)
> | p a      = a : filter p x
> | otherwise =      filter p x

>filter isEven [2,3,4]
[2,4]

```

Figure 6.7. The `filter` function in Haskell.

Discussion:

Fold collapses the elements of a list using a binary operation. The operation is conceptually placed between each of the elements in the list. In situations where the order is important, *Fold* can be dealt with either as right *Fold* or as left *Fold*, depending on the desired result of the operation

Example:

The following example (Figure 6.8) shows the declaration and usage of the `fold` function in Haskell. This example is borrowed from [Thompson 1996].

```

>fold :: (t -> t -> t) -> [t] -> t

>fold f [a]      = a
>fold f (a:b:x) = f a (fold f (b:x))

>fold (||) [False,True,False]
True
>fold (*) [1..6]
720

```

Figure 6.8. The `fold` function in Haskell.

Higher-level Patterns: *Function*

Other Related Patterns: *Simple Function, Higher Order Function*

Pattern 6.6 Data Type and Declaration**Motivation:**

Data Types are programmatic conceptualizations of objects that can be manipulated by functions. Simple types can also be used to create more complex *Data Types*. Functional programs operate on data, but not on variables. A variable consists of a name associated with a memory location in an imperative program. Data in a functional program is a conceptual entity that exists in a declared way. *Data Declarations* are the mechanisms by which conceptual entities are explicitly declared in functional programs.

Discussion:

Despite the fact that functional programming attempts to model computation and not computers, the origin of *Data Types* is very similar to the imperative approach, where *Data Types* are largely abstractions of memory locations.

The most pervasively used *Data Type* in functional programming is the list type. Lists accommodate a recursive approach to computation, which is important to avoid side effects.

Functional programming accommodates various kinds of types. Algebraic types include base types, such as integer, floating point, boolean and character. It also includes composite types, such as tuples, lists, and functions. In addition, enumerated types, product types, recursive types, and polymorphic types are also supported. All of these types can be encapsulated in type classes.

Some functional languages, such as Haskell, support type inference, which permits the programmer to ignore the explicit typing of data, as long as the representations of data are internally consistent.

Examples:

The following example (Figure 6.9) shows the use of explicit typing of functions in Haskell. This example is borrowed from [Thompson 1996].

```

>Type Pair = (Int, Int)
>addTwo :: Pair -> Int
>addTwo (first, second) = first + second

```

Figure 6.9. Explicit typing of functions in Haskell.

Discussion:

Data in functional programming can be viewed as a constant associated with a name. A name can be declared to be associated with some value, but will never be a variable in the imperative sense, since it can never change.

This approach to data is one of the functional programming concepts that is most foreign to imperative programmers. Data can be declared, but never changed. The reason is that the intermediate results of operations exist conceptually, but are never changed in practice.

Example:

If we let x represent the value 5, we might represent this as $x < -5$ (in a non-language specific pseudo-code). If we then seek an answer involving two separate functional compositions, such as $f(g(x))$, we might write it as $f(g(x < -5))$. Let's assume that g is a function that multiplies integers by 2, and f is a function that adds 4 to integers. We can view this composite function at four different levels, all of which conceptually co-exist. The integer 5 has value independent of all operations. The label x refers to the integer 5, and always will. $g(x < -5)$ will be equivalent to the integer value 10, and $f(g(x < -5))$ will be equivalent to the integer value 14. All of these intermediate values will co-exist, and none of these will be destroyed. This is in stark contrast to the same operation employed in an imperative way, in which these individual values will no longer exist after the operation is performed.

The following example (Figure 6.10) shows the declaration and initialization of a data structure in LISP. This example is borrowed from [Wilensky 1984].

```
-> (setq text1 (vector 'Mary 'had 'a 'little 'lamp))  
vector[5]
```

Figure 6.10. The declaration and initialization of a data structure in LISP.

Lower-level Patterns: *List, Polymorphic Type, Infinite Type*

Higher-level Patterns: *Functional Program*

Other Related Patterns: *Function, Expression, Input/Output*

Pattern 6.7 List

Motivation:

The *List* data type is very important in functional programming. While it is available in imperative programming, it has properties that make it particularly useful for functional programming.

Discussion:

In general a *List* is a collection of objects of any type. When a *List* contains elements that are all of the same type, this is a homogeneous *List*. In general, this is the default usage of *Lists* in functional programming. Heterogeneous *Lists* are also supported. In Haskell, these are known as tuples.

Since there is a strong mathematical basis for functional programming, including recursive function theory, *Lists* provide an easy mechanism for recursive behavior, since *Lists* can be dealt with recursively in a very natural way. In addition, *Lists* are not limited to particular types, and so provide generic mechanisms for manipulating data, while allowing for type inference within a given list. So important is the *List* structure in functional Programming that the LISP (LISt Processing) language is based almost entirely upon it as a fundamental data type.

Lists may be used to facilitate generic functions, in which the *List* traversal mechanism carries a particular function to apply to *List* elements. The function may be different depending on the type of the *List* elements.

Examples:

The following example (Figure 6.11) shows the use of *Lists* in LISP. This example is borrowed from [Wilensky 1984].

```
-> (cons 'a (cons 'b (cons 'c nil)))
(a b c)
-> (cons 'a (cons (cons 'b (cons 'c nil)) (cons 'd nil)))
(a (b c) d)
```

Figure 6.11. Creating *Lists* in LISP.

The following example (Figure 6.12) shows the use of *Lists* in Haskell. This example is borrowed from [Thompson 1996].

```
>[3,4,5] :: [Int]
>[2] ++ [3,4,5]
[2,3,4,5]
```

Figure 6.12. Creating and manipulating *Lists* in Haskell

Higher-level Patterns: *Data Type and Declaration*

Other Related Patterns: *Polymorphic Type, Infinite Type*

Pattern 6.8 Polymorphic Type**Motivation:**

In mathematical reasoning, there is not a strong concept of type. Certainly types exist (for example, real numbers, integers, natural numbers). But imperative types grew out of a need to efficiently represent data in memory stores. With the functional programming perspective operating irrespective of data stores, types tend in general to be more polymorphic, with less constraint for particular types to be nailed down ahead of time. This leads to a greater reuse among functional components, since the same functions can be used on different data types, irrespective of the types of elements to be pushed through them.

Discussion:

A good example of *Polymorphic Typing* (and perhaps its most common usage) is the generic list structure. Such a structure can be operated on in a generic way, irrespective of the types of its elements. For example, a list of items can be examined, and its head returned. Or its tail could be returned. Or the entire list except for head or tail. Or the length of the list can be computed. All these operations are oblivious to the types of the elements in the list and can operate in a polymorphic fashion. These functions can therefore be reused on any list structure, irrespective of the type of the elements in the lists.

Examples:

The following example (Figure 6.13) shows the use of *Polymorphic Types* in Haskell. This example shows both the polymorphic nature of the `length` function as well as the recursive mechanism used to compute the length of a list of any type. This example is borrowed from [Thompson 1996].

```
>length [] = 0
>length (a:x) = 1 + length x
```

Figure 6.13. The *Polymorphic Type* length function in Haskell.

Higher-level Patterns: *Data Type and Declaration*

Other Related Patterns: *List, Infinite Type*

Pattern 6.9 *Infinite Type*

Motivation:

Many functional languages use lazy evaluation (as opposed to eager evaluation). Stated simply, this means that expressions are evaluated only when they are needed. This also means that they are only evaluated to the extent that they are needed. This principle is important to functional programming, since it preserves the mathematical purity of the paradigm, and permits the conceptualization of *Infinite Types*.

Discussion:

In imperative programming, a data structure has to be represented entirely in memory to be manipulated. As structures grow very large, this is very impractical and inefficient. But functional programming is motivated by the desire to manipulate mathematical concepts as effectively as possible. Lazy evaluation allows elements of certain types to be manipulated, even when the element could not actually be completely represented in physical memory if that were required. The most common case in which this becomes important is with infinite lists. The use of infinite structures allows us to programmatically manipulate mathematical concepts (like the set of all even numbers, or the set of all prime numbers) that can't be dealt with in an imperative approach.

Examples:

Consider a situation in which an infinite series is represented as a list of the individual pieces that constitute the data element. We may want to manipulate each

individual piece of the infinite series and then effect some combination of elements that causes the infinite series to be collapsed into something more representable in a computer. With lazy evaluation, we can represent an infinite list conceptually, and manipulate it in the same way, so long as we are not required to do something that explicitly traverses the entire list (which is, of course, logistically impossible).

The following example (Figure 6.14) shows a declaration of the infinite set of natural numbers in Haskell. In this example, an infinite series is defined where `f` is the function that propagates the series, and `a` is the starting point. This example is borrowed from [Thompson 1996].

```
>series f a = a : series f (f a)
>nats = series (+1) 0
```

Figure 6.14. An infinite series defined in Haskell.

Higher-level Patterns: *Data Type and Declaration*

Other Related Patterns: *List, Polymorphic Type*

Pattern 6.10 *Expression*

Motivation:

Functional *Expressions* are programmatic representations of mathematical expressions, and follow very similar forms. An *Expression* is formed by applying a function or operator to its arguments. The arguments can be literal values, or *Expressions* themselves. It is in *Expressions* that the bulk of the work is done in functional programming.

Discussion:

The treatment of *Expressions* in functional programming is unique when compared to expressions in other programming paradigms. *Expressions* in functional

programming have referential transparency. This means that the value of an *Expression* depends only on the values of its well-formed subexpressions. Any two *Expressions* that are equivalent can be exchanged without affecting the overall evaluation of the *Expression*. Another way of expressing this is to say that there are no side effects outside of a given *Expression*, or outside of the context in which an *Expression* is used.

Examples:

In a simple *Expression*, such as $((2+3)+4)$, there can be no side effects because each of the elements in the expression is a number. But if the expression is composed of functions, such as $((a+b)+c)$, then the issue of referential transparency becomes much more important. Each of these functions may be viewed as a subexpression, since each has other computations to perform that contribute to the final solution.

Higher-level Patterns: *Functional Program*

Other Related Patterns: *Function, Data Type and Declaration, Input/Output*

Pattern 6.11 <i>Input/Output</i>

Motivation:

Without *Input/Output* there is no way for a programmer to indicate desired values on which to perform operations, nor ways to view the results of the operation. The challenge with functional programming is that the functional model is conceptually at odds with *Input/Output*. *Output* is by its nature a side effect. To write an answer to disk is to store something in a memory location, and forfeit transparency. Similarly, to write something to a screen is to leave a side effect in the system. This seems quite innocuous to the imperative programmer, but to the functional programmer, it is non-trivial. It is particularly difficult when the functional perspective is held firmly, and side-effects avoided with religious fervor. The resultant compromise leads to various tricks that allow a functional programmer to get information in and out of the system without affecting the rest of the system.

Discussion:

There are two common approaches to resolving the Input/Output dilemma in functional programming. One approach is to view input and output as lists. This is sometimes referred to as the stream model. The advantage here is that the model is consistent with the rest of the functional programming model, and functions that operate on input and output can be reused. The disadvantage is that this approach is not particularly extensible or flexible when dealing with larger systems.

Another approach, commonly used is monadic *Input/Output*. With monads we actually define types that correspond to input and output mechanisms, and that can operate on them. Monads provide mechanisms for sequencing events, since interactions with users (*Input*) and *Output* devices branch into the time-sensitive domain. Monads attempt to capture the inherent side effects of *Input* and *Output* and encapsulate them in such a way that they do not adversely affect the rest of the system.

Examples:

The following example (Figure 6.15) shows a streams approach to the *Input/Output* problem of functional programming. This example is borrowed from [Thompson 1996].

```
>type Input = String
>type Output = String
>read :: Read t => String -> t
```

Figure 6.15. *Input/Output* using streams in Haskell.

The following example (Figure 6.16) shows an approach to the *Input/Output* problem of functional programming using monads. This example is borrowed from [Thompson 1996].

```
>getLine :: IO String
>putLine :: String -> IO ()

>readWrite :: IO ()
>readWrite = getLine >=> putLine
```

Figure 6.16. *Input/Output* using monads in Haskell.

Higher-level Patterns: *Functional Program*

Other Related Patterns: *Function, Data Type and Declaration, Expression*

7.0 Functional Analysis and Design Patterns

Those paradigms that are both well-established and popular (such as imperative and object-oriented programming) have equally well-established and popular methodologies for designing within the paradigm. Unfortunately, although functional programming is certainly established as a discipline, and popular enough among certain circles of computer scientists and mathematicians, there is no well-established and popular methodology for designing within a functional paradigm. The reason for this lack of a methodology for functional design strikes at the heart of the evolution of any design methodology, and so has applicability in our ultimate quest in this research for a multiparadigm design methodology.

Design methodologies tend to be somewhat market driven. Design methodologies typically are intended to make the tools and perspectives of a particular programming paradigm accessible to the masses, those for whom programming in the new paradigm is not necessarily intuitive but who, for one reason or another, need or want access to it. Indeed, for most of the early adopters of any new paradigm, the approaches and world views of the paradigm are already somewhat intuitive, otherwise they wouldn't have seen its value in the first place. The masses who ignore a new paradigm (or any new technology for that matter) are predominantly composed of those who can't see the applicability of the paradigm without some help. The largest market for popular books on design methodology (such as the current object-oriented design wave) is for those who see the market presence and growing popularity of the technology and don't want to be left behind, but need considerable help coming up to speed in using it.

The tendency of early proponents of a new technology to view its benefits and usage as intuitive is, I believe, not unique to functional programming.⁸ However, functional programming proponents seem to consistently express some surprise at (or at least lack of interest in) the notion of “functional design.” This is best typified by functional programming expert Simon Peyton-Jones' statement that “we don't really do

⁸ See *Crossing the Chasm* [Moore 1991] for an insightful analysis of the “psychographics” of technology adopters, from “innovators” and “early adopters” to the “late majority” and the “laggards”.

design” [Peyton-Jones 1996]. They don’t do design, I believe, because the experts—those on the leading edge of functional programming—apply design principles in an intuitive, non-explicit way. There is no question that design is occurring, it simply isn’t communicated, in general. Davie put it similarly when he wrote, “we hope that readers feel that this is an easy and natural way to program” [Davie 1992]. The reality is that for most programmers, raised and educated in the imperative mindset, functional programming is not necessarily “easy and natural”. However, for mathematicians untainted with prior exposure to imperative programming, functional programming might indeed seem “easy and natural”. In any case, functional programming lacks general popularity, and hence it lacks popular design methodologies.

As imperative and then object-oriented programming have risen in popularity, design methodologies have sprung up to support them. The pattern sets for imperative and object-oriented design and analysis (Chapters 3 and 5) drew heavily on existing popular methodologies, some of which are already decades old. The creation of this pattern set for functional design and analysis patterns has not benefited from a similar availability of resource material. It is, therefore less complete in its treatment than its imperative and object-oriented counterparts. On the other hand, it is more complete in its treatment of the subject than anything that has yet come out of the functional programming community.

Most of the design patterns in this set have been culled from conversations with functional programmers, and from small multi-sentence gems of design insight in Simon Thompson’s book *Haskell: The Craft of Functional Programming* [Thompson 1996]. Pipes and Filters is dealt with in depth as an architectural pattern in [Buschmann 1996] and is adapted here as a design pattern. This pattern set is a first attempt, and represents a potential area of future research.

- *Functional Analysis and Design*
 - *General Design Approaches*
 - *Adapting Similar Problems*
 - *Stepwise Refinement*
 - *Functional Data Design*
 - *Recursive Refinement*
 - *Type Refinement*
 - *Functional Design Perspectives*
 - *Pipes and Filters*
 - *Functional Composition*

Figure 7.1. Hierarchy of functional analysis and design patterns

Pattern 7.1 *Functional Analysis and Design*

Aliases: *Applicative Analysis and Design*

Motivation:

The functional perspective views a system as being composed of functions that interact in various ways to create a result, given some input. Functional problem analysis therefore involves reducing a problem into its requisite functional pieces. *Functional Design* involves designing a system using functions and the operations that combine those functions.

Discussion:

Each of the analysis and design techniques in this set of patterns involves some way of viewing a problem and designing a system as a set of interacting functions. The common thread of all these approaches is that none of them violate the basic principles of functional programming, and all of them yield a system of interacting functions.

Lower-level Patterns: *General Design Approaches, Functional Data Design, Functional Design Perspectives*

Other Related Patterns: *Imperative Design, Object-Oriented Design, Logical Design*

Pattern 7.2 *General Design Approaches*

Motivation:

Many design principles are paradigm-independent (or at least not tightly bound to a particular paradigm). Functional design relies heavily on several of these *General Design Approaches*.

Discussion:

While there are no design methodologies specific to the functional paradigm, there exist design approaches that rely on the modularization made possible by functions. These approaches have typically grown out of the imperative paradigm, but are used by functional programmers. The *Stepwise Refinement* pattern in this set is an example of this kind of *General Design Approach*.

One common design approach employed by functional programmers involves *Adapting Similar Problems*. This pattern is also included in this set.

Higher-level Patterns: *Functional Analysis and Design*

Lower-level Patterns: *Adapting Similar Problems, Stepwise Refinement*

Other Related Patterns: *Functional Data Design, Functional Design Perspectives*

Pattern 7.3 *Adapting Similar Approaches*

Motivation:

One of the guiding principles of functional programming is code reuse. We modularize things into functions so that they can be reused by other functions. In this way we shouldn't have to solve the same problem over and over and again. Design should follow similar principles. If a design problem has been solved once, it shouldn't have to be solved over and over again. Therefore, we should be able to look at existing problem solutions, and borrow from the design used in those solutions for the current design problem.

Discussion:

This approach to functional design is not particularly original, nor does it require any particular approach or methodology. It is simply guided by the principle that one should not invent that which has been done before. This is particularly true in functional programming, where the functions that result from one problem solution embody design principles, and can therefore be used in solving other problems where such solutions are applicable.

Another approach to functional design and code reuse is that higher-level problems tend to be composed of lower-level problems. If any of these lower-level problems has been previously solved, it should be possible (and desirable) to reuse the solution rather than reinvent the wheel.

Functional programming lends itself strongly to this kind of code and design reuse because of its consistent use of functional composition as a problem solving and programming perspective.

Higher-level Patterns: *General Design Approaches*

Other Related Patterns: *Stepwise Refinement*

Pattern 7.4 Stepwise Refinement

Alias: *Top-Down Design*

Motivation:

We presented a *Top-Down Design* pattern as part of the set of Imperative Analysis and Design Patterns of Chapter 5. This *Stepwise Refinement* pattern is essentially the same, but is dealt with here from a more strictly functional programming perspective. The basic concepts of problem decomposition that underlie *Stepwise Refinement* are no more the exclusive domain of imperative programming than functions are.

Discussion:

When breaking down a problem using *Stepwise Refinement* for a functional solution, the pieces we look for are functional. We can view a problem as being composed of individual tasks. Each of these tasks can ultimately be represented as a function.

Unique to the functional perspective is that these functions that we design do not act exclusively on data (as their imperative counterparts do), but can act on any part of the program, including other functions. In other words, the refinement of the problem through design steps can result in functions that produce or consume other functions. These dynamic functions can also be used as part of the solution to the problem.

Because of referential transparency, functions can be swapped with other functions without impacting performance, so long as the behavior of the two functions is the same. This permits a certain degree of maintainability in functional programs.

Higher-level Patterns: *General Design Approaches*

Other Related Patterns: *Adapting Similar Problems*

Pattern 7.5 <i>Functional Data Design</i>

Motivation:

One of the most effective approaches to functional design involves a data-driven design approach that influences the design and creation of functions.

Discussion:

The *Functional Data Design* patterns included in this set (*Recursive Refinement* and *Type Refinement*) are related to the data-driven approaches of imperative design. These patterns focus on data as a guide for creating the functions that render an effective solution.

Higher-level Patterns: *Functional Analysis and Design*

Lower-level Patterns: *Recursive Refinement, Type Refinement*

Other Related Patterns: *General Design Approaches, Functional Design Perspectives*

Pattern 7.6 Recursive Refinement

Motivation:

Functional programming lends itself to recursion as a programming or problem-solving approach. The solutions to many problems can be framed as successive instances of smaller problems that follow the same form. As long as the data being acted upon grows smaller, but maintains the same form, the same function can be applied to the problem recursively. *Recursive Refinement* involves analysis and design in which the problem is addressed from this perspective.

Discussion:

Recursive Refinement is not specific to functional programming, but is particularly applicable, given the dependence of functional programming on recursion as a mechanism for data traversal.

When using *Recursive Refinement*, the overriding question concerns whether the problem is regular enough in its solution to permit refinement into smaller problems of the same form. The process of design involves first identifying the form of the problem, and then assessing whether the data being manipulated moves through a state that resembles the original problem, but in a smaller form. In some situations, this recursive view of the problem can take various forms, with no particular form necessarily the “correct” one.

The key step in this refinement is to determine the final state of the data being manipulated when it arrives at its smallest form. At this point, a base case must be designed that can be easily solved, such that, as the recursive levels are popped back, the correct answer is ultimately achieved and presented.

One of the benefits of this design approach is that the efficiency of a given algorithm can be reasonably analyzed using difference equations, where efficiency is of

concern. Note that methods for assessing algorithmic complexity are not the exclusive domain of recursive functions, but mechanisms do exist here and could be utilized.

Examples:

Any of the classic recursive problems could be used as an appropriate example for *Recursive Refinement*. For example, if we need to traverse a binary tree to print its output, we can easily view this problem as a set of nearly identical problems that we face over and over again. Our process of recursive refinement involves two distinct phases. First, we identify what the normal state looks like that we pass through repeatedly. Second, we identify what the base case looks like. For each of these, we determine what we do in each of these two states.

Our normal state is to find ourselves visiting a node that has a value, a left branch, and a right branch. Either of these branches may be NIL, meaning there are no children on one side or the other. Our base case is to be in a node that has neither left nor right branch. This is a leaf node. If we assume in-order traversal of the tree, our normal behavior will be to visit the left branch (visit its node and print its value), print our value, visit the right branch (visit its node and print its value), and then return. Our behavior in the leaf node is essentially the same, and in this example requires no special behavior for the base case; it will fail to visit the left branch, print its value, fail to visit the right branch, and then return.

The key to applying these design principles was first recognizing that the overall problem is not just composed of many smaller steps, but that the overall problem can be viewed as being composed of the same *step* repeated over and over again for subproblems of differing sizes.

Higher-level Patterns: *Functional Data Design*

Other Related Patterns: *Type Refinement*

Pattern 7.7 <i>Type Refinement</i>

Alias: *Type Transformation*

Motivation:

Functions accept inputs of certain types and produce outputs of certain types. These types do not have to be the same, and are sometimes different. One way to visualize problem solving in the functional perspective is to view the solution to the problem as having to do with the transformation of a data set of a particular type into another data set of another type. We refer to this approach as *Type Refinement*.

Discussion:

This approach will rarely stand alone as the only design to apply for a particular problem, but conceptualizing problems in this way can be extremely helpful. Some of the tricks of functional programming involve getting the types to come out right. The process of *Type Refinement* involves refinement since we have to break down a problem into finer pieces to see how the type mapping will take place.

Examples:

It is often the case that solving mathematical equations or physics problems involves making sure that the types come out right. In these kinds of problems we sometimes work from both directions in an attempt to reduce the equation to something recognizable, so that we can solve for our answer. Similarly, using *Type Refinement* in functional programming involves understanding the nature of the data being brought into the system, and the nature of the answer being produced. This approach is strongly associated with the *Pipes and Filters* approach.

As an example, suppose we have a stream of characters that we understand to contain tokens, and our task is to search and count the number of occurrences of a particular token. The most brute force approach to solving this problem would be to attempt to match the token against characters in the string beginning with every character. This approach would work, but is not particularly efficient. However, we know the string of characters contains tokens, so we could first process the list of characters to produce a list of tokens. This is the first *Type Refinement*. At this point we can compare our token

with each token in the list of tokens. This might be acceptable, but is still not necessarily optimal. We could conceptualize another transformation on the input data, by converting the list of tokens into a sorted list of tokens. For a single token search, this would not be efficient. But if we had many comparisons to perform, a sorted list of tokens would make subsequent comparisons much quicker. At this point our token can be given as input to a function along with the sorted list of tokens, and number produced. Our transformation then, is as follows:

((List of characters -> List of Tokens -> Sorted List of Tokens), Target Token) -> Integer

Note that this notation is not specific to any functional programming language, but is intended to indicate the transformation types for the original list of characters, and then the ultimate production of a value of another type (Integer) given the two inputs.

At this point our *Type Refinement* is more or less complete, and we can apply other principles to create the functions that perform each of these transformations.

Higher-level Patterns: *Functional Data Design*

Other Related Patterns: *Recursive Refinement*

Pattern 7.8 *Functional Design Perspectives*

Motivation:

There are design perspectives that tend to strongly lend themselves to functional design. Understanding and recognizing these *Functional Design Perspectives* can aid us in functional design.

Discussion:

Functional programming is typically viewed in one of two broad ways: 1) as functional composition; 2) as a piping of values through functions. This pattern set includes two patterns that describe these *Functional Design Perspectives* (*Pipes and Filters, Functional Composition*).

Higher-level Patterns: *Functional Analysis and Design*

Lower-level Patterns: *Pipes and Filters, Functional Composition*

Other Related Patterns: *General Design Approaches, Functional Data Design*

Pattern 7.9 Pipes and Filters

Motivation:

One way to conceptualize functional programming is to view a system as being composed of pipes that carry data between functions or filters. Each of these functions does something to the data and then ships it on to the next function. Design in this approach involves building the pipeline between the right kinds of filters.

Discussion:

This approach to functional design is used in a variety of programming approaches, from signal processing in data communications to text processing in a Unix environment. [Buschmann 1996] describes an architectural pattern called *Pipes and Filters*. This pattern deals with the high-level perspective of viewing systems that break down in this way, and is a superb resource for understanding this pattern. In this discussion, our *Pipes and Filters* pattern is a design pattern that applies specifically to functional programming, and is intended to guide one form of functional design.

In practice, this design approach would lead a programmer to first examine the available tool set. This encourages code and function reuse, and is very desirable. The most natural solutions created in this way involve the piping of data between known functions. When it is discovered that some bridge cannot be crossed because of the lack of a functional piece, this piece can be written. Once written, it would be ideal for this filter to be made available to others and hence grow the global tool set among a team of functional programmers.

The *Pipes and Filters* pattern is somewhat related to type refinement, because each function along the pipe performs transformations to the data that it receives,

potentially modifying the type. However, *Pipes and Filters* could be applied in elaborate ways, and producing acceptable results, without impacting the type at any juncture.

Examples:

In the UNIX environment individual utilities, such as `sed`, `grep`, and `awk` can be configured through command-line parameters to perform filtering functions on a stream of text input. The mechanism for moving data between them is, appropriately enough, the “|” or pipe command.

The following example (Figure 7.2) shows the use of UNIX utilities to filter input to a desired state using filtering programs connected by pipes. This example is borrowed from [Welsh 1995]. In this example, `du` reports “disk usage” for different disk areas, `sort -rn` sorts character input in reversal numerical order, and `more` causes only enough output at a time as can fit on a single screen. Just producing a list of disk usage using `du` is not usable enough if we want to see who’s hogging the most disk space. By applying our functions connected by pipes, we can streamline our output and see the most important things first.

```
$ du
10      ./zoneinfo/Australia
13      ./zoneinfo/US
9       ./zoneinfo/Canada
3       ./zone/Chile
20      ./zoneinfo/SystemV
118     ./zoneinfo
298     ./ghostscript/doc
183     ./ghostscript/examples
3289    ./ghostscript/fonts

$ du | sort -rn | more
34368   .
16005   ./emacs
16003   ./emacs/19.25
13326   ./emacs/19.25/lisp
4039    ./ghostscript
3289    ./ghostscript/fonts
```

Figure 7.2. Pipes and Filters used in a UNIX command-line environment.

Higher-level Patterns: *Functional Design Perspectives*

Other Related Patterns: *Functional Composition*

Pattern 7.10 *Functional Composition*

Motivation:

The theoretical background of functional programming derives from the mathematical concepts of functional composition, in which mathematical functions can be viewed as being composed of other functions. This gives us another way of conceptualizing functional design, by viewing functions as being composed of other functions. The lower-level functions can be thought of as being encapsulated by higher-level functions.

Discussion:

From a mathematical perspective, this approach is natural, and is particularly useful when using functional programming to solve mathematical problems where order of execution must not affect the outcome of the problem. One positive result of this perspective is clearly that solving mathematical problems can be done more naturally in functional programming than with other programmatic notions.

This approach is different from *Pipes and Filters*, where we view a transformation of data starting at one end of a pipe and existing the other.

Examples:

As an example, suppose we want to create a composite function h from f and g . Assume that g produces output of the input type required by f . Further assume that the input to g is the desired input to h , and that the output of f is the desired output of h . Using mathematical terms, we could describe h in the following way: $h = f \circ g$. Looking at functions more programmatically, we could view it in the following way: $h = f(g())$.

Discussion:

Functional Composition also lends itself to a view of computation that includes encapsulation. Suppose we are designing a database system, in which we desire to capture certain actions together into atomic units, so that no part of a set of database instructions would ever go through without the others. We can view this in some cases as a form of transaction tracking system, in which transactions are confirmed in their entirety, then locked down. In its extreme, it also includes the notion of backing out transactions in case of problems.

Example:

Continuing this perspective, we can view a series of database transactions as a composition of functions on an existing database. For example, if we have functions to add a record, delete a record and query for a record, and an existing database `db`, we could encapsulate a series of actions on the database in the following way:

```
add(a, add(b, delete(c, add(d, delete(e, add(f, db)))))
```

where `a`, `b`, `c`, `d`, `e`, and `f` are records of the type appropriate for this database.

Higher-level Patterns: *Functional Design Perspectives*

Other Related Patterns: *Pipes and Filters*

8.0 Logical Programming Patterns

The history of modern logical programming began in the early 1970's with the introduction of Prolog, a language originally intended to be used for parsing natural language (specifically, French). The name Prolog was as an abbreviation for "PROgrammation en LOGique". Its approach was a hybrid between natural language processing and automated theorem proving. [Colmerauer 1993]

Kamin defines logical programming in the following way:

A programming language concept in which programs are regarded as assertions in a logic, and computation corresponds to proving or satisfying the assertions. Logic programming languages differ in the logic on which they are based, and in the additional, non-logical, features they include. PROLOG is by far the best known logic programming language, based upon the logic of Horn clauses; its most prominent non-logical feature is the cut. [Kamin 1990]

Since its introduction in 1971, Prolog has come to stand almost completely alone as the sole representative of logical programming languages. Given its preeminent position among logical programming languages, Prolog is the primary source for the logical programming patterns in this set.

A secondary source for logical programming patterns in this set is SQL, a popular database programming language (or database query language) that can loosely be considered logical. If we strictly follow Kamin's definition above, SQL will not fully qualify. Indeed, some authors place SQL in a separate class of database languages [Appleby 1991]. However, others consider logical programming to be an extension (or superset) of database programming, and hence closely related [Sterling 1983].

There are strong parallels between SQL and Prolog at the level of fundamental programmatic elements. Both are largely declarative, both establish known facts, and both allow the derivation of additional information based upon the given facts, the rules of inference, and some form of query or question. Also importantly, both Prolog and SQL permit the creation of intermediate results, derived from established facts and rules of inference, that subsequently carry the weight of "facts" in establishing further results.

Prolog and SQL also share some preoccupation with performance (thereby violating a purely declarative approach to programming). In Prolog, the main performance concern is in avoiding excessive (or infinite) backtracking. In SQL the size (and therefore the efficiency) of intermediate results can sometimes be controlled by the order in which certain operations (joins and reductions) are performed.

The programmatic match between Prolog and SQL is, of course, not perfect. But the parallels are close enough that we will consider SQL as a logical language for the purpose of this study, since it lends some insight into the general class of logical languages.

In this chapter, we have attempted to capture the essential elements of logical programming in pattern form. This pattern set will be used as a building block to analyze aspects of logical design, and to explore multiparadigm programming and design. We do not expect this pattern set to be exhaustive in revealing everything that could be said about logical programming (although it could provide an excellent foundation on which to build such an analysis).

This taxonomy of logical programming elements was culled from a thorough analysis of programming elements in selected logical languages including Prolog and SQL. The essential programmatic elements were identified, and are listed in the pattern hierarchy below (Figure 8.1). Each of these elements is discussed in greater detail in the patterns that follow.

- *Logical Program*
 - *Facts*
 - *Rules of Inference*
 - *Queries*
 - *Relations*
 - *Unification Function*
 - *Control Elements*

Figure 8.1. Hierarchy of Logical Programming Patterns.

Pattern 8.1 Logical Program

Alias: *Relational Program*

Motivation:

A certain number of problems can be readily represented via the propositional calculus. These problems are of a deductive nature, and the solution to these problems can be generally determined by following a set of assumptions and rules of inference to a logical conclusion. *Logical Programming* provides a programmatic vehicle for representing problems in this form.

Discussion:

For those problems that are easily represented by logical deduction, *Logical Programming* is an easy and natural way to program a solution. In fact, *Logical Programming* is a largely declarative approach to programming, requiring the programmer to characterize the problem in terms of a language like Prolog. The programmer does not identify how the solution is to be achieved, only the form of the problem. The *Logical Programming* language has the responsibility to deduce from the information provided what the solution is to a given query. *Logical Programming* involves identifying the basic facts that are known about a problem, and the rules of inference that govern what can be deduced from things that are known. Information is extracted through queries that are posed to the system.

Examples:

The first and preeminent example of *Logical Programming* is the language Prolog, which was developed in France in the early 1970's. Also considered in this pattern set is SQL, a database query language that shares many characteristics of *Logical Programming*. The similarities and differences between these two languages are discussed in the introduction to this pattern set.

Lower-level Patterns: *Facts, Rules of Inference, Queries, Control Elements*

Other Related Patterns: *Imperative Program, Object-Oriented Program, Functional Program*

Pattern 8.2 *Facts*

Aliases: *Assumptions, Axioms*

Motivation:

In propositional calculus, there are *Facts* that are known about any problem that may be helpful in deriving results from deductive queries. These *Facts* form the foundation of knowledge upon which inference rules can be applied to obtain useful information.

Discussion:

Facts (also referred to as *Assumptions* or *Axioms*) are those things we know that pertain to a given problem and its potential solution. The assessing of known *Facts* is the single easiest step in logic programming, since it is very straightforward.

In Prolog, *Facts* are asserted in simple statements. In SQL, *Facts* are established through the creation of relations (or tables) containing tuples of information.

Examples:

One of the easiest and most common examples of logic programming involves deducing information from a family tree. The kinds of questions that can be asked in such a system might include things like, “What’s my exact relationship to Uncle Luke, given that he had some relationship to my mother?” The *Facts* of a logical program would include all the things I already know about my family tree. For example, my mother and father’s names, their parents’ and children’s names, etc. From this information I can ask for lists of all the first cousins once removed that share some common set of great grandparents.

In the following example (Figure 8.2), some *Facts* relating to the author’s family are established in a Prolog program.

