

1996

Reverse Engineering Low-Level Design Patterns From Object-Oriented Code.

Chandra Shrivastava
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_disstheses

Recommended Citation

Shrivastava, Chandra, "Reverse Engineering Low-Level Design Patterns From Object-Oriented Code." (1996). *LSU Historical Dissertations and Theses*. 6163.
https://repository.lsu.edu/gradschool_disstheses/6163

This Dissertation is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

**A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600**

**REVERSE ENGINEERING LOW-LEVEL DESIGN PATTERNS
FROM OBJECT-ORIENTED CODE**

A Dissertation

**Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment for the degree of
Doctor of Philosophy**

in

The Department of Computer Science

by

Chandra Shrivastava

BS in Physics, Fergusson College, Poona, 1987

MS in Computer Science, Poona University, Poona, 1989

May, 1996

UMI Number: 9628317

**UMI Microform 9628317
Copyright 1996, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

This work is dedicated to my parents and my family

Acknowledgements

I take this opportunity to record my gratitude towards the many people who touched my life in significant ways; particularly those who helped me during the time I was enrolled in the doctoral program at Louisiana State University.

My parents are amongst the wisest teachers I have and they have taught me the fine arts of survival, communication, and humanity. To convert my weaknesses into my strengths; to comprehend the importance of understanding fundamental concepts in their entirety; to fully realize the harm that any form of deception can cause; to learn to adapt to new environments and coexist with a wide cross-section of humanity; to always do my best in any endeavor I choose to undertake; to accept disappointments, rewards, appreciation and criticism with equanimity; to be honest, truthful, enterprising, resourceful and hardworking; these are some of the principles they strove to inculcate in all their children. Their single-minded dedication and commitment to the goal of ensuring a better future for their children by giving them the best and highest possible education has been a motivating and driving factor in all my endeavours. Their consistent and unconditional love and support have been my greatest source of solace, strength and security. The enormous effort, dedication and commitment required for a dissertation were revealed to me by my elder sister, Dr. Indira Shrivastava. Her gentle guidance and encouragement have helped me along. To my elder brother Major Rajesh K. Shrivastava I owe special thanks for urging me to complete my studies. I would also like to record my appreciation for the help extended and concern shown by Rajan Chandras towards my entire family.

Dr. Doris L. Carver, my advisor, guide and friend has shown me how wonderful a student-teacher relationship can be. Her professionalism, versatility in the different branches of software engineering, enthusiasm to learn something new, intelligence and humanity have evoked my deepest respect and admiration. Her dedication and commitment to work and quality is a source of inspiration to me. She has always taken the time to listen to my ideas, has patiently answered my questions, both technical and otherwise and has helped me build my self-confidence. She has unhesitatingly lent her shoulder for me to lean on and I have gratefully accepted it. I am singularly fortunate to have interacted with a person of such strength and character. I would also like to thank her for allowing me to use the facilities and equipment of the Software Engineering Laboratory.

I would especially like to thank Jigang Liu for the smooth running of the SELAB computer network and for providing intellectually challenging discussions. I would like to thank all the members of the software engineering research group for their camaraderie and for the wonderful atmosphere in SELAB.

I would like to thank all the members of my doctoral advisory committee, Dr. Ghosh, Dr. Iyengar, Dr. Jones, Dr. Tyler and Dr. Smolinsky for taking the time to review my dissertation and offer their invaluable suggestions.

I owe very special thanks to Dr. Kraft for admitting me into the doctoral program in 1989 and again in 1991. The financial assistantship provided by Dr. Geske of the EDAF department is gratefully acknowledged. The decision of the assistantship committee in the Computer Science Department, to appoint me as a teaching assistant from January 1992 is very gratefully acknowledged. To Gina Mounfield and Jerry Weltman I owe very special thanks for their support and tolerance of my idiosyncracies and for accomodating many special requests. Without

the excellent systems management support provided by the hardworking group of Elias Khalaf, Deky Gouw and Amit Nanavati I would not have been able to meet many assignment and conference deadlines. The enthusiasm and readiness with which they addressed all my problems are appreciated. I would like to thank Jane Crawford for keeping me abreast of paperwork and for being a wonderful friend and helping me through each semester. Special thanks are owed to Betty Rushing and Mary Adcock at Graduate School for their co-operation and help whenever I solicited it.

I would like to acknowledge Drs Wesley and Demetria Mcjulien for alleviating some of the homesickness and loneliness I felt when I first came to Baton Rouge. The concern and love shown by Mr. Morris Reynaud, Mrs. Jean Reynaud and Mrs. Ruby Lantz has also helped me adjust to Baton Rouge.

Vinayak Hegde, Sankar Krishnamurthy, Raghuram Yedatore, Dipti Sonak, Amit Nanavati and Sundar Vedantham made my stay at LSU a fun-filled experience; their wonderful company and friendship will be missed. Especial thanks to a dear friend of mine, Raghuram, for granting me many favors and coming to my aid whenever I requested his help. A special word of thanks to a very good friend of mine, Dr. Ramana Rao who helped me get the right perspective on several problems on more than one occasion. Sai Pinepalli's pleasant company provided welcome relief.

My friends from Poona University, Ranjit Mavinkurve, Tapasi Ray, Dr. Varsha Apte and Vaishali Khandekar exercised every means of communication – email, letters, telephone, photographs, visits, books, music, movies – and often boosted my flagging spirits. Very special thanks to R. Venkatesh for his insightful comments and suggestions on inheritance and other class relationships. The selflessness with which

he helps and educates people have made a deep impact on me. He is responsible for teaching me the yes-no-black-white game and for sparking a deep interest in object-orientation. Philosophical interludes with Tapasi Ray have helped me mature and become more tolerant and patient.

To the faculty at Poona University I owe my deepest gratitude for enlightening me with their perceptions and ideas of the fundamental concepts underlying the fast-changing field of computer science. Especially worthy of mention are Prof. H.V. Sahasrabudde, Mr. S.N. Sapre, Dr. H.Diwakar, Mr. R.P. Mody and Prof. K.V. Nori. I am indebted to Mr. S.N. Sapre for sharing his depth of knowledge and expertise in wide-ranging areas with me.

The generous and enveloping love and understanding of my eldest sister-in-law, Cynthia D. Reynaud has made my life and stay in Baton Rouge a beautiful experience. The innumerable discussions we have had, have helped me understand and cross the wide gaps of culture, religion, language, diets, music, east and west. She, more than anybody else, taught me the beauty of abstraction. She has shouldered my responsibilities thereby freeing me to devote my time to research. She is significantly responsible in helping me towards the doctor of philosophy degree.

My eldest brother, Dr. Rajendra K. Shrivastava has been a mentor and a role model since a very early age. I would not have been able to do this dissertation without his moral support and guidance. I would like to acknowledge Nikhil Shyamani for giving me sound advice and for being a terrific friend. One of the best.

Contents

Acknowledgements	iii
List of Tables	xi
List of Figures	xii
Abstract	xiv
Chapter	
1 Introduction	1
1.1 The Problem	1
1.1.1 Understanding the Behavior of Software Systems	2
1.1.2 Understanding the Structure of Software Systems	6
1.1.3 Graph Representations of Software Systems	10
1.1.4 Understanding Software Systems	12
1.2 The Context	13
1.2.1 Role of a Maintainer in the Software Lifecycle	14
1.2.1.1 Analysis	15
1.2.1.2 Design	15
1.2.1.3 Implementation	17
1.2.1.4 Maintenance	17
1.3 Modeling Object-Oriented Software Systems	20
1.3.1 Overview of LLSA and LLDP	21
1.3.2 Design Rationale of the Low-Level Software Architecture Model	24
1.3.3 Using LLSA and LLDPs for Maintenance of Object-Oriented Systems	25
1.4 Pulse	25
1.5 Overall Organization	27
2 Motivation and Background	29
2.1 Motivation	29
2.2 Issues in Understanding Object-Oriented Systems	31
2.3 Inconsistent Documentation	32
2.4 Maintenance Aids	34
2.5 Background	36
2.5.1 Software Engineering	36
2.5.2 Forward Engineering	39

2.5.3	Reverse Engineering	40
2.5.4	Restructuring	41
2.5.5	Reengineering	42
2.5.6	Representations of Program Structure	42
2.5.7	Object-Oriented Concepts	45
2.5.7.1	Abstraction	47
2.5.7.2	Encapsulation	47
2.5.7.3	Inheritance	48
2.5.7.4	Polymorphism	49
2.6	Object Oriented Design	50
2.7	Summary	51
3	Related Research	53
3.1	Programmer's Apprentice	53
3.2	Desire	55
3.3	Valhalla	56
3.4	Demeter	57
3.5	OOTME	58
3.6	CIA, XREF/XREFDB, SAM	59
3.7	Browser	59
3.8	SCRUPLE	60
3.9	PERPLEX	61
3.10	Restructuring	62
3.11	Related Research in Reverse Engineering	63
3.12	Summary	65
4	Low-Level Software Architecture of OO Systems	68
4.1	Introduction	68
4.2	LLSA Conceptual Model	69
4.2.1	Theoretical Model	70
4.2.2	Graph Representation and Views	72
4.2.2.1	Control Flow Graph View	75
4.2.2.2	Component Domain Graph View	76
4.2.2.3	Rooted Component Subgraph View	77
4.3	Low-Level Software Architecture of C++ Programs	79
4.3.1	Component Description	80
4.3.2	LLSA Components of a C++ Software System	80
4.3.3	Component Interface	83
4.3.4	Component Interactions	86
4.4	Representational Support of LLSA	91
4.5	How will LLSA be used ?	92

4.6	Summary	92
5	Low-Level Design Patterns	94
5.1	Introduction	94
5.2	Pattern Languages	95
5.3	Design Patterns	97
5.4	Low-Level Design Patterns	99
5.4.1	Low-Level Design Pattern Structures	100
5.4.2	LLDP Structure that Exposes Hidden Dependencies	101
5.4.3	LLDP Structure Embedded in the LLSA Description of a Component	103
5.4.4	Low-Level Design Pattern Template	103
5.4.5	Polymorphism	105
5.4.5.1	Ad-hoc Polymorphism	105
5.4.5.2	Polymorphism And Reuse	108
5.4.5.3	Polymorphism Using Inheritance and Dynamic Bind- ing	108
5.4.6	Decoupling	111
5.4.6.1	Decoupling a Class from its Representation	113
5.4.6.2	Decoupling for Flexible Design	113
5.4.6.3	Decoupling a Function from a Class	115
5.4.7	Messages	115
5.4.7.1	Messages Between an Object and a SubObject	117
5.4.7.2	Messages Between Objects of a Class	117
5.4.7.3	Messages Between Objects of Different Classes	120
5.5	Summary	120
6	Using LLSA and LLDP for Maintaining OO Systems	122
6.1	LLSA as an Aid for Software Maintenance	122
6.1.1	Using LLSA for Understanding the Structure	123
6.1.1.1	Logical and Physical Organization	123
6.1.1.2	Static and Dynamic Structure	124
6.1.2	Using an LLSA Description to Understand a Component	126
6.1.3	Code Navigation	128
6.2	LLDP as an Aid for Software Maintenance	128
6.2.1	Using LLSA and LLDP in Code Modifications	130
6.3	Summary	132
7	Reverse Engineering LLDPs	134
7.1	Overall Architecture of pulse	136
7.2	Symbol Table Organization	138