```

father(Allen, Chuck).
mother(Marjorie, Chuck).
brother(Bob, Chuck).
brother(Wes, Chuck).
sister(Julie, Chuck).
sister(Cindy, Chuck).

```

Figure 8.2. Known *Facts* in a Prolog program.

In the following example (Figure 8.3) SQL statements are used to create a simple relation and initialize it. This example was borrowed from [Celko 1995] and modified.

```

CREATE TABLE Customer
(custname CHAR(15),
 custaddr CHAR(15),
 status INTEGER NOT NULL);

INSERT INTO Customer (custname, custaddr, phone)
VALUES ('Julie', 'Jesup, Iowa', 1);

```

Figure 8.3. Known *Facts* in an SQL program.

Higher-level Patterns: *Logical Program*

Other Related Patterns: *Rules of Inference, Queries, Control Elements*

Pattern 8.3 *Rules of Inference*

Motivation:

The key to deductive reasoning involves understanding the rules by which inference may take place within a logical system. The *Rules of Inference* are those rules that relate objects to other objects, and can hence be used to deduce information from a system based upon known facts in order to produce an answer to a query.

Discussion:

Rules of Inference include things like how to know what the definition of a sibling is (an individual who shares one or more parents with another individual). But we could look closer at sibling, and separately look at those that share only one parent (half-brother or half-sister) and those that share two parents (brother or sister). We could define sibling either as the set of all individuals that share one or more parents, or the set of individuals that are all some form of brother or sister together, with respect to one or more parents.

The first and most significant difficulty encountered in logic programming is correctly characterizing the *Rules of Inference*. In many situations the *Rules of Inference* can be derived from more than one perspective. Cousins can be individuals that share grandparents, or individuals whose parents are siblings, etc.

In addition, the logical process by which answers are deduced may involve infinite recursion that must be protected against. Control elements (discussed in a separate pattern) are provided to protect against such infinite recursion.

In Prolog, *Rules of Inference* are created through a series of relations combined programmatically. These relations in Prolog can be easily reused by Prolog programs. In SQL, *Rules of Inference* are inherent in the nature of the relations (or tables) and the ways in which tables and fields are selected in an SQL query.

Examples:

Continuing our example of family relationships, we can use *Rules of Inference* to establish relationships between family members. In the following example (Figure 8.4), a rule is established to identify brothers. In this example, `Brother` and `Sibling` are two individuals whose relationship we are attempting to establish through logical inference. This program segment says that `Brother` and `Sibling` must each have the same `Parent`, must be `male`, and must not be the same person. This example is borrowed from [Sterling 1986].

```
brother(Brother, Sibling) :-
    parent(Parent, Brother),
    parent(Parent, Sibling),
    male(Brother),
    Brother != Sibling.
```

Figure 8.4. *Inference Rules* in Prolog.

In the following example (Figure 8.5) the WHERE clause of this SQL statement performs the *Inference Rules* for this query. In a sense, the WHERE clause provides similar rules for inclusion in some set as the predicates of relation do in Prolog. This example uses the relation definition from Figure 8.3.

```
SELECT *
FROM Customer
WHERE status <= 5 AND custaddr = 'Jesup, Iowa';
```

Figure 8.5. Use of Inference Rules via the WHERE clause in SQL.

Higher-level Patterns: *Logical Program*

Other Related Patterns: *Facts, Queries, Control Elements*

Pattern 8.4 *Queries*

Motivation:

Once a logical program exists with facts and rules of inference, usable information can be extracted from it. *Queries* are the mechanisms by which information is extracted from a logical program.

Discussion:

Once the facts and rules of inference are in place, creating queries is a relatively straightforward process, and is really the fun part of logical programming, because the

answers just sort of pop out. The mechanism that generates these answers is a search engine that performs a depth first traversal of the solution space, using backtracking to seek values that would cause the query to be fulfilled.

Queries typically take one of two forms. The first form is a boolean validation or rejection of a premise. In this approach, the query might be something like, “Is Bob the father of Jess?” The answer is either true or false.

The second form follows a similar pattern, but leaves out key information, such as in the following Query: “X is the father of Jess.” In this situation, the search engine seeks to create a true answer, so it will search for values of X that satisfy the statement. In this case, the engine will cause X to be bound to the value “Bob” so that the statement will be true. If more than one value will satisfy the statement, all values may potentially be returned. If no value will satisfy the statement, the result of the query will be false. This is at the heart of the unification function (discussed in a separate pattern).

In Prolog, *Queries* take the form of simple statements such as those just described. In SQL, the user creates (sometimes elaborate) *Queries* that may include elements that are, in fact, primarily composed of rules of inference

Examples:

In the rules of inference pattern, we shared an example rule for establishing whether two individuals are brothers. In this example (Figure 8.6), we will put those rules together with facts, and establish relationships via *Queries*.

```

parent(Allen, Bob).
parent(Allen, Chuck).
parent(Allen, Wes).
male(Bob).
male(Chuck).
male(Wes).

brother(Brother, Sibling) :-
    parent(Parent, Brother),
    parent(Parent, Sibling),
    male(Brother),
    Brother != Sibling.

?-brother(Bob, Chuck)
True
?-brother(X, Chuck)
Bob
Wes

```

Figure 8.6. A Prolog program with *Queries* for sibling relationships.

In the following example (Figure 8.7) a complex SQL *Query* is shown. This example is borrowed from [Celko 1995].

```

CREATE GLOBAL TEMPORARY TABLE OrderSummary
(custid INTEGER NOT NULL,
orderid INTEGER NOT NULL,
ordertotal DECIMAL(12,2) NOT NULL,
PRIMARY KEY (custid, orderid));

INSERT INTO OrderSummary
SELECT O0.custid, O0.orderid, SUM(I0.price * OI0.qty)
FROM Orders AS O0, OrderItems AS OI0, Inventory AS IO
WHERE IO.partid = OI0.partid
AND OI0.orderid = O0.orderid
GROUP BY O0.custid, O0.orderid;

```

Figure 8.7. A complex SQL *Query*.

Higher-level Patterns: *Logical Program*

Other Related Patterns: *Facts, Rules of Inference, Control Elements*

Pattern 8.5 Relations**Motivation:**

An additional form of computational power is available in logic programming through the use of *Relations*. *Relations* capture relationships, and can be progressively reused to form increasingly complex *Relations*. The *Relation* concept is strongly related to the theoretical underpinnings of relational databases.

Discussion:

Relations can be viewed as a type of query, since they rely on facts and rules of inference at a basic level to understand and identify relationships within the system. But while queries extract information for the end user, *Relations* capture relationship information that builds on facts and rules of inference, and so become part of the logical program. Any programmatic statements that build upon existing facts and *Relations* to create additional information that can in turn be used, can be termed a *Relation*.

Examples:

In the example for the rules of inference pattern, we created a rule that establishes what a brother is. We can treat this now as a relation, and use it to build increasingly complex rules. For example, we can add a rule for sister, and then establish a generic sibling *Relation*.

```

brother(Brother, Sibling) :-
    parent(Parent, Brother),
    parent(Parent, Sibling),
    male(Brother),
    Brother != Sibling.

sister(Sister, Sibling) :-
    parent(Parent, Sister),
    parent(Parent, Sibling),
    female(Sister),
    Sister != Sibling.

sibling(Sibling1, Sibling2) :-
    brother(Sibling1, Sibling2) |
    sister(Sibling1, Sibling2).

```

Figure 8.8. Building complex rules using *Relations* in Prolog.

In the following example (Figure 8.9) the relationship between siblings is stored in a temporary table, or *Relation*, called Siblings.

```

CREATE TABLE Family
(parentname CHAR(15),
 name      CHAR(15),
 sibname  CHAR(15));

CREATE GLOBAL TEMPORARY TABLE Siblings
(name      CHAR(15),
 sibname  CHAR(15));

INSERT INTO Siblings
SELECT name, sibname
FROM Family
WHERE Family.name = chuck
GROUP BY Family.parentname;

```

Figure 8.9. *Relations* in SQL.

Discussion:

In the multiparadigm programming language Leda, *Relations* are the programmatic interfaces through which imperative or object-oriented programs access the

capabilities of logic search engines. From an imperative perspective, *Relations* in Leda look like function calls. Within the *Relation* itself, it looks predominantly logical.

In the following example (Figure 8.10) *Relations* in Leda are used to determine if two individuals are siblings. This example is borrowed from [Budd 1995a].

```
function sibling (byRef left, right : name)->boolean;
var
  parent : name;
begin
  {true if they share a parent in common }
  return parentOf(parent, left) & parentOf(parent, right) & (left <>
right);
end;
```

Figure 8.10. The use of *Relations* in Leda.

Higher-level Patterns: *Queries*

Other Related Patterns: *Unification Function*

Pattern 8.6 Unification Function

Motivation:

Logical programs can be used for validating a boolean query, confirming that the indicated relation is either true or false. But relations can also include variables that do not have values. In this situation logic programming seeks to unify the values of the query in such a way that the value of the relation can be made true. This *Unification Function* holds some of the most significant power of logical programming.

Discussion:

The *Unification Function* becomes relevant in logic programming when one of the values passed into a query is a variable that is not bound to a value. The search engine will attempt to bind this variable to values that it encounters that would render the evaluated value of the relation to be true. This yields a bi-directional characteristic to

functional programming that is somewhat unique from other programming paradigms. One need not explicitly identify the direction of parameters to a relation. If the parameter is already bound to a value, the direction is input to the relation. If the parameter is not bound to a value, the direction is output from the relation, so long as values are available to be bound to the parameter. Each time a value is bound to the parameter, the relation is true, and the value is returned. Relations can be called successively to extract all possible values that render a relation true.

Examples:

The following example (Figure 8.11) depends on the `brother` relation presented in Figure 8.6. In the first query, both values are known to the engine, so the query is treated as a boolean question (the answer to which is `True`). In the second query, the first variable, `X`, is not a name bound to anything, so the search engine seeks to find values for `X` that cause the `brother` relation to be true. These values are then printed out by the engine.

```
?-brother(Bob, Chuck)
True
?-brother(X, Chuck)
Bob
Wes
```

Figure 8.11. Input and output variables to a relation in Prolog.

Higher-level Patterns: *Queries*

Other Related Patterns: *Relations*

Pattern 8.7 Control Elements**Motivation:**

Logic programming is typically viewed as a declarative programming paradigm. This means that we identify the characteristics of a proper solution, and allow a search engine to find the correct answer. In the purest sense, there is no need for *Control Elements* in this kind of system. *Control Elements* should be the concern of imperative programming languages, not declarative ones. Unfortunately, in practice, it is extremely difficult to build a generic search engine that will always search in the most efficient way possible, or that will avoid infinite recursion in certain situations. For this reason, logic programming includes room for *Control Elements* to be used in specific situations.

Discussion:

There are two driving motivations that lead logic programming languages to permit *Control Elements* in an otherwise declarative paradigm—efficiency, and infinite recursion.

Efficiency problems include both time and memory. When performing searches on a problem space, it is possible to dramatically vary the amount of either memory or processing time required to solve a particular problem. This typically has to do with the order in which certain operations are performed. For example, some intermediate results because of the nature of the query, may be larger than is desirable. In many of these situations, changing the order of the traversal will dramatically reduce the space required. A significant place where this efficiency must be dealt with is in SQL queries that require intermediate joins of relations prior to trimming the joined table down to size by selecting certain fields. The order in which joins are invoked can lead to smaller intermediate tables, and hence aid both memory requirements and execution efficiency.

Efficiency is a serious concern in logical programming. The search engine performs a depth-first search of the potential solution space using backtracking. Occasionally, the search engine will encounter a place in the search tree that has already been visited previously. This point in the search tree may have branches that have already

been determined to be fruitless. If these pointless path are followed, the result will be inefficient execution at best, and infinite recursion at worst. Cuts are used to inhibit the search engine from taking paths known to be fruitless. The cut therefore provides a mechanism by which the search tree can be dynamically pruned, yielding a more efficient execution of the program.

Another control mechanism available in logic programming is the fail. Fail causes the traversal to stop further searching when it can be determined that any additional searching is futile. This naturally requires an understanding on the part of the programmer of the solution space (in essence violating the notion of a purely declarative language), but saves considerable search time when further examination is hopeless.

Examples:

The following example (Figure 8.12) shows the use of the cut operation in a Prolog program to find the minimum of two numbers. In this code segment, X is determined to be the minimum number when it is less than or equal to Y. Y is determined to be the minimum number when it is strictly greater than X. The cut used here allows computation to stop as soon as the answer is found, since if one of these relations is found true, the other one must be false. This example is borrowed from [Sterling 1986].

```

minimum(X,Y,X) :-
    X <= Y, !.
minimum(X,Y,Y) :-
    X > Y, !.

?-minimum(3,4,Min)
3

```

Figure 8.12. The use of cuts in Prolog.

Higher-level Patterns: *Logical Program*

Other Related Patterns: *Facts, Rules of Inference, Queries*

9.0 Logical Analysis and Design Patterns

In our introduction to the pattern set for Functional Analysis and Design Patterns (Chapter 7), we identified the relationship between programming paradigms and the subsequent rise of design methodologies for these paradigms. Like functional programming, logical programming suffers from a lack of popular design methodologies. Despite the fact that Prolog was first introduced more than 25 years ago, logical programming remains a niche area for programmers, and one in which most professional programmers do not venture. The same principles that govern the rise of design methodologies apply here, and there is an unfortunate void.

However, our study requires that we analyze design approaches for this methodology as well as the others, so we have attempted to cull design principles from the available literature. This search for methodological gems has been difficult and at times frustrating and fruitless. Some research has attempted to explore the cognitive models of Prolog programmers [Bergantz 1991] [Ormerod 1993] [Ormerod 1994], but have not led to what we would commonly view as design methodologies.

Most of the design gems have been found in the form of idiomatic hints— suggestions about how to deal with specific programmatic situations. But very little has been written concerning overarching approaches to analyzing and designing for a logical solution. The single richest source for logical design insights has been [Sterling 1986]. It is likely that the reasons for this dearth of logical design methodologies is similar to those for functional programming. Despite these limitations, we have gleaned a handful of design patterns. We are confident that others exist, and that this material deserves a fuller treatment.

This set of design patterns is the result of a three-pronged investigation. First, we looked at the logical programming language Prolog, and tried to identify what might be natural correlations between the programming language and a suitable design methodology. Second, we gleaned programmatic and design hints from several authors and tried to expand these ideas from the idiomatic level up to an overarching design level. Third, we have examined SQL for design insight. The majority of our success has come from the first two approaches. As we examined SQL for design insights, we found that

the most significant design principles were similar to those learned from our research into Prolog. This had the effect of solidifying our prior findings, but contributed little that was new or useful.

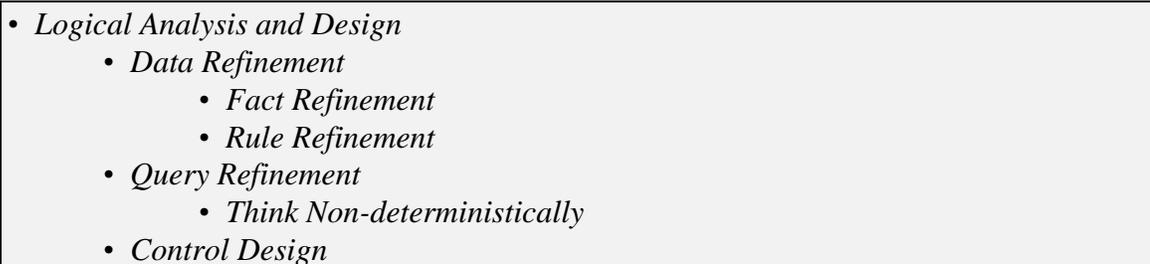


Figure 9.1. Hierarchy of Logical Analysis and Design Patterns.

Pattern 9.1 *Logical Analysis and Design*

Motivation:

Logical programming, like all other programming paradigms, involves a design phase, followed by an implementation phase in which a program is written. There are no formal logical design and analysis methodologies extant, and yet design certainly occurs in logical programming, whether implicit or explicit.

Discussion:

A logical programming system consists of two basic types of elements: data and queries. Data can be viewed as consisting of both facts and rules, where facts are those things immediately known, and rules provide the means for deriving additional knowledge. In performing logical design, there is a natural need to refine each of these programmatic components through some methodology.

To summarize, some representation of the problem must be made in fact and rule refinement phases. Then the queries that are used to drive to a solution must themselves be refined. Within each of these high-level logical design phases, there are increasingly lower-level patterns that quickly fall into the domain of idioms. It is not the intent of this

pattern language to take this discussion to that level. Rather, the intent is to capture a broad view of logical design and analysis, providing a handful of principles in each of these refinement phases.

There is very little available in the literature of what would typically be referred to as analysis and design for logical programming. Design elements that can be found are typically at an idiomatic level, such as “don’t abuse cuts” and “lists can be overused.” It seems common in the more declarative paradigms (such as logical and functional) that the practitioners involved in these paradigms are not accustomed to appealing to a general audience. Either that, or their own programming perspectives are so obvious to them that they are unable or unwilling to explain it at a high level to the novice. In any event, there is currently little research into logical design principles that can be leveraged to create an effective analysis and design methodology. There are some gems to be mined, such as *Think Non-Deterministically* that give a broader overview of how to approach logical problems in general. We have attempted to capture these patterns in this pattern set.

Lower-level Patterns: *Data Refinement, Query Refinement, Control Design*

Other Related Patterns: *Imperative Analysis and Design, Object-Oriented Analysis and Design, Functional Analysis and Design*

Pattern 9.2 Data Refinement

Motivation:

Any logical program first exists as a problem that needs solving. Despite the declarative and deductive nature of logical programming, the essence of a problem is contained in data structures of some kind that can be accessed and manipulated by the program. As a logical program is designed, the representation of the problem must be refined, typically before queries begin to be formed.

Discussion:

In a logical programming language like Prolog, *Data Refinement* might take the form of determining which data structures will be used to capture key information that

can later be the subject of queries. In query languages like SQL, the key information will be encapsulated in relations consisting of tuples of information (which can loosely be viewed as heterogeneous lists). In either case, the data structures must be created effectively, and the failure to create them well can lead either to a more difficult time creating effective queries later, or in some cases can lead the designer down a path from which no solution can be obtained.

Examples:

The following example is taken from [Ross 1989] and gives some insight into the thought processes an experienced Prolog programmer might go through in determining the form of data to be used to solve a particular puzzle.

Puzzles that give a sum of two numbers, but with digits replaced by letters, frequently appear in papers and magazines. Different letters in these puzzles are meant to denote different digits. For example,

CROSS + ROADS = DANGER

The problem here is to write a PROLOG program to solve such puzzles....

The first question is how to represent the problem. Since we want to deal with individual letters, it seems at first sight sensible to represent each row as a list, such as [C, R, O, S, S], [R, O, A, D, S] and [D, A, N, G, E, R]. The reason is that we are going to want to deal with columns: a letter from the top row, a letter from the bottom row and a letter from the answer, and (implicitly) a carry digit from the previous row. We could represent each column as a list, of course—but that would mean having as many lists as columns, and dealing with an unknown number of arguments is usually slightly more awkward than dealing with a fixed number (such as three). Here each element of the list is a variable, so that, provided the three lists are part of the same term, whenever a variable in one of the lists is assigned a value it will be instantiated to the same value in the other two lists... [Ross 1989].

This discussion continues on for several pages as the author brainstorms his *Data Refinement* task on paper. In this example, we don't see a systematic methodology. Rather we are given insight into some of the intuitive decision making processes in the

mind of this Prolog programmer. Clearly *Data Refinement* is the first concern of the designer in this example.

Lower-level Patterns: *Fact Refinement, Rule Refinement*

Higher-level Patterns: *Logical Analysis and Design*

Other Related Patterns: *Query Refinement, Control Design*

Pattern 9.3 *Fact Refinement*

Motivation:

One of the fundamental building blocks for a logical program is the set of information initially available to the program. These facts must be laid down in a form that can be used as the foundation for the deduction of answers to queries. *Fact Refinement* is the process by which these facts are laid down and coded.

Discussion:

There is very little written about this process of *Fact Refinement*. There is clearly some dependence on the data structures that are chosen to represent either a problem or its solution. In Prolog this refining stage involves laying out facts as programming statements. In SQL this process involves the creation of tables (a key step in *Data Refinement*) and then filling these tables with initial database information. See Chapter 8 for specific examples of *Fact Refinement* in Prolog and SQL.

Higher-level Patterns: *Data Refinement*

Other Related Patterns: *Rule Refinement*

Pattern 9.4 Rule Refinement

Motivation:

Some of the most challenging aspects of logical programming occur in the *Rule Refinement* phase. During this phase decisions are made that determine the means by which deductive reasoning can take place.

Discussion:

In Prolog, *Rule Refinement* constitutes the bulk of design and programming effort for any given program. During this stage of design, all of the facilitative relationships between facts are established, as well as the relationships between rules. This phase involves creating chains of implication that will ultimately permit the program to accept a query and generate meaningful answers.

In SQL, *Rule Refinement* is somewhat programmatically inseparable from *Query Refinement*, since they occur together. Still, the phase takes place in SQL and enables the database engine to fulfill the query.

Higher-level Patterns: *Data Refinement*

Other Related Patterns: *Fact Refinement*

Pattern 9.5 Query Refinement

Motivation:

Once a logical problem has been analyzed and represented in data (through a process of data refinement), a set of queries must be created to evoke an appropriate solution to the problem. This process of creating the queries is referred to here as *Query Refinement*, and constitutes a large part of the work of logical program design.

Discussion:

The creation of a data representation for a problem will impact what can be done during the *Query Refinement* phase. But independent of the influence of data refinement,

queries must be created that can effectively extract answers from the possible solution space.

During *Query Refinement*, there are a number of idiomatic patterns that become a concern, including efficiency (both space and time), infinite recursion, and traversal styles (including recursion and failure-driven loops). A complete discussion of these programming idioms is beyond the scope of this pattern set, but is clearly part of the process of *Query Refinement*.

Higher-level Patterns: *Logical Analysis and Design*

Lower-level Patterns: *Think Non-Deterministically*

Other Related Patterns: *Data Refinement, Control Design*

Pattern 9.6 *Think Non-Deterministically*

Motivation:

One of the unique aspects of logical programming is the existence of a sequential search and backtracking mechanism that searches a potential solution space in a depth-first traversal, seeking a potential solution to the query. This existence of backtracking permits the programmer to view problems non-deterministically

Discussion:

The ability to *Think Non-Deterministically* is unique within the logical perspective, particularly in Prolog, where depth-first traversal with backtracking permits a non-deterministic view of the world.

The most common version of non-deterministic programming is the “generate-and-test” approach in which one process generates potential solutions, while another tests them. Since logical programming is declarative, the programmer does not need to be overly concerned with the order in which tasks are performed, and the set of potential solutions to be generated can be viewed as universally available to the search engine in a

non-deterministic fashion. As it turns out, by pushing the tester inside the generator, optimal Prolog programs tend to emerge.

Examples:

The following discussion, taken from [Sterling 1986] sheds more light on forms of non-deterministic thought.

It is easy to write logic programs that, under the execution model of Prolog, implement the generate-and-test technique. Such programs typically have a conjunction of two goals, in which one acts as the generator and the other tests whether the solution is acceptable, as in the following clause:

```
find(X) <- generate(X), test(X).
```

This Prolog program would actually behave like a conventional, procedural generate-and-test program. When called with `find(X)?`, `generate(X)` succeeds, returning some `X`, with which `test(X)` is called. If the test goal fails, execution backtracks to `generate(X)`, which generates the next elements. This continues iteratively until the tester successfully finds a solution with the distinguishing property, or the generator is exhausted of alternative solutions. [Sterling 1986]

Discussion:

Two other forms of nondeterminism come into play in Prolog programming. The first is referred to as “don’t-care nondeterminism” and involves a situation in which the choice of next candidate for searching in the solution space is unimportant and will equally lead to a solution. The second is referred to as “don’t-know nondeterminism” and involves a situation in which the choice of next candidate for searching in the solution space is indeed important, but is unknown at the time of query execution. [Sterling 1986]

Given the issues of efficiency within logical programming, in practice don’t care nondeterminism is not encouraged, and is, in fact, rarely appropriate, since rarely are the possible paths to a solution equivalent. Hence, the vast majority of logical programming involves don’t-know nondeterminism. It can be said that writing a program that captures don’t know nondeterminism is really creating and utilizing a depth-first search algorithm

on the possible solution space for the problem. This is clearly a significant part of many Prolog programs.

Higher-level Patterns: *Query Refinement*

Pattern 9.7 Control Design

Motivation:

Despite the declarative nature of logical programming, designers must still be concerned with issues of program flow control in order to manage efficiency and to avoid debilitating problems such as infinite backtracking.

Discussion:

Effective *Control Design* in Prolog involves the effective use of cuts and fails to avoid infinite or inefficient solutions. It violates the declarative nature of logical programming, and requires that a programmer understand the way in which a prolog engine traverses the solution space. In SQL, *Control Design* involves creating SQL queries in such a way that the overall time and space constraints of joins are minimized for efficiency.

Higher-level Patterns: *Logical Analysis and Design*

Other Related Patterns: *Data Refinement, Query Refinement*

10.0 Methodological Patterns

Design methodologies exist to guide engineers in the design and implementation of programs within a particular paradigm. Clearly successful design methodologies must have some direct applicability within the target programming paradigm, or they would have very little usefulness in practice. As Johnson said, “It’s a good rule of thumb that the value of a method is inversely proportional to its generality. A method for solving all problems can give you very little help with any particular problem” [quoted in Buschmann 1996]. So we find, in general, that successful design methodologies have tended to spring up around a particular programming paradigm, or indeed, sometimes around a particular programming *language*.

In this research, we have captured essential elements of design approaches in each of four paradigms and have presented those here as four pattern sets (see Chapters 3, 5, 7 and 9). We have also captured essential elements of programming within each of these same four paradigms, and have presented those elements as an additional four pattern sets (see Chapters 2, 4, 6 and 8). Having captured both design and programming pattern sets for these paradigms provides an opportunity for analysis of these pattern sets. In particular, we have studied the relationships that exist between programming and design patterns within a given paradigm. Clearly we would expect relationships to exist since the design patterns should be supportive of the programming patterns.

As a result of this research, we have discovered higher order patterns that relate programming patterns with design patterns within a given paradigm. Further, we have found that these patterns, while more visible within certain paradigms, are applicable in all four paradigms. The discovery of these higher order patterns that relate design patterns with programming patterns is significant, because it suggests an approach that might be used to generate design patterns once programming patterns are known (and possibly vice versa). We refer to these patterns as methodological patterns because they may suggest a methodology for mapping a set of programming elements to a corresponding design methodology.

This concept of higher-order patterns that relate sets of patterns to one another is somewhat novel (if not heretical) to the current object-oriented design patterns movement

whose view of patterns is largely limited to that of a form for capturing a solution to a problem within a context. This notion stems from Alexander’s statement that, “Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution” [Alexander 1979]. However, Alexander also said, “Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns within which it is contained....It is the network of these connections between patterns which creates a language.”

The following example (Table 10.1) helps to illustrate the nature of the relationships that have been discovered between these pattern sets. This table shows the imperative programming patterns on the left and the imperative design patterns on the right. The elements in this table have been combined to a general level for simplicity of discussion. Notice that there is a general matching up between them in the most important and fundamental areas. This natural close relationship led to the pattern *Expect Isomorphism*. But casual observation reveals that there are more design patterns than language patterns within most paradigms. The pattern *Expect Non-Isomorphism* attempts to discover more about these inconsistencies.

Table 10.1. Example of a comparison between programming and design pattern sets.

Programming Pattern	Described by Design Pattern
Procedure	Data Flow Diagram
Data Statement	Data Dictionary
Executable Statement	Process Specification
(none)	Entity/Relation Diagram
(none)	State Transition Diagram

The ultimate objective of this research is the creation of a pattern-based multiparadigm design methodology. Once we have successfully created a pattern set for multiparadigm programming, we will apply this set of methodological patterns to help create a multiparadigm design methodology. Figure 10.1 shows the methodological patterns contained in this pattern set.

- *Expect Isomorphism*
- *Expect Non-Isomorphism*
 - *Sources of Non-Isomorphism*
 - *High-Level Programming Patterns*
 - *Environmental Forces*
 - *Destinations of Non-Isomorphism*
 - *New Language Features*
 - *Stretching the Paradigm*
 - *System-Specific Language Elements*

Figure 10.1. Hierarchy of Methodological Patterns.

Pattern 10.1 *Expect Isomorphism*

Motivation:

The search for methodological patterns involves a search for relationships between programming language elements and design methodologies within a given programming paradigm. In general we expect to find a correlation between language elements and design elements. In practice this doesn't fully occur (see *Expect Non-Isomorphism*), but this does not lessen the importance of seeking isomorphism as a starting point.

Discussion:

Any minimal design methodology must include design methods that at least cover the essential programming language elements, or it will not be at all adequate to the task. Therefore, when creating a new design methodology, we should be able to begin with a set of patterns describing the language elements and build a minimal design methodology by filling out the set of design patterns to create an isomorphic relationship. The nature of the design methods used in the set of design patterns is not important, but it is important that all the key language elements have corresponding methods in the set of design patterns.

We believe that the natural relationship between language elements and design elements is a general isomorphic equilibrium. If there are language elements that are not represented in a design methodology, the methodology is clearly inadequate to the extent that these areas are needed for a particular solution. When design methodologies evolve to transcend the language or paradigm for which they were originally created, the language will tend to evolve to match the methodology, or in some cases, stretch completely and give rise to a new programming paradigm. (These inconsistencies are discussed in the pattern *Expect Non-Isomorphism*.)

Examples:

Table 10.1 shows a simplified view of the relationship between imperative programming and design patterns. The sets of imperative programming and design patterns described in Chapters 2 and 3 have an isomorphic core, but also contain some design elements that transcend traditional imperative programming. We believe this is because work on imperative languages is largely stagnant, while software design is largely giving way to object-oriented approaches. The two imperative design patterns that do not have corresponding programming patterns are essentially dealt with naturally within the object-oriented paradigm.

Object-oriented language and design elements (see Chapters 4 and 5) have qualities, both isomorphic and non-isomorphic, that are similar to imperative elements, but for potentially different reasons. Despite the relative newness of object-oriented programming, we still see design elements that transcend programming elements. However, the three design patterns for which there are no corresponding programming patterns are relatively new additions to object-oriented design, and are largely environmentally imposed. We believe that over time object-oriented languages will evolve to natively handle the concepts represented by these design patterns.

Table 10.2. Comparison of object-oriented programming and design pattern sets.

Programming Pattern	Described by Design Pattern
Forms of Reuse	Object Relationship Model
Object Class	Object Behavior Model
Points of Interaction	Object Interaction Model
(none)	Object Distribution
(none)	Object Concurrency
(none)	Real-Time Objects

Other Related Patterns: *Expect Non-Isomorphism*

Pattern 10.2 *Expect Non-Isomorphism*

Motivation:

Despite the fact that we expect in general to find isomorphism between programming and language elements (see *Expect Isomorphism*), in practice we find inconsistencies between them (typically design elements for which there are no corresponding programming elements). This is because languages, design methodologies, computing environments, and even paradigms themselves are constantly evolving, and so occasionally fall out of lock-step.

Discussion:

There are forces at play between programming elements and design elements. There are also forces between programming paradigms (including programming and design) and computing environments. These forces tend to strive toward an isomorphism between language and design elements. In practice, however, there are typically gaps between programming language and design elements. Most commonly these gaps occur in the form of design elements for which there is no corresponding native language capability. The converse is seldom the case when there are applicable design methodologies extant. This is because languages typically precede design methodologies. As methodologies are created, they tend to include approaches for utilizing the major

features of languages that are representative of the paradigm. When new languages evolve for which design methodologies are not sufficient, those methodologies tend to evolve to cover the applicable features. In a certain sense, we could say that languages and design methodology “leap frog” each other over time, but tend primarily to rest with methodology ahead of language.

There is a great deal that can be learned from these gaps. Part of the learning concerns the origin of the gaps (see *Sources of Non-Isomorphism*) and part of the learning concerns the implications of the gaps (see *Destinations of Non-Isomorphism*).

Lower-level Patterns: *Sources of Non-Isomorphism, Destinations of Non-Isomorphisms*

Other Related Patterns: *Expect Isomorphism*

Pattern 10.3 Sources of Non-Isomorphism

Motivation:

As we seek general patterns that lend insight to the creation of design methodologies, we observe that there is typically some degree of non-isomorphism between language and design elements within a given paradigm. By examining the nature of these inconsistencies, specifically the sources of the observable non-isomorphism, we can learn about the creation of design methodologies.

Discussion:

In our study of programming and design patterns, we determined two fundamental areas of non-isomorphism whose sources were of particular value to this research. These are described in *High-Level Programming Patterns* and *Environmental Forces*.

Higher-Level Patterns: *Expect Non-Isomorphism*

Lower-level Patterns: *High-Level Programming Patterns, Environmental Forces*

Other Related Patterns: *Destinations of Non-Isomorphism*

Pattern 10.4 High-Level Programming Patterns**Motivation:**

There exist situations in which a set of design patterns contains a pattern for which there is not a natural counterpart among the corresponding language patterns within the same paradigm. By seeking to understand these differences we hope to be able to apply general principles in the creation of other design methodologies. Some of these differences occur when the design pattern reflects a weakness in the programming language, and suggests *High-Level Programming Patterns*.

Discussion:

Design methodologies tend initially to represent the programming elements of a language or paradigm in a straightforward manner. But as a language or paradigm becomes more regularly used, and more practitioners begin to use it, design methodologies tend to stretch to include design approaches that transcend language elements. In some of these situations, we say that these design methodologies incorporate *High-Level Programming Patterns*. These design patterns occasionally suggest modifications to the programming language, which may lead to language feature enhancements for the language in question, new higher-level languages within the programming paradigm or to the evolution and creation of a different paradigm altogether (see *Destinations of Non-Isomorphism*).

Examples:

A good example of this pattern is the *Entity/Relation Diagram* in the imperative design pattern set. This design pattern has no natural counterpoint in the imperative programming pattern set (although it can be coded imperatively). However, it fits nicely into the *Forms of Reuse* pattern of the object-oriented programming pattern set, and the *Object Relationship Model* of the object-oriented design pattern set. This design principle could be used within an imperative programming language, but found its place more comfortably when these same principles were embodied in the object-oriented paradigm.

Higher-level Patterns: *Sources of Non-Isomorphism*

Other Related Patterns: *Environmental Forces*

Pattern 10.5 <i>Environmental Forces</i>

Motivation:

When there are discrepancies between language and design elements, sometimes the differences are induced by *Environmental Forces*. These are forces that act on languages and design methodologies and are inspired by the introduction of technologies related to the computing environments in which programs run.

Discussion:

The most clear example of the influence of environmental forces from our study of imperative language and design patterns is the *State Transition Diagram*, which is an attempt to focus on and model the time-dependent behavior of a system. This modeling constraint is irrelevant in a batch processing environment, but becomes a significant issue in modern real-time systems.

Similarly, multi-processing and multi-threaded systems require different perspectives for design, even when the programming is done in an imperative fashion. Designing such systems often requires a perspective in which entities exist and interact via message passing or semaphores, or some other signaling mechanism.

Real-time systems have also had an influence on design, imposing timing constraints on the design process, even when languages have no easy way to accommodate the requirements.

Higher-level Patterns: *Sources of Non-Isomorphism*

Other Related Patterns: *High-Level Programming Patterns*

Pattern 10.6 *Destinations of Non-Isomorphism***Motivation:**

The fact that non-isomorphism exists is easy enough to demonstrate. It is also relatively easy to speculate on the causes of these inconsistencies. But once design patterns exist that do not have corresponding programming patterns, there are a number of potential effects that can spring forth as a result.

Discussion:

In our study of programming and design elements, we observed several different outcomes when there is non-isomorphism between the programming pattern set and the design pattern set for a particular paradigm. In most cases, later iterations of the programming language or paradigm evolve to handle the issues indicated by the non-isomorphic design patterns (see *New Language Features* and *System-specific Language Elements*). However, in some case the discrepancy hints strongly at a new emerging paradigm, and may represent a complete stretching (or breaking) of the paradigm (see *Stretching the Paradigm*).

Higher-level Patterns: *Expect Non-Isomorphism*

Lower-level Patterns: *New Language Features, Stretching the Paradigm, System-specific Language Elements*

Other Related Patterns: *Sources of Non-Isomorphism*

Pattern 10.7 *New Language Features***Motivation:**

When there are discrepancies between programming and design elements, it is sometimes the case that a design pattern has captured some high-level language pattern, such that the set of language elements does not contain the pattern, but the set of design elements does contain it. One of the potential outcomes in this situation is for the

language to evolve and include the higher-level elements in a subsequent iteration of the language, or in a subsequently created new language within the same paradigm.

Discussion:

Designers often capture high-level patterns of language elements and represent them in certain ways. These high-level patterns are captured by designers because they are useful for solving certain problems. As programming languages evolve, some of the most useful of these design patterns become part of the next generation of the language, or become part of another language that springs up within the same paradigm.

Examples:

The easiest and most readily accessible mechanism for language evolution in the imperative and object-oriented paradigms is through function or class libraries. In the strictest sense, the evolution of libraries is not an actual growth of the language. But when certain libraries become standard operating equipment for all compilers for a certain programming language, it may as well be an extension of the language itself.

Within the model of library extension, one of the clearest examples is the current availability of C++ and Java class libraries from companies (like Rogue Wave of Corvallis, Oregon) that provide high-level functions such as sorting and searching algorithms for C++ and Java compilers. Once an easy-to-use class library provides the ability to do a quicksort, for example, there is little reason for an engineer to start from scratch to build such a sorting routine.

In other situations, the language itself may evolve to accommodate a higher-level feature than was originally provided. As an example, all languages do not provide the same data types as all the others. For instance, a programming language may not provide a boolean value, but the concept of a place holder for true/false values is important to imperative design. In cases where there is no boolean value (such as C), an `int` can easily hold zero or non-zero values and do the same job as a boolean. Still, design methodologies will refer to this value as a boolean value, identifying the purpose of the

variable even when the language doesn't strictly provide it. The need for such values has given rise to the existence of explicit boolean variables in languages such as Pascal.

Similarly, procedure calls have not always existed in all assembly languages, but became more generally available as structured programming became more popular. Similarly, early assembly languages provided rudimentary test, increment and jump instructions, which were the tools of choice for looping constructs. Modern assembly languages tend to provide loop instructions that combine these features.

Higher-level Patterns: *Destinations of Non-Isomorphism*

Other Related Patterns: *Stretching the Paradigm, System-Specific Language Elements*

Pattern 10.8 *Stretching the Paradigm*

Motivation:

Sometimes there are design patterns that pertain to a particular paradigm for which there is not a natural mechanism in available programming languages. Occasionally these design methodologies actually capture design approaches that transcend the particular paradigm in which the design is being done. In these cases, the paradigm may be stretched, resulting in either an expanded view of the paradigm or the creation of an entirely new paradigm

Examples:

The imperative design patterns we explored contain a very clear example of this pattern. *Entity/Relation Diagrams* are used to demonstrate relationships between data in a graphical way. These relationships can involve either composition or inheritance. While *Entity/Relation Diagrams* grew out of imperative programming, they contain the seeds of object-oriented organization of class and instance hierarchies. It is interesting to note that while they can be used for imperative design, they capture some of the elements of imperative programming that are common in object-oriented design. It's also worth

pointing out that object-oriented programming can be done from an imperative perspective as long as object-oriented design principles are followed.

Higher-level Patterns: *Destinations of Non-Isomorphism*

Other Related Patterns: *New Language Features, System-Specific Language Elements*

Pattern 10.9 System-Specific Language Elements

Motivation:

When design elements evolve from system constraints, it is sometimes the case that programming languages arise that assume (or at least enable the use of) a particular operating environment, and hence contain *System-Specific Language Elements*.

Discussion:

The most common mechanism for linguistic extension is through system calls or libraries that provide the desired functionality. However, the forces between languages and design methodologies tend to seek equilibrium, and occasionally languages embed key design capabilities, even system-specific ones into the language itself. Two examples serve to illustrate.

Examples:

Ada is a language that provides support for concurrency. Parallel processes in Ada are called “tasks,” and a concurrent program can be viewed as a number of interacting tasks. Tasks synchronize via the “rendezvous.” Ada also provides explicit language elements that provide for timing constraints to be met, making real-time programming somewhat natural. These language elements are clearly geared toward making certain aspects of a presumed computing environment more accessible to the programmer.

Java grew up in the 90’s when the internet and the world wide web were becoming ubiquitous. Java applets are destined to run as small application threads under web browsers. For this reason, the concept of multi-threading was foundational in the

creation of the Java language. Java has built-in language mechanisms for multi-threaded programming, including synchronization between threads and scheduling of individual threads based on priority. Again, this programming language incorporates language elements that were inspired by a target operating environment.

In both of these cases, languages already existed that could not easily take advantage of the technology of new standard operating environments. The additional features of these languages were strongly influenced by environment.

Higher-level Patterns: *Destinations of Non-Isomorphism*

Other Related Patterns: *New Language Features, Stretching the Paradigm*

11.0 Multiparadigm Programming Patterns

11.1 Introduction

There are several ways to create a set of multiparadigm programming patterns. Each of these approaches has specific implications for the concept of what multiparadigm programming patterns are, or should be. One of the most obvious approaches is to examine multiparadigm programming languages, of which there are few. Leda [Budd 1995a] stands out as an obvious example, as does G [Placer 1991a] to a lesser degree. But beyond these two, the landscape is a bit bleak for multiparadigm programming languages. We will focus our attention on Leda as an important source for mining multiparadigm programming patterns.

Another approach to creating multiparadigm programming patterns is to look at the programming elements of various languages that might be viewed as common between paradigms. This can be done through an examination of the patterns already presented in previous pattern sets. One of the key questions to ask when addressing multiparadigm programming involves the way in which the various paradigms combine within a multiparadigm language, or the ways in which elements can be combined within existing languages that pertain to a particular paradigm.

11.2 Adoptive versus Hybrid Combination

As we explore the ways in which programming features from different paradigms can be combined, we discover two broad approaches, which are here referred to as *adoptive* versus *hybrid* combination. Paradigm adoption occurs when some programming element of one paradigm is borrowed, and then joined to an existing language or paradigm in such a way that there is very little genetic material exchanged, similar to adopting a child into a family. The child takes the family name, and is integrated in a functional way, but not in an inherently genetic way. This happens when some interface is created between programming functions or elements of differing paradigms in such a way that calls can be made between them, but the language elements are not necessarily integrated tightly. For example, we might create a logical module with an interface that

looks like a function call to an imperative program, but looks to the logical module like a relation. Calls can be made into the logical module, but there are no inherently logical elements in the normal flow of the program.

Hybrid combination involves a tighter meshing of genetic material. This is analogous to the birth of a child to a couple. The child is a functional member of the family, like the adopted sibling, but unlike the adopted sibling, shares genetic material from both parents. This analogy extends to multiparadigm language elements that tightly integrate disparate paradigmatic material at a sometimes idiomatic level. For example, we might permit the integrating of imperative elements within logical relations that are not ordinarily permitted. We might use logical backtracking to perform iteration through an embedded language feature similar to that provided by Leda. The biggest problem with identifying hybrid adoption is that the combinations happen at such an idiomatic level that they sometimes fall below the scope of this study. Still, there is value in exploring these hybrid combinations, as not all of them fall outside the scope of this research.

11.3 Programmatic Equivalence Classes and Inherent Commonality

In our effort to create a multiparadigm programming pattern set, we studied each of the pattern sets for the four paradigms included in this research. Through this study, it became clear that, despite their obvious differences, all of the programming pattern sets shared some common traits. We refer to this phenomenon as *inherent commonality*, meaning that all programming paradigms share certain traits that are common, not just to solutions, but to problems themselves.

For example, all the programming paradigms permit the manipulation of data in some form. The philosophical particulars might differ (for example, imperative programming implies side effects, while functional programming eschews side effects, etc.), but the fact remains that in any problem solving endeavor, there is data involved in some way. Even in non-von Neumann computing (such as neural networks) data is introduced into the system in some form. Indeed, in any problem solving environment, whether related to computing or not, there is at the very least a problem and its solution, both of which can be loosely termed as *Data* (in the form of input and output).

Similarly, in any problem solving endeavor, some mechanism is employed to achieve a solution, whether that mechanism is by its nature imperative, declarative, or something in between. Even the philosophically simplest form of problem solving—the asking of the oracle—involves a mechanism, although the operational details of the mechanism may not be known to the asker. We have referred to these mechanisms in this pattern set as *Operative Units*.

Operative units must have access to both data and other operative units in order to perform their appropriate actions. At the very least, an operative unit must be able to accept data as input and produce data as output. Within various programming paradigms the means of accessing data and operative units may vary greatly. In some paradigms, such as object-oriented, these concepts are tightly coupled, with data encapsulated with operative units in object classes. We refer to these means of interaction between operative units and data as *Access Points*.

These three areas of inherent commonality (data, operative units, and access points) form broad equivalence classes of which all programming languages are composed. We can now study multiparadigm programming by exploring the various ways in which programmatic elements can be drawn from these equivalence classes and combined in creative ways (whether through adoptive or hybrid combination). We will find some of these combinations in existing multiparadigm programming languages, such as Leda. Others will be new and original.

11.4 “Multiple Paradigm” versus “Multiparadigm”

It is important to differentiate two separate styles of multiparadigm programming, strongly related to the type of combination used to create new programming elements or concepts. In this research, we will use the term *multiple paradigm* to refer to situations in which the integrity of an individual paradigm is preserved within some narrow context. In other words, adoptive combination has brought a programming element from one paradigm into contact with a programming element from another paradigm without dramatically compromising the paradigmatic purity of either.

We will use the term *multiparadigm* to refer to situations in which the integrity of each of the combined paradigms is not preserved. In other words, hybrid combination has brought a programming element from one paradigm into contact with a programming element from another paradigm in such a way that the new element or concept does not strictly pertain to either paradigm, but pertains either to both, or to some new, unique paradigm.

Figure 11.1 lists the patterns contained in this pattern set. All of the principles described in these introductory sections (such as inherent commonality, adoptive and hybrid combination, etc.) are embodied in these individual patterns. For the patterns that relate to the combination of programmatic elements, no more than two elements are combined to create a resultant multiparadigm or multiple paradigm programming pattern. To do more would perhaps yield interesting results, but would have increased the complexity of this task to a prohibitive level. Future research might explore combining more than two elements either by combining several elements from different paradigms, or by combining multiparadigm elements with other elements (in essence blending three or more elements without inherently expanding the complexity of the task).

Following this pattern set, Section 11.5 contains a detailed study and analysis of the multiparadigm programming language Leda in light of this new multiparadigm programming pattern set.

- *Multiparadigm Program*
- *Inherent Commonality*
 - *Program*
 - *Data*
 - *Operative Unit*
 - *Access Point*
- *Paradigmatic Interchangeability*
- *Adoptive Combination*
 - *Multiple Paradigm Programs in a Single System*
 - *Multiple Paradigm Data Declaration*
 - *Multiple Paradigm Operative Unit*
 - *Multiple Paradigm Access Point*
- *Hybrid Combination*
 - *Multiparadigm Data Declaration*
 - *Multiparadigm Operative Unit*
 - *Multiparadigm Access Point*

Figure 11.1. Hierarchy of Multiparadigm Programming Patterns.

Pattern 11.1 *Multiparadigm Program*

Motivation:

The concept of problem solving within a computational model involves the creation of programs. Other patterns have presented the programming concept from one of four paradigmatic perspectives, including *Imperative Program*, *Object-Oriented Program*, *Functional Program*, and *Logical Program*. Strictly speaking, any program that combines elements of different paradigms in some unified way can be referred to as a *Multiparadigm Program*.⁹ The purpose for which multiple paradigms are combined is to enable characteristics of different paradigms to be applied where most appropriate in a single problem solution.

⁹ Note that this term ignores for the moment our vernacular that draws a distinction between “multiple paradigm” and “multiparadigm”.

Discussion:

The most fundamental attribute of a *Multiparadigm Program* is that a program in some way combines programmatic elements from various paradigms. In practice, almost all programming languages are at least minimally multiparadigm, since all languages typically borrow from other paradigms, even if only slightly. For example, functional programming supports input/output via monads, and logical programming supports execution control mechanisms, such as cuts.

Our concern here is with a more overt form of borrowing between paradigms that leads the programmer away from individual paradigms, and into new arenas where thought and problem solving change in varying degrees. We believe that within these new multiparadigm arenas of thought and design that certain elegant solutions may hide, undiscovered by traditional methods of programming and design.

Other Related Patterns: *Inherent Commonality, Paradigmatic Interchangeability, Adoptive Combination, Hybrid Combination, Imperative Program, Object-Oriented Program, Functional Program, Logical Program*

Pattern 11.2 <i>Inherent Commonality</i>

Motivation:

Despite the fact that different programming paradigms approach problem solving in different ways, all problem solving (particularly computational problem solving) shares certain commonalities. We use the term *Inherent Commonality* to refer to the fact that certain programmatic elements are common throughout all paradigms.

Discussion:

At the highest level, we can see commonality among the four paradigms under discussion in the simple fact that solutions in each of these paradigms yield programs. Programs in these paradigms also share commonalities in that they all contain data, operative units, and access points. Each of these areas of inherent commonality were

discussed in the introduction, and are treated in greater detail in subsequent lower level patterns.

The principle of *Inherent Commonality* is very critical to understanding hybrid and adoptive combination as well as paradigmatic interchangeability.

Lower-level Patterns: *Program*

Other Related Patterns: *Multiparadigm Program, Paradigmatic Interchangeability, Adoptive Combination, Hybrid Combination*

Pattern 11.3 Program

Motivation:

Despite the obvious differences between programming paradigms, and the ways in which they foster decomposition of problems and composition of solutions, all the major paradigms deal in programs that embody solutions within a particular paradigm. This simple fact that programs are created within paradigms becomes a potential building block that we can use to explore the nature of multiparadigm programming and the ways in which programmatic elements can be combined.

Discussion:

Once we view individual programs as potentially interchangeable, we can view multiparadigm architectural solutions as being composed of individual programs, each pertaining to a particular paradigm. Without this interchangeable concept, this approach to multiparadigm design does not immediately spring to the mind.

Example:

The interchangeability of whole programs is particularly visible in Unix systems in which pipelines are created, composed of separate programs piped together, through which data passes in some form. In this kind of system, the particular paradigms or other characteristics of the individual programs in the pipeline are of little or no interest, so long as they perform the desired operations on the data passing through.

Higher-level Patterns: *Inherent Commonality*

Lower-level Patterns: *Data, Operative Unit, Access Point*

Pattern 11.4 Data

Motivation:

Regardless of the particular paradigm being used, all programs within the four paradigms under consideration operate in some way on *Data*. This concept of *Data* can be abstracted away from the paradigm, and dealt with separately in such a way that unique programmatic and design techniques become more obvious.

Discussion:

The following sections describe the nature of *Data* in each of four paradigms.

Imperative: *Data* consists of variables, which are memory locations identified by symbolic names. *Data* can be declared globally, or it can be declared locally according to the scoping rules of the language being used.

Object-Oriented: *Data* is the same as in imperative programming, but it should be bundled with methods that operate on it within object classes. This preserves information hiding, and limits the negative influence of side effects.

Functional: *Data* is not considered to consist of memory locations, but as information that is symbolically representable and not destructible. Conceptually there are no side effects, since there are no variables.

Logical: *Data* are represented as facts that are stated at the beginning of a logical program. These facts can also be viewed as initial table entries in a (typically relational) database system.

Even though these approaches to *Data* differ dramatically from one another, it can be seen that they all essentially deal with the same thing—identifying the information that is known to the system at the time the problem solving begins, or otherwise laying out the data formats that will be used to solve the problem. If we look at these different

approaches from a distance, and recognize each of them as instances of the single concept of *Data*, we can view them as interchangeable, and begin to look at the ways in which these can be combined in interesting and innovative ways.

Higher-level Patterns: *Program*

Other Related Patterns: *Operative Unit, Access Point*

Pattern 11.5 *Operative Unit*

Motivation:

Regardless of the paradigm used to solve a problem, every paradigm includes the concept of some action being taken somewhere in the system, either explicitly stated by the programmer or implicitly understood by the system. This action provides a solution (or partial solution) to a given problem. We refer to these providers of solution as *Operative Units*.

Discussion:

The following sections describe the nature of *Operative Units* in each of four paradigms.

Imperative: *Operative Units* are functions or procedures which encompass the explicit instructions necessary to perform operations on data structures.

Object-Oriented: *Operative Units* are methods inside object classes that operate on the data with which they are encapsulated.

Functional: *Operative Units* are functions that exhibit referential transparency, and perform consistent operations on data that is passed through them. These functions are first class citizens and can be used anywhere that data can be used.

Logical: *Operative Units* are logical relations that can be accessed as part of a query by the programmer, or by other relations. Accessing a relation causes a logical search engine to be invoked that searches the potential solution space for an acceptable solution.

Even though these approaches to *Operative Units* differ dramatically from one another, it can be seen that they all essentially deal with the same thing—providing some means for all or part of a given problem to be solved, whether that solution is specified by the programmer or deduced by the programming environment or system. If we look at these different approaches from a distance, and recognize each of them as a single concept of *Operative Unit*, we can view them as interchangeable, and begin to look at the ways in which these can be combined in interesting and innovative ways.

Higher-level Patterns: *Program*

Other Related Patterns: *Data, Access Point*

Pattern 11.6 *Access Point*

Motivation:

Since all paradigms include the concept of operative units and data, there must exist means by which operative units may interact with data or with other operative units. Without this capability, there would be no ability to obtain the solutions or parts of solutions that these operative units render. We use the term *Access Points* to refer to the means by which operative units access data as well as other operative units.

Discussion:

The following sections describe the nature of *Access Points* in each of four paradigms.

Imperative: *Access Points* are function or procedure calls through which operative units are invoked, and data is transferred to and from the operative units. They may also be viewed as any logical grouping of instructions that operate on some data, regardless of the scope of that data.

Object-Oriented: *Access Points* are messages passed between objects. These messages indicate the object whose services are desired, and any data necessary for the operative unit to perform its function. Messages also carry answers back to the caller.

Functional: *Access Points* are function calls through which parameters are passed. Parameters may exhibit polymorphism, indicating that the functions are capable of performing operations on data of different types.

Logical: *Access Points* are logical relations used by the programmer to bring logical information together. Parametric elements that are bound to solutions are viewed as facts to the search engine, and unbound elements are viewed as holes in need of filling.

Even though these approaches to *Access Points* differ dramatically from one another, it can be seen that they all essentially deal with the same thing—providing some means by which an operative unit interfaces with data and other operative units. If we look at these different approaches from a distance, and recognize each of them as a single concept of *Access Point*, we can view them as interchangeable, and begin to look at the ways in which these can be combined in interesting and innovative ways.

Higher-level Patterns: *Program*

Other Related Patterns: *Data, Operative Unit*

Pattern 11.7 Paradigmatic Interchangeability

Alias: *Paradigmatic Substitution*

Motivation:

From the principle of inherent commonality comes the concept that there are common elements shared between various paradigms, so long as we view the paradigms from afar and ignore areas of conflict at the micro level. Once we see all programs as being composed (at least in part) of these shared elements, we can also look at the possibility of playing with the composition of the elements that are shared. We refer to this principle as *Paradigmatic Interchangeability*.

Discussion:

The principle of *Paradigmatic Interchangeability* is important to both adoptive and hybrid combination, although it is most readily visible in adoptive. If all programs

have a data portion, for example, we can look at the ways in which data is declared, allocated, or identified in the various paradigms. Using adoptive combination, we could simply use a data section from one paradigm embedded within a program from another paradigm. Using hybrid combination, we could permit the content of the data portion to combine elements from various paradigms into a hybrid multiparadigm data portion that can be interchanged for a data section of any given paradigm, or any combination of paradigms.

Examples:

To illustrate the principle of *Paradigmatic Interchangeability*, Figure 11.2 shows an example of a pipelined solution to a problem in which data, operative units and access points are identified. The particular paradigm employed for each of these elements is unknown at this level of design, and can theoretically be from any of the four paradigms we have discussed (in the case of adoptive combination) or be multiparadigm units (in the case of hybrid combination).

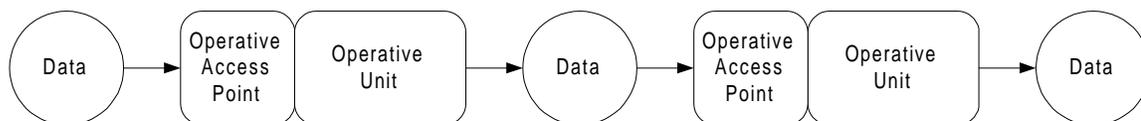


Figure 11.2. Problem solution with data, operative units and access points.

Other Related Patterns: *Multiparadigm Program, Inherent Commonality, Adoptive Combination, Hybrid Combination*

Pattern 11.8 Adoptive Combination

Motivation:

When combining elements of different paradigms, one approach is to “adopt” elements from one paradigm into programs from another paradigm. When an element from a paradigm is adopted into a program from another paradigm, it must be

accompanied by some clearly defined means through which elements from the program interact with elements within the unit being adopted. In this situation no “genetic” material is swapped, but the adopted element serves the same function as a similar unit in the same paradigm as the rest of the program. We refer to this as *Adoptive Combination*.

Discussion:

The ability to deal with *Adoptive Combination* depends heavily on the principle of paradigmatic interchangeability. This presupposes that there is an interchangeable piece of a given program whose place can be swapped with an element from another paradigm. Programs or programmatic elements that use *Adoptive Combination* to blend elements from different paradigms yield “multiple paradigm” programs or elements (as opposed to “multiparadigm” programs or elements).

Examples:

At the highest level, this principle might involve a system composed of different programs, each written in a different language, each following a different programming paradigm.

At a lower level, *Adoptive Combination* might involve the integration of functions from different languages or paradigms into a single program. Our experience is that in this case, there would tend to be a primary paradigm that controls the overall flow of the program. *Adoptive Combination* tends to yield uneven mixes of paradigmatic elements with one paradigm dominant and controlling, and the other subservient. As a practical example, modules might be created as object files or libraries in any paradigm or language, so long as an interface existed between the caller and the module. This was the model used many years ago with the Borland Turbo Pascal and Turbo Prolog compilers. The Turbo Prolog compiler had the capability of generating functions that could be called from Turbo Pascal. In this way Prolog functions could be accessed in a Pascal program. This is an example of early multiparadigm programming, but is very adoptive in nature.

Figure 11.3 illustrates *Adoptive Combination* by recasting the components of the problem solution used in Figure 11.2. In this example, `Text File` is probably

imperative data. Message Passing is an object-oriented access point. Object Class Method is an object-oriented operative unit. List of Tokens is probably functional data. Function Call is a functional access point. Logical Relation is a logical operative unit. Spell Check Results is probably imperative data. So, in this example, we exercise the principle of programmatic interchangeability by building our system from different pieces, borrowed from the most appropriate paradigm.



Figure 11.3. An example of *Adoptive Combination*.

Lower-level Patterns: *Multiple Paradigm Programs in a Single System, Multiple Paradigm Data Declaration, Multiple Paradigm Operative Unit, Multiple Paradigm Access Point*

Other Related Patterns: *Multiparadigm Program, Inherent Commonality, Paradigmatic Interchangeability, Hybrid Combination*

Pattern 11.9 Multiple Paradigm Programs in a Single System

Motivation:

One of the most accessible approaches to multiparadigm programming using adoptive combination involves the creation of two or more programs in some environment that interact in some way to form a functional system. Each of these programs can be written in a separate paradigm or language, and it is hidden to the system, since only the interfaces between the programs is a concern to the system as a whole.

Discussion:

To utilize *Multiple Paradigm Programs in a Single System*, a unifying architecture must exist that ties the programs together. For example, we can view a system wherein individual programs communicate by sending messages (or packets of information) to one another. So long as individual programs can send and receive messages, and respond to them, the paradigm or language used internally to solve their parts of the problem are irrelevant.

Similarly, an architectural pattern such as Pipes and Filters can be used to cause separate programs to interact as a single system. In this model, programs receive input through a pipe and send responses out another pipe. The individual programs are linked via pipelines that route output from one program as input to another. Once again, the paradigm or language of each of these programs is of no concern to the system so long as they can each receive input via pipes and send output via pipes.

Examples:

Figure 7.2 (in the set of functional Design Patterns) provides a programmatic example of a filtering mechanism implemented on a Unix system by linking separate filtering programs with pipes. The actual Unix command line instruction that forms this system is:

```
$ du | sort -rn | more
```

Figure 11.4 provides a graphic view of this implementation of *Multiple Paradigm Programs in a Single System*. Although these programs are probably implemented in either C or C++ on most Unix systems, the point of this illustration is that they need not be limited to any particular paradigm. Although this illustration shows a system comprised of data (the output of `du` through various transformations), access points (pipes), and operative units (individual programs `du`, `sort`, and `more`), each of these operative units is itself an individual program comprised of data, operative units, and access points.

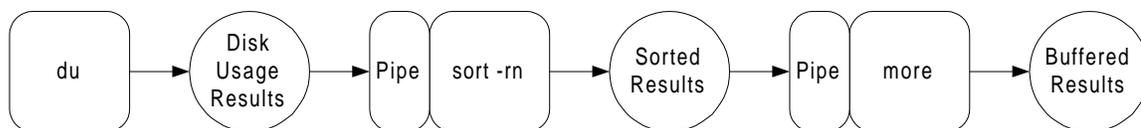


Figure 11.4. Example of *Multiple Paradigm Programs in a Single System*.

Higher-level Patterns: *Adoptive Combination*

Other Related Patterns: *Multiple Paradigm Data Declaration, Multiple Paradigm Operative Unit, Multiple Paradigm Access Point*

Pattern 11.10 *Multiple Paradigm Data Declaration*

Motivation:

Following the principle of paradigmatic interchangeability, *Multiple Paradigm Data Declaration* is a form of adoptive combination in which a program from a particular paradigm includes a data declaration from a different paradigm.

Discussion:

One of the important keys to effectively utilizing *Multiple Paradigm Data Declaration* is to realize that there is one paradigm (here referred to as the “primary paradigm”) that controls the overall flow of the program being written. The data area of the program, pertains to another paradigm (here referred to as the “secondary paradigm”), different from the primary paradigm. So, for example, combining imperative and logical paradigms using *Multiple Paradigm Data Declaration* might involve an imperative program with logical data declaration, or a logical program with imperative data declarations. It should be clear that these two scenarios, while involving the same two paradigms in combination, are potentially very different, depending on which paradigm is primary and which is secondary.

To fully appreciate the various ways of combining two paradigms using *Multiple Paradigm Data Declaration*, Table 11.1 summarizes the forms that such combination would take in each of the possible scenarios. In this table, items bolded and highlighted

represent contributions that are useful and/or somewhat interesting or unusual. Other items (not highlighted) are either available in pure programming paradigms or are not particularly interesting, because they contribute very little to new thought. The following key provides explanations for the latter:

- (A) The combination is already essentially provided for within the Primary Paradigm.
- (B) The combination is already essentially provided for within the Secondary Paradigm.
- (C) The combination is possible, but either is not particularly useful, or excessively violates the nature of the Primary Paradigm.

Table 11.1. Adoptive Combinations for *Multiple Paradigm Data Declaration*.

Primary Paradigm	Secondary Paradigm	Description
Imperative	Object-Oriented	Data types in data declarations are actually object classes, so variables declared can (and should) be accessed through appropriate methods from within the object class. This approach is almost exactly the same as a pure object-oriented approach to data declaration, since object-oriented data declaration is syntactically nearly identical to imperative. To fully merge these approaches, the declaration of a variable from an object class would permit the imperative program to peek inside. This would violate the purpose of the object class, and would probably not be valuable. (B)
Imperative	Functional	Named constants are declared, but cannot be modified. (A)
Imperative	Logical	Variables are declared through the use of statements of logical facts. The names given become variable names to the imperative program, and the values given are the default values of the variables.

Table 11.1, Continued

Primary Paradigm	Secondary Paradigm	Description
Object-Oriented	Imperative	Data declarations can be made either within objects, or within functions, or as global variables. (A)
Object-Oriented	Functional	Data declarations are done in a basically imperative fashion, but instead of variables, named constants are declared, and cannot be modified. (A)
Object-Oriented	Logical	Data declarations are done in a basically imperative fashion, but variables are declared through the use of statements of logical facts. The names given become variable names to the imperative program, and the values given are the default values of the variables.
Functional	Imperative	Data declarations are variables that occupy memory space, and can be altered. Unfortunately, this approach violates the essence of the functional program in a critical way. Declaring variables serves no essential purpose within a functional program perhaps other than paving the way for other uses of imperative components. (C)
Functional	Object-Oriented	Data declarations are variables defined from object classes. The same caveats apply here as to the use of imperative data declarations in a functional program. (C)
Functional	Logical	Logical facts are used to provide names and initial values. These names are not variables, and cannot be modified.
Logical	Imperative	The set of logical facts is presented via imperative variables and initial values, either done explicitly through assignment statements, or through initial assignment.

Table 11.1, Continued

Primary Paradigm	Secondary Paradigm	Description
Logical	Object-Oriented	The set of logical facts is presented either in a strictly imperative way (subsumed by the object-oriented perspective) or are accessed through some object class designed to create and initialize the logical facts. This could possibly be done through constructors.
Logical	Functional	The set of logical facts is set up through functional statements that give symbolic names to values that can then be manipulated.

Higher-level Patterns: *Adoptive Combination*

Other Related Patterns: *Multiple Paradigm Programs in a Single System, Multiple Paradigm Operative Unit, Multiple Paradigm Access Point*

Pattern 11.11 *Multiple Paradigm Operative Unit*

Motivation:

Following the principle of paradigmatic interchangeability, *Multiple Paradigm Operative Unit* is a form of adoptive combination in which a program from a particular paradigm utilizes an operative unit from a different paradigm.

Discussion:

One of the important keys to effectively utilizing *Multiple Paradigm Operative Unit* is to realize that there is one paradigm (here referred to as the “primary paradigm”) that controls the overall flow of the program being written. The operative unit pertains to another paradigm (here referred to as the “secondary paradigm”), different from the primary paradigm. So, for example, combining imperative and logical paradigms using *Multiple Paradigm Operative Unit* might involve an imperative program with logical

operative units (in the form of logical relations), or a logical program with imperative operative units (in the form of functions). It should be clear that these two scenarios, while involving the same two paradigms in combination, are potentially very different, depending on which paradigm is primary and which is secondary.

To fully appreciate the various ways of combining two paradigms using *Multiple Paradigm Operative Unit*, Table 11.2 summarizes the forms that such combination would take in each of the possible scenarios. In this table, items bolded and highlighted represent contributions that are useful and/or somewhat interesting or unusual. Other items (not highlighted) are either available in pure programming paradigms or are not particularly interesting, because they contribute very little to new thought. The following key provides explanations for the latter:

- (A) The combination is already essentially provided for within the Primary Paradigm.
- (B) The combination is already essentially provided for within the Secondary Paradigm.
- (C) The combination is possible, but either is not particularly useful, or excessively violates the nature of the Primary Paradigm.

Table 11.2. Adoptive Combinations for *Multiple Paradigm Operative Unit*.

Primary Paradigm	Secondary Paradigm	Description
Imperative	Object-Oriented	The object-oriented operative unit is a method, and can be invoked in a manner similar to any other function or procedure within an imperative program. (A)
Imperative	Functional	A purely functional operative unit will take some input and transform it. It will not be recognizably different from any other imperative operative unit. (A)
Imperative	Logical	A logical operative unit could be used in place of a function and use its backtracking mechanism to solve relational problems.

Table 11.2, Continued

Primary Paradigm	Secondary Paradigm	Description
Object-Oriented	Imperative	Despite the fact that a well-designed object-oriented program will have classes and methods, there is nothing to limit the program's ability to call imperative operative units. Viewed another way, individual methods are typically imperative already. (A)
Object-Oriented	Functional	The object-oriented program could access functional operative units in the same way that the imperative program does. Another approach is for the method in a given class to be a functional operative unit. This would permit the use of functional programming to perform certain operations, with the results stored by the object and side effects limited.
Object-Oriented	Logical	The object-oriented program could access logical relations in the same way that the imperative program does. Another approach is for the method in a given class to be a logical relation, so that the backtracking method could be used solve relational problems.
Functional	Imperative	An imperative operative unit can be built in such a way that it exhibits referential transparency. If done this way, calls to an imperative operative unit will appear the same as calls to functional operative units. (A)
Functional	Object-Oriented	The object-oriented operative unit is a method, and can be invoked in a manner similar to any other function within a functional program. (A)
Functional	Logical	A logical operative unit could be used in a manner similar to that of an imperative primary paradigm (above) with the exception that the logical unit would have to exhibit referential transparency to be useful.

Table 11.2, Continued

Primary Paradigm	Secondary Paradigm	Description
Logical	Imperative	Instead of a logical relation with backtracking, an imperative function or procedure could be substituted to perform other straightforward operations that are more suited to imperative programming.
Logical	Object-Oriented	This approach is very similar to using imperative as the secondary paradigm, except that the object-oriented method is used as an operative unit.
Logical	Functional	Instead of a logical relation with backtracking, a functional operative unit is substituted to perform other straightforward operations that are more suited to functional programming.

Higher-level Patterns: *Adoptive Combination*

Other Related Patterns: *Multiple Paradigm Programs in a Single System, Multiple Paradigm Data Declaration, Multiple Paradigm Access Point*

Pattern 11.12 *Multiple Paradigm Access Point*

Motivation:

Following the principle of paradigmatic interchangeability, *Multiple Paradigm Access Point* is a form of adoptive combination in which a program from a particular paradigm includes an access point between two or more operative units from different paradigms.

Discussion:

One of the important keys to effectively utilizing *Multiple Paradigm Access Point* is to realize that there is one paradigm (here referred to as the “primary paradigm”) that controls the overall flow of the program being written. The access point to some data or operative unit pertains to another paradigm (here referred to as the “secondary

paradigm”), different from the primary paradigm. So, for example, combining imperative and logical paradigms using *Multiple Paradigm Access Point* might involve an imperative program that uses logical relations to access an imperative function, using the relational elements of a logical program to facilitate flexible parameter passing. Likewise, it might involve a logical program that issues imperative function calls to other operative units. It should be clear that these two scenarios, while involving the same two paradigms in combination, are potentially very different, depending on which paradigm is primary and which is secondary. To fully appreciate the various ways of combining two paradigms using *Multiple Paradigm Access Point*, Table 11.3 summarizes the forms that such combination would take in each of the possible scenarios.

An additional point to keep in mind is that *Multiple Paradigm Access Point* only infers that the access point pertains to a unique paradigm. It does not infer anything about the nature of the data or operative unit itself. For instance, in the previous example, we could use a logical relation to allow an imperative program to call into an imperative function call, by taking advantage of the logical mechanism for calling between operative units. Similarly, we could use object-oriented message passing between two operative units, irrespective of the paradigms of the operative units in question.

To fully appreciate the various ways of combining two paradigms using *Multiple Paradigm Access Point*, Table 11.3 summarizes the forms that such combination would take in each of the possible scenarios. In this table, items bolded and highlighted represent contributions that are useful and/or somewhat interesting or unusual. Other items (not highlighted) are either available in pure programming paradigms or are not particularly interesting, because they contribute very little to new thought. The following key provides explanations for the latter:

- (A) The combination is already essentially provided for within the Primary Paradigm.
- (B) The combination is already essentially provided for within the Secondary Paradigm.
- (C) The combination is possible, but either is not particularly useful, or excessively violates the nature of the Primary Paradigm.

Table 11.3. Adoptive Combinations for *Multiple Paradigm Access Point*.

Primary Paradigm	Secondary Paradigm	Description
Imperative	Object-Oriented	Two imperative programs can communicate using object-style messages, or individual routines or threads in an imperative program can communicate using object-style messages. Data is passed between operative units in the messages or accessed via an appropriate message to a method that controls the data.
Imperative	Functional	There is not a significant difference between imperative and functional function calls. However, some aspects of functional calls, such as parametric polymorphism could expand the typical capabilities of imperative function calls.
Imperative	Logical	Instead of typical function calls, logical relational interfaces would be used to access operative units. This could provide great flexibility in the ways parameters are passed, allowing the natural binding capabilities of relations to determine what is in and what is out.
Object-Oriented	Imperative	This capability typically exists already in object-oriented programming languages, whether calls are made from within an object class or to some function. This could also be achieved by access to global variables by imperative methods, which is permitted in many object-oriented languages. (A)
Object-Oriented	Functional	This capability also typically exists already in Object-Oriented programming languages, whether calls are made from within an object class or to some function. The same caveats apply as for imperative as a primary paradigm (above). (A)

Table 11.3, Continued

Primary Paradigm	Secondary Paradigm	Description
Object-Oriented	Logical	Instead of typical calls to functions or methods, logical relational interfaces would be used to access operative units. This could provide great flexibility in the ways parameters are passed, allowing the natural binding capabilities of relations to determine what is in and what is out.
Functional	Imperative	There is not a significant difference between imperative and functional function calls. Any use of imperative parameter passing would probably critically violate the integrity of the functional program body. (C)
Functional	Object-Oriented	Functional programs could use message passing as a mechanism for transferring functional control similar to that of imperative as a primary paradigm (above).
Functional	Logical	Functional programs could use a relational interface in a similar way to imperative as primary paradigm (above).
Logical	Imperative	A logical program could make an imperative function call instead of a relational call. This would impose some order in the way parameters were passed, and would reduce some of the flexibility of logical parameter passing, but could be used for tasks that were not particularly logical in nature.
Logical	Object-Oriented	A logical program could use an object style message to communicate with an operative unit. There should be nothing that precludes the logical program using this mechanism to interface with an operative unit.

Table 11.3, Continued

Primary Paradigm	Secondary Paradigm	Description
Logical	Functional	A logical program could make a functional function call instead of a relational call. This would impose some order in the way parameters were passed, and would reduce some of the flexibility of logical parameter passing, but could be used for tasks that were not particularly logical in nature.

Higher-level Patterns: *Adoptive Combination*

Other Related Patterns: *Multiple Paradigm Programs in a Single System, Multiple Paradigm Data Declaration, Multiple Paradigm Operative Unit*

Pattern 11.13 Hybrid Combination

Motivation:

It is possible to combine elements of different paradigms in such a way that the genetic material of the two paradigms merges into something new. Such combination combines elements of both contributing paradigms in a very tightly integrated way. We refer to this meshing of paradigmatic elements as *Hybrid Combination*.

Discussion:

Hybrid Combinations tend to occur at a relatively low level, commonly at the idiomatic level. When these combinations occur, the resultant element is not immediately recognizable as pertaining clearly to either paradigm, but has elements of both. In the process, some of the pure paradigmatic elements may be compromised.

This principle is somewhat related to that of paradigmatic interchangeability, but is not limited to the same scope. By this, we mean that once we have identified interchangeable parts within a program, the pieces we insert may be of one or more paradigms, but may also pertain to something of a hybrid nature. Therefore, some of our

research depends on these principle of paradigmatic interchangeability, and explores *Hybrid Combination* within the scope of interchangeable pieces.

However, *Hybrid Combination* is not limited to this perspective because any combination of paradigmatic elements can occur at any appropriate idiomatic level, and need not be limited to areas that exhibit paradigmatic interchangeability. In other words, there may be areas in which there is not a general interchangeability between paradigms, but where a particular idiom can be grown, extended, changed or adapted to some purpose because of some hybrid combination of elements from two or more paradigms.

Lower-level Patterns: *Multiparadigm Data Declaration, Multiparadigm Operative Unit, Multiparadigm Access Point*

Other Related Patterns: *Multiparadigm Program, Inherent Commonality, Paradigmatic Interchangeability, Adoptive Combination*

Pattern 11.14 *Multiparadigm Data Declaration*

Motivation:

By combining aspects of data declaration from various programming paradigms in hybrid combination, we may be able to view data declaration in a new way, even more profound than just swapping data declarations of various paradigms through paradigmatic interchangeability. This process should yield unique, innovative *Multiparadigm Data Declarations*.

Discussion:

The approach of combining data declarations from different paradigms is a largely unexplored territory in multiparadigm programming research. To be fair, there isn't much that dazzles here, but there are potential ways of viewing the declaration of data, or indeed, the formation of problem statements, that could benefit by a careful exploration of hybrid combination at this level. See Section 11.5 for instances of *Multiparadigm Data Declaration* in Leda.

Most of the practical implementations of *Multiparadigm Data Declaration* will occur at the idiomatic level, and hence will be beyond the scope of this research.

In adoptive combination, one paradigm is identified as primary while the other is secondary. At first glance, it would appear that in hybrid combination these differentiations would not be needed, since elements are combined from each paradigm in a more egalitarian manner. Hence, it would appear on the surface that there would not necessarily be a dominant paradigm, but rather two (or more) paradigms contributing equally. However, despite this conceptual egalitarian participation between paradigms, in practice one paradigm tends to dominate the other. Put another way, in exploring hybrid combination, we may tend to begin with a first paradigmatic element, and then add characteristics of a second paradigmatic element to it. For example, we may consider the impact of adding logical features to imperative data, or the impact of adding imperative features to logical data. These two approaches will tend to yield different results, because of the dominant influence of the first paradigm. In hybrid combination we refer to first and second paradigms where elements of the second paradigm are added to the first. This differs from adoptive combination where we referred to primary and secondary paradigms because of the much stronger influence of the first paradigm.

To fully appreciate the various ways of combining two paradigms using *Multiparadigm Data Declaration*, Table 11.4 summarizes the forms that such combination would take in each of the possible scenarios. For each of these scenarios, the Description column is intended to be a high level view of the kinds of things that might be possible, and not an exhaustive treatise on the actual combinations, many of which occur at the idiomatic level.

In this table, items bolded and highlighted represent contributions that are useful and/or somewhat interesting or unusual. Other items (not highlighted) are either available in pure programming paradigms or are not particularly interesting, because they contribute very little to new thought. The following key provides explanations for the latter:

(A) The approach is already essentially provided for within the two paradigms.

(B) The approach embodies fundamental conflicts that cannot be easily resolved.

Table 11.4. Hybrid Combinations for *Multiparadigm Data Declarations*.

First Paradigm	Second Paradigm	Description
Imperative	Object-Oriented	As soon as elements of object-oriented data declaration are added to an imperative programming language, it becomes almost entirely object-oriented. (A)
Imperative	Functional	Flexible elements of functional typing could be leveraged in imperative declarations.
Imperative	Logical	Imperative declarations could add logical elements either as part of initialization, or as a form of typing.
Object-Oriented	Imperative	Object-oriented data declaration is already the result of such hybrid combination. It incorporates elements of imperative data declaration in a hybrid way. (A)
Object-Oriented	Functional	Flexible elements of functional typing could be leveraged in object-oriented declarations within objects.
Object-Oriented	Logical	Combination could be similar to that achieved by combining logical with imperative.
Functional	Imperative	Imperative elements of data structure could be leveraged in functional declarations.
Functional	Object-Oriented	Object declarations could be directly used as types within functional data declarations.
Functional	Logical	Combination could be similar to that achieved by combining functional with imperative.
Logical	Imperative	Logical facts could include imperative elements.
Logical	Object-Oriented	Combination could be similar to that achieved by combining logical with imperative.
Logical	Functional	Combination could be similar to that achieved by combining logical with imperative.

Higher-level Patterns: *Hybrid Combination*

Other Related Patterns: *Multiparadigm Program, Multiparadigm Operative Unit, Multiparadigm Access Point*

Pattern 11.15 <i>Multiparadigm Operative Unit</i>

Motivation:

By combining aspects of operative units from various programming paradigms in a hybrid combination, we may be able to view operative units in a new way, even more profound than just swapping operative units of various paradigms through paradigmatic interchangeability. This process potentially yields unique, innovative *Multiparadigm Operative Units*.

Discussion:

Most of the research in multiparadigm programming has focused on programmatic operations, those statements that constitute the bulk of any given *Multiparadigm Operative Unit*. Indeed, there is much research still to be done in this area and a wide variety of ways in which elements from different paradigms can be leveraged and merged. See Section 11.5 for instances of *Multiparadigm Operative Units* in Leda.

Many of the practical implementations of *Multiparadigm Operative Units* will occur at the idiomatic level, and hence will be beyond the scope of this research.

To fully appreciate the various ways of combining two paradigms using *Multiparadigm Operative Unit*, Table 11.5 summarizes the forms that such combination would take in each of the possible scenarios. For each of these scenarios, the Description column is intended to be a high level view of the kinds of things that might be possible, and not an exhaustive treatise on the actual combinations, many of which occur at the idiomatic level.

In this table, items bolded and highlighted represent contributions that are useful and/or somewhat interesting or unusual. Other items (not highlighted) are either available in pure programming paradigms or are not particularly interesting, because they contribute very little to new thought. The following key provides explanations for the latter:

(A) The approach is already essentially provided for within the two paradigms.

(B) The approach embodies fundamental conflicts that cannot be easily resolved.

Table 11.5. Hybrid Combinations for *Multiparadigm Operative Units*.

First Paradigm	Second Paradigm	Description
Imperative	Object-Oriented	The methods of object-oriented programs are typically imperative, so there is nothing distinguishable here. (A)
Imperative	Functional	A large part of this already happens in practice in imperative expression evaluation. Allow imperative instructions to dynamically create functions.
Imperative	Logical	Allow logical statements within imperative expressions.
Object-Oriented	Imperative	The methods of object-oriented programs are typically imperative, so there is nothing distinguishable here. (A)
Object-Oriented	Functional	Same contribution as imperative with functional.
Object-Oriented	Logical	Since object-oriented methods are typically imperative within their context, same contribution as imperative with logical.
Functional	Imperative	Adding imperative elements to functional elements presents a fundamental conflict. (B)
Functional	Object-Oriented	Same problem as functional with imperative. (B)
Functional	Logical	Allow logical statements within functional expressions.
Logical	Imperative	Allow imperative statements or expressions within logical relations.
Logical	Object-Oriented	Same contribution as logical with imperative.
Logical	Functional	Same contribution as logical with imperative.

Higher-level Patterns: *Hybrid Combination*

Other Related Patterns: *Multiparadigm Program, Multiparadigm Data Declaration, Multiparadigm Access Point*

Pattern 11.16 <i>Multiparadigm Access Point</i>

Motivation:

By combining aspects of access points from various programming paradigms in a hybrid combination, we may be able to view access points in a new way, even more profound than just swapping access points of various paradigms through paradigmatic interchangeability. This process should yield unique, innovative *Multiparadigm Access Points*.

Discussion:

There has been little research on the subject of merging different programmatic elements to create hybrid access points. But some of the potential power of multiparadigm programming could ultimately be derived from the flexibility in accessing operative units and data, particularly in distributed environments. See Section 11.5 for instances of *Multiparadigm Access Points* in Leda.

Most of the practical implementations of *Multiparadigm Access Points* will occur at the idiomatic level, and hence will be beyond the scope of this research.

To fully appreciate the various ways of combining two paradigms using *Multiparadigm Access Point*, Table 11.6 summarizes the forms that such combination would take in each of the possible scenarios. For each of these scenarios, the Description column is intended to be a high level view of the kinds of things that might be possible, and not an exhaustive treatise on the actual combinations, many of which occur at the idiomatic level.

In this table, items bolded and highlighted represent contributions that are useful and/or somewhat interesting or unusual. Other items (not highlighted) are either available in pure programming paradigms or are not particularly interesting, because they contribute very little to new thought. The following key provides additional information related to these uninteresting combinations:

- (A) The approach is already essentially provided for within the two paradigms.

(B) The approach embodies fundamental conflicts that cannot be easily resolved.

Table 11.6. Hybrid Combinations for *Multiparadigm Access Points*.

First Paradigm	Second Paradigm	Description
Imperative	Object-Oriented	This form of combination is already typically in place in object-oriented languages, particularly those, like C++ that originated from imperative languages. (A)
Imperative	Functional	Allow the return statement of an imperative function to return a function. (Note that there are other possible forms of combination that are in fundamental conflict.)
Imperative	Logical	Use logical forms of binding in imperative function calls.
Object-Oriented	Imperative	This form of combination is already typically in place in object-oriented languages, particularly those, like C++ that originated from imperative languages. (A)
Object-Oriented	Functional	Same combination as imperative with functional. Also combine message passing with functional composition.
Functional	Imperative	Imperative elements of function calls that are different from functional tend to be at philosophical odds and are probably not able to be resolved. (B)
Functional	Object-Oriented	Object-oriented mechanisms to access methods could be combined with functional calls.
Object-Oriented	Logical	Same combination as imperative with logical.
Functional	Logical	Logical binding concepts are fundamentally incompatible with functional calling. (B)
Logical	Imperative	Allow fuller mechanism for passing information to a relation using imperative parameters.
Logical	Object-Oriented	Same contribution as logical with object-oriented.
Logical	Functional	Same problem as functional with logical. (B)

Higher-level Patterns: *Hybrid Combination*

Other Related Patterns: *Multiparadigm Program, Multiparadigm Data Declaration, Multiparadigm Operative Unit*

11.5 An Analysis of Leda Using Multiparadigm Programming Patterns

This study of Leda is intended as a practical validation of the pattern set presented in this chapter. Having created this pattern set, it is important to compare this taxonomy with a multiparadigm language, such as Leda. The features of Leda are listed here, and are categorized according to the pattern category that best fits the multiparadigm language characteristic of Leda.

One of the challenges in analyzing Leda is that we can draw two distinctions relative to the multiparadigm nature of this language. First, the language incorporates hybrid (as well as adoptive) multiparadigm elements. Second, the language provides flexibility, so that multiparadigm elements can be created using the language. These are both important distinctions, and both contribute to the value of the language. By exploring specific language elements of Leda, this study will focus on the first. It would be a much more arduous task to explore the implications for more broad-reaching approaches to multiparadigm programming via Leda, and would be beyond the scope of this research.

This study shows that all of the essential elements of Leda have been sufficiently identified and described (at least at the categorical level) in this pattern set. A more detailed study of this pattern set should yield other potential multiparadigm programming elements (particularly of the hybrid variety) that may yet be incorporated into Leda in the future.

The table below follows the basic outline of the Leda programming language provided in Appendix A of [Budd 1995a]. The left column identifies specific language features, and the right column contains discussion of these features relative to the pattern set presented in this chapter.

Table 11.7. Analysis of the Multiparadigm Programming Language Leda.

Features	Categorizations and Discussion
1. Program Structure	
Pascal-ish overall structure.	Imperative. Somewhat paradigm independent, but leaning heavily on imperative.
Constant declarations.	Imperative, functional. Inherently functional, but not explicitly so, since it operates through an explicit imperative mechanism.
2. Variables and Assignment	
Variable declaration.	Imperative. Very traditional imperative mechanism for allocating memory location and giving it a name.
3. Types	
Function types.	Imperative, object-oriented, functional. There is a very functional flavor here, because this use of function types is intended to support the use of functions as first class values (see next item).
Functions as first class values.	Functional. At a language level, it is an idiomatic level hybrid merging of imperative variable declaration with functional function types. This may occur at any of the levels of granularity, as a data declaration, within an operative unit, or via an access point.
Class types	Object-oriented, imperative. This approach closely follows the object-oriented approach, which is itself a hybrid combination of object-oriented class types with imperative variable declarations.
4. Statements	
Assignment statement.	Imperative. Traditional imperative mechanism for changing the value of the memory location represented by the variable name.
Compound statement.	Imperative. Traditional imperative mechanism for combining several statements into one compound unit.
Conditional statement.	Imperative, logical. The conditional statement in Leda must return either a Boolean (traditional imperative) or a relation (logical). This involves a hybrid combination of imperative and logical elements at an idiomatic level within an operative unit.
While loop statement.	Imperative. Traditional conditional loop construct.

Table 11.7, Continued

Features	Categorizations and Discussion
For loop statement.	Imperative, logical. In Leda, the For loop is commonly used in conjunction with relations. Discussed below in Section 8: Relations.
Procedure call statement.	Imperative, functional. Discussed below in Section 5 (Functions).
Return statement.	Imperative, functional. Discussed below in Section 5 (Functions).
5. Functions	
Function declaration.	Imperative, functional. Leda's approach is fairly typical imperative syntax.
No return value.	Imperative. The only purpose for calling a function that does not return a value is to cause a side effect.
Return value.	Imperative, functional. An inherent commonality.
Parameter pass by value.	Imperative, functional. Default form of parameter passing in Leda. An inherent commonality.
Parameter pass by reference.	Imperative. Changes inside cause side effects outside.
Parameter pass by name.	Imperative, functional. Used to cause lazy evaluation. An inherent commonality.
6. Classes	
Object-oriented classes	Object-oriented. There are no other high-level wrappers in Leda. The overall object-oriented classes provide a wrapper within which paradigmatic interchangeability allows multiple paradigm operative units to be swapped.
Class declaration.	Object-oriented. Same as object-oriented classes above.
Constructor.	Object-oriented, functional. Common to object-oriented. Avoids the assignment side effect problem of imperative to potentially create a more functional view of data initiation.
7. Expressions	
Literal values.	Imperative, functional. Inherent commonality.
Pattern-matching expression (is).	Imperative, logical. Conditional statement in an imperative flavor, but performs logical binding, depending on conditions.

Table 11.7, Continued

Features	Categorizations and Discussion
Creating values of type function.	Imperative, functional. Hybrid combination of functional and imperative. Functions can be constructed and used on the fly via expressions.
Operator overloading.	Object-oriented, functional. Leda supports this form of ad hoc polymorphism.
8. Relations	
Relational data types.	Object-oriented, logical. Hybrid data declaration, permitting the declaration of logical relations within Leda.
Relational parameters.	Imperative, logical. Pass by reference parameters are used as the access point into relations, permitting values to be determined by the backtracking mechanism and passed back to the caller. Hybrid combination of imperative and logical.
Relational assignment.	Imperative, logical. Permits a form of assignment that causes the relation to be invoked, values to be created and assigned as a side effect. Hybrid combination of imperative and logical.
Conditional if statement.	Imperative, logical. Causes relation to be invoked to see if condition can be resolved. Hybrid combination of imperative and logical.
For loop statement.	Imperative, logical. Causes relation to be invoked repeatedly to provide satisfactory values for iteration. Hybrid combination of imperative and logical.
Imperative control flow in Logical statements.	Imperative, logical. Hybrid combination of imperative statements in a logical relation.
9. Parameterized Types	
Type parameters.	Imperative, functional, object-oriented. Provides the capability for parametric polymorphism. Hybrid combination of imperative and functional that is now viewed as common in object-oriented.

12.0 Multiparadigm Analysis and Design Patterns

12.1 Introduction

To this point in our research we have presented pattern sets for programmatic elements of our four target paradigms. We have also presented pattern sets for design elements for the same four paradigms. We have presented a set of methodological patterns, as well as a pattern set for multiparadigm programming. These ten pattern sets form a base of material that can now be drawn upon to explore issues related to multiparadigm analysis and design.

Our basic method for creating this current pattern set is best explained with the aid of Figure 12.1. On the left side of this figure are pattern sets for programming elements from each of the four paradigms studied. On the right side of this figure are pattern sets for analysis and design for these same four paradigms. In the center of this figure are methodological patterns that provide potential generative mappings from programming patterns to design patterns. At the bottom left are multiparadigm programming patterns. At the bottom right are multiparadigm design patterns.

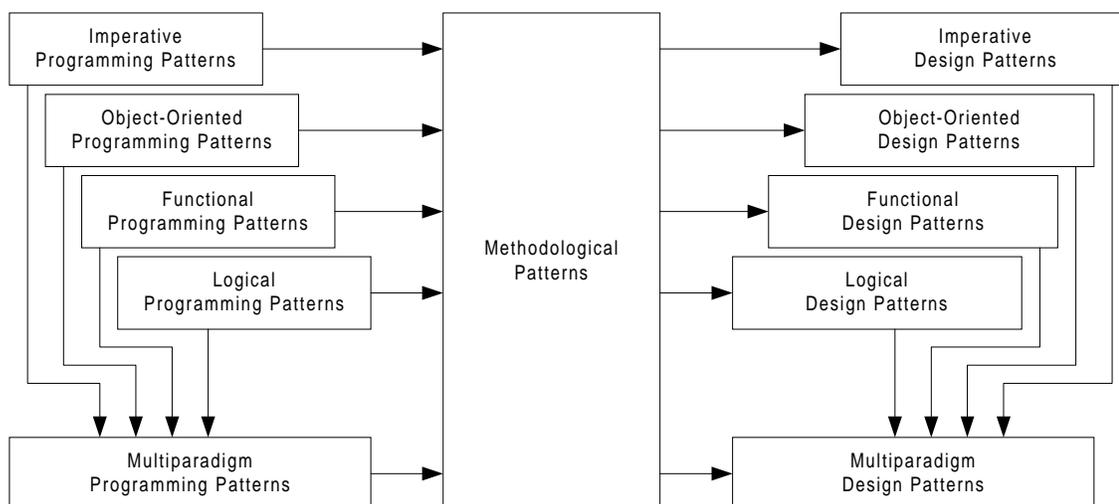


Figure 12.1. Organization of pattern sets created for this research.

The arrows in Figure 12.1 indicate the creative flow between pattern sets. Specifically, the mappings between programming patterns and design patterns gave rise to the set of methodological patterns in the center. The sets of programming patterns from the four paradigms contributed greatly to the set of multiparadigm programming patterns. Similarly, the set of design patterns from the four paradigms contributed significantly to the set of multiparadigm design patterns. Finally, the set of methodological patterns was applied in a generative fashion to the set of multiparadigm programming patterns to help contribute to the set of multiparadigm design patterns. Of greatest interest in this creative flow is the creation of a set of patterns for multiparadigm design. This pattern set is presented in this chapter, and its creation is described in greater detail in the next section.

12.2 Creating a Pattern Set for Multiparadigm Design

In our search for multiparadigm analysis and design patterns, we have explored two fundamental approaches: top-to-bottom and left-to-right. Their relative directional flow has reference to the flow of pattern creation in Figure 12.1. The patterns contained in this set are the result of following these two approaches.

The first approach involves movement from top to bottom. In this approach we examine each set of design patterns for each of the paradigms, looking for patterns and commonality between them. This is the basic approach that was used in creating the set of multiparadigm programming patterns presented in the previous chapter. Following a similar approach, we merged various design principles from the sets of design patterns that pertain to the four paradigms, and produced a set of multiparadigm design patterns.

The second approach involves movement from left to right. For each paradigm, we previously created a set of programming patterns and a set of design patterns. We also looked for patterns that could describe relationships between each pair of pattern sets (programming and design) pertaining to a single paradigm. We found generative patterns, and referred to them as methodological patterns, meaning that they captured elements of methodology that aided in the creation of design patterns given a particular set of programming patterns. These patterns have been captured in a separate pattern set and were presented in Chapter 10. This second approach to creating multiparadigm design

pattern involves looking at this set of methodological patterns and applying them to the set of multiparadigm programming patterns. The use of generative methodological patterns does not imply that the creation of resultant patterns is formulaic or deterministic. These methodological patterns inspire the creation of design patterns, but do not mandate their form or content. Indeed, this is one of the important characteristics of generative patterns—they inspire production within a given scope and purpose, but do not necessitate the form or content of the outcome.

12.3 Results of Top-to-Bottom Evaluation

The analysis of design patterns immediately revealed commonalities between the sets of paradigm-specific design patterns, which were, not surprisingly, similar to the ones that had been discovered when analyzing the various paradigm-specific programming patterns. This result is not surprising, given the evolving nature of design methodologies and multiparadigm programming languages. Design patterns tend to flow from programming patterns and multiparadigm programming tends to flow from paradigm-specific programming. If these two principles hold, design patterns should apply to multiparadigm programming the same way they do to paradigm-specific programming. During this research, we tried to distance ourselves from the experience of having created the previous pattern sets, in an attempt to not bias these results. But the common points were so strong that they could not be ignored. After identifying all the common patterns, and grouping them in a manner similar to the programming patterns, we discovered other patterns that did not fit. These all shared the common characteristic of being concerned with overall multiparadigm design principles. We therefore grouped these together in the last pattern of this set (*Guiding Principles of Multiparadigm Design*).

One of the implications of this result is a suggestion that the concept of multiparadigm design may be founded primarily on the principle of paradigmatic interchangeability. If we settle on the assumption that these multiparadigm programming patterns are an appropriate embodiment of multiparadigm programming principles and