7.3	Scanner	140
7.4	Parser	140
7.4.1	Fundamental Patterns and Programming Constructs	143
7.5	LLSA Generator	144
7.5.1	Class Component LLSA	145
7.5.1.1	Static Interface	145
7.5.1.2	Dynamic Interface	147
7.5.2	Function Component LLSA	150
7.5.3	Object Component LLSA	150
7.6	LLDP Recognizer	153
7.6.1	Identification of the Polymorphism LLDPs	154
7.6.2	Identification of the Decoupling LLDPs	156
7.6.3	Identification of the Message LLDPs	158
7.7	Sample Session Using pulse	160
7.8	Summary	162
8	Conclusion	163
8.1	Contributions	163
8.2	Extensions and Future Work	169
	Bibliography	172
	Appendix : Session Listing	180
	Vita	186

List of Tables

1.1	Graph-theoretic Definition of a Call Graph	6
1.2	Graph-theoretic Definition of a Dependency Graph	9
3.1	Summary of Different Approaches	63
4.1	Low-Level Software Architecture of Object-Oriented Systems	73
4.2	Transitive Closure of A Relation	75
4.3	Graph-theoretic Definition of a Rooted Component Subgraph	77
4.4	Definition of an LLSA Graph in terms of Rooted Component Subgraphs	79
6.1	View Construction from LLSA Representation	126
7.1	Fundamental Patterns and Programming Constructs	144
7.2	Class LLSA Interactions and Fundamental Patterns	150
7.3	Class LLSA Interactions Heuristics	153
7.4	Function LLSA Interactions and Fundamental Patterns	153
7.5	Polymorphism LLDPs and LLSA Interactions	156
7.6	Decoupling LLDPs and LLSA Interactions	158
7.7	Message LLDPs and LLSA Interactions	158

List of Figures

1.1	Call Graph Representation of a Software System	5
1.2	A Software Maintenance Process Model	19
1.3	Views of a Software System	23
1.4	A Software Maintenance Process Model Using LLSA and LLDP	26
2.1	Abstract Representations of Software System Structure	43
4.1	Graph Representations of a Software System	74
4.2	Object Component Description Template	80
4.3	Class Component Description Template	81
4.4	Function Component Description Template	82
5.1	Geometric Patterns	95
5.2	Relationship between LLDPs, LLSA, Fundamental Patterns and Language Constructs	101
5.3	Low-Level Design Patterns Defined Over LLSA	102
5.4	Low-Level Design Pattern Template	104
5.5	Polymorphism LLDP-1	107
5.6	Polymorphism LLDP-2	109
5.7	Polymorphism LLDP-3	110
5.8	Decoupling LLDP-1	112
5.9	Decoupling LLDP-2	114
5.10	Decoupling LLDP-3	116
5.11	Messages LLDP-1	118
5.12	Messages LLDP-2	119
5.13	Messages LLDP-3	121

7.1	Overall Architecture of pulse	136
7.2	Symbol Table Data Structure	139
7.3	Algorithm to Compute the Static Interface of a Class	148
7.4	Algorithm to Compute the Dynamic Interface of a Class	149
7.5	Algorithm to Compute the Dynamic Interface of a Function	151
7.6	Algorithm to Compute the Dynamic Interface of an Object	152
7.7	LLDP Recognizer Algorithm	155
7.8	Algorithm to Recognize Polymorphism LLDPs	157
7.9	Algorithm to Recognize Decoupling LLDPs	159
7.10	Algorithm to Recognize Message LLDPs	160
7.11	A C++ Software System	161

Abstract

The purpose of this research is to develop an automatically extractable abstract representation model of object-oriented (abbreviated as OO) software systems that captures the structure of the system and code dependencies in order to aid maintenance. The research results include the development of two abstract representation models – the *low-level design pattern (LLDP)* abstract model and the *low-level software architecture (LLSA)* abstract model. The LLDP model is at a higher level of abstraction than the LLSA model. The LLSA model acts as an intermediate representation between the LLDP model and an OO software system. The design of the LLSA and LLDP representation models and the automatic extraction of these models from an OO software system are significant contributions of this research.

An LLDP representation is a textual description of common OO strategies. Three sets of LLDPs – *polymorphism*, *decoupling* and *messages* are defined. LLDPs describe the structure, the benefits and consequences of a strategy. The design of the LLSA model considers the complexities inherent in OO systems and the requirements of a maintainer from such a model. The LLSA model defines software components, static and dynamic interfaces of components, and static and dynamic interactions between components. Software components are defined in terms of OO programming language constructs, and interactions between the components are defined in terms of OO relationships that exist between the components. Understanding the relationships is necessary to understand what code dependencies occur and why they occur. The LLSA abstract model in conjunction with the LLDPs provides a view of software systems that captures the dependency relationships between code, the nature of the dependencies and the reasons why the dependencies must exist and be preserved. The LLSA model of C++ software systems in particular

are defined. The usefulness of the LLSA and LLDP models from the maintenance perspective are explored.

A prototype CASE tool, *pulse*, was implemented to demonstrate the feasibility of automatic extraction of both models. Reverse engineering and code analysis techniques were developed to extract the LLSA relationships and interfaces and to recognize the LLDP model.

Chapter 1

Introduction

1.1 The Problem

Understanding a software system is a difficult problem. To understand something is to know its meaning. In order to grasp the meaning of something fully, one must know the reason for its existence and its nature. The purpose of this research was to design a model which represents the nature of object-oriented software systems to aid program understanding from the maintenance point of view.

Every software system has a reason for its existence. User requirements to automate some process or activity often results in the development of a software system. A user may be a person, a company, a programmer, a hardware device or another software system. The requirements of a user are analyzed in the light of many factors even before the decision to develop software is made. The objective of requirements analysis is to obtain a clear picture of the real needs of the user. The requirements are then closely examined to determine if they can be automated (called *feasibility study*) and the effort that would have to be expended in the automation (called *cost analysis*). This study of user requirements typically results in a collection of documents which contains a precise specification of the user's needs. *Software development* is the activity of transforming the user's needs into a software system. Therefore, a software system meets the user's needs and the user's requirements justify the software's existence. The intent of a software system can

be gleaned from the requirements specification documents. Understanding the techniques that were employed in designing the software is a much more complicated task.

There are two aspects to the complex nature of software system – behavior and structure. The response of a system to some input is referred to as the behavior of the system. The structure of a software system is determined by the logical and physical organization of code and the relationships that exist within it. A thorough understanding of a software system is possible if the behavior and the structure can be explained. The behavior and structure of a system are mutually dependent aspects; the structure of the system permits the software to behave in a desired way and the behavior that is expected from a software is the reason the software is structured (or organized) in a particular way.

1.1.1 Understanding the Behavior of Software Systems

A *well-behaved* software system is one that responds in a predictable manner to all conceivable inputs. An *ill-behaved* system is one that behaves erratically and with unpredictable responses on some or all inputs.

Understanding well-behaved systems can be done by performing an *execution trace* on various inputs and examining the input, the trace and the output. An execution trace of a software system is the complete path of execution that a software system follows on a particular input. We perform a trace by starting from a particular function (or procedure), examining the functions that it calls, and then examining the functions that the called functions themselves call until a point where no more functions are called is reached. A trace therefore is a complete sequence

of function calls and this sequence explains the step-wise response of the software system to a particular input. We shall refer to this as a forward trace.

Understanding the behavior of an ill-behaved system is difficult because the trace of the system on some input is incomplete and provides partial information, whereas the trace of a well-behaved system contains complete information. There are at least two common situations that allow us to classify a software system as unpredictable – abnormal termination and infinite looping. Abnormal termination of a system is the situation when the execution of the system is abruptly and externally terminated due to some violation performed by the system or due to some abnormal event created by the system. A system is said to be in an infinite loop if it performs the same set of instructions over and over again and the condition for the system to come out of the loop can never be true. In either event, locating the precise point (or function) at which the software started behaving abnormally is necessary. The point the software system reached before it started behaving erratically becomes the starting point in understanding *why* the system behaved abnormally. The function that is next examined is the function that called the function that caused the system to behave unpredictably. Thus, understanding of the behavior of ill-behaved software systems progresses in a direction opposite to that of understanding the behavior of well-behaved systems. We shall refer to this backward process as a backward trace. The starting point for understanding well-behaved systems is the starting point of execution, whereas the starting point of understanding ill-behaved systems is the termination point of execution or the point of endless execution.

Determining whether a software system is well-behaved or not is contingent on the inputs to the system and it is entirely possible for a system to be well-behaved

with respect to some inputs and ill-behaved with respect to others. Determining if a software system is well-behaved under all circumstances and with respect to all input is virtually impossible. Therefore, any model that attempts to represent the behavior of a software system must aid in the understanding of both kinds of behavior.

A *call graph* is an abstract representation model of software that precisely captures all possible execution paths that exist in the system. The call graph of a program can be represented textually as well as graphically; both these representations are illustrated in figure 1.1. Figure 1.1 shows a program written in C (fig 1.1 (a)), a graphical representation of the call graph of the program (figure 1.1 (b)) and the graph-theoretic representation of the call graph of the program (fig 1.1 (c)). The program computes the i^{th} number in the Fibonacci series. The call graph does not indicate the order in which the calls are made. The number of times a procedure or a function is called is also not indicated in the call graph. For example, the function `main` in figure 1.1 (a) calls `printf` twice and the function `fib` calls itself recursively twice but the graph shows one directed edge between `main` and `printf` and one arrow between `fib` and itself. In the graphical representation, the starting point of execution of a program is indicated as a double circle. A formal graph-theoretic definition of a call graph for a software system is defined in Table 1.1.¹

In table 1.1, S denotes a software system, and $CG(S)$ denotes the call graph of S . The set of vertices V is a collection of names of procedures or functions in S . E , the set of directed edges consists of tuples (v_i, v_j) . A tuple (v_i, v_j) represents a call from procedure v_i to procedure v_j .

¹Figure 1.1 (b) adapted from *The Study of Programming Languages by Ryan Stansifer [Sta94]*