practices, then the design patterns should naturally follow some kind of isomorphism with these patterns, and describe how to effectively utilize these patterns.

Having discovered patterns of multiparadigm design, we observed that these patterns naturally tend toward different levels of abstraction, with idiomatic patterns at the lowest level, design patterns in the middle, and architectural patterns at the highest level. The pattern *Multiparadigm Design Levels* introduces and expands on this concept.

12.4 Results of Left-to-Right Evaluation

Our left-to-right evaluation was rather straightforward, largely due to the opportunity to apply the methodological patterns of Chapter 10 in a straightforward fashion. The essence of this approach is to look first for isomorphism, secondly for non-isomorphism, and then to understand sources and destinations for any inconsistencies. This helps lay out explanations when differences are observed, and hints at potential sources for anything out of the ordinary. Applying these patterns to the multiparadigm programming patterns yielded essentially the same patterns as the set obtained by the first approach, but with some differences.

The most obvious unique characteristic of this pattern set is the use of design levels as a mechanism for introducing hierarchy to multiparadigm design patterns. These design level patterns clearly did not arise from our study of design patterns within other paradigms, nor was it strictly the product of applying methodological patterns to multiparadigm programming patterns. Rather, it was a synthesis of all of these efforts, coupled with a study of software design patterns and overall guiding principles of pattern-based design [Buschmann 1996]. I believe that these design level patterns are fundamental to design principles that apply in a pattern-based system, and so are an important part of a sufficiently powerful multiparadigm design approach that uses patterns as a foundation.

This pattern set first introduces *Multiparadigm Design Levels*, which forms an overarching structure for this pattern set and expands on the hierarchical principles espoused by [Buschmann 1996]. The rest of this set involves deeper examination of

architectural and design patterns as they apply to multiparadigm analysis and design. Figure 12.2 shows the hierarchy of patterns for this pattern set.

- *Multiparadigm Design Levels*
 - *Multiparadigm Architecture*
 - *Multiparadigm Design*
 - *Multiparadigm Idiom*
- *Multiparadigm Program Architecture*
 - *Multiparadigm Data Design*
 - *Multiparadigm Operative Unit Design*
 - *Multiparadigm Access Point Design*
- *Guiding Principles of Multiparadigm Design*

Figure 12.2. Hierarchy of Multiparadigm Analysis and Design Patterns.

Pattern 12.1 *Multiparadigm Design Levels*

Motivation:

When dealing with analysis and design issues, one of the most fundamentally difficult things to deal with is the appropriate level of abstraction at which to discuss the design of a system. By characterizing systems as being composed of architectures, designs, and idioms, we are more able to focus our analysis and design appropriately. We are also able to more effectively apply patterns of multiparadigm programming to problem solutions.

Discussion:

I've chosen to borrow the terms “architectural patterns,” “design patterns,” and “idiomatic patterns” from [Buschmann 1996]. These terms refer to varying levels of abstraction at which design can take place, from highest (architectural) to lowest (idiomatic). Each of these is discussed in greater detail in the lower-level patterns which follow. These hierarchical layers are not paradigm-specific. Buschmann deals with them

primarily in an object-oriented perspective, and we use them here in a multiparadigm perspective.

One of the most significant applications of these different levels of abstraction is that the multiparadigm programming patterns (which were presented in Chapter 11) fit comfortably into these three groups. Once grouped in this manner, they can be effectively analyzed and described from a design perspective.

Figure 12.3 shows the distribution of multiparadigm programming patterns into the three levels of abstraction. Each of these groupings is discussed in greater detail in appropriate subsequent patterns.

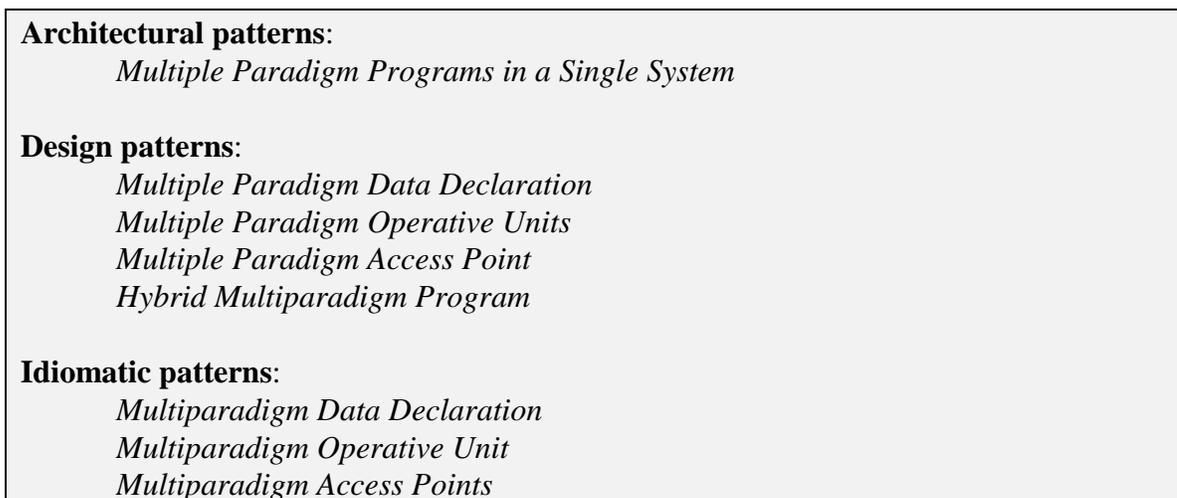


Figure 12.3. Multiparadigm programming patterns by level of abstraction.

Lower-level Patterns: *Multiparadigm Architecture, Multiparadigm Design, Multiparadigm Idiom*

Other Related Patterns: *Multiparadigm Program Architecture, Guiding Principles of Multiparadigm Design*

Pattern 12.2 *Multiparadigm Architecture*

Motivation:

In order to effectively perform multiparadigm design, one must first characterize the basic architectural design of the system under construction and make decisions that affect the overall focus and design of the system. This discussion and investigation falls under the domain of *Multiparadigm Architecture*.

Discussion:

“An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.” [Buschman 1996]

Within the context of multiparadigm design, this means that various separate paradigms or combined paradigmatic elements may be used to describe the overall structural organization of the system.

Buschmann [1996] cites three examples of architectural patterns: *Layers, Pipes and Filters*, and *Blackboard*. We have described *Pipes and Filters* in our set of functional design patterns, and utilize *Layers* in one of our case studies in Chapter 13. In all of these examples, a pattern is expressed that captures elements that might pertain to varying paradigms. This schema guides the overall form of the system, but does not necessitate a particular paradigmatic approach to a solution.

Of the multiparadigm programming patterns presented in Chapter 11, only one pattern was identified as pertaining to the set of architectural patterns (see Figure 12.4). Of the eight multiparadigm programming patterns involving adoptive and hybrid combination, only this pattern involves issues at a high enough level to affect overall architectural decisions. The others are applicable primarily at a lower level.

<i>Multiple Paradigm Programs in a Single System</i>

Figure 12.4. Architectural patterns for multiparadigm programming.

Higher-level Patterns: *Multiparadigm Design Levels*

Other Related Patterns: *Multiparadigm Design, Multiparadigm Idiom*

Pattern 12.3 <i>Multiparadigm Design</i>

Motivation:

Once a system has been architected, many design decisions must follow. The *Multiparadigm Design* patterns fall lower in abstraction than the *Multiparadigm Architecture* patterns, but higher than the *Multiparadigm Idiom* patterns.

Discussion:

“A *design pattern* provides a scheme for refining the subsystems of components of a software system, or the relationship between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context” [Buschmann 1996].

Based on our pattern language for multiparadigm programming, this can easily be seen to apply to the various interchangeable elements that compose almost all programs (data, operative units, access points). *Multiparadigm Design* patterns can be applied where these programmatic components are used, and will help determine their internal composition (leveraging adoptive and hybrid combination).

Buschmann [1996] cites several examples of design patterns, in several arenas of design, including: Structural Decomposition (*Whole-Part*), Organization of Work (*Master-Slave*), Access Control (*Proxy*), Management (*Command Processor, View Handler*), Communication (*Forwarder-Receiver, Client-Dispatcher-Server, Publisher-Subscriber*). The majority of the object-oriented design patterns movement has concerned itself with patterns at this level of abstraction.

It is not our intention to recreate the large (and rapidly growing) body of work in design patterns. Rather, our intention is to address issues that affect the intermingling of elements from different paradigms to create cohesive solutions to difficult problems.

Of the multiparadigm programming patterns presented in Chapter 11, four patterns were identified as pertaining to the set of architectural patterns (see Figure 12.5).

<p><i>Multiple Paradigm Data Declaration</i> <i>Multiple Paradigm Operative Units</i> <i>Multiple Paradigm Access Point</i> <i>Hybrid Multiparadigm Program</i></p>

Figure 12.5. Design patterns for multiparadigm programming.

Higher-level Patterns: *Multiparadigm Design Levels*

Other Related Patterns: *Multiparadigm Architecture, Multiparadigm Idiom*

<p>Pattern 12.4 <i>Multiparadigm Idiom</i></p>

Motivation:

At a certain point in any design process, decisions must be made that reach a low level where we move closer to actual implementation, with greater dependency on a target programming language. This level is referred to as the idiomatic level.

Discussion:

“An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationship between using the features of the given language” [Buschmann 1996].

Notice that we are now very close to the language-specific level. In the case of multiparadigm programming, this means that we are merging paradigmatic elements at

such a tight level that it may require some overlap of paradigm within the language being used.

Budd's text describing Leda [Budd 1995a] contains many idiomatic patterns, but none of them are named or identified as patterns. We have explored some of the idiomatic patterns in Leda in a previous study [Knutson 1996a]

Very little of the work in the object-oriented design patterns movement has concerned itself with idiomatic patterns, although some have pushed into this domain (see [Coplien 1997]).

Of the multiparadigm programming patterns presented in Chapter 11, three patterns were identified as pertaining to the set of idiomatic patterns (see Figure 12.6).

<p><i>Multiparadigm Data Declaration</i> <i>Multiparadigm Operative Unit</i> <i>Multiparadigm Access Points</i></p>

Figure 12.6. Multiparadigm programming patterns by level of abstraction.

Higher-level Patterns: *Multiparadigm Design Levels*

Other Related Patterns: *Multiparadigm Architecture, Multiparadigm Design*

<p>Pattern 12.5 <i>Multiparadigm Program Architecture</i></p>

Motivation:

There exist a number of architecture patterns that describe the overall structure of systems or programs. The *Multiparadigm Program Architecture* pattern describes the use of architectural patterns in such a way that paradigmatic elements can be effectively shared.

Discussion:

The process of determining an appropriate program architecture involves two separate phases. First, an overall structure must be selected that is suitable for the problem being solved. Once an architecture has been selected, one must then ask questions concerning the appropriate paradigm or paradigms to be used within that architecture at a course-grained level.

Certain architectures may suggest a particular paradigm. Although a problem may well be solved completely in a single paradigm, that decision does not have to be determined at this point in the design process. We should only be concerned with an overall perspective (for example, a client-server architecture with one server and some number of clients).

In traditional design, we tend at this level to determine an overall perspective based on a particular paradigm. But in fact, we typically encounter instances of paradigmatic interchangeability if we are aware that they exist. Still, it is common for one paradigm to dominate our view, even when some amount of paradigmatic mixing occurs at the architectural level. It is most common (particularly in Leda) to follow a primarily object-oriented perspective for overall program design, allowing individual pieces to be designed in an appropriate way later in the process. This is because the object-oriented approach provides the greatest degree of flexibility at the architectural level among the four paradigms in this study (see [Knutson 1997a]). Still, some problems may seem best suited to an overall functional or logical treatment, and these options should not be limited.

Once an overall perspective has been determined, individual design pieces should be identified at a course-grained level. Note that an architecture can include more than one program. In such a case, each program can be created using different paradigms. However, if we limit our discussion to a single program, our architecture now brings us to look in a course grained way at the individual pieces that comprise a multiparadigm program (specifically data, operative units, and access points).

The particular paradigmatic inclinations of the various pieces of a program (or system composed of several programs) are largely irrelevant during the architectural

phase. For this reason the architectural patterns operate above the design level, and are concerned only with course grained demarcations. Multiparadigm elements become an issue in architecture when a single system is designed in such a way that the individual pieces of the system come from different paradigms, but are function as a cohesive program. During this architectural phase, we examine, in a broad way the relationships between the data, operative units, and access points that will make up our system. The specific design decisions that relate to each of these components occur during the next phase (multiparadigm design).

In Chapters 3, 5, 7 and 9, we identified paradigm-specific analysis and design patterns. Of these patterns, several pertain predominantly to *Multiparadigm Program Architecture*. These patterns are identified in Figure 12.7. As we create multiparadigm architectures, we should draw on these paradigm-specific patterns, and employ the principles outlined in the patterns for adoptive and hybrid combination. This set of paradigm-specific patterns then forms a pool of raw materials from which to draw.

Imperative	<i>Structured Analysis</i>
Object-Oriented	<i>Object-Oriented Analysis and Design</i>
	<i>Object Distribution</i>
	<i>Object Concurrency</i>
Functional	<i>Functional Analysis and Design</i>
Logical	<i>Logical Analysis and Design</i>

Figure 12.7. Analysis and design patterns for *Multiparadigm Program Architecture*.

Lower-level Patterns: *Multiparadigm Data Design, Multiparadigm Operative Unit Design, Multiparadigm Access Point Design*

Other Related Patterns: *Multiparadigm Design Levels, Guiding Principles of Multiparadigm Design*

Pattern 12.6 *Multiparadigm Data Design*

Motivation:

During program design, data must be laid out and designed. This process is referred to as *Multiparadigm Data Design*.

Discussion:

During the design process it is almost always the case that data is created in some form, and defined and declared in some way. The design process should involve looking at the data that needs to be declared, and should include an analysis of its basic nature. Certain kinds of data may be best conceptualized within a particular paradigm. Some of this decision may flow from the overall programmatic approach chosen previously as an architectural decision. We also may be unaware of certain kinds of data declaration that may be available to us. This process should lead us to explore the set of patterns related to data in Chapter 11.

If the nature of the data is such that a single paradigm is both sufficient and elegant to represent it, then a particular paradigm's data design should be used. This is referred to in general as multiple paradigm design, since it employs adoptive combination (see *Adoptive Combination* in Chapter 11).

In a situation where elements of several paradigms are combined, we are more fully at the mercy of the richness of the multiparadigm programming language that we are using, and must depend to some extent on the idioms available in the language. This is referred to in general as multiparadigm design, since it employs hybrid combination (see *Hybrid Combination* in Chapter 11).

In Chapters 3, 5, 7 and 9, we identified analysis and design patterns. Of these patterns, several pertain predominantly to *Multiparadigm Data Design*. These are identified in Figure 12.8. As we create designs for data, we should draw on these paradigm-specific patterns, and employ the principles outlined in the patterns for adoptive and hybrid combination. This set of paradigm-specific patterns then forms a pool of raw materials from which to draw.

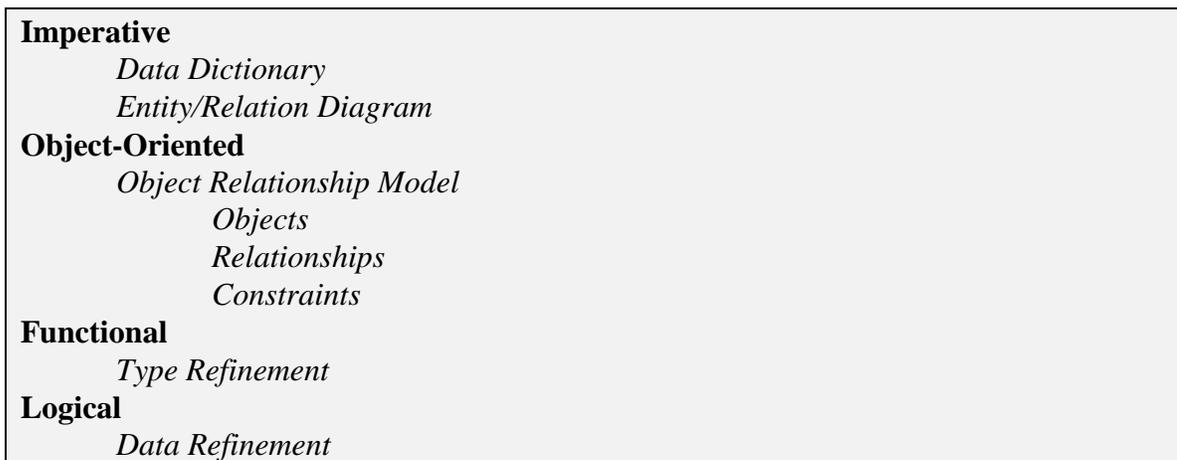


Figure 12.8. Analysis and design patterns for *Multiparadigm Data Design*..

Higher-level Patterns: *Multiparadigm Program Architecture*

Other Related Patterns: *Multiparadigm Operative Unit Design, Multiparadigm Access Point Design*

Pattern 12.7 Multiparadigm Operative Unit Design

Motivation:

As a program is being designed, individual operative units must be identified, and their purpose and function described. This process is referred to as *Multiparadigm Operative Unit Design*.

Discussion:

During design process, individual operative units are described. Each of these units must be individually designed. Each operative unit may be designed within a particular paradigm, or some hybrid unit might evolve. In this process, we should be referring to patterns from Chapter 11 that guide us in the process of combining elements

of different paradigms in different situations, and identifying inappropriate or impractical combinations.

If the nature of the operative unit is such that a single paradigm is both sufficient and elegant, then a particular paradigm's operative unit should be used. This is referred to in general as multiple paradigm design, since it employs adoptive combination (see *Adoptive Combination* in Chapter 11).

In a situation where elements of several paradigms need to be combined, we are more fully at the mercy of the richness of the multiparadigm programming language that we are using, and must depend on the idioms available in the language. This is referred to in general as multiparadigm design, since it employs hybrid combination (see *Hybrid Combination* in Chapter 11).

In Chapters 3, 5, 7 and 9, we identified analysis and design patterns. Of these patterns, several pertain predominantly to *Multiparadigm Operative Unit Design*. These are identified in Figure 12.9. As we create designs for operative units, we should draw on these paradigm-specific patterns, and employ the principles outlined in the patterns for adoptive and hybrid combination. This set of paradigm-specific patterns then forms a pool of raw materials from which to draw.

Imperative	<i>Process Specification</i>
	<i>State Transition Diagram</i>
Object-Oriented	<i>Object Behavior Model</i>
	<i>States</i>
	<i>Triggers and Transitions</i>
	<i>Actions</i>
	<i>Timing Constraints</i>
	<i>Real-Time Objects</i>
Functional	<i>Functional Composition</i>
Logical	<i>Query Refinement</i>

Figure 12.9. Analysis and design patterns for *Multiparadigm Operative Unit Design*.

Higher-level Patterns: *Multiparadigm Program Architecture*

Other Related Patterns: *Multiparadigm Data Design, Multiparadigm Access Point Design*

Pattern 12.8 <i>Multiparadigm Access Point Design</i>

Motivation:

During design, one must be concerned with the interactions that take place between operative units and either data or other operative units (or both). These design decisions are made at around the same time that decisions concerning operative units are made. We refer to this process as *Multiparadigm Access Point Design*.

Discussion:

During the design of a program or system, interface issues arise. These may take a number of forms, depending on the environment in which the system is being designed. For example, in a distributed system, the interface may involve a design and understanding of the distributed mechanism for remote communication and the particular formatting of information in packets. Within a single program, this typically takes the form of function calls, or relational queries, or methods. In multiparadigm design, the access point can be separated from the operative units and data.

If the nature of the access point is such that a single paradigm is both sufficient and elegant to represent it, then a particular paradigm's access point design should be used. This is referred to in general as multiple paradigm design, since it employs adoptive combination (see *Adoptive Combination* in Chapter 11).

In a situation where elements of several paradigms are combined, we are more fully at the mercy of the richness of the multiparadigm programming language that we are using, and must depend to some extent on the idioms available in the language. This is referred to in general as multiparadigm design, since it employs hybrid combination (see *Hybrid Combination* in Chapter 11).

In Chapters 3, 5, 7 and 9, we identified analysis and design patterns. Of these patterns, several pertain predominantly to *Multiparadigm Access Point Design*. These are identified in Figure 12.9. As we create designs for access points, we should draw on these paradigm-specific patterns, and employ the principles outlined in the patterns for adoptive and hybrid combination. This set of paradigm-specific patterns then forms a pool of raw materials from which to draw.

<p>Imperative</p> <ul style="list-style-type: none"> <i>Data Flow Diagram</i> <i>Data Flow</i> <i>Control Flow</i> <p>Object-Oriented</p> <ul style="list-style-type: none"> <i>Object Interaction Model</i> <p>Functional</p> <ul style="list-style-type: none"> <i>Type Refinement</i> <i>Pipes and Filters</i> <i>Functional Composition</i> <p>Logical</p> <ul style="list-style-type: none"> <i>Query Refinement</i>

Figure 12.10. Design patterns for *Multiparadigm Access Point Design*.

Higher-level Patterns: *Multiparadigm Program Architecture*

Other Related Patterns: *Multiparadigm Data Design*, *Multiparadigm Operative Unit Design*

Pattern 12.9 Guiding Principles of Multiparadigm Design

Motivation:

In the various collections of design methodologies that have been created and used over the years, there are a collection of design principles that are not particular to a specific paradigm, but can be applied freely to multiparadigm design. These paradigm-independent principles are collected here in this pattern.

Discussion:

Each of these design patterns exists and is described in the pattern set from which it is borrowed. It is, therefore, not necessary to revisit each of these patterns here. However, we should point out that these patterns exist, that they were derived from individual design approaches specific to particular paradigms, and that they all have applicability to multiparadigm design.

In Chapters 3, 5, 7 and 9, we identified analysis and design patterns. Of these patterns, several fall into the category of Guiding principles of Multiparadigm Design. These are identified in Figure 12.11.

Imperative	<i>Top-Down Design</i>
Functional	<i>Adapting Similar Problems</i>
	<i>Stepwise Refinement</i>
	<i>Recursive Refinement</i>
	<i>Type Refinement</i>
Logical	<i>Think Non-Deterministically</i>

Figure 12.11. Design patterns for *Guiding Principles of Multiparadigm Design*.

Other Related Patterns: *Multiparadigm Design Levels, Multiparadigm Program Architecture*

13.0 Applying Multiparadigm Analysis and Design Patterns

13.1 Introduction

We have presented 11 pattern sets intended to aid in multiparadigm analysis and design. We seek to answer two final questions in this research: 1) How are these pattern sets best utilized in performing multiparadigm analysis and design? 2) Can these patterns and principles actually be applied effectively to real problems? We will answer the first question with a discussion of design methodologies and a proposed design methodology for multiparadigm design. We will answer the second question with three separate case studies in which the patterns and principles espoused in this study are applied to real examples.

13.2 Patterns and Design Methodologies

The traditional use of design methodologies follows predictable patterns. These methodologies typically lay out general steps to follow in solving a problem (or most problems) within a given paradigm. Such methodologies attempt to bring a general framework to the process of problem solving, without fully providing substantive material to contribute to a particular solution.

The currently popular object-oriented design patterns approach, on the other hand, moves in a nearly opposite direction. It presupposes no (or any) design methodology or paradigm for design (except for its strong object-oriented bias). Instead, it attempts to capture knowledge and solutions to various kinds of problems that may be applicable within particular problem domains.

The most appropriate approach to utilizing a pattern system effectively is to merge these two perspectives to solve a particular problem. For example, if your paradigm of choice is object-oriented, you could still use the Booch method, but at particular points where these decisions have to be made, design decisions can be fed in part by the patterns available in domain-specific pattern sets. We are a bit more limited in the multiparadigm design arena, since prior to this research there has never existed a formalized multiparadigm design methodology. The pattern systems that we've presented in this

research provide a number of things that may help, such as guiding principles, gleaned from the various paradigms.

In a general sense, it could be argued that one could use any design methodology by ignoring the particular biases (object-oriented or imperative, for example), and then applying multiparadigm patterns (such as those presented in this research) where applicable. Before presenting our multiparadigm design methodology, we should better understand potentially effective ways of merging traditional design methodologies with sets of domain-specific design patterns.

The following methodology for pattern-based design is provided by [Buschmann 1996], and lends insights into potential solutions to merging these two design approaches (emphasis is from the original):

1. **Specify the problem.** This is important because it establishes a problem domain from which to select appropriate patterns.
2. **Select the pattern category** that corresponds to the design activity you are performing.
3. **Select the problem category** that corresponds to the general nature of the design problem.
4. **Compare the problem descriptions. Select the patterns** whose problem descriptions and forces best match your design problem.
5. **Compare benefits and liabilities** of the candidate patterns.
6. **Select the variant** that best implements the solution to your design problem.
7. **Select an alternative problem category.**

Another approach to design methodology brought out by Buschmann involves an attribute that is common to many design pattern systems. In problem domain specific systems, the individual patterns typically include an “implementation” section. By selecting a single high-level pattern that approaches a solution to the general problem, and following the particular steps outlined in the implementation section (most of which will reference other patterns), we can use this as a form of methodology for doing concrete design using pattern systems. This is effective in some arenas, but is not particularly useful for our study of multiparadigm design, since such domain-specific pattern sets do not exist for us.

Finally, Buschmann gives a concise summary of an appropriate way to combine existing methodologies with pattern systems:

- Use any method you like to define an overall software development process and the detailed activities to be performed in each development phase.
- Use an appropriate pattern system to guide your design and implementation of solutions to the specific problems.
- If the pattern system does not include a pattern for your design problem, try to find a pattern from other pattern sources you know.
- If no pattern is available, apply the analysis and design guidelines of the method you are using.

At a surface level, this discussion seems somewhat underwhelming, providing very little insight. But upon closer examination, this is probably about the best one can do so long as one views patterns only as the repository of solutions to specific problems within specific contexts under specific forces. The pattern system we've presented here uses a more loose approach to patterns, allowing the pattern sets to capture and express relationships between patterns at a meta-level. In contrast, within the popular design pattern perspective, one must use patterns as sort of a dictionary look up in the middle of using a particular methodology (which will naturally be bound to a particular paradigm). This approach is not particularly suited to our goal of multiparadigm design, since we do not want to be limited by a particular paradigmatic bias.

In a sense, this approach solves one of the most fundamental shortcomings of general design methodologies. As stated by Jackson [1983]:

Failure to focus on problems has harmed many projects. But it has caused even more harm to the evolution of development METHOD. Because we don't talk about problems we don't analyze them or classify them. So we slip into the childish belief that there can be universal development methods, suitable for solving all development problems. We expect methods to be panaceas—medicines that cure all diseases. This cannot be. It's a good rule of thumb that the value of a method is inversely proportional to its generality. A method for solving all problems can give you very little help with any particular problem.

So we can see that the utilization of patterns in this context provides the necessary material to plug into a given process. Typically design methodologies provide a form of

scientific method, but devoid of content. Design patterns provide the content that makes it all work.

However, Coplien takes this issue a bit further, and suggests that patterns are not necessarily a universal plug-in either. He says, “One fear I harbor for patterns is that designers will look to them first for their design solutions” [Coplien 1996b]. Notice that Coplien does not say that designers should never look to patterns, but he suggests that there are other guiding approaches that should be considered, and that design patterns can be utilized as part of them. This seems to fit in well with the perspective expressed by Buschmann, but the following statement by Coplien sheds additional light:

In the recent past, we’ve tried to use object tools to solve everything. Patterns take us outside pedestrian object design methods, often into structures that are handled well by no existing paradigm. To me, that’s where patterns shine—the dark corners of design. To me, patterns cover only small holes in the design space: the broader design space lends itself well to the common techniques of well-known paradigms, and we should seek to use those paradigms where they fit. [Coplien 1996b]

It’s been the intent of this project to use patterns to shine light into some of the dark corners of design, particularly multiparadigm design. The result is a set of patterns that sheds light on design within various paradigms, and within two or more of these paradigms at the same time. Coplien and Buschmann advocate utilizing the techniques of well-known paradigm as appropriate, and utilizing patterns as a repository of domain-specific material from which to draw. To place these perspectives in context, however, one must remember that these authors operate largely in an object-oriented world where “design pattern” and “object-oriented” have quickly become synonymous. They may have more accurately written that “object-oriented design patterns” can be used as domain-specific resources to any “object-oriented design methodology.” We still encounter the challenge that “techniques of well-known paradigms” bring bias that can mask multiparadigm design, which inhibits our ultimate goal.

Our view of pattern-based design represents a departure from the mainstream patterns community. The patterns movement views patterns within a fairly narrow scope. According to them, a pattern captures the solution to a problem within a certain context

(including certain forces). These pattern systems therefore contain a sort of dictionary of potential solutions to particular problems. However, this is not exclusively what Alexander wrote about in his seminal works. Alexander's focus was on capturing in some structure those things that facilitate the living things that people do. Granted, it's easier to look at some of these things in the architectural world, but we believe that the object-oriented software community has gone to an extreme. They have moved now so far off of Alexander's original intentions that they ignore the flexibility of patterns to capture things other than solutions to specific problems within specific contexts.

The system of patterns presented in this dissertation has a much broader scope. The things that we've attempted to capture here relate not just to the systems being constructed, but to the design processes and thought patterns that programmers engage in during analysis and design. Specifically, we capture not only patterns that relate to programming and design activities, but patterns that describe the relationship between these activities. This approach is more closely related to Alexander's original work than it is to the current object-oriented patterns movement.¹⁰

The pattern sets presented in this research are not a collection of solutions to specific problems. Those domain-specific collections exist, and are proliferating. The solutions contained therein can be effectively used according to the approach suggested by Buschmann above and cited here. The patterns presented here relate to the business of analyzing, designing, and programming in a multiparadigm fashion. They can and should be used in conjunction with domain-specific patterns for particular problem areas.

¹⁰ Alexander states, "Each pattern then, depends both on the smaller patterns it contains, and on the larger patterns within which it is contained.... And it is the network of these connections between patterns which creates the language....In this network, the links between the patterns are almost as much a part of the language as the patterns themselves....It is, indeed, the structure of the network which makes sense of individual patterns, because it anchors them, and helps make them complete." With respect to this space between patterns, he further states, "Beyond its elements each building is defined by certain patterns of relationships among the elements....The fact is that the elements *themselves* are patterns of relationship" [Alexander 1979]. Hence, Alexander advocates a holistic view of patterns in which patterns may describe the relationships between patterns in infinitely higher and lower directions of abstraction.

13.3 A Pattern Based Methodology for Multiparadigm Design

The system of patterns presented in this research provides a way to view programs from the perspective of analysis, design, and programming in such a way that multiparadigm solutions can be obtained. In this section we present a basic methodology to guide the designer in using these pattern sets.

This methodology flows naturally from the multiparadigm programming and design pattern sets presented in Chapters 11 and 12. Our approach to multiparadigm design follows three distinct steps (see Figure 13.1). The headings in **bold** identify the broad distinctions in the phases of architecting, designing, and implementing. Within each of these are sections enclosed in brackets and *italicized* that point the designer to specific pattern sets for help. All of these begin with the word “Explore” and end with a reference to a specific pattern set. Seeking these pattern sets will provide guiding principles for the actions that follow in normal text. After this pointer to helpful patterns are the specific design steps that must be followed to complete the appropriate section of the overall system. Figure 13.1 contains a more complete summary of our multiparadigm design methodology.

We will follow this design methodology in three case studies in the sections that follow. These case studies were chosen from differing technology areas so that the domain-specific knowledge would vary greatly. Each of these three case studies also lends itself to a different architectural view. Each pertains to a very common technology, and each is benefited by multiparadigm design and implementation.

In all three of these case studies, we will contrast the solution obtained using our methodology with traditional solutions in these domains. It is neither our expectation nor our quest to derive solutions that are unusual, or radically different from those obtained following traditional design approaches, just for the sake of being different. We seek effective designs, and believe that incorporating elements from different paradigms will lead to such designs. We have presented a multiparadigm design methodology, and it can now be examined to see what kind of designs can be obtained using it. The three case studies that follow represent a preliminary study, and more research needs to be done in this area.

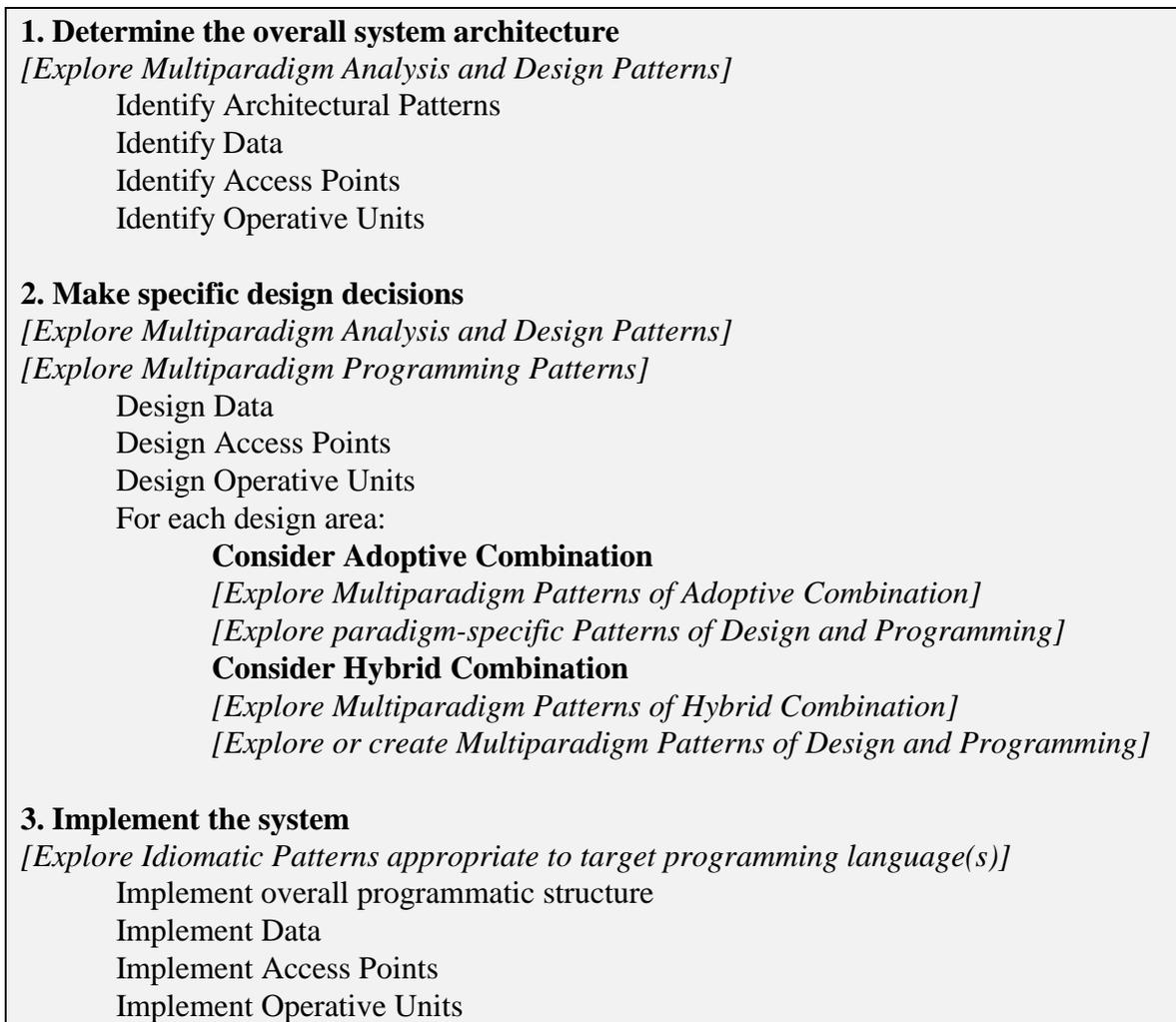


Figure 13.1. A multiparadigm design methodology.

We believe that our methodology can be used to produce designs that span the spectrum from paradigmatically pure to those that involve a high degree of integration of elements from the four paradigms. In these case studies, we expect to show several things. First, we expect that our methodology will lend insight into the opportunities for multiparadigm design, which in turn should produce designs that take fuller advantage of elements from different paradigms. These insights come, in part, from following our proposed methodology, and exploring the potential contributions of the various paradigms at appropriate stages in the design process. Second, we expect our approach to validate, rather than invalidate, some traditional design decisions relating to well-

understood technologies (like the three selected for these case studies). Still, we will show that in situations where traditional design decisions are effective, our methodology and pattern sets provide tools to understand why the design is effective, especially in light of other possible choices. Third, we will demonstrate that our designs tend to incorporate multiparadigm elements even when we accept an overall design architecture that could be commonly viewed as traditional.

13.4 Case Study: Multiparadigm Compiler Design

One of the first studies of multiparadigm design was published in a paper entitled “A Multiparadigm Approach to Compiler Construction” [Justice 1994]. The approach taken in this study focused on an implementation of a compiler written in the multiparadigm programming language Leda. Throughout this paper, the authors make design decisions that lead to a multiparadigm implementation of their compiler. Some of the design decisions that were made are shared after the fact, but little insight is given into their creative process. In fairness to the authors, their paper was more concerned with the merits of the multiparadigm programming language Leda than with broader issues related to design. Because of this, their design details are naturally sparse, and their focus is primarily at an idiomatic level, exposing the strengths of Leda. Our focus in this case study is specifically away from the idiomatic level and more focused on design issues. Hence, we provide a fuller treatment of multiparadigm design issues. This should not be viewed as a deficiency in the work by Justice et al, since our respective purposes are different.

As a case study, we will approach the same problem as Justice, but will employ the pattern-based design methodology previously described. Our approach to this case study will be two-fold. First, we will take an opportunity to comment on the study performed by Justice. Second, we will do our own design of a compiler using the multiparadigm design methodology and multiparadigm pattern sets that were previously introduced.

The problem as stated by Justice is: “The design and implementation of a compiler for an imperative language.” As they point out, “Compilers have traditionally

been modeled as a sequence of phases.” A good description of this traditional modeling can be found in [Aho 1986]. Figure 13.2 shows these phases.

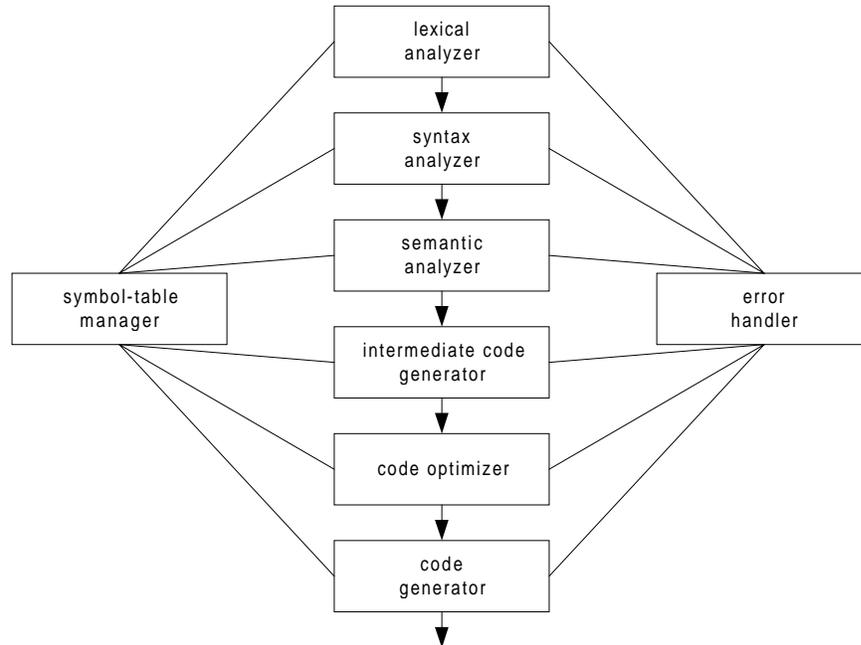


Figure 13.2. Phases of a compiler.

In contrast to this multi-phased approach, Justice et al. present what they refer to as a “multiparadigm compiler model”. Figure 13.3 shows their compiler design. First some comments about the study done by Justice. We believe that this "multiparadigm" compiler design is not inherently more multiparadigm than the "traditional" compiler model. It is an interesting and effective design, but both the old and the new models can be viewed as "a group of interacting computing agents." This new design is largely an object-oriented design in which the methods contained in the objects are permitted to be of differing paradigms, such as logical or functional. Data is also captured in objects with methods to provide access. From an object-oriented view, this new model assumes a certain amount of inherent multitasking, as each object is conceptually viewed as operating independently of others and interacting with them via message passing as needed. But all these descriptions could apply equally to the traditional model. So this

new model is fundamentally an object-oriented compiler design with multiparadigm methods.

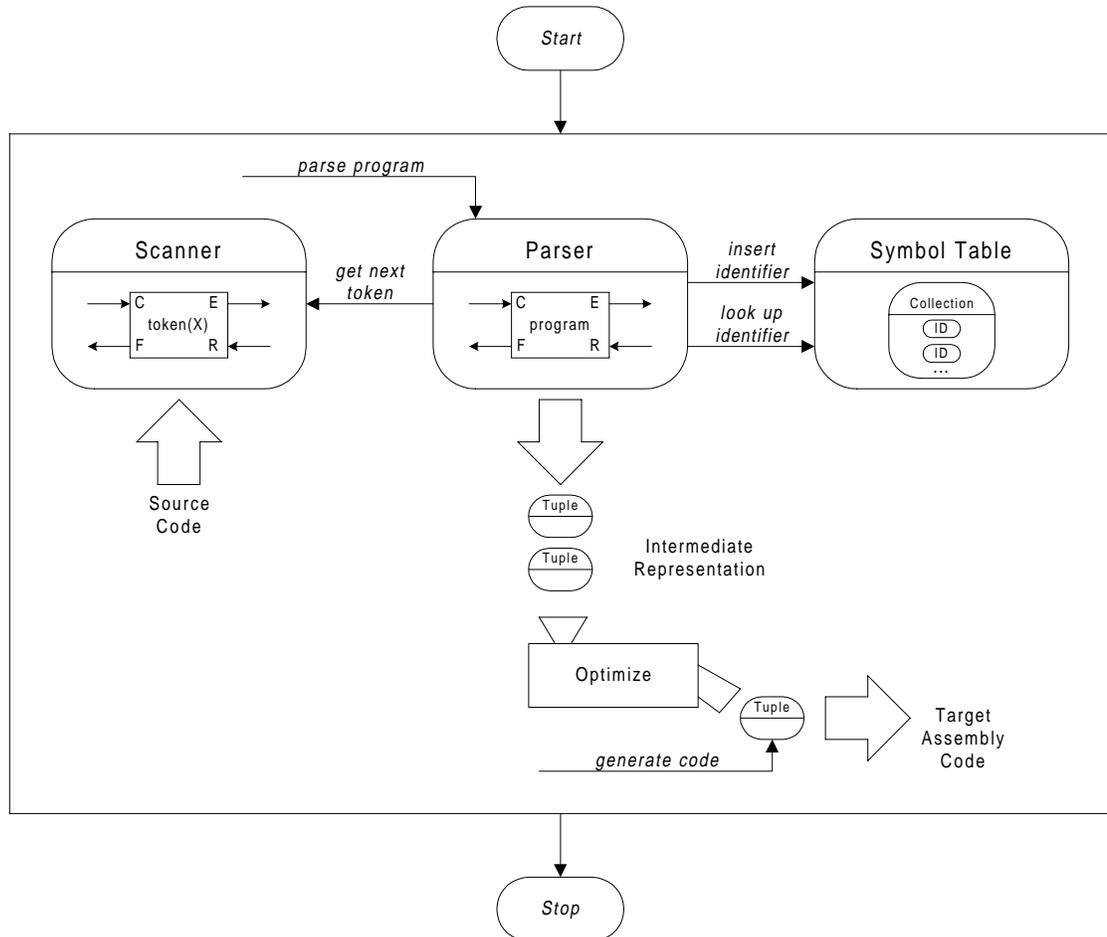


Figure 13.3. A Multiparadigm Compiler Model.

We capture this concept in this research in the pattern *Multiple Paradigm Operative Unit*. Specifically, Leda is strongly slanted toward the use of object-oriented design with interchangeable methods borrowed from various paradigms. The design taken by Justice is worthwhile and valuable, and illustrates multiparadigm design and implementation very well. But while it is a multiparadigm design, we don't believe it captures the essence of the "multiparadigm design" of a compiler. Neither do we believe that the traditional compiler architecture must be rejected as being single paradigm.

Specifically, we don't believe that we have to go to such a dramatic redesign of compiler organization to capture the essence of multiparadigm design. We can work within the "traditional" design model and get the job done just fine.

In the following sections of this case study, we present our description of the multiparadigm design of a compiler. Each of these sections follows the multiparadigm design methodology presented in Figure 13.1.

13.4.1 Determine Overall System Architecture

According to our methodology, the first step is to characterize our overall architecture. This step should involve some perusing of existing architectural patterns, most of which are largely paradigm independent, since they only loosely define data, operative units, and access points. The specific paradigms for these components can be filled in later. Buschmann et al. provide an excellent discussion of architectural patterns, and pointers to other collections [Buschmann 1996]. Mary Shaw [Shaw 1995] [Shaw 1996] has also made a study of architectural patterns. In finding an appropriate architectural pattern, the steps laid out by Buschmann and shared above are appropriate. We may find that our particular problem does not fall into one of the patterns available to us. In this case we begin to look at the overall interaction between operative units via access points, and look at data.

However, in this case, the traditional compiler model falls neatly into six phases, and involves a transformation from source code to object code through various type transformations. This should immediately bring to mind the *Pipes and Filters* pattern. Here is a thumbnail sketch of *Pipes and Filters* from [Buschmann 1996]: "The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems."

This pattern is also captured in Chapter 9 of this research among our set of functional design patterns. This architectural view is primarily functional, since it involves the transformation of input data to output data without changing the input, and

involves the composition of functions on the original input. Using the traditional model, we can conceptually represent this functional design pattern as:

```
Object Code = (Code Generator (Optimizer (Intermediate Code
Generator (Semantic Analyzer (Syntax Analyzer (Lexical
Analyzer (Source Code)))))))))
```

Another functional design pattern applies to this traditional compiler model. *Type Refinement* is a design approach that views the decomposition of the system based upon the transformations that the input goes through before it comes out the other end as output. The traditional compiler is a classic example of this functional design approach. In this case, the type refinement moves through these distinct phases:

```
Source Code -> Tokens -> Parse Tree -> Parse Tree ->
Intermediate Representation -> Optimized Intermediate
Representation ->Target Assembly Code
```

Not surprisingly, since *Type Refinement* and *Pipes and Filters* are both functional design patterns, they can be used in a complementary fashion such as in this case.

We have explored the functional approach to overall design, and have found a very applicable architectural pattern and some very useful design patterns that we can use to help us. It is useful at this point to briefly examine the other paradigms to see if another more obvious choice comes to mind. The imperative approach to compiler design is available, but it lends nothing to our functional design, so we reject it as more or less unpromising. The logical approach imposes a strong relational bias too early, so we reject it as not applicable in this situation at the architectural level. The object-oriented approach is very promising, and permits us to view the system as being composed of interacting objects. This is the approach taken by [Justice 1994]. If the traditional model is largely ignored, and a concise object-oriented design is sought, it would probably look very similar to their new model. However, we can also apply the object-oriented perspective to our *Pipes and Filters* approach, and view each phase as an object, with the pipes consisting of some form of message passing. The option of applying object-oriented principles to the next design phase is still available to us within the context of this functional view of the system.

One other decision ought to be made at this point. We have in our pattern sets a multiparadigm programming pattern called *Multiple Paradigm Programs in a Single System*. This approach leads us to create individual programs with defined access points. The alternative is to create subsystems within a given program. From one perspective, these two approaches are very similar and lead to similar architectural decisions. But they in fact lead to very different approaches. If we implement our compiler as multiple programs, we have unique decisions to be made as we design each phase as a separate program. For example, the programming language or paradigm we use in each phase is completely up for grabs. If we implement these as part of a single program, those decisions are largely made for us. Further, if we implement these together, they will necessarily share some programmatic glue that does not have to be there in the same way with multiple programs in a single system. In practice, compiler construction tools are examples of compiler construction as multiple programs in a single system. Parser generators can create syntax analyzers that examine source code and save output in some intermediate representation that can be used as input to a semantic analyzer. Similarly, scanner generators automatically generate lexical analyzers. Automatic code generators can take the intermediate representation of code with a set of transformation rules and generate final code. So, we see that in practice, compilers are indeed often composed of multiple programs functioning as a single system. Each of these programs could be written in different languages according to different paradigms.

There are clearly pros and cons to using multiple programs versus a single program. Multiple programs provide greater flexibility to a broader audience. These tools are typically adaptable depending upon the rule base that is included as input to the tool. Less flexible, but simpler, is a built-in system that moves data internally without having to deal with external representations. Because we seek a simple example, and we don't need flexibility or extensibility, we will choose to design our compiler as a single program with internal pieces to perform the functions of the six phases.

It is not our intent to reinvent any wheel that works well in practice. We don't intend to walk through the decades of history that have led to the six phases that are now commonly understood to be part of a functioning compiler. This makes our task simpler,

and permits us to move forward. At the architectural level, we will outline our system as being basically composed of six operative units: Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Optimizer, and Code Generator. We will view the access points between these operative units as loosely related to the concept of a pipe, but will permit some variance in the actual design as each interaction point is defined. For example, not all data that passes between operative units looks like a stream of data. For example, the input to the Semantic Analyzer is a Parse Tree. This tree needs to be created by the Syntax Analyzer and passed whole to the Semantic Analyzer. Similarly, when the Semantic Analyzer outputs its own tree, it should be passed in its entirety to the Intermediate Code Generator. Each of these access points should be designed separately to best meet the needs of the two Operative Units that need to share information.

We can look at data in a broad sense as originating as Source Code input to the program, moving through transformations courtesy of the operative units, and exiting as Object Code output from the program. The particular data design issues pertaining to each operative unit (and hence to each access point) should be designed at those junctures. The overall sense of input and output data should be loosely understood at this level. Source Code input should be in the form of a file of ASCII text containing the program to be compiled. Object Code output should be in the form of a file containing binary executable code appropriate to the target environment of the program.

Finally, we add two other operative units: a Symbol Table Manager, and an Error Manager. These two operative units are accessible to the other six operative units that make up the pipeline. In this sense, the addition of these two operative units breaks some of the purity of the functional *Pipes and Filters* approach to this overall architecture. This system could be designed to absolutely preserve the functional purity of the *Pipes and Filters* pattern, but at a high cost. It is much easier to allow access from all operative units to the Symbol Table Manager and the Error Manager. The addition of these two operative units introduces a kind of object-oriented twist to the overall architecture. The flow of data transformation from Source Code to Object Code flows through the pipeline, but

these two operative units exist to provide services to the pipeline, which is a very object-oriented perspective.

13.4.2 Make Specific Design Decisions

In this compiler system, there are six operative units in the pipeline, five access points between these operative units, two access points defining input and output to the system, and seven data definitions pertaining to the seven access points. In addition, there are two operative units providing additional services, and access points between these two operative units and all the other operative units in the pipeline. To design this system, each of these items needs to be designed. While we have yet to look at the design of each operative unit, we naturally begin to think about the overall nature of the operative units and access points to be designed. We will admit to a bias toward object-oriented structure to each of these, because the encapsulation mechanisms of object-oriented programming provide such a neat delineation of components, and message passing provides such a relatively clean interface. This does not mean that we are bound to use the object-oriented paradigm as the wrapper for these operative units, but it does mean that there will be a strong natural pull in that direction.

Note that at this point, we have chosen a broadly functional architecture (with an object-oriented twist), but we have not limited our ability to now choose from different available paradigms for each of the data, access point, or operative unit designs. In the following sections, we will discuss the design issues for each of these items, and the ways in which various paradigms can be combined for effective design.

Operative Unit: Lexical Analyzer

Ins and Outs: A Lexical Analyzer reads text and converts it into tokens.

Traditionally the input to a Lexical Analyzer is viewed as a stream of ASCII text, and its output is viewed as a stream of tokens. In this model the input is streaming into the Lexical Analyzer as fast as it can handle them, and it spits out tokens as quickly as it creates them. But this stream-centric view of data flow is not essential to the model of a Lexical Analyzer. For example, we can view the Lexical Analyzer as reading all input

into some kind of buffer (or being given the entire buffer, or grabbing the buffer) and tokenizing it on demand. We could also view the Lexical Analyzer as waiting for a request for a token before it begins reading sufficient text to create the next token. The stream-based view of the Lexical Analyzer tends toward a functional perspective since the input is transformed into some output according to some transformational algorithm. But we can also view the Lexical Analyzer from a more object-oriented perspective. From a demand-based perspective, the Lexical Analyzer can wait for a message from the Syntax Analyzer requesting that it deliver the next token. From a supply-based perspective, the Lexical Analyzer can create tokens and then deliver messages to the Syntax Analyzer that contain new tokens. This discussion about the input and output of the Lexical Analyzer needs to be formed into final design decisions during the discussion on the appropriate access points, but is impacted by the design decisions about the internal behavior of the Lexical Analyzer.

Internal Behavior: The Lexical Analyzer must interpret strings according to some rules and render them into tokens. Parsing input strings into tokens has traditionally been done imperatively. But the logical paradigm lends itself very well to this task. Logical programming tends to shine in situations that can be described by facts and rules. The task of parsing for tokens is based upon facts that define what tokens are made of, and rules that define how individual characters are combined into tokens. The rules of combination can even be made to reject characters that are not valid, triggering the error handler when appropriate.

Internal Data: In this logical design, internal data is captured by the facts and the rules associated with the relations. Strings and tokens are handled internally, but are primarily associated with the access points coming in and going out.

Summary: The behavior of the Lexical Analyzer is predominantly logical, but the wrapper is object-oriented. To be consistent with the overall pipeline architecture we've selected, the Lexical Analyzer receives a stream of characters from the Program Input access point and delivers tokens through the Token Passing access point to the Syntax Analyzer (supply-based delivery).

Operative Unit: Syntax Analyzer

Ins and Outs: The Syntax Analyzer receives tokens from the Lexical Analyzer and delivers a Syntactic Parse Tree to the Semantic Analyzer. Since we are defaulting to a push view of the interactions along the pipeline, the Syntax Analyzer will wait for tokens, and will actively deliver a parse tree to the Semantic Analyzer when it has been built.

Internal Behavior: The creation of a Syntactic Parse Tree will follow strict rules that describe legal statements in the programming language for which this compiler is being built. Logical relations will be very appropriate for this task, for reasons similar to those used for the Lexical Analyzer—the facts and rules of the language syntax can be immediately captured with logical programming. The Syntax Analyzer will create a parse tree that can be used by the Semantic Analyzer. The kinds of design decisions that lead to this representation are myriad, and beyond the scope of this study. We will be satisfied to say that some parse tree representation is created by this phase, and made available through an access point to the Semantic Analyzer.

Internal Data: The Syntax Analyzer creates the parse tree internally, but then makes it available through some mechanism to the Semantic Analyzer. The decisions of how to make the parse tree available is dealt with in the access point discussion for Syntactic Tree Passing.

Summary: The wrapper for the Syntax Analyzer is object-oriented, and internal data is encapsulated, but the mechanics of parsing are all done logically.

Operative Unit: Semantic Analyzer

Ins and Outs: The Semantic Analyzer receives as input a Syntactic Parse Tree, already known to be syntactically correct, and generates a Semantic Parse Tree that can be used for code generation.

Internal Behavior: The biggest task of the Semantic Analyzer is to perform static type checking. The Semantic Analyzer may also insert elements into the parse tree that will cause code to be generated to perform dynamic type checking, depending on the runtime environment. The rules for type checking and programmatic behavior can also be captured logically, so we will use logical relations to perform the work of parsing.

Internal Data: Similar to the Syntax Analyzer, the Semantic Analyzer will create a parse tree that can be used by the next phase in the pipeline, in this case the Intermediate Code Generator. The issues therefore are essentially the same.

Summary: The wrapper for the Semantic Analyzer is object-oriented, and internal data is encapsulated, but the mechanics of parsing are done logically.

Operative Unit: Intermediate Code Generator

Ins and Outs: The Intermediate Code Generator receives a Semantic Parse Tree from the Semantic Analyzer, and generates an Intermediate Representation of the program. The Parse Tree comes in the form of a pointer that provides access to the tree. We can view this pointer as a data access point shared between operative units. The Intermediate Representation of the program takes the form of a list of intermediate instructions. This list is delivered to the Optimizer.

Internal Behavior: The Intermediate Code Generator traverses the Semantic Parse Tree and creates a list of instructions in an intermediate form that can be used to generate final source code. Once again, the logical perspective is helpful. The traversal of the parse tree is easily done logically, and the rules defining what to do once certain things are encountered also lend themselves to a relational representation.

Internal Data: Some data structure must be created to contain this new representation of the program. This structure, once created is potentially quite large, and should be not be passed in its entirety.

Summary: The wrapper for the Semantic Analyzer is object-oriented, and internal data is encapsulated, but the mechanics of generating an Intermediate Representation of the program are done logically.

Operative Unit: Optimizer

Ins and Outs: The Optimizer receives an Intermediate Representation from the Intermediate Code Generator and delivers the same basic material to the Code Generator. The only difference is that the output of the Optimizer is an Intermediate Representation that has been optimized.

Internal Behavior: The Optimizer rearranges and modifies Intermediate Representation of code in order to optimize its size, speed, or both. It will typically perform these optimizations based upon some kind of rule set that recognizes instruction patterns and understands how to transform these into more efficient patterns. The pattern recognition portions might lend themselves to logical search for identification. Indeed, the bi-directional nature of relational parameters might lend itself to replacement. Otherwise, there are filter functions that can also identify some of these patterns, and pertain more to the functional perspective.

Internal Data: The internal representation is consistent with the inputs and outputs. There is no special data that has to be created here.

Summary: The Optimizer is organized as an object, but its methods could be of functional, logical, or imperative nature. All of these could provide some service to the task of optimization.

Operative Unit: Code Generator

Ins and Outs: The Code Generator accepts an Optimized Intermediate Representation and produces Target Object Code. Part of this process includes a translation from the input form to the output form. But it also includes the addition of code specific to the target environment (operating system and hardware platform).

Internal Behavior: At the most basic level, the Code Generator converts Intermediate Representation into final Target Object Code. But in the process, it also adds considerable insight into the target environment. It also pulls information from the symbol table and uses it in the ultimate sense to make things work in the target code. The behavior must include memory mapping issues, instruction selection (dependent on the target instruction set), register allocation (dependent on the hardware architecture). It must implement address resolutions of various kinds (static and run-time), stack allocation issues, as well as a variety of issues that pertain to the programming language being used (structures, function calls, etc.).

Internal Data: There is no significant additional data structure needed internally. There is knowledge embodied in the Code Generator, but this is not necessarily present in data like it is for an Error Handler (for example).

Summary: While captured in an object, the work of the Code Generator is very straightforward, following fairly defined steps. This lends itself to imperative programming within its methods.

Operative Unit: Symbol Table Manager

Ins and Outs: The Symbol Table Manager may receive symbols and information from the Lexical Analyzer, the Syntax Analyzer, and the Semantic Analyzer. It may provide information for any of the pipeline phases. Input to the Symbol Table Manager takes the form of messages that are particular to individual Symbol types. Output takes a similar form, with the caller initiating the interaction with a request for information about a particular symbol, and the response taking the form of an appropriate message.

Internal Behavior: At the simplest level, the Symbol Table Manager is maintaining a table or database of useful information. It merely records meaningful information and retrieves it when called upon. These tasks do not beg to be solved in any particular paradigm, and any table look up approach will do the job.

Internal Data: There are several ways to manage the internal data structures used for the Symbol Table, most commonly including lists, and hash tables. Since the Symbol Table Manager is object-oriented, it has methods that manipulate the internal data structure to record and provide information.

Summary: The overall approach to the Symbol Table Manager will be object-oriented, with the data maintained within the object and access to it through the methods, accessible to the outside world via messages.

Operative Unit: Error Handler

Ins and Outs: The Error Handler may receive information from any of the six operative units that constitute the pipeline. Unlike the Symbol Table Manager, the Error Handler does not send information to the operative units. Rather, it receives Error messages as input, and produces error messages to the user as output.

Internal Behavior: The behavior of the Error Handler is relatively simple. It does not have to maintain the information it receives, but produces messages that are accessible to the user either to the console or to an integrated environment. The behavior

of the Error Handler is embodied in methods that act in accordance with the messages received, and produce the correct messages to the user.

Internal Data: The Error Handler must have access to text strings that indicate the kinds of errors that can occur. These strings are static, but must be available through some kind of query, depending on the message.

Summary: The Error Handler is an object that receives error messages and produces appropriate output for the user.

Data: Source Code

The form of Source Code is common and must be supported in its most basic form: ASCII text containing the program to be compiled. Individual constraints may be placed on the Source Code, such as the length of lines, the overall size of the code, etc. The program should (but might not) conform to syntactically correct forms that define the language of the program being compiled. Source Code is typically found in text files but could be the output of some other program, such as a pretty printer or other preprocessor. So the access point to the Lexical Analyzer should not presuppose a file, but rather, an input stream of text.

Summary: Source Code fundamentally resides in its static form in the very imperative ASCII text file. However, it may also may take the dynamic form of output from a preprocessor, and must be deliverable to the Lexical Analyzer.

Data: Tokens

Tokens are produced by a Lexical Analyzer and consumed by a Syntax Analyzer. Tokens are small strings consisting of contiguous legal characters. In the original input stream of Source Code, the tokens were separated by white space or some other physical or logical separation. But as tokens they are (typically) small strings containing individual words, numbers or symbols. The tokens could be viewed as pertaining to a list of strings or they could be viewed as a set of strings. That representation is not as critical as that the tokens be delivered in sequential order to the Syntax Analyzer.

Summary: We won't do anything fancy with tokens. These are small strings created by the Lexical Analyzer and consumed by the Syntax Analyzer. The concept of a

sequential set of strings can be viewed as common between imperative, object-oriented, and functional paradigms.

Data: Syntactic Parse Tree

The particular representation of the Syntactic Parse Tree is beyond the scope of this study, since there are many design decisions that must be made. We will assume for the sake of simplicity that the Syntactic Parse Tree is created with some appropriate representation, probably a dynamic tree structure. Even the concept of a data structure to represent a Parse Tree presupposes a particular approach of objective examination of the Source Code to create this representation. This is consistent with a logical perspective. But we could have chosen a recursive descent parsing approach in which the structure of the imperative function calls mimics the structure of the target programming language syntax. In this approach, the syntactic parsing is inherent in the calling structure of the Syntax Analyzer, and other components, such as the Semantic Analyzer would potentially have to piggy back on the Syntax Analyzer in a more tightly coupled way.

Summary: The Syntactic Parse Tree is a data structure created by the Syntax Analyzer, and is consistent with imperative, object-oriented, and functional perspectives.

Data: Semantic Parse Tree

The particular representation of the Semantic Parse Tree is, like the Syntactic Parse Tree, beyond the scope of this study, since there so many domain-specific decisions to be made. But we can make similar assumptions concerning the nature of this data structure and the way in which it can be created, shared, and manipulated.

Summary: The Semantic Parse Tree is a data structure created by the Semantic Analyzer, and is consistent with imperative, object-oriented, and functional perspectives

Data: Intermediate Representation

The intermediate Representation of the program is a list of program statements. Each program statement can be represented in a number of ways, with the most common form being the three-address statement. This approach consists of code that looks much like assembly with symbolic names created to hold the place of final addresses and offsets. This representation permits further manipulation by the Optimizer. [Justice 1994]

views this data as a set of tuples, and this representation is totally appropriate. We can also view this as a list of statements, each comprised of several elements, including an operand, two arguments, and (potentially) a result. These values may potentially require access into the symbol table, so some connection must be maintained.

Summary: This list of tuples may be represented in an object-oriented way (as in [Justice 1994]), or in either an imperative or functional way, similar to the passing of the parse trees in earlier parts of the pipeline. We will view the Intermediate Representation in a similar way to the parse trees, and will pass a pointer to the data structure.

Data: Optimized Internal Representation

Summary: The data structure issues are identical to Intermediate Representation, since they deal with exactly the same material, merely optimized.

Data: Target Object Code

The data comprising the Target Object Code is simply a sequence of byte values contained in a file (viewed as either an executable file or at least as an object file that can be linked for executability).

Summary: This representation is very straightforward, traditional, and imperative.

Data: Symbols

Symbols consist of names (typically text strings) and information. This information can come in the form of a heterogeneous list, a structure, or any tuple representation. Symbols that are received by the Symbol Table Manager will be stored in a variety of possible forms. But the nature of Symbols will typically not be consistent since the kinds of information represented by Symbols of different types may vary. A reasonable way to deal with the data that represents Symbols is for a variety of messages to be created that pertain to the various types of symbols. This approach could use polymorphism to create a consistent entry point to the Symbol Table Manager, and let its methods deal with the contents of messages to effectively deal with and store the information.

Summary: An object-oriented approach is used with messages being sent that pertain to different Symbol types.

Data: Errors

Interfacing with the Error Handler involves similar kinds of challenges (but some strong dissimilarities as well). A similar approach to the Symbol structure could permit messages pertaining to different error conditions to be sent to the Error Handler.

Summary: An object-oriented approach is used with messages being sent that pertain to different Errors.

Access Point: Program Input (Source Code to Lexical Analyzer)

Although Source Code typically resides as a text file, it cannot be imposed upon the Lexical Analyzer to count on it being in that form. The Program Input access point must deliver a stream of text to the Lexical Analyzer, and it must be able to either read that text from a file or receive it via some other delivery method from some preprocessor. We can explore several approaches to this access point. If our environment supports pipes, Program Input can simply consist of the Lexical Analyzer reading from standard input. It is then incumbent upon the individual executing the compilation to pipe the input program to the lexical analyzer (in the case of separate programs) or into the compiler (in the case of a single program). Another approach to this Program Input is for the name of the text file to be communicated to the compiler (or Lexical Analyzer) via command-line parameter. In this case, the portion of the Program Input on the Lexical Analyzer side consists of code to open the file and read its content. Access to this Source Code may also be dependent on the working environment surrounding the compiler. For example, one might have an integrated environment (such as Microsoft Visual C++) in which you can edit and then compile what you've been editing. Important information may be stored in configuration space. In this situation, the Program Input access point is contained in the integrated environment which must get the correct text and deliver it to the Lexical Analyzer.

Summary: We assume a simple system in which the entire compiler is in one program, and the Source Code is contained in a file. The name of the file must be given on the command line as input to the compiler or be received via pipeline from some other operative unit (such as a preprocessor). In the command-line case, the access point

consisting of Program Input begins when the user types the file name, and includes the operating system's ability to deliver the name to the program, and code that opens the file, reads its contents and delivers a stream of characters to the Lexical Analyzer. In the pipeline case, the access point consisting of Program Input involves the pipeline from the preprocessor, and bypasses the file reading mechanism. This yields an essentially imperative access point.

Access Point: Token Passing (Lexical Analyzer to Syntax Analyzer)

There are two basic approaches to moving tokens between the Lexical Analyzer and the Syntax Analyzer: push and pull. In the push model, the Lexical Analyzer initiates the delivery of tokens. In the pull model, the Syntax Analyzer initiates the delivery. Either approach can be done in several paradigms. If object-oriented, messages can be sent in either direction initiating delivery. If imperative or functional, function calls can be made in either direction to initiate delivery. To be consistent with the pipeline model we have chosen to keep the flow of communication between compiler phases moving from input to output. So the Lexical Analyzer has responsibility to initiate the delivery.

Summary: Tokens will be passed via messages initiated by the Lexical Analyzer and received by the Syntax Analyzer.

Access Point: Syntactic Parse Tree Passing (Syntax Analyzer to Semantic Analyzer)

The Syntactic Parse Tree can conceivably be quite large. For this reason, the most practical approach to sharing this information from the Syntax Analyzer to the Semantic Analyzer is to pass a pointer to the root node of the parse tree. This approach violates to some degree the principles of information hiding. This could be remedied in part by grouping Syntax Analyzer and Semantic Analyzer together so that they each become privy to the same parse tree information. This is the approach taken by [Justice 1994]. Grouping them together also provides the opportunity to more carefully consider a recursive descent approach in which the access point is essentially subsumed in melding together of the two functions.

Summary: For our purposes, we will conceptualize the sharing of this parse tree as passing a pointer to the parse tree via a message from the Syntax Analyzer to the Semantic Analyzer.

Access Point: Semantic Parse Tree Passing (Semantic Analyzer to Intermediate Code Generator)

The issues for sharing the Semantic Parse Tree between operative units are very similar to the issues for the Syntactic Parse Tree.

Summary: In keeping with our quest for simplicity and a forward flow of information down the pipeline, we will conceptualize this sharing as the passing of a pointer to the parse tree via a message from the Semantic Analyzer to the Intermediate Code Generator.

Access Point: Intermediate Representation Passing (Intermediate Code Generator to Optimizer)

The movement of Intermediate Representation from the Intermediate Code Generator to the Optimizer involves access to a list of tuples that represent programmatic statements. These tuples are conceptualized in our design as a list of tuples, where each tuple is delivered to the Optimizer in sequence.

Summary: This implementation is very similar to the passing of tokens, and can be done in an object-oriented fashion.

Access Point: Optimized Intermediate Representation Passing (Optimizer to Code Generator)

Summary: The mechanism for passing Optimized Intermediate Representation of code is identical to Intermediate Representation, since the material is identical, only optimized.

Access Point: Target Object Code Output (Code Generator to Output)

Object Code is ultimately written to a file in a form that can be executed by an operating system. This process is very analogous to the initial reading of the Source Code into the Compiler, with the exception that the output of the compiler is not typically piped

into another program (a post-processor) while obtaining input from a preprocessor is not that uncommon.

Summary: The Object Code is written to a file in a very traditional, straightforward imperative way.

Access Point: Symbol Passing (Pipeline to Symbol Table Manager)

The Symbol Table Manager is an object-oriented operative unit that encapsulates the Symbol Table and all of its functions. It receives information as it becomes available during compilation, and it delivers appropriate information as requested. It fits very neatly into the object-oriented perspective.

Summary: Symbols are passed to and from each of the pipeline phases via object-oriented message passing.

Access Point: Error Passing (Pipeline to Error Handler)

The Error Handler is similar in interaction to the Symbol Table Manager. It potentially receives information from all of the pipeline phases. It manages the creation and maintenance of error messages. As such, it must either spill messages to a standard place (like standard err) or interact with the user through an integrated environment. In either event, the Error Handler maintains its own world, and can easily be communicated to through message passing.

Summary: Error messages are passed to the Error Handler via object-oriented message passing.

13.4.3 Multiparadigm Compiler Design Summary

By applying our multiparadigm design methodology, we were able to create a paradigmatically diverse design for a compiler. This design does not represent the only viable design for a multiparadigm compiler, nor is it the only obvious solution even given the particular design process we followed or the patterns we used. The nature of pattern-based design is that the same pattern set can lead to the production of an infinitely diverse set of instantiations. This flows from the fact that patterns capture the essence of principles and solutions without mandating a particular implementation.

Notice that this design differs in significant ways from the design presented by Justice et al. Their multiparadigm compiler primarily involved object-oriented operative units with methods pertaining to different paradigms. Our design also uses this type of combination. However, their design did not address issues of paradigmatic interchangeability with respect to data or to access points. We can see, therefore how our patterns of *Inherent Commonality* and *Paradigmatic Interchangeability* lend insight into the challenge of multiparadigm design.

Table 13.1 summarizes the design decisions that were made during the design of this compiler, with the paradigms that were used in the design. In addition, this table also includes a summary of the design decisions that were made by [Justice 1994] in their multiparadigm compiler.

Table 13.1. Paradigmatic summary of multiparadigm compiler designs.

Compiler Component	Paradigms Used	
	[Knutson 1998]	[Justice 1994]
Operative Units		
Lexical Analyzer	Object-oriented with Logical	Called Scanner. Object-oriented with Logical.
Syntax Analyzer	Object-oriented with Logical	Combined together into the Parser. Object-oriented with Logical.
Semantic Analyzer	Object-oriented with Logical	
Intermediate Code Generator	Object-oriented with Logical	Method within Parser. Assume Object-oriented.
Optimizer	Object-oriented with Logical, Functional, Imperative	Functional
Code Generator	Object-oriented with Imperative	Method within Tuple. Object-Oriented.
Symbol Table Manager	Object-oriented	Object-oriented
Error Handler	Object-oriented	Not addressed
Data		
Source Code	Imperative	Not addressed, assume Imperative
Tokens	Imperative, Object-oriented, Functional	Object-oriented

Table 13.1, Continued

Compiler Component	Paradigms Used	
	[Knutson 1998]	[Justice 1994]
Syntactic Parse Tree	Imperative, Object-oriented, Functional	Not addressed, subsumed in Parser.
Semantic Parse Tree	Imperative, Object-oriented, Functional	Not addressed, subsumed in Parser.
Intermediate Representation	Imperative, Object-oriented, Functional	Data portion of Tuple. Object-oriented.
Optimized Intermediate Representation	Imperative, Object-oriented, Functional	Data portion of Tuple. Object-oriented.
Target Object Code	Imperative	Not addressed, assume Imperative.
Symbols	Object-oriented	Object-oriented
Errors	Object-oriented	Not addressed
Access Points		
Program Input	Imperative	Object-oriented
Token Passing	Object-oriented	Object-oriented
Syntactic Parse Tree Passing	Object-oriented with Imperative	Not addressed, subsumed in Parser
Semantic Parse Tree Passing	Object-oriented with Imperative	Not addressed, subsumed in Parser
Intermediate Representation Passing	Object-oriented	Not addressed, assume Object-oriented.
Optimized Interm. Representation Passing	Object-oriented	Not addressed, assume Object-oriented.
Target Object Code Output	Imperative	Not addressed, assume Imperative
Symbol Passing	Object-oriented	Object-oriented
Error Passing	Object-oriented	Not addressed

13.5 Case Study: Multiparadigm Database Design

One of the earliest case studies that we pursued in our preliminary investigations on multiparadigm design methodologies involved the design of a relational database [Knutson 1997a]. That research effort predated our research into patterns as a mechanism for capturing knowledge and experience. The result of this early research was a very interesting look at database design from a multiparadigm perspective, but ultimately a

failure in effective design. One of the main reasons is that the level of complexity involved in asking multiple paradigm questions without a guiding design methodology quickly became unwieldy in practice. Further, our research did little to differentiate architectural, design, and idiomatic issues. Perhaps most importantly, at that time we were not equipped with pattern systems to guide our thinking (or to capture recurring thought) in multiparadigm design. The application of these pattern systems to this particular problem is a somewhat satisfying case study, because the contrast of that first effort with our current insights is striking.

The following overview analysis of relational databases is taken from [Knutson 1997a].

In the relational model, a database is a collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values. The columns of a table represent attributes that can be held by items in the database. Each row of a table contains information that relates instances of attributes to one another. Commonly “relation” is used in place of “table,” and “tuple” is used in place of “row.”

A set of relations then, comprises the physical manifestation of a given database. A number of actions can be performed on a database. Relations can be created (given a unique name, and set of attributes, each assigned to a separate column) or deleted. Modifying a relation is done through manipulation of the tuples. Given a relation, tuples can be added, deleted or modified. Modification involves changing one or more attribute values for a given tuple.

Finally, queries can be performed on databases, and can involve one or more relations. One of the most common methods for users to submit queries is through SQL (Structured Query Language). The basic structure of an SQL expression consists of three clauses: `select`, `from`, and `where`. `select` is used to list the attributes desired in the result of a query. `from` lists the relations to be scanned in the execution of the expression. `where` consists of a predicate involving attributes of the relations that appear in the `from` clause.