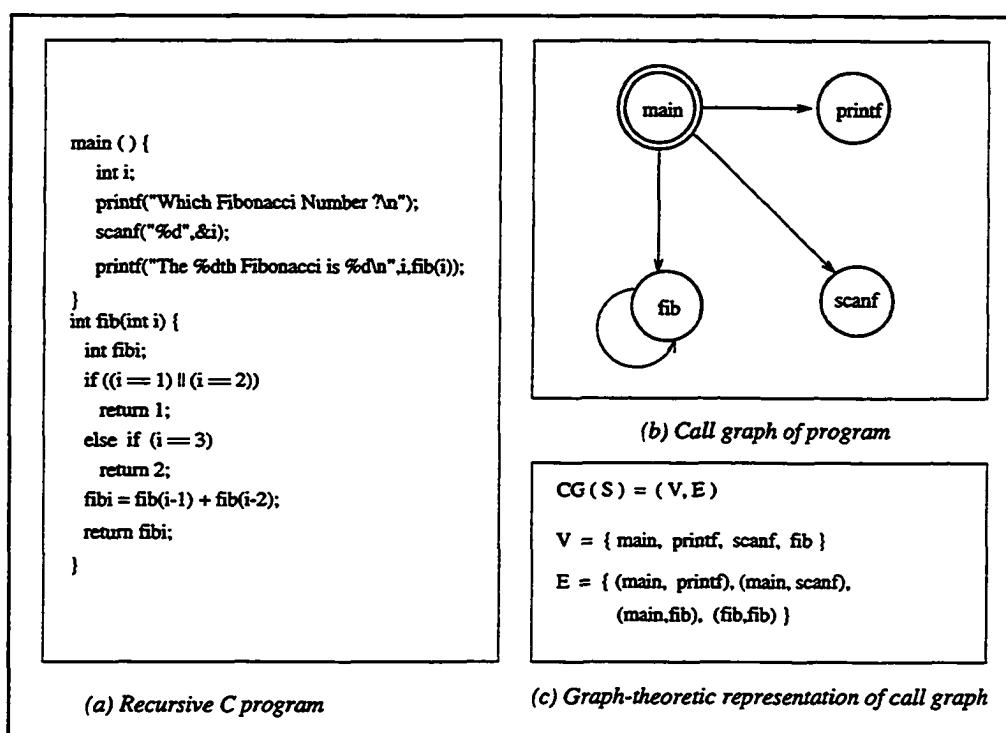


Figure 1.1: Call Graph Representation of a Software System

Table 1.1: Graph-theoretic Definition of a Call Graph

V	=	$\{v_1, v_2, \dots, v_k\}$
E	=	$\{(v_i, v_j) \mid v_i, v_j \in V \text{ and } v_i \text{ calls } v_j\}$
$CG(S)$	=	$\{V, E\}$

The *call graph* of a software system is a graph in which each function or procedure in the system is represented as a node in the graph and each edge in the graph corresponds to a function call in the graph. The edge connects the caller function with the called function. Forward and backward traces can be easily performed on the graph by simply starting at a node and then traversing the edges to reach other nodes. Call graphs aid in understanding the behavior of a software system by depicting the different execution paths possible by function calls. Other graph representations of software systems are described in section 1.1.3.

1.1.2 Understanding the Structure of Software Systems

There is no simple definition for describing the structure of a software system. Software structure has two aspects to it – organization and relationships within the system. There are two kinds of organization in a software system – logical and physical. The logical organization of code is the outcome of mapping and preserving the logical design of the system. The logical design of a system is the representation of the solution in terms of interacting logical components. Logical components are determined by the overall approach or paradigm adopted for software development. Hence, in the object-oriented approach the logical design of a system is expressed in terms of interacting objects and classes, key concepts in object orientation. Logical organization of code refers to the distribution of behavior over different components

and the subsequent interaction between the components to realize the original needs of a user.

The physical organization of code corresponds to the allocation of code to different files and the organization of the files. Typically, the physical organization of code does not correspond to the logical organization of code and determining the logical structure from the code itself becomes a difficult task. Both logical and physical organizations produce logical and physical dependencies between code fragments; alternatively, the logical and physical dependencies respectively determine the logical and physical organization embedded in the code.

Static and dynamic relationships are the two kinds of relationships that can exist between logical components of a system. A static relationship is a fixed, unchanging relationship that establishes a strong and predictable connection between components. A dynamic relationship is indicative of a weak association between components. Components associate dynamically with each other in the context of some *event*. An event is some occurrence that causes the system to change its configuration or state. Events cause components to associate dynamically in order to effect the change in configuration. Once the configuration has changed, the association is no longer necessary and ceases to exist. Thus different events cause different dynamic relationships between components and determining dynamic relationships is based on understanding the events that can occur in the system. The statically related components lay the groundwork for dynamic interactions to occur in a system and therefore the dynamic relationships that are possible can be discerned from the code itself.

Experienced programmers develop techniques that combine programming language constructs and features in elegant ways so that the software system is well-structured and the static and dynamic relations specified in the logical design of the system are realized in the software implementation. An experienced programmer often spends long hours devising a technique that will enable the system to behave in a specific way as well make the system flexible, reusable and maintainable. Programmers are likely to reuse good techniques and therefore maintainers should study and understand the existing techniques, the structure, the benefits and consequences of the techniques so that any code modification performed as a part of the maintenance activity does not destroy the techniques employed by the original developers. In order to detect all possible dynamic relations that exist in a software system, a maintainer must first be aware of all dynamic relations that can exist in a software system and then discover the techniques that develop the static frameworks that allow dynamic interaction between components. Static and dynamic relationships cause complicated and non-trivial dependencies in the code.

A well-structured software system exhibits a logical design and the physical organization of code follows the logical decomposition to the extent possible. A well-structured system is a system that is well-designed and properly implemented. A well-designed system is a system that possesses desirable design properties. Desirable design properties are listed in section 1.2.1. An ill-structured system is one that has either an unclear or complicated logical structure, or one in which the implementation differs vastly from the original design.

A *dependency graph* is a representation model of software systems that captures static and dynamic code dependencies as well as logical and physical code dependencies. The nodes in a dependency graph represent some programming entity (for

example, a function or a variable) and the edges connecting nodes represent different kinds of dependencies that exist among the connected nodes [WHH89]. The call graph model is a special kind of dependency graph.

Notation : Let S denote a software system, and $DG(S)$ the dependency graph of S . Let the set VN represent the names of all the variables in S and the set PN represent the names of all procedures and functions in S . Let the set of vertices V represent the collection of names of programming entities (procedure or variable) in S . Let D_i represent a dependency relationship between programming entities. Each D_i consists of tuples (v_i, v_j) . A tuple (v_i, v_j) represents a dependency from entity v_i to entity v_j . Let the set of directed edges E be the set of all directed edges in the graph. A formal set-theoretic definition of a call graph for a software system is defined in table 1.2.

Table 1.2: Graph-theoretic Definition of a Dependency Graph

VN	=	$\{v_1, v_2, \dots, v_k\}$
PN	=	$\{p_1, p_2, \dots, p_k\}$
V	=	$VN \cup PN$
D_1	=	$\{(v_i, v_j) \mid v_i, v_j \in V \text{ and } v_i \text{ calls } v_j\}$
D_2	=	$\{(v_i, v_j) \mid v_i, v_j \in V \text{ and } v_i \text{ defines } v_j\}$
D_3	=	$\{(v_i, v_j) \mid v_i, v_j \in V \text{ and } v_i \text{ modifies } v_j\}$
D_k	=	$\{(v_i, v_j) \mid v_i, v_j \in V \text{ and } v_i \text{ depends on } v_j\}$
E	=	$\cup_{i=1 \dots k} D_i$
$DG(S)$	=	$\{V, E\}$

A dependency graph is called a directed multigraph due to the multiple kinds of dependency edges that connect nodes in a dependency graph.

1.1.3 Graph Representations of Software Systems

There are two kinds of graph representations of programs – (i) flow graphs and (ii) data-flow graphs. Flow graphs model the control flow structure and dependencies in a software system [ASU86, FOW87, GJM91]. A flow graph consists of nodes and directed edges. A node (vertex) in a flow graph is a *basic block of statements and expressions*. A basic block as defined in [ASU86] is

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

A node B_1 is connected to another node B_2 by a directed edge if control can flow from B_1 to B_2 in some execution sequence. Flow of control can be transferred by the if-then, if-then-else, while loop, goto, function call statements. In [ASU86], algorithms for partitioning a program into basic blocks and constructing flow graphs from the partitions are found.

The nodes in a flow graph represent a block of sequential computation and the edges represent a transfer of flow of control. In essence, a flow graph abstracts multiple statements as a single basic block and models transfer of control (irrespective of precisely how the transfer was achieved in the software) as an edge between the blocks. If the basic blocks are restricted to be procedures or functions only, the flow graph is referred to as a *call graph*. The call graph of a program depicts the functional decomposition of the program, and captures the *calls/uses* relationship between functions and the dependencies between the functions. From the maintenance point of view, in addition to providing an abstract view of function decomposition, call graphs are also useful in determining the functions that will be affected by code modifications.

Ghezzi et al [GJM91] provide a different definition of control-flow graphs; in their definition, the nodes represent entry into and exit from a single statement (for example an if-then statement) and the edges between nodes represent the statement itself. The conventional view of control-flow graphs is that given in [ASU86] where nodes represent a set of statements and edges represent transfer of control-flow.

Besides the computational basic blocks, there are other entities (such as variables, data structures) present in a software system which are not represented in a flow graph since the nodes in a flow graph represent computation and not data storage. Data-flow graphs model the modification of data in a program and emphasize data flow over control flow. A data flow diagram (or data flow graphs) as defined by DeMarco [DeM78] is :

A Data Flow Diagram is a network representation of a system. The system may be automated, manual or mixed. The Data Flow Diagram portrays the system in terms of its component pieces, with all interfaces among the components indicated.

A data flow graph has five graphical symbols; a bubble represents a function, an arrow represents data flow, a data store is indicated as an open box and I/O boxes represent input/output operations that result in data initializations. An arrow between two function bubbles indicates data flow between the functions. Ghezzi et al [GJM91] give an overview of data flow diagrams.

Data flow diagrams describe the functions that access and modify the data in a system. The relationship connecting functions is the data that is exchanged between the functions. This graph provides information about the data structures in a program, the functions in a program and the relationship between functions and data structures. Such information is useful in determining the functions that are affected when data structures are modified.

Program dependence graphs or system dependence graphs [LC93, FOW87] capture more than one relationship (such as control flow and data flow relationships) between the nodes in the graph (see table 1.2 for a formal definition of dependence graphs). A node in a program dependence graph can be any programming construct such as declarations, assignment statements, control statements. Edges between the nodes represent different kinds of dependencies between the nodes. A program dependence graph is referred to as a multigraph since it has more than one kind of edge connecting the nodes. Program dependence graphs aid in understanding the system design, exposing dependencies between components in a system, aiding maintenance [LR92, LC93].

Yau and Tsai provide a graph-theoretic definition of a software component interconnection graph (CIG) in [YT87]. The CIG captures the interconnection behavior of the software components of a large-scale software system. The labeled nodes in the graph are abstract representations of software components and the labeled arcs represent the allowable inter-connection among subsystems. The nodes can represent a compilable unit of a procedural programming language, a module, a file, a procedure, a function, a data file or a command procedure. The interconnections permitted between the nodes are determined by the implementation language.

1.1.4 Understanding Software Systems

Understanding a software system is an activity that includes understanding the software system's overall structure, current design and architecture, behavior, documentation, maintenance records, implementation language, development paradigm, popular strategies and techniques peculiar to its implementation language and its

paradigm, dependency relationships, the logical, physical, static and dynamic aspects of its design as well as the design rationale.

Understanding software systems is aided by understanding the information contained in simple abstract representation models such as the call-graph and dependency graph models explained in sections 1.1.1, 1.1.2 and 1.1.3 . The construction of these models is a complex and time-consuming process. The aim of the models is to aid human understanding of the system itself and not to understand the construction of the model itself. The process of *reverse engineering* extracts information from an existing software system and constructs the abstract representation model [CC90]. The goals of reverse engineering techniques are to design abstract representation models of software systems that aid understanding and to automate the construction of the abstract representation.

The focus of this research is :

1. To determine and analyze the factors that contribute to the complexity of object-oriented systems.
2. To design an abstract representation model of object-oriented software systems that aids program understanding from the software maintenance point of view.
3. To design techniques which enable the automatic extraction of the model from a software system.

1.2 The Context

The program understanding problem is considered in the context of object-oriented software maintenance. In order to understand the problems faced by a

maintainer, we must understand the overall software development process and the activities a skilled software maintainer is expected to perform.