To summarize, the relational database model consists of the following elements:

- 1) A database, consisting of
 - Relations (or tables), consisting of
 - A unique name
 - A list of attributes (column names)
 - Tuples (or rows), consisting of
 - Attribute data in appropriate columns
- 2) Actions on Relations, including
 - Add a relation
 - Delete a relation
- 3) Actions on Tuples, including
 - Add a tuple
 - Delete a tuple
 - Modify a tuple
- 4) Queries on database, consisting of
 - SQL statements, consisting of
 - `select` clause, consisting of
 - A list of result attributes
 - `from` clause, consisting of
 - A list of relations
 - `where` clause, consisting of
 - A predicate with desired attributes from selected relations

13.5.1 Determine Overall System Architecture

We can characterize a relational database as a large piece (or set of pieces) of data and a lot of operations that you can do with it or to it. When faced with such a broad characterization, there are several patterns that immediately spring to mind. The first pattern is simply the quintessential object-oriented objects, which consist of data and methods encapsulated together. Data is contained in and defined in classes, and methods are created that perform operations on the data. The analysis above lays out a set of data descriptions and the functions that must be performed on them. While it is true that the choice of a particular paradigm at this point is premature, even by our own methodological guidelines, the object pattern is so strong in this database example, that it grabs almost immediate attention. We can essentially view this entire architecture as the database data encapsulated in an object with methods relating to database, relations,

tuples, and queries. From a course-grained view, this is an incredibly simple overall system architecture.

However, most database systems do not stand alone as a database engine (or back-end). They require the ability to access the database from some other part of an overall system. And so database front-ends are created that encapsulate user interface issues—the front-end interacts with the user and then communicates with the back-end or database engine. As we look at architectural patterns, one stands out as a clear match: *Client-Server*. In the *Client-Server* model, one object is a provider of services (the Server) which are available through some access mechanism by one or more service accessors (Clients). The *Client-Server* pattern is a very strong pattern, and occurs abundantly in software including databases, networking, and almost any arena in which distribution is involved. But it has come to be most tightly related to database design. In fact, so strong is the association between database design and the *Client-Server* model, that in many circles the term “client-server” strongly implies a database with front-end and back-end (typically distributed, but not necessarily), despite the fact that the *Client-Server* pattern is quite broad and in no way limited to the database domain.

In summary, our overall architecture for the database is simple. A database object contains data and operative units. These data structures and operative units are described in some detail in the analysis above. There is a user interface front-end that allows user input. There is an access point between the client (front-end) and server (database engine)

This architecture is so simple that we will quickly move through the design decisions involved at this level, and then look more closely at the database engine itself as a separate architectural entity.

13.5.2 Make Specific Design Decisions: Overall System Architecture

At the highest view of the system, there are only five components to be concerned with: Two operative units (the Database Engine, and the Database Front-End), two data components (Structural and Query Information) and one access point (Database Transport). Figure 13.5 shows this high-level view of database architecture

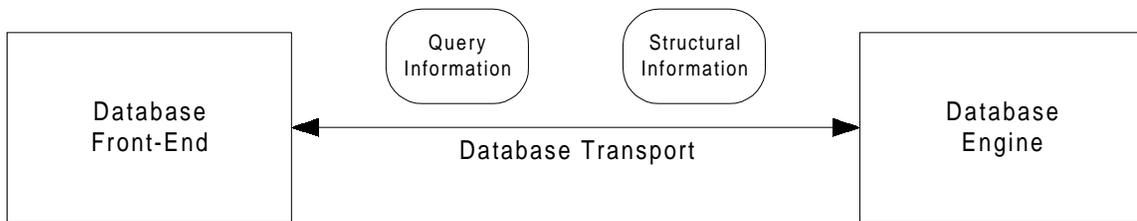


Figure 13.4. A high-level view of a client-server database system.

In the following sections, we will discuss the design issues for each of these components, and the ways in which various paradigms can be combined for effective design.

Operative Unit: Database Engine

Ins and Outs: The Database Engine will give and take two different kinds of information: Structural and Query. Structural Information concerns adding, deleting and modifying information in a database. Query Information involves retrieving the contents of a database according to a query.

Internal Behavior: The Database Engine has two primary responsibilities, loosely connected to the two kinds of information it has to deal with. First, the Database Engine maintains the Database itself, based upon structural information received from the Database Front-End. Second, the Database Engine retrieves query information from the Database based upon requests made by the Front-End.

Internal Data: The Database is a potentially complex configuration of relations and tuples and a great deal of effort can potentially be spent creating and effectively managing the data structures that comprise a database. A more detailed view of this will be dealt with in the next section where the Database Engine is designed.

Summary: At this level, we can simply view the Database Engine as an object with data encapsulated together with methods. We do not have to make design decisions at this point concerning the nature of its data or these methods.

Operative Unit: Database Front-End

In this database design case study, we are less concerned with the Database Front-End than we are with the Database Engine. But it is at this point that the user gains access and has a view of the Database system. Many powerful tools exist to aid in the construction of Database Front-Ends, and these are overwhelmingly object-oriented, since they involve a tremendous amount of graphic user interface construction (which lends itself well to object-oriented construction).

Ins and Outs: The Database Front-End interacts with the user via an appropriate interface. It deals in Structural and Query Information, just like the Database Engine, but in obviously complementary ways. The Database Front-End produces structural information to impact the contents of the database. It also produces queries to send to the Database Engine, and consumes query results to display to the user.

Internal Behavior: The Database Front-End primarily serves as a mechanism by which the user can interact with the Database Engine. It manages the user interface and the flow of Information to and from the Database Engine. The Database Front-End may also be application-specific, depending on its usage. For example, it could be a generic SQL query interface, or it could be domain specific (such as an airline reservation system) with a very specific approach to displaying information.

Internal Data: There is very little internal data associated with the Database Front-End to be concerned with at this point other than those required to trade information with the Database Engine and those required to deal with the user interface.

Summary: The Database Front-End is probably an object-oriented operative unit. We will make the decision here that it is a GUI application that has the ability to bundle communicative information to interact with the Database Engine.

Data: Structural Information

Structural information must be communicated to the Database Engine in such a way that the it can effectively perform its appropriate functions. This means that specific database, relation, and tuple names and fields must be dealt with. The paradigm used for this information is somewhat open. For example, we can define information packets that

either correspond to a particular implementation of a Database Engine (such as the airline reservation scheduler) or that are generic (such as SQL for queries). Sending this information constitutes a message, and so is fundamentally object-oriented. The information can be a packaged data structure corresponding to the internal representation of the Database information, and so can be very imperative. The fact that this information is not directly modifiable (that is, there are no side effects between the Front-End and Back-End in this exchange) is largely a functional concept. We could further identify the Structural Information as a set of logical facts. All of these forms of information could be sent as text to be interpreted by the Database Engine, or in some encapsulated form that pertains to a predefined mechanism of the Database Engine to understand it.

Summary: The overall data concept here is that of an object-oriented message. We will assume Structural Information, similar to SQL, that allows us to embed information strings in messages that can be sent to the Database Engine and be interpreted by it for operations on the database, relations, or tuples. The nature of this information language could easily be either imperative or logical. Our design here leans toward logical, but does not preclude imperative.

Data: Query Information

The basic principles for Structural Information apply equally to Query Information. In this case, we will use SQL, which is a language that is largely logical in its flavor, but with elements of imperative. The logical nature of SQL comes from the fact that the mechanisms for fulfilling the query are not given, but a largely declarative statement of the problem is given to the Database Engine. It has an imperative flavor to the extent that one still has to designate which tables to look in.

Summary: Query Information is basically of the form of logical queries with some imperative flavor carried as the content of object-oriented messages.

Access Point: Database Transport

The Database Transport can be designed in a variety of ways. The simplest way is to visualize a simple network peer-to-peer communication from one point to another with packets of information exchanged by communication mechanisms at either point. This is

somewhat of a puristic object-oriented approach, in that messages are exchanged between sides with few other additional constraints being imposed. On the other hand, one of the most common ways to communicate from a Database Front-End to a Database Engine is through remote procedure calls (RPC) which allow the Front-End to make what appear to be function calls, but for those calls to be translated at a lower level into packets deliverable across a network. We prefer to view the access point between the Database Engine and Database Front-End as moving across some other delivery mechanism that allows either side to bundle messages and have them delivered, without having to mimic an imperative function call protocol. This approach has elsewhere been referred to as an “unconstrained object” perspective [Knutson 1997b].

Summary: The approach taken is best described as object-oriented, since it embodies the essence of unconstrained message passing in either direction.

13.5.3 Multiparadigm Database Overall Design Summary

In this section, we have treated only the high-level view of database interaction, and have been able to create a paradigmatically diverse view of database systems. Indeed, the current traditional “three-tiered” relational database model is, in fact, a multiparadigm approach. Our methodology provides us with more natural ways to describe this multiparadigm design, as well as the tools to explore other elements of potential multiparadigm design approaches.

Table 13.2 summarizes the design decisions that were made during the overall design of this database system, with the paradigms that were used in the design. The overall database design is similar to the design we proposed in our earlier study [Knutson 1997a], although in that study we were ill-equipped to articulate those decisions and to drive much further than this point in our design.

Table 13.2. Paradigmatic summary of multiparadigm database system design.

Database Component	Paradigms Used
Operative Units	
Database Engine	Object-Oriented
Database Front-End	Object-Oriented
Data	
Structural Information	Object-Oriented with Logical or Imperative
Query Information	Object-Oriented with Logical and Imperative
Access Points	
Database Information Transport	Object-Oriented

13.5.4 Determine Database Engine System Architecture

At our current level of design, we are primarily concerned with the Database Engine itself, and the architecture that will best describe various components and their interactions. As we stated during the previous design level, the data structures of the database fall very nicely into an object-oriented style of packaging, since there are pieces composed of other pieces, with each piece having certain operations that can be performed on it. Figure 13.5 shows one possible view our Database Engine.

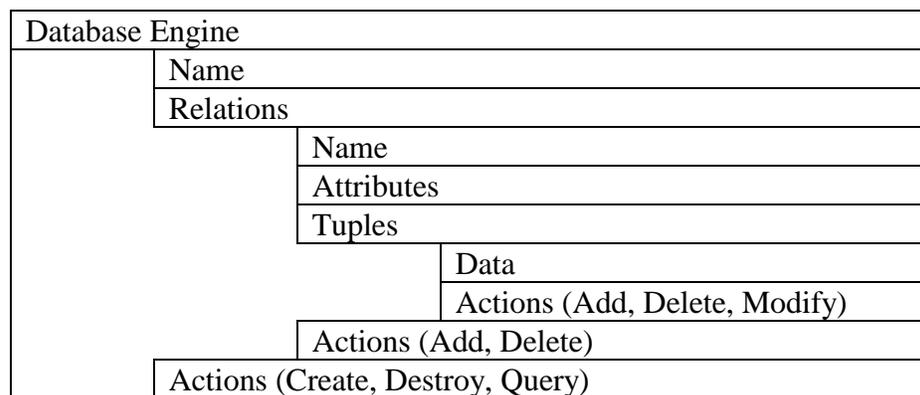


Figure 13.5. One view of a Database Engine.

This is somewhat of an object-oriented view, showing each piece of the database engine as a composite of other pieces, including both data and methods. To get a nice graphic view of this system, we really need to build out from the inside. Figure 13.6 shows one conceptual view of a tuple.

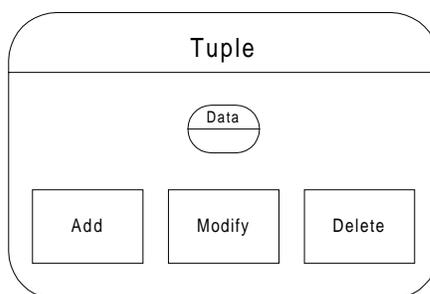


Figure 13.6. A tuple composed of data and actions.

Figure 13.7 shows a similar conceptualization of a relation composed of Names, Attributes, Tuples, and Actions.

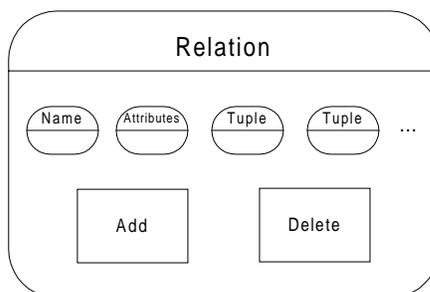


Figure 13.7. A relation composed of tuples.

Figure 13.8 shows a view of a Database composed of Names, Relations, and Actions.

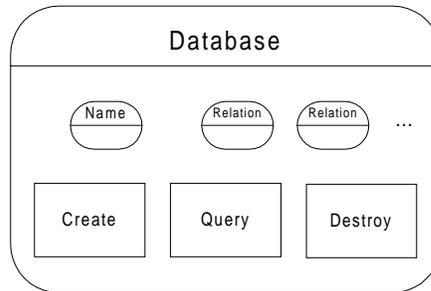


Figure 13.8. A Database composed of relations.

Since the object-oriented form of encapsulation allows us to embed data and methods in the same system, this last drawing of the Database not only defines the structure of the database, but also defines the Database Engine itself, since the methods that do all the work are encapsulated in the Database.

13.5.5 Make Specific Database Engine Design Decisions

The Database Engine falls out into eight operative units (three for Database, two for Relation, and three for Tuple), and six Data units. These are described in the sections which follow.

Operative Unit: Database Create

Ins and Outs: Information must flow into this operative unit containing appropriate data for creating a Database. This information might include a Database name, and it might extend to the identification of particular Relations that will constitute the database.

Internal Behavior: The actual behavior of this operative unit is very simple, and is used to initialize a new Database structure.

Internal Data: The actual Database structure is embedded within the same class as this method.

Summary: The operative unit that creates a Database can be viewed as a constructor for the Database class.

Operative Unit: Database Destroy

Ins and Outs: This operative unit receives the name of the Database to be destroyed.

Internal Behavior: The Database is destroyed as well as all relevant Relations and Tuples.

Internal Data: The internal data depends on the definitions for the Database, Relations and Tuples.

Summary: The operative unit that destroys a Database can be viewed as a destructor for the Database class.

Operative Unit: Database Query

Ins and Outs: This operative unit receives SQL statements and generates output information that corresponds to the desired results.

Internal Behavior: The process of performing a relational query typically involves elaborate steps that include joining tables and then reducing them. This process is most easily done with a logical approach in which the conditions of a successful query are laid out and the logical search engine can be employed to discover the answer.

Internal Data: The data used is the same as the structures defined by the classes.

Summary: While this operative unit exists as an object-oriented method, its behavior is almost entirely logical.

Operative Unit: Relation Add

Ins and Outs: Information concerning the new Relation flows into this operative unit. Information can include the name of the Relation, its fields, and their types.

Internal Behavior: The Relation is created and initialized.

Internal Data: The actual Relation structure is embedded within the same class as this method.

Summary: The operative unit that adds a Relation can be viewed as a constructor for the Relation class.

Operative Unit: Relation Delete

Ins and Outs: Information should come into this operative unit that identifies the Relation to be deleted.

Internal Behavior: The Relation is deleted from the Database.

Internal Data: The internal data depends on the Database that has been created.

Summary: The operative unit that deletes a Relation can be viewed as a destructor for the Relation class.

Operative Unit: Tuple Add

Ins and Outs: Information concerning the new Tuple flows into this operative unit. Information can include the name of the relation in which the new Tuple will be added, and the contents of its fields.

Internal Behavior: The Tuple is created and initialized.

Internal Data: The actual Tuple structure is embedded within the Relation class. This operative unit extends the Relation and adds information.

Summary: The operative unit that adds Tuples can be viewed as a constructor for the Tuple class.

Operative Unit: Tuple Delete

Ins and Outs: Flowing into this operative unit is enough information to uniquely identify the Tuple or Tuples to be deleted from the Relation.

Internal Behavior: The Tuple is removed from the appropriate Relation. The mechanism to search for a match of fields to the desired key fields is probably most easily done through a logical perspective.

Internal Data: The internal data depends on the Relation in which the Tuple lives.

Summary: The operative unit that deletes Tuples can be viewed as a destructor for the Tuple class.

Operative Unit: Tuple Modify

Ins and Outs: This operative unit receives adequate information to uniquely identify a Tuple, and replacement information for particular fields in the Tuple.

Internal Behavior: The appropriate fields are updated depending on the information submitted. The mechanism to search for a match of fields to the desired key fields is probably most easily done through a logical perspective.

Internal Data: Data depends upon the definition of the Tuple.

Summary: This method uses logical programming to identify desired Tuples and then changes appropriate fields.

Data: Database

We could declare a Database in traditional object-oriented fashion (which includes imperative data structures encapsulated in object classes). We could also utilize logical declarations to describe the Database.

Summary: The data associated with a Database is based on object-oriented classes, and is composed of Relations. The initialization of a Database could be done equally using object-oriented or logical constructs.

Data: Relation

We could declare a Relation in traditional object-oriented fashion (which includes imperative data structures encapsulated in object classes). We could also utilize logical declarations to describe the Relation.

Summary: The data associated with a Relation is based on object-oriented classes, and is composed of Tuples. The initialization of a Relation could be done equally using object-oriented or logical constructs.

Data: Tuple

We could declare a Tuple in traditional Object-Oriented fashion (which includes imperative data structures encapsulated in object classes). We could also utilize logical declarations to describe the Tuple.

Summary: The data associated with a Relation is based on object-oriented classes, and is composed of Fields. The initialization of a Tuple could be done equally using object-oriented or logical constructs.

Data: Field

We could declare a Field in traditional Object-Oriented fashion (which includes imperative data structures encapsulated in object classes). We could also utilize logical declarations to describe the Field.

Summary: The data associated with a Relation is based on object-oriented classes, and is composed of that data which fill the fields.

Data: Structural Request

The Structural Request is the message that is passed to the Database Engine from the Database Front-End that tells it which things to Add, Delete, or Modify. This message could come in the form of SQL-like statements, or some other format.

Summary: The Structural Requests could come in either object-oriented message form in a more logical SQL-style form.

Data: Query

The Queries that the Database Engine deals with are a significant part of the Database system. For our system, these will be designed in the form of SQL statements. We view SQL as a logical programming language (see Chapter 8, Logical Programming Patterns, and Chapter 9, Logical Analysis and Design Patterns) and so these instructions, although arriving in the form of object-oriented messages are very logical in nature.

Summary: Queries are logical instructions to be interpreted by the Database Engine.

Access Points

In this object-oriented design, all the access points between various components of the Database Engine are object-oriented messages, including calls into the methods of particular object classes.

13.5.6. Multiparadigm Database Engine Design Summary

In this section, we have explored the design of a database engine, and its composite parts. By applying our multiparadigm design methodology, we were able to

create a paradigmatically diverse design. This design does not represent the only viable design for a multiparadigm database engine. Indeed, the design of databases is a well-studied and explored field. Many of the resultant designs lean toward an object-oriented architecture, as does ours. However, our methodology provides us with more natural ways to describe these designs, as well as the tools to explore other elements of potential multiparadigm design approaches, such as the strength of logical design for key components of the database engine.

Table 13.3 summarizes the decisions that were made during the design of this database system, with the paradigms that were used in the design. These results are not dramatically different from those achieved in [Knutson 1997a], nor is the basic approach used here much different in from the approach most commonly used in [Budd 1995a], that is, using an object-oriented wrapper with methods pertaining to different paradigms. This case study is important for several reasons. The same design methodology that produced such a paradigmatically diverse solution for the compiler example was flexible enough to produce a much simpler solution here, where it was applicable. This design effort explored more significant issues than the previous effort (including data and access point design), and did so more efficiently and with a clearer understanding of the implications of these decisions, and the potential alternative decisions.

Table 13.3. Paradigmatic summary of multiparadigm database engine design.

Database Engine Component	Paradigms Used
Operative Units	
Database Create	Object-Oriented
Database Destroy	Object-Oriented
Database Query	Object-Oriented with Logical
Relation Add	Object-Oriented
Relation Delete	Object-Oriented
Tuple Add	Object-Oriented
Tuple Delete	Object-Oriented with Logical
Tuple Modify	Object-Oriented with Logical
Data	
Database	Object-Oriented or Logical
Relation	Object-Oriented or Logical

Table 13.3, Continued

Database Engine Component	Paradigms Used
Tuple	Object-Oriented or Logical
Field	Object-Oriented or Logical
Structural Request	Object-Oriented or Logical
Query	Logical
Access Points	
All Access Points	Object-Oriented

13.6 Case Study: Multiparadigm Network Protocol Design

This final case study involves another common area of design and software construction: network protocols. Much has been written about network protocols, most prominently the seven layers of the OSI model. For our example, we will look at another protocol stack that is much simpler, the Infrared Data Association (IrDA) stack. This stack is simpler because it only involves point-to-point communication across an infrared connection over a one meter distance. Because of this point-to-point, line of sight model, the IrDA protocol stack can be made much simpler than standard internet or LAN protocol stacks such as TCP/IP or IPX/SPX.

13.6.1 Determine Overall System Architecture

Figure 13.9 shows the basic layers of the IrDA stack. The concept of a “stack” is common to protocols, in which several layers interact, each owning responsibility for performing certain kinds of behavior and then sending information either up or down the stack, depending on the situation. An excellent discussion of this approach is given in [Buschmann 1996] in their pattern *Layers*. Their thumbnail description is as follows: “The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.” They then go on to talk about the OSI model, as well as the TCP/IP model.

IAS	Applications
IrLMP	
IrLAP	
Framer/Driver	
Physical Layer	

Figure 13.9. The IrDA Protocol Stack

It has become somewhat common to view communication protocols from the perspective of this *Layers* pattern, so much so that the term used for protocol systems is “stack”. We have no argument with this approach. The flexibility provided by this approach permits a tremendous amount of flexibility. For example, because of this separation of abstraction layers, the Framer/Driver at the bottom of the IrDA stack isolates hardware issues from the rest of the stack, allowing flexible portability of upper layers. By using the *Layers* pattern, we can implement an IrDA stack on an embedded operating system, and then port it with relative ease to different hardware platforms by changing parts of the Framer/Driver to adapt it to particular CPU and infrared transceiver hardware.

Establishing a layered architecture for protocol stacks does not constrain us to any particular paradigm. If we had to characterize this architecture, we could view it functionally, since the stack somewhat resembles a pipeline (like the compiler case study). But whereas the compiler pipeline processes information without permanently altering it, the protocol stack tends to change information as it processes it. The protocol stack also typically is bound by real-time constraints, and so some information hiding is typically sacrificed for speed and efficiency. This is different from most other architectures that we deal with, and is markedly different from the two architectures that we have dealt with in the previous case studies.

The stack approach to protocols is commonly burdened by a very strong imperative bias, since packets move up and down stacks and are altered in the process. However, if we draw the picture in another way, it lends itself to a more functional perspective. Figure 13.10 shows the same IrDA stack as encapsulated layers exposing a view to layers above, but hiding those layers below.

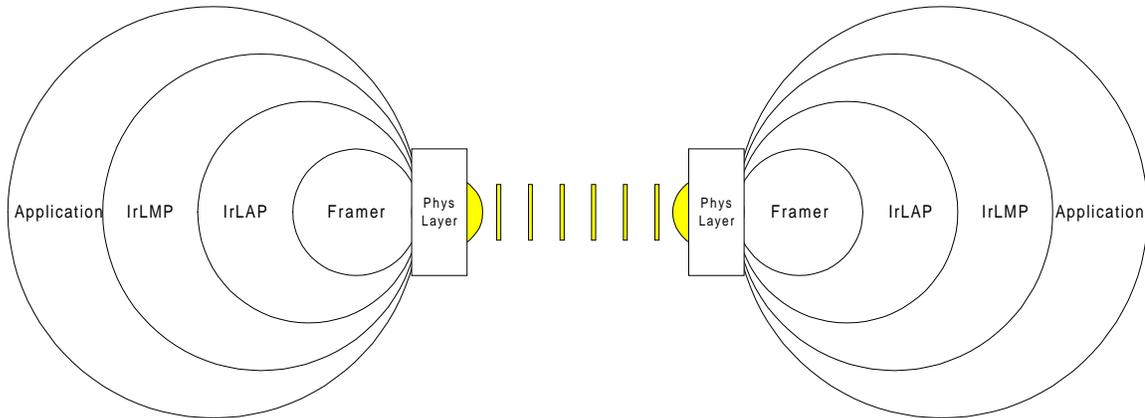


Figure 13.10. Two IrDA applications communicating.

In this figure, an Application's view of the interaction with the other Application is limited to the view given by IrLMP. In a sense, this presents a functional encapsulation of all layers beneath. Similarly, IrLMP has a view of the interaction with the other IrLMP layer that is presented by IrLAP. Each of these nested layers can be designed under separate paradigmatic influence (for example, the Application can live with a high-level view, while the Framer must know about the hardware, including the infrared transceiver).

It can reasonably be argued that the two views of an IrDA stack presented in Figures 13.9 and 13.10 are identical, and that neither pushes a particular paradigm. In a sense this is true. However, the two views may lend different understandings to different viewers. Certainly the concentric circles in the second figure suggest a strong encapsulation, reminiscent of functional composition. We could draw the IrDA stack in yet another way, with the layers separated and pipes between, suggesting a pipes and

filters approach to design. All of these are valid and suggest at least slight variations in architectural approaches. For our design, we will focus primarily on the first layered representation, since this is the figure most commonly used in the IrDA community.

13.6.2 Make Specific Design Decisions

The IrDA protocol stack, like other protocol stacks, is already well understood and, at a coarse level, its organization is well-known. The required pieces of the stack are identified in Figure 13.9 and consist of four operative units (the physical layer is included in the figure for clarity, but is not one of the protocol stack layers). There is an interface between each of the pieces that join along a horizontal border. This interface consists of two pieces: a down call, and an up call. These two calls can be viewed differently since they effect different paths of execution through the stack. On the other hand, it's not unacceptable to look at them each as a bi-directional pipe. For this design, we will look at up calls and down calls as similar elements, and lump all up calls together, as well as all down calls. We will only differentiate these two for the hardware, where the nature of these calls is considerably different. We will also define a user API that applications can use to access the stack.

The stack consists of three fundamental layers (Framer, IrLAP, IrLMP) and one optional service layer (IAS). Like the TCP/IP model, there are other higher-level protocols, but they are not essential to the nature of the basic stack, and so are not included in this case study. Each of these layers is described in the appropriate operative unit descriptions in the sections that follow.

Operative Unit: IAS (Information Access Service)

Ins and Outs: Inputs to IAS consist of requests to register some application with the service so it is accessible to those querying across the link. Outputs from IAS occur when a device makes an IAS query, looking for an application of a particular type.

Internal Behavior: IAS maintains a very simple list of devices that have registered. When an application registers, IAS records it. When it receives a query for some application or device, it looks for entries of the appropriate type.

Internal Data: The table maintained by IAS is very small and not complex at all. It can be viewed as any appropriate list structure, but in practice a small static table is sufficient and takes less overhead to maintain.

Summary: The function of IAS is very naturally object-oriented. It maintains a data structure, and controls access to that structure.

Operative Unit: IrLMP (Infrared Link Management Protocol)

Ins and Outs: Access is made through a user API, sending data to be sent with information concerning whom to send it to. Information comes back through IrLMP and is made available to the user application.

Internal Behavior: IrLMP manages multiple virtual connections over a single IrLAP connection. It includes flow control capability, and bundles data to be sent across the link.

Internal Data: IrLMP relies heavily on the data that is either sent from the user application or that comes up from lower levels.

Summary: It's tempting to want to view IrLMP as a functional pipeline element, like our compiler example. But in the end, the speed and memory requirements that are typically placed upon IrDA stacks force us to consider allowing IrLMP to reuse data structures that are passed from the user, or to declare stack structures into which data is copied, allowing the calling application to release its data. Our design of IrLMP is therefore strongly imperative.

Operative Unit: IrLAP (Infrared Link Access Protocol)

Ins and Outs: IrLAP receives IrLMP packets from above, and IrLAP packets from below.

Internal Behavior: IrLAP manages the physical connection via infrared light. Its behavior is similar to IrLMP in that it receives packets from above and adds header information before sending it downward. When it receives information from below, it strips off header information and sends it upward.

Internal Data: It generally reuses the structures that are passed to it from above or below.

Summary: Like IrLMP, IrLAP faces stiff memory and speed constraints. To alleviate these, data structures are typically reused up and down a protocol stack, which means there are inherent side effects. It also means that data encapsulation is not fundamentally achieved. This design is therefore imperative.

Operative Unit: Framer

Ins and Outs: The Framer receives IrLAP packets from above to be sent out. It sends these across the infrared communication link using the hardware capabilities of the target platform. It also receives IrDA packets from the link and makes these available to IrLAP.

Internal Behavior: The Framer serves two basic purposes. First, it packages (or “frames”) data sent to it from IrLAP in such a way that the information can be understood across the physical link and correctly received. This includes placing additional header information at the beginning of packets as well as CRC (cyclic redundancy check) data at the end. Second, the Framer is typically the device driver for the infrared transceiver on the target hardware. This means the Framer is very hardware and operating system specific. This leads to a large degree of exposure (and access) to the internals of the system.

Internal Data: The Framer has to maintain a buffer pool for receiving packets. Any list of structures is appropriate, with a circular linked list being most common.

Summary: Framers are typically device driver level software components with intimate knowledge of the target operating system and hardware. This lends itself to an imperative approach.

Data: User Data

The user passes a raw bunch of data to the stack with destination information attached so the stack can deliver it. It also receives information that has been received by the stack and made available.

Summary: Given the speed and memory constraints typically imposed upon these kinds of systems, our view of User Data will involve the passing a pointer to a data structure held by the user application. This approach is very imperative. It is conceivable

to imagine the entire stack as an object to which data can be given, so that the delivery mechanism can deliver the message to another application on the other end of the link. This object-oriented approach to message passing could make this interface cleaner without excessively compromising the speed and memory constraints.

Data: IrDA Packet

Framers send and receive IrDA Packets, which are streams of data constituting a valid packet of information for an IrDA stack.

Summary: The IrDA Packet can be viewed at one level as the embodiment of a data structure in a packet form to be transmitted across an infrared link. At another level, this is the essence of a sophisticated object-oriented message passed between two interacting objects.

Access Point: User API

The user must have some access point to the infrared stack to utilize its services. We are under essentially no constraints as to how to conceptualize this interface. The imperative approach is simple, with a function call. Similarly, if we view an object class as being the keeper of the stack, then we can access its methods to deliver our goods. More difficult is receiving data, since this happens asynchronously. This is dealt with in the Interrupt Service Routine.

Summary: The user API could be done in any paradigm, with object-oriented and imperative being most obvious, depending on the overall programmatic environment for which the API is being built.

Access Point: Up Calls

The Up Calls are the mechanism by which a lower layer stack component delivers information to a higher level. The Up Calls happen asynchronously, so the lower component must know about the message or entry point to alert the higher component to the arrival of data. This alert could involve a direct function call to the higher component. It could also involve making the data available in some common location and alerting the higher component via a signal, semaphore, or some other mechanism.

Summary: Given speed and space constraints, this is probably best done imperatively, but it could also be done object-oriented, with the Up Calls consisting of messages delivered to higher components.

Access Point: Down Calls

The Down Calls are analogous to the Up Calls, with the direction reversed.

Summary: The same reasoning exists, leading us to lean toward imperative, but suggesting a look at object-oriented.

Access Point: Hardware Access

The Framer must directly access the hardware in order to function as the device driver for the infrared transceiver. This access involves direct manipulation of memory mapped registers and other low-level features of the target hardware.

Summary: Accessing the hardware is naturally low-level, hardware-dependent, laden with side effects, and therefore imperative.

Access Point: Interrupt Service Routine (ISR)

This is analogous to the Up Call component for the Framer. The nature of infrared transceiver hardware is that when packets are received, an interrupt is triggered in the hardware. The only thing the hardware can do is call the interrupt service routine associated with the particular interrupt. The focus of the ISR is to move the information as quickly as possible, alert the appropriate module that data has arrived, and go away. Because it runs in interrupt mode, it can't be allowed to take more than a minimal amount of time.

Summary: The Interrupt Service Routine is an approach largely dictated by existing hardware, and it is by its nature very low-level and imperative.

13.6.3 Multiparadigm Network Protocol Design Summary

This multiparadigm design of an IrDA protocol stack is interesting for several reasons. While it utilizes an overall architecture that can be viewed as either object-oriented, functional, or imperative, specific design decisions lead us to consider all four

paradigms, but to ultimately choose imperative in many situations. Still, we are aided, when appropriate, to see how other paradigms can influence our design.

Table 13.4 summarizes the design decisions that were made during this design of the IrDA protocol stack, with the paradigms that were used in the design. Notice the strong imperative influence throughout. This is significant for two reasons. First, our multiparadigm design methodology was shown to be adequate for constructing somewhat traditional, imperative-flavored software. Second, despite the strong pull toward an imperative solution, we were still able to easily weave in elements from other paradigms where applicable, without compromising other aspects of the design.

Table 13.4 Paradigmatic summary of multiparadigm design of network protocol.

IrDA Protocol Stack Component	Paradigms Used
Operative Units	
IAS	Object-Oriented
IrLMP	Imperative
IrLAP	Imperative
Framer	Imperative
Data	
User Data	Object-Oriented with Imperative
IrDA Packet	Object-Oriented with Imperative
Access Points	
User API	Object-Oriented or Imperative
Up Calls	Imperative or Object-Oriented
Down Calls	Imperative or Object-Oriented
Hardware Access	Imperative
Interrupt Service Routine	Imperative

14.0 Conclusions

14.1 Summary

In this research, we have captured, in pattern form, key elements of programming and design in four programming paradigms (imperative, object-oriented, functional and logical) as well as multiparadigm programming. These pattern sets have formed a foundation upon which we were able to build a deeper understanding of multiparadigm programming and design.

For each paradigm, we identified sets of programming patterns. In addition, we identified design patterns for those paradigms that already have design methodologies (imperative and object-oriented). For those that do not (functional and logical), we created design pattern sets that may yet play a seminal role in formal design methodologies for those paradigms. From the relationships between programming and design patterns, we were able to identify and record methodological patterns that provide generative mappings between programming patterns and design patterns.