1.2.1 Role of a Maintainer in the Software Lifecycle

The object-oriented software life-cycle is comprised of four major phases (*i*) *analysis*, (*ii*) *design*, (*iii*) *implementation* (*iv*) *maintenance*. Each phase is performed by a team of software engineers and it is not unusual to find the same team participating in more than one phase. Each of the first three software development phases - analysis, design and implementation, results in some output, for example textual documents describing characteristics of either the problem or the solutions. Each output has significant implications for the software maintainer. Though a clear distinction between the phases is useful for understanding the software development process, in practice the boundaries between the phases are not precise.

Several object-oriented methodologies describe the object-oriented approach to software development. Rumbaugh's *Object Modeling Technique* [RBP+91], Booch's *Object-Oriented Analysis and Design* [Boo93], Jacobsen's *Use-Cases Approach* are representative methodologies. Each of these methodologies advocate the same object-oriented principles and techniques, but differ significantly in specifying the order in which the techniques must be applied. Describing each object-oriented methodology is beyond the scope of this work; instead we provide a brief overview of each phase and the significance of the output of each development phase on the maintenance phase.

1.2.1.1 Analysis

The analysis phase consists of understanding the nature of the problem, decomposing it into its subproblems to reduce the complexity of the problem, understanding, representing and partially solving each subproblem and finally composing the solutions to give an integrated solution to the entire problem [Boo93]. The central concern of object-oriented analysis is to discover the *components* of the problem; component discovery results in the identification of objects in the problem domain. The object identification process is referred to as object-oriented decomposition where data abstractions are considered to be more important than procedural abstraction.

Object-oriented analysis results in specification documents which contain information pertaining to *what* the system is expected to do. These documents serve to explain and specify the expected behavior of the system. Specification documents provide an overview of the purpose of the software and give some insight into the problem being solved and the issues that were identified and addressed in the analysis phase. Object-oriented analysis often results in a *object model diagram* [Boo93, RBP⁺91] which models the problem in terms of interacting objects. In the object-oriented approach, behavior of the system is described in terms of collaborations among the objects.

1.2.1.2 Design

The object model produced as a result of object-oriented analysis is scrutinized from the perspective of object-oriented design. Designers accomplish the difficult task of providing a solution to a problem with the added constraints of endowing

the solution with desirable properties like low-coupling, high cohesion, information-hiding, flexibility, extensibility, reusability, readability, understandability, maintainability, efficiency, and performance [GJM91]. In essence, it is not sufficient for the design team to simply provide a working solution; it is their responsibility to investigate as many possible alternatives as they can devise and then provide a stable design.

The design team commits itself to designing a solution with some subset of the desirable properties and this subset constitutes design goals. The final design is the outcome of a process of elimination in which each alternative solution is subjected to a thorough and critical examination from the perspective of each desirable property. Each solution is either accepted or rejected. This design-phase commitment to obtaining a “good” or “elegant” design may result in a larger and more complicated object model than that produced by the analysis team. One of the goals of a good design is to aid software maintenance by ensuring that the software possesses certain qualities.

Design documents can be a combination of textual and graphic descriptions of the system. These documents which explain *how* the system achieves its expected behavior and functionality, are useful in understanding the design architecture of the system. For example, class diagram models, state transition models, interaction models, state-transition models, data-flow models, dynamic models and object models are the typical outcome of the analysis and design phases of the object-oriented software development methodologies.

1.2.1.3 Implementation

Implementation of the design involves transforming or realizing the design as actual executable code. This transformation addresses the question of *how* to preserve the design and its associated properties in the implementation. Some of the design-level concepts may be directly supported by the language of implementation and some may not. Lack of direct implementation support of design-level concepts results in strategies that map design-level concepts to combinations of programming language concepts. We shall call design-preserving strategies *programming techniques*. Programming techniques may result from actually mapping design-level concepts to implementation or from ensuring some desirable property of the design.

1.2.1.4 Maintenance

The job of a maintainer is to maintain software and it is the software or code itself that is the maintainer's prime concern. Maintaining software is a generic term that encompasses a wide variety of activities such as adding functionality to the software system, adapting the system to a new environment, correcting defective code, or modifying the system. Each activity implies performing modifications on the software system. Code modifications can be performed after precisely determining and locating the actual code fragments that must be modified. The maintenance process model [GL91]² (see figure 1.2) depicts how a maintenance request is performed. Initially, a maintenance request is analysed and classified as a request that requires existing code to be modified or a request that requires new code to be

²Reproduced from *Using Program Slicing in Software Maintenance* [GL91], IEEE Transactions on Software Engineering, Aug 1991.

added or a request that requires existing code to be deleted. The goals of a good maintenance strategy are to – (i) minimize the introduction of defects as a result of code modifications (ii) minimize the time, effort and manpower expended on maintenance (iii) reduce the cost of maintaining software (iv) maximize customer satisfaction. The change is then performed after pending requests and their priorities have been examined and the priority of the request is ascertained. Changing software is comprised of two activities – (i) understanding the existing software system and (ii) incorporating changes in the software to accommodate the maintenance request, subject to the maintenance constraints stated above. The revalidation phase consists of two steps – (i) testing and validating the changes before integrating them into the system (ii) testing and validating the integrated system. The revalidation phase consists of ensuring that the system meets its original and new objectives as well as determining if code modifications adversely affected the original functionalities or performance of the system.