From the sets of programming patterns, we were able to derive a pattern set for multiparadigm programming. We were also able to perform a critical analysis of the multiparadigm programming language Leda using this pattern set. Finally, we were able to apply the methodological patterns to this multiparadigm programming pattern set to aid in our search for multiparadigm design patterns. We were also able to derive insight into multiparadigm design patterns by studying the pattern sets for each of the four paradigms studied.

Armed with this rich pattern system, we then created and presented a new pattern-based methodology for multiparadigm design. Finally, we applied our methodology and our pattern sets to three common design problems to show the viability of our new pattern-based multiparadigm design approach. We found that this new methodology lent new insights into software design, and suggested the strong role that multiparadigm programming and design can play in many aspects of software creation.

14.2 Contributions

This research has contributed to the field of Computer Science in the following ways:

- *Survey and analysis of language and paradigm elements.* A set of taxonomies of this kind provides a strong foundation on which to build discussions of multiparadigm programming. It also represents a capturing of programming language and paradigmatic elements in pattern form for the first time.
- *Survey and analysis of design methodologies.* Some work has been done in this area, particularly in imperative and object-oriented design, but little design work has been done in functional and logical paradigms. In addition, this effort represents a capturing of design elements in pattern form in a way that has not been done before. This establishes a strong foundation on which to build discussions of multiparadigm design.
- *Analysis of multiparadigm programming languages.* This type of analysis hasn't been done before, and helps lay the foundation for further study of multiparadigm programming languages. It also contributes in a significant way to potential advances in the design of multiparadigm languages, such as Leda.
- *Functional and logical design methodologies.* Functional and logical design methodologies are generally either weak or nonexistent. To provide foundations for an effective discussion of design patterns, some form of functional and logical design methodology had to be created. Creating these design methodologies are clearly on the fringe of the scope for this research effort. However, to create a synthesized design methodology requires some building blocks within each of the paradigms. The functional and logical design methodologies presented here are minimal, but point to further research.
- *Exploring patterns of relationship between programming and design.* Such a pattern set has been shown to be useful as a tool in creating a set of multiparadigm design patterns; it also captured principles that can be used for the creation of design patterns in general.

- *Exploring multiparadigm programming and design patterns.* There has been no other research done on multiparadigm patterns prior to this effort. The results presented here have the potential to spark the new field of multiparadigm design patterns as well as contributing to the analysis of existing programming and design paradigms.
- *Creation of multiparadigm design patterns.* This set of patterns has become a working example of the viability of multiparadigm patterns, and a tool set for practical design examples. It forms the nucleus of a demonstrable multiparadigm design methodology based on patterns.
- *A pattern-based multiparadigm design methodology.* The methodology presented in this research captures essential principles that describe the effective manner in which to utilize multiparadigm design patterns. It is the first formal design methodology for multiparadigm programming, and one of the first pattern-based design methodologies of any kind.
- *Case studies of real-world problems.* This is an important deliverable, since it demonstrates the use of the methodology. These case studies serve as examples of the viability of multiparadigm programming and design.

14.3 Conclusion

Our initial motivation to explore multiparadigm programming and design grew initially out of an interest in program comprehension and programmer/designer cognition. At the conclusion of this research, our interest is even deeper, because we have seen the interactions of language and thought on design. We have seen old problems appear new and fresh, with potentially new design solutions suddenly available where before solutions seemed locked within a specific, narrow mindset dictated by a particular paradigm.

The fortuitous rise of the design patterns movement shortly before the commencement of this research effort provided an excellent vehicle to capture the concepts and principles that were discovered. We have experienced the value of patterns as a literary form, but also as building blocks with which to see deeper relationships between language and design.

Almost all programmers with significant experience in the industry know what it is like to be competent for years, and then suddenly to become “paradigmatically challenged” by the sweeping advent of something new that they didn’t teach you in college ([Ross 1997] explores the specific challenge of migrating from structured programming to object-oriented). It seems like each new generation of programmers comes equipped with the tools to eclipse the status quo, who in turn, must adapt or become obsolete. We believe that multiparadigm programming is a wave of the future, whether this results from the rise of pure multiparadigm languages, or through the steady adoption of multiparadigm programming elements in general purpose programming languages. This set of programming and design patterns, together with this pattern-based multiparadigm design methodology should contribute to that on-going evolution.

14.4 Future Research

Our research effort ultimately raises as many questions as it answers. In a field as unexplored as multiparadigm design, there are far more areas discovered but left unexplored than we have managed to examine. Many of these research areas deserve additional discussion and research. The sections below suggest future research in these areas.

14.4.1 Programmer Cognition

There is considerable on-going research in the domain of program comprehension and programmer cognition. Of particular interest to this research is the exploration of cognitive models by which programmers build an understanding of software and systems. The pattern approach may perhaps lend some interesting applications to these problems, since patterns intend to capture essential elements of solutions to recurring problems. In particular, some of our early research into programmer cognition identified recurring recognizable patterns that programmers use to build mental models of software systems during comprehension tasks. The relationship of such concepts as “beacons” [Brooks 1983], or “rules of discourse” [Soloway 1984] to “patterns” is somewhat obvious, and is worthy of further exploration.

14.4.2 Patterns

The software patterns movement, while still relatively young, and growing rapidly, has displayed considerable provincialism; the patterns movement has followed a very narrow self-definition. The movement is almost exclusively viewed and practiced as a component of object-oriented design, despite explicit statements from some that patterns are not paradigm-specific [Vlissides 1997]. It would seem that the patterns movement must at some point be permitted to branch freely beyond object-oriented programming. This research may contribute in some way to that effort.

The patterns movement has also used patterns almost exclusively as a mechanism for capturing recurring problems (the rule of three stating that it must be observed in nature three times to be deemed a pattern). This is not necessarily bad, but is not the entire scope within which patterns lend value to software. For example, the software patterns movement informally defines patterns as solutions to recurring problems within certain contexts in which certain forces act. That's only part of the value that Alexander espoused. For Alexander, patterns didn't have to appear in nature several times to be real. He saw the value of patterns as forms for capturing creative efforts, including patterns for phenomena that had not yet been observed even once, but which have been only conceptualized. He identified specific architectural patterns that were first created in the minds of architects, then implemented, and only then began to recur [Alexander 1979]. That is part of the power of patterns, not just the capturing (or mining) of existing knowledge, but as a tool to extend the existing base of knowledge. The potential of patterns in this sense has been virtually ignored by the object-oriented software patterns movement.

Finally, patterns as a basis for design has been (ironically) largely ignored by the patterns community. In some prominent circles in the patterns community, the main activity is the mining of patterns, or the cataloguing of existing patterns, discovered through research of completed software projects. The application of specific design methodologies to patterns (or the application of pattern-specific design methodologies) has been largely ignored, although occasionally mentioned. One of the notable exceptions is the excellent (but brief) discussion of pattern-based design by [Buschmann 1996] cited

earlier in this research. The concept of capturing design principles or methodologies in pattern form has been completely ignored prior to this research. This is ironic, considering Alexander's perspective on the effectiveness of patterns in capturing relationships. According to Alexander, patterns capture nothing concrete, but capture relationships between things, each of which can be described or represented by lower-level patterns [Alexander 1979]. This concept is at the heart of design methodologies—dealing with the relationships between things. So it appears that the software patterns community has locked onto patterns as the way to capture the concreteness of things, rather than the relationships between things.

14.4.3 Patterns as Pedagogy

Some research has begun to explore the issues surrounding the use of patterns for instructing Computer Science undergraduates in various arenas. Certainly one of the most fundamental areas of learning for Computer Science students lies in programming languages. The programming patterns presented in this dissertation could provide an excellent foundation for future study into the effectiveness of such patterns in presenting the essential attributes of programming perspectives to students largely unfamiliar with the concepts. This research could be coupled with pedagogical research into multiparadigm programming languages as tools for teaching a broader view of programming to students.

14.4.4 Multiparadigm Programming

This research was not fundamentally about multiparadigm programming, although the overlap and influence is obvious. Based upon the programming patterns previously created, we created a set of multiparadigm programming patterns. These patterns captured a variety of principles that can be used to describe the various ways in which elements of different paradigms can be combined. We also compared the multiparadigm programming language Leda against this taxonomy in a brief analysis. A more complete analysis of Leda could be very valuable, particularly at an idiomatic level, and could lead to yet additional programmatic elements being added to Leda to make it a more robust

and complete multiparadigm programming language. In addition, Leda has the potential to be used to create other multiparadigm combinations. The implications of its capabilities could also be explored within this context.

Our study of hybrid and adoptive combinations of programmatic elements limited its scope to the combination of no more than two elements at a time. Further research should explore the implications of combining three or more programmatic elements to produce new multiparadigm programming constructs. This combination could either be achieved by examining three or more elements from separate paradigms at the same time, or by combining multiparadigm elements (each of which already brings characteristics of the source paradigms).

The study of hybrid combinations of programmatic elements yielded three tables (Tables 11.4, 11.5, and 11.6). These tables suggest possible ways in which elements can be tightly combined into multiparadigm data declarations, multiparadigm operative units, and multiparadigm access points. Further study should be done to explore these possible combinations at an idiomatic level, yielding particular programming elements to support these hybrid combinations.

14.4.5 Software Design

In commercially viable areas of programming, design methodologies tend to spring up like weeds. These have varying degrees of effectiveness, but all attempt to capture the analysis and design principles that allow programmers to analyze problems, and then break them down in such a way that they can be effectively coded in the target language (or paradigm). We have created one approach or methodology to multiparadigm design, but we don't believe that others don't (or shouldn't) exist. We feel that this is a first, preliminary attempt at such a methodology, and that if multiparadigm programming ever becomes significantly more popular, methodologies will spring up to aid in the design process.

Similar challenges exist in two of the programming paradigms examined in this research. There really are no codified methodologies for functional and logical design. It is possible to find authors who lay out design principles, or whose examples can be used

to extricate such principles. But you won't find *Logical Design: The Booch Method*, or *Feltis Functional Design with ML* on shelves at the local book store. There are significant and understandable market reasons for the lack of these titles. But it nonetheless leaves the functional and logical paradigms inequitably deprived of effective design methodologies. We have tried to capture or suggest some elements of design in the logical and functional design pattern sets, but we do not believe that these pattern sets are complete, despite their sufficiency for our current cause.

References

- [Agarwal 1995] Rakesh Agarwal and Patricia Lago. "PATHOS: A Paradigmatic Approach to High-Level Object-Oriented Software Development." *ACM SIGSOFT Software Engineering Notes*, 20(2):36-41, April 1995.
- [Aho 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
- [Alagic 1978] Suad Alagic and Michael A. Arbib. *The Design of Well-Structured and Correct Programs*. New York: Springer-Verlag, 1978.
- [Alexander 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.
- [Alexander 1979] Christopher Alexander. *A Timeless Way of Building*. New York: Oxford University Press, 1979.
- [Ambler 1992] Allen L. Ambler, Margaret M. Burnett, and Betsy A. Zimmerman. "Operational Versus Definitional: A Perspective on Programming Paradigms." *IEEE Computer*, 25(9):28-43, September 1992.
- [ANSI 1966] ANSI. *ANSI Standard FORTRAN*. New York: American National Standards Institute, 1966.
- [ANSI 1974] ANSI. *American National Standard Programming Language COBOL*. New York: American National Standards Institute, ANSI X3.23-1974, 1974.
- [Appleby 1991] Doris Appleby. *Programming Languages: Paradigm and Practice*. New York: McGraw-Hill, 1991.
- [Arnold 1996] Ken Arnold and James Gosling. *The Java Programming Language*. Reading, MA: Addison-Wesley, 1996.
- [Backus 1978] John Backus. "Can Programming Be Liberated from the von Neumann style? A Functional Style and Its Algebra of Programs." *Communications of the ACM*, 21(8):613-641, August 1978.
- [Bailey 1989] T.E. Bailey, and Kris Lundgaard. *Program Design with Pseudocode, 3rd Edition*. Pacific Grove, CA: Brooks/Cole, 1989.
- [Bailey 1990] Roger Bailey. *Functional Programming with Hope*. New York: Ellis Horwood, 1990.

- [Beck 1994a] Kent Beck. "Patterns and Software Development." *Dr. Dobbs Journal*, vol. 19, issue 2, February 1994.
- [Beck 1994b] Kent Beck and Ralph Johnson. "Patterns Generate Architectures." In *Proceedings of ECOOP '94*, p.139-149, Bologna, Italy, July 1994, Springer-Verlag.
- [Bergantz 1991] D. Bergantz and J. Hassell. "Information relationships in Prolog Programs: How do Programmers Comprehend Functionality?" *International Journal of Man-Machine Studies*, 35:313-328, 1991.
- [Bird 1988] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [Birrell 1985] N.D. Birrell, and M.A. Ould. *A Practical Handbook for Software Development*. New York: Cambridge University Press, 1985.
- [Bishop 1986] Judy Bishop. *Data Abstraction in Programming Languages*. Reading, MA: Addison-Wesley, 1986.
- [Bohm 1966] Corrado Bohm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM*, 9(5):366-371, May 1966.
- [Booch 1994] Grady Booch. *Object-Oriented Analysis and Design, 2nd Edition*. Redwood City, CA: Benjamin Cummings, 1994.
- [Brilliant 1996] Susan S. Brilliant and Timothy R. Wiseman. "The First Programming Paradigm and Language Dilemma." *Proceedings of ACM SIGSCE 27th Technical Symposium*, Philadelphia, Pennsylvania, February 1996.
- [Brogi 1994] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. "Modular Logic Programming." *ACM Transactions on Programming Languages and Systems*, 16(4):1361-1398, July 1994.
- [Brooks 1983] Ruven Brooks. "Towards a Theory of Comprehension of Computer Programs." *International Journal of Man-Machine Studies*, 18(543-554), 1983.
- [Budd 1987] Timothy A. Budd. *A Little Smalltalk*. Reading, MA: Addison-Wesley, 1987.
- [Budd 1991] Timothy A. Budd. "Blending Imperative and Relational Programming." *IEEE Software*, 8(1):58-65, August 1991.
- [Budd 1995a] Timothy A. Budd. *Multiparadigm Programming in Leda*. Reading, MA: Addison-Wesley, 1995.

- [Budd 1995b] Timothy A. Budd, Timothy P. Justice, Rajeev K. Pandey. "General-Purpose Multiparadigm Programming Languages: An Enabling Technology for Constructing Complex Systems." *Proceedings of the first IEEE International Conference on Engineering of Complex Computer Systems* (Ft. Lauderdale, Florida, November 6-10), 1995.
- [Budd 1997] Timothy A. Budd. *An Introduction to Object-Oriented Programming, Second Edition*. Reading, MA: Addison-Wesley, 1997.
- [Budgen 1994] David Budgen. *Software Design*. Reading, MA: Addison-Wesley, 1994.
- [Buschmann 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley, 1996.
- [Callegarin 1996] Giuseppe Callegarin and Carlo Salvagno. "L3P: A Language for 3 Programming Paradigms." Unpublished information available from callegar@unive.it.
- [Catarci 1996] T. Catarci, M.F. Cotabile, A. Massari, L. Saladini, G. Santucci. "A Multiparadigmatic Environment for Interacting with Databases." *SIGCHI Bulletin*, 28(3):89-96, July 1996.
- [Celko 1995] Joe Celko. *Instant SQL Programming*. Birmingham, England: Wrox Press Ltd., 1995.
- [Chamberlain 1976] D.D. Chamberlain, M.M. Astrahan, K.P. Eswaran, P.P. Griffiths, R.A. Lorie, J.W. Mehl, P. Reisner, and B.W. Wade. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control." *IBM Journal of Research and Development*, 20:6(560-575), November, 1976.
- [Clocksin 1981] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Berlin: Springer-Verlag, 1981.
- [Coad 1991] Peter Coad and Ed Yourdon. *OOA: Object-Oriented Analysis, 2nd Edition*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- [Coad 1992a] Peter Coad. "Object-Oriented Patterns." *Communications of the ACM*, 35(9):152-159, September 1992.
- [Coad 1992b] Peter Coad. "Workshop Report—Patterns." *Addendum to the Proceedings of OOPSLA '92*, p. 93-95, Vancouver, British Columbia, Canada, October 5-10, 1992.

- [Coad 1995] Peter Coad, David North, and Mark Mayfield. *Object Models: Strategies, Patterns, and Applications*. Englewood Cliffs, NJ: Yourdon Press, 1995.
- [Cohen 1985] Jacques Cohen. "Describing Prolog by its Interpretation and Compilation." *Communications of the ACM*, 28(12):1311-1324, December 1985.
- [Cohen 1990] Jacques Cohen. "Constraint Logic Programming Languages." *Communications of the ACM*, 33(7):52-68, July 1990.
- [Colmerauer 1993] Alain Colmerauer and Phillippe Roussel. "The Birth of Prolog." The Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II), *ACM SIGPLAN Notices*, 28(3):37-52, March 1993.
- [Constantine 1979] Larry L. Constantine and Ed Yourdon. *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
- [Coplien 1995a] James O. Coplien. "A Development Process Generative pattern Language," In J.O. Coplien and D.C. Schmidt, Editors. *Pattern Languages of Programs*. Reading, MA: Addison-Wesley, 1995.
- [Coplien 1995b] James O. Coplien and Douglas C. Schmidt, Editors. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.
- [Coplien 1996a] James O. Coplien. "History of Patterns." Unpublished web page manuscript, <http://c2.com/cgi-bin/wiki?HistoryOfPatterns>, March 1996.
- [Coplien 1996b] James O. Coplien. Pattern Mailing List Reflector, V96 #35, April 1996, cited in [Buschmann 1996].
- [Coplien 1997] James O. Coplien. "Idioms and Patterns as Architectural Literature." *IEEE Software*, 14(1):36-42, January 1997.
- [Cox 1986] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach*. Reading, MA: Addison-Wesley, 1986.
- [Curtis 1995] Bill Curtis. "Objects of Our Desire: Empirical Research on Object-Oriented Development." *Human-Computer Interaction*, 10:337-344, 1995.
- [Dale 1994] Nell Dale, Chip Weems, and John McCormick. *Programming and Problem Solving with Ada*. Lexington, MA: D.C. Heath and Company, 1994.
- [Davie 1992] Antony J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. New York: Cambridge University Press, 1992.

- [Delrieux 1991] Claudio Delrieux, Pablo Azero, and Fernando Tohme. "Towards Integrating Imperative and Logic Programming Paradigms: A WYSIWYG approach to PROLOG Programming." *SIGPLAN Notices*, 26(3):35-44, March 1991.
- [Dijkstra 1968] Edsger Dijkstra. "Go To Statement Considered Harmful." *Communications of the ACM*, 11(3):147-148, 538, 541, March 1968.
- [Eckel 1995] Bruce Eckel. *Thinking in C++*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
- [Embley 1992] David W. Embley, Barry D. Kurtz, and Scott N. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, NJ: Yourdon Press, 1992.
- [Embley 1996] David W. Embley, Stephen W. Liddle, and Yiu-Kai Ng. "On Harmonically Combining Active, Object-Oriented, and Deductive Databases." In *Proceedings of the Fourth Annual Workshop on Advances in Database Systems* (Moscow, Russia, September 10-13, 1996), pp. 21-30.
- [Fisher 1995] Gerhard Fischer, David Redmiles, Lloyd Williams, Gretchen I. Puhr, Atsushi Aoki, and Kumiyo Nakakoji. "Beyond Object-Oriented Technology: Where Current Approaches Fall Short." *Human-Computer Interaction*, 10:79-119, 1995.
- [Flanagan 1996] David Flanagan. *Java in a Nutshell*. Sebastopol, CA: O'Reilly & Associates, Inc., 1996.
- [Gabriel 1991] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. "CLOS: Integrating Object-Oriented and Functional Programming." *Communications of the ACM*, 34(9):28-38, September 1991.
- [Gabriel 1996] Richard Gabriel. *Patterns of Software*. New York: Oxford University Press, 1996.
- [Gamma 1991] Erich Gamma. *Object-Oriented Software Development Based on ET++: Design Patterns, Class Library, Tools* (in German). PhD Thesis, University of Zurich *Institut fur Informatik*, 1991.
- [Gamma 1993] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Design Patterns: Abstraction and Reuse of Object-Oriented Design," In *Proceedings of ECOOP '93*, pp. 406-421, Kaiserslautern, Germany, July 26-30, 1993.
- [Gamma 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

- [Glaser 1984] Hugh Glaser, Chris Hankin, and David Till. *Principles of Functional Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [Gomaa 1993] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems*. Reading, MA: Addison-Wesley, 1993.
- [Graham 1994] Ian Graham. *Object Oriented Methods, Second Edition*. Reading, MA: Addison-Wesley, 1994.
- [Griswold 1990] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [Hailpern 1986] Brent Hailpern. "Multiparadigm Languages and Environments." *IEEE Software*, 3(1):6-9, January 1986.
- [Halbert 1987] Daniel C. Halbert and Patrick D. O'Brien. "Object-Oriented Development." *IEEE Software*, 4(5):71-79, September 1987.
- [Hartel 1995] Pieter H. Hartel and L.O. Hertzberger. "Paradigms and Laboratories in the Core Computer Science Curriculum: An Overview." *SIGCSE Bulletin*, 27(4):13-20, December 1995.
- [Henderson 1980] Peter Henderson. *Functional Programming: Application and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
- [Hogger 1984] Christopher John Hogger. *Introduction to Logic Programming*. Orlando, Fla.: Academic Press, Inc., 1984.
- [Hudak 1989] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages." *ACM Computing Surveys*, 21(3):359-411, September 1989.
- [Hudak 1992] Paul Hudak and Joseph Fasel. "A Gentle Introduction to Haskell." *ACM Sigplan Notices*, 27(5):1-53, May 1992.
- [Hughes 1987] Joan K. Hughes, Glen C. Michtom, and Jay I. Michtom, *A Structured Approach to Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [Hughes 1989] John Hughes. "Why Functional Programming Matters." *The Computer Journal*, 32(2), April 1989.
- [Ingalls 1978] D. Ingalls. "The Smalltalk-76 Programming System, Design and Implementation." *Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978.

- [Iverson 1962] Kenneth E. Iverson. *A Programming Language*. New York: Wiley, 1962.
- [Jackson 1975] Michael A. Jackson. *Principles of Program Design*. London: Academic Press, 1975.
- [Jackson 1983] Michael A. Jackson. *System Development*. Englewood Cliffs, NJ: Prentice Hall, 1983.
- [Johnson 1992] Ralph Johnson. "Documenting Frameworks Using Patterns." *In Proceedings of OOPSLA '92* (Vancouver, B.C. Oct. 1992), pp. 63-76.
- [Johnson 1994] Ralph Johnson. "Why a Conference on Pattern Languages?" *Software Engineering Notes*, 19(1):50-52, January 1994.
- [Jones 1979] Capers Jones. "A Survey of Programming Design and Specification Techniques." *In IEEE Proceedings, Specifications of Reliable Software*, 1979.
- [Justice 1994] Timothy P. Justice, Rajeev K. Pandey, and Timothy A. Budd. "A Multiparadigm Approach to Compiler Construction." *ACM SIGPLAN Notices*, 29(9):29-37, September 1994.
- [Kamin 1990] Samuel N. Kamin *Programming Languages: An Interpreter-Based Approach*. Reading, MA: Addison-Wesley, 1990.
- [Kay 1993] Alan C. Kay. "The Early History of Smalltalk." *The Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*, *ACM SIGPLAN Notices*, 28(3):69-75, March 1993.
- [Keene 1989] Sandra E. Keene. *Object-Oriented Programming in Common Lisp*. Reading, MA: Addison-Wesley, 1989.
- [Kelly 1987] J.C. Kelly. "A Comparison of Four Design Methods for Real-Time Systems." *Proceedings of the Ninth International Conference on Software Engineering*, (Monterey, California, March 30 – April 2, 1987), IEEE Computer Society Press, pp. 46-61.
- [Kernighan 1978] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
- [King 1988] K.N. King. *TopSpeed Modula-2, Language Tutorial*. London: Jensen & Partners, 1988.
- [Knuth 1968] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms, Volume 1*. Reading, MA: Addison-Wesley, 1968.

- [Knuth 1974] Donald Knuth. "Structured Program with Go To Statements." *Computing Surveys*, 6(4):261-301, December 1974.
- [Knuth 1984] Donald E. Knuth. "Literate Programming." *The Computer Journal*, 27(2):97-111, February 1984.
- [Knutson 1996a] Charles D. Knutson, Timothy A. Budd, and Curtis R. Cook. "Multiparadigm Patterns of Thought and Design." *Proceedings of the Third Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, September 4-6, 1996.
- [Knutson 1996b] Charles D. Knutson, Timothy A. Budd, and Curtis R. Cook. "Multiparadigm Iteration Patterns." *Technical Report 96-60-14*, Oregon State University, 1996.
- [Knutson 1997a] Charles D. Knutson, Hugh Vidos, and Timothy A. Budd. "Multiparadigm Design of a Simple Relational Database: A Preliminary Study." *Technical Report 97-60-1*, Oregon State University, 1997.
- [Knutson 1997b] Charles D. Knutson, Scott N. Woodfield, and Ben Brown. "Suitability Criteria for LAN-based Unconstrained Distributed Objects." *Technical Report 97-60-5*, Oregon State University, 1997.
- [Knutson 1997c] Charles D. Knutson and Scott N. Woodfield. "A Comparison of Five Object-Oriented Static Declarative Models." *Technical Report 97-60-9*, Oregon State University, 1997.
- [Knutson 1997d] Charles D. Knutson and Scott N. Woodfield. "A Comparison of Five Object-Oriented Behavioral Models." *Technical Report 97-60-10*, Oregon State University, 1997.
- [Knutson 1997e] Charles D. Knutson and Scott N. Woodfield. "A Comparison of Five Object-Oriented Interactive Models." *Technical Report 97-60-11*, Oregon State University, 1997.
- [Koffman 1981] Elliot B. Koffman. *Problem Solving and Structured Programming in Pascal*. Reading, MA: Addison-Wesley, 1981.
- [Kuhne 1994] Thomas Kuhne. "Higher Order Objects in Pure Object-Oriented Languages." *ACM SIGPlan Notices*, 29(7):15-20, July 1994.
- [Lea 1994] Doug Lea. "Christopher Alexander: An Introduction for Object-Oriented Designs." *ACM SIGSOFT Software Engineering Notices*, 19(1):39-46, January 1994.

- [Leska 1996] Chuck Leska, John Barr, and L.A. Smith King. "Multiple Paradigms in CS I." *Proceedings of ACM SIGSCE 27th Technical Symposium*, Philadelphia, Pennsylvania, February 1996.
- [Liskov 1974] B. Liskov and S. Zilles. "Programming with Abstract Data Types." *ACM Sigplan Notices*, 9(4):50-59, September 1974.
- [Liskov 1986] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. New York: McGraw-Hill, 1986.
- [Lloyd 1984] John W. Lloyd. *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1984.
- [Loia 1993] Vincenzo Loia and Michel Quaggetto. "Extending CLOS towards Logic Programming: A Proposal." *OOPS Messenger*, 4(1):46-51, January 1993.
- [Lopez 1996] Antonio M. Lopez, Jr. "Multiparadigm Software Design and Evolving Intelligence in Information Systems." *Software Engineering Technology*, June 1996.
- [Luker 1989] Paul A. Luker. "Never Mind the Language, What About the Paradigm?" *Proceedings of the Twentieth SIGCSE Tehcnical Symposium on Computer Science Education*, pp. 252-256, Louisville, Kentucky, February 1989.
- [MacLennan 1990] Bruce J. MacLennan. *Functional Programming, Practice and Theory*. Reading, MA: Addison-Wesley, 1990.
- [Mandell 1985] Steven L. Mandell and Colleen J. Mandell. *Understanding Pascal: A Problem Solving Approach*. St. Paul, MN: West Publishing Company, 1985.
- [Martin 1997] Robert Martin. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1997.
- [McCabe 1983] C. Kevin McCabe. *Forth Fundamentals, Volume 1: Language Usage*. Beaverton, Oregon: Dilithium Press, 1983.
- [McCarthy 1960] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine." *Communications of the ACM*, 3(4):184-195, April 1960.
- [McCarthy 1962] John McCarthy. *Lisp 1.5 Programmer's Manual*. Cambridge, MA: The MIT Press, 1962.

- [Meszaros 1996] Gerard Meszaros and Jim Doble. "MetaPatterns: A Pattern Language for Pattern Writing." *Proceedings of the Third Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, September 4-6, 1996.
- [Meyer 1985] Bertrand Meyer. *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1985.
- [Microsoft 1993a] Microsoft. *Microsoft Visual Basic Development System for Windows*. Redmond, WA: Microsoft Corporation, 1993.
- [Microsoft 1993b] Microsoft. *Microsoft Visual C++ Development System for Windows*. Redmond, WA: Microsoft Corporation, 1993.
- [Milner 1990] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. Boston, MA: MIT Press, 1990.
- [Moore 1991] Geoffrey A. Moore. *Crossing the Chasm*. New York: Harper Collins, 1991.
- [NASA 1993] National Aeronautics and Space Administration. "Software Design Methods for Multiple Paradigm Programming." Goddard Space Flight Center, September 1993.
- [Ng 1995] K.W. Ng and C.K. Luk. "I+: A Multiparadigm Language for Object-Oriented Declarative Programming." *Computer Language*, 21(2):81-100, February 1995.
- [Nygaard 1981] K. Nygaard and O-J Dahl. "The Development of the Simula Languages." In *History of Programming Languages*. Ed. R. Wexelblat. New York, NY: Academic Press, 1981.
- [Ormerod 1993] Thomas C. Ormerod and Linden J. Ball. "Does Programming Knowledge or Design Strategy Determine Shifts of Focus in Prolog Programming?" In *Empirical Studies of Programmers '6*, Palo Alto, CA:Ablex, 1993.
- [Ormerod 1994] Thomas C. Ormerod and Linden J. Ball. "An Empirical Evaluation of TEd, A Techniques Editor for Prolog Programming." In *Empirical Studies of Programmers '6*, Palo Alto, CA:Ablex, 1994.
- [Parnas 1972] D. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM*, 15(12):1053-58, December 1972.
- [Paulson 1991] L.C. Paulson. *ML For the Working Programmer*. New York: Cambridge University Press, 1991.

- [Peyton-Jones 1987] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
- [Peyton-Jones 1993] Simon Peyton-Jones and Philip Wadler. "Imperative Functional Programming." *ACM Principles of Programming Languages*, 1993.
- [Peyton-Jones 1996] Simon Peyton-Jones. Personal conversation with the author.
- [Pinson 1991] Lewis J. Pinson, and Richard S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Reading, MA: Addison-Wesley, 1991.
- [Placer 1991a] John Placer. "The Multiparadigm Language G." *Computer Language*, 16(3):235-258, 1991.
- [Placer 1991b] John Placer. "Multiparadigm Research: A New Direction in Language Design." *ACM SIGPLAN Notices*, 26(3):9-17, March 1991.
- [Placer 1993] John Placer. "The Promise of Multiparadigm Languages as Pedagogical Tools." ACM 21st Annual Computer Science Conference, Indianapolis, Indiana, February 1993.
- [Polivka 1975] Raymond P. Polivka and Sandra Pakin. *APL: The Language and Its Usage*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
- [Pree 1994] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.
- [Pree 1997] Wolfgang Pree. "Essential Framework Design Patterns." *Object Magazine*., (3):34-37, March 1997.
- [Pressman 1987] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 1987.
- [Prieto-Diaz 1987] Rubén Prieto-Diaz and Peter Freeman. "Classifying Software for Reusability." *IEEE Software*, (1):6-16, January 1987.
- [Rentsch 1982] T. Rentsch. "Object-Oriented Programming." *SIGPLAN Notices*, 17(12):51, September 1982.
- [Robertson 1995] Stephanie A. Robertson and Martin P. Lee. "The Application of Second Natural Language Acquisition Pedagogy to the teaching of Programming Languages – A Research Agenda." *SIGCSE Bulletin*, 27(4):9-20, December 1995.
- [Ross 1989] Peter Ross. *Advanced Prolog: Techniques and Examples*. Reading, MA: Addison-Wesley, 1989.

- [Ross 1997] John Minor Ross and Huazhong Zhang. "Structured Programmers Learning Object-Oriented Programming: Cognitive Considerations." *SIGCHI Bulletin*, 29(4):93-99, October 1997.
- [Rumbaugh 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- [Sapir 1921] Edward Sapir. *Language: An Introduction to the Study of Speech*. New York: Harcourt Brace Jovanovich, 1921.
- [Schmidt 1995] Douglas C. Schmidt. "Using Design Patterns to Develop Reusable Object-Oriented Communication Software." *Communications of the ACM*, 38(10):65-74, October 1995.
- [Schmidt 1996] Douglas C. Schmidt, Ralph E. Johnson, Mohamed Fayad. "Software Patterns." *Communications of the ACM*, 39(10), October 1996.
- [Schneider 1981] G. Michael Schneider and Steven C. Bruell. *Advanced Programming and Problem Solving with Pascal*. New York: John Wiley & Sons, 1981.
- [Shaw 1995] Mary Shaw. "Patterns for Software Architectures." In *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.
- [Shaw 1996] Mary Shaw. "Some Patterns for Software Architectures." In *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.
- [Shlaer 1988] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, 1988.
- [Shlaer 1992] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, NJ: Yourdon Press, 1992.
- [Soloway 1984] Elliot Soloway and Kate Ehrlich. "Empirical studies of programming knowledge." *IEEE Transactions on Software Engineering*, 10(5):595-609, September 1984.
- [Steele 1978] Guy L. Steele and Gerald J. Sussman. "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One and Two)." *MIT AI Memo No. 453*, May, 1978.
- [Sterling 1986] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Cambridge, MA, MIT Press 1986.

- [Stroustrup 1991] Bjarne Stroustrup. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1991.
- [Tepfenhart 1997] William M. Tepfenhart and James J. Cusick. "A Unified Object Topology." *IEEE Software*, 14(1):31-35, January 1997.
- [Tesler 1985] Larry Tesler. *Object Pascal Report*. Santa Clara, CA: Apple Computer, 1985.
- [Thompson 1996] Simon Thompson. *Haskell: The Craft of Functional Programming*. Reading, MA: Addison-Wesley, 1996.
- [Vlissides 1996] John Vlissides, James Coplien and Norman Kerth, Editors. *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley, 1996.
- [Vlissides 1997] John Vlissides. "Patterns: The Top Ten Misconceptions." *Object Magazine*, (3):31-33, March 1997.
- [Wadler 1990] Philip Wadler. "Comprehending Monads." *ACM Conference on LISP & Functional Programming*, Nice, France, 1990.
- [Wadler 1992] Philip Wadler. "The Essence of Functional Programming." *ACM Principles of Programming Languages*, 1992.
- [Warnier 1977] J.D. Warnier. *Logical Construction of Programs*. New York: Van Nostrand, 1977.
- [Wells 1989] Mark B. Wells and Barry L. Kurtz. "Teaching Multiple Programming Paradigms: A Proposal for a Paradigm-General Pseudocode." *Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, pp. 246-251, Louisville, Kentucky, February 1989.
- [Welsh 1995] Matt Welsh and Lar Kaufman. *Running Linux*. Sebastopol, CA: O'Reilly, 1995.
- [Whitaker 1996] William A. Whitaker. "Ada—The Project: The DoD High Order Language Working Group." *History of Programming Languages II*. Reading MA: Addison-Wesley, 1996.
- [Whorf 1956] Benjamin Lee Whorf. *Language Thought and Reality*. Cambridge, MA: MIT Press 1956.
- [Wikstrom 1987] Ake Wikstrom. *Functional Programming Using Standard ML*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

- [Wiener 1984] R. Wiener and R. Sincovec. *Software Engineering with Modula-2 and Ada*. New York: Wiley Press, 1984.
- [Wilensky 1984] Robert Wilensky. *LISPcraft*. New York: Norton, 1984.
- [Winfield 1983] Alan Winfield. *The Complete Forth*. New York: Wiley Press, 1983.
- [Winograd 1979] Terry Winograd. "Beyond Programming Languages." *Communications of the ACM*, 22(7):391-401, July 1979.
- [Wirfs-Brock 1990a] Rebecca Wirfs-Brock and Ralph E. Johnson. "A Survey of Current Research in Object-Oriented Design." *Communications of the ACM*, 33(9):104-124, 1990.
- [Wirfs-Brock 1990b] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- [Wirth 1971a] Niklaus Wirth. "The Programming Language Pascal. Technical Report 1." Fachgruppe Computer-Wissenschaften, ETH, Nov. 1970; *Acta Informatica*, (1):35-63, 1971.
- [Wirth 1971b] Niklaus Wirth. "Program Development by Stepwise Refinement." *Communications of the ACM*, 14(4):221-7, March 1971.
- [Wirth 1985] Nicklaus Wirth. *Programming in Modula-2*. New York, NY: Springer-Verlag, 1985.
- [Wulf 1972] "A Case Against the GOTO." *Proceedings of the 25th National ACM Conference*, pp. 791-97, August 1972.
- [Xu 1995] Dianxiang Xu and Guoliang Zheng. "Logical Objects with Constraints." *ACM SIGPlan Notices*, 30(1):5-10, January 1995.
- [Yourdon 1989] Ed Yourdon. *Modern Structured Analysis*. Englewood Cliffs, NJ: Yourdon Press, 1989.
- [Zave 1983] Pamela Zave and Michael Jackson. "Conjunction as Composition." *ACM Transactions on Software Engineering and Methodology*, 2(4):379-411, October 1983.
- [Zave 1989] Pamela Zave. "A Compositional Approach to Multiparadigm Programming." *IEEE Software*, 15(9):15-25, September 1989.
- [Zettel 1995] Leonard Zettel. "Toward an Object-Oriented Forth." *ACM SIGPlan Notices*, (31):4, April, 1996.