Understanding software systems is necessary in order to perform careful, structure-preserving maintenance. A maintainer must be aware of the effect of a modification on the rest of the software system. The components affected by a modification can be determined from the dependency graph of the software system. Determining the code to be modified is possible if the logical structure is known to the maintainer, whereas the physical structure of the system enables the actual location of the code fragment. Understanding the various dependencies that exist in the code as well as the overall structure of the code is crucial to the maintenance activity. Among the constraints any structure-preserving maintenance activity must fulfill are:

1. Software maintenance must be performed in such a manner that the structure and properties of the system prior to maintenance are retained in the

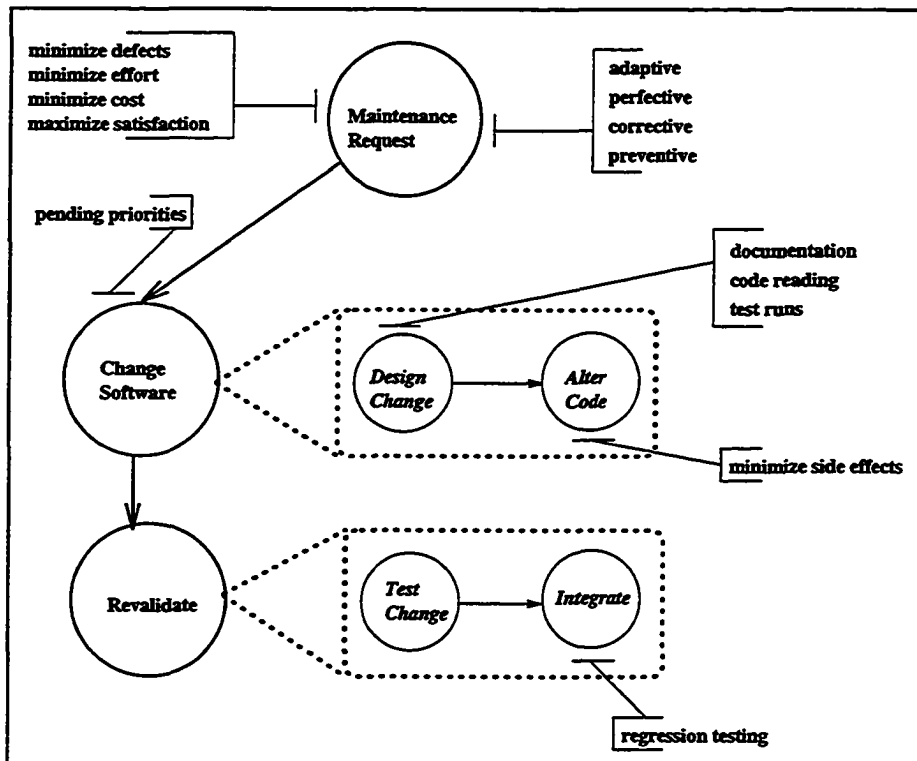


Figure 1.2: A Software Maintenance Process Model

modified code to the extent possible. Retaining structure is only possible if the structure is known to the maintainer; once the structure is known, modifications must be performed in a manner that preserves the structure of the system. Determining how to perform modifications in a structure-preserving manner requires knowledge and experience in programming and programming techniques.

2. Maintenance must ensure that the modifications made to code are minimized and localized. This constraint forces the maintainer to closely examine and understand the various kinds of dependencies that exist within the software system. Maintainers spend a considerable amount of time devising techniques to perform maintenance in a way that localizes and minimizes the effect of maintenance.

1.3 Modeling Object-Oriented Software Systems

This research is aimed at providing software maintainers with a conceptual model of the architecture (or the structure) of a software system in order to aid structure-preserving software maintenance. The *low-level software architecture model (LLSA)* defined in chapter 4 is an abstract representation of object-oriented software systems. The LLSA representation of an object-oriented software system consists of textual descriptions of software components, the interfaces of each component and the interactions between components. The *low-level design patterns (LLDP) model* represents a collection of the textual description of some object-oriented programming techniques. The LLDP representation is at a higher level of abstraction

than the LLSA model. Both models of representation aid in software maintenance and an overview of each is provided in section 1.3.1.

1.3.1 Overview of LLSA and LLDP

Two abstract representations of software structure are presented – (i) low-level software architecture representation and (ii) low-level design patterns (see fig 1.3). At the bottom of the figure the software system is viewed as source code. Source code consists of multiple lines of code possibly distributed over separate files. Each file can be viewed as a module. In a well-structured software system, each file specifies the library support it requires and the external functions that it requires. This information establishes a physical (or a compilation) dependency between the modules. Physical dependencies do model logical relationships and dependencies to some extent; however, implementation limitations and restrictions make it difficult for an implementor to model the software organization to totally reflect logical dependencies.

We define an abstract view of source code (the LLSA) that captures physical and logical dependencies between software components. In the LLSA model, a software component is defined to be a specific set of programming constructs. The rationale for selecting certain constructs over others is provided in sections 1.3.2 and 4.3. The components of a software architecture represent logical concepts in the design of the system. Each software component has an *interface*. The interface describes the static and dynamic behavior of the component. Software components interact with each other and these interactions set up logical relationships and dependencies between the components. The LLSA model represents relationships between components by including information pertaining to the relationships in the

description of software components. A detailed explanation of the design of the model and its uses is provided in chapter 4. The LLSA model provides a structured view of a software system capturing important components and relationships present in the software system.

At the next higher level of abstraction, we define a collection of low-level design patterns. Low-level design patterns describe *programming techniques* that occur repeatedly in a software system. LLDPs are constructed over the components, interfaces and relationships defined in the LLSA model. An LLDP has a name and a distinctive structure. The name indicates the programming technique being described and the structure of an LLDP enables identification and recognition of the technique in the software system. The structure of an LLDP is comprised of components and the logical relationships connecting the components. LLDPs explain the reason for the connections between components. LLDPs interact among themselves to create more complex relationships. LLDPs are therefore useful in software maintenance because they introduce the maintainer to existing programming techniques and enable the maintainer to check if a modification can either reuse the technique or if a modification disrupts a structure that destroys the technique used by the original developers. The collection of LLDPs is shown as an open system because it is a collection that can be extended by including more programming techniques. It represents a higher level of abstraction in which the information content of the patterns in the model corresponds to programming experience and expertise. LLDPs are explained in detail in chapter 5.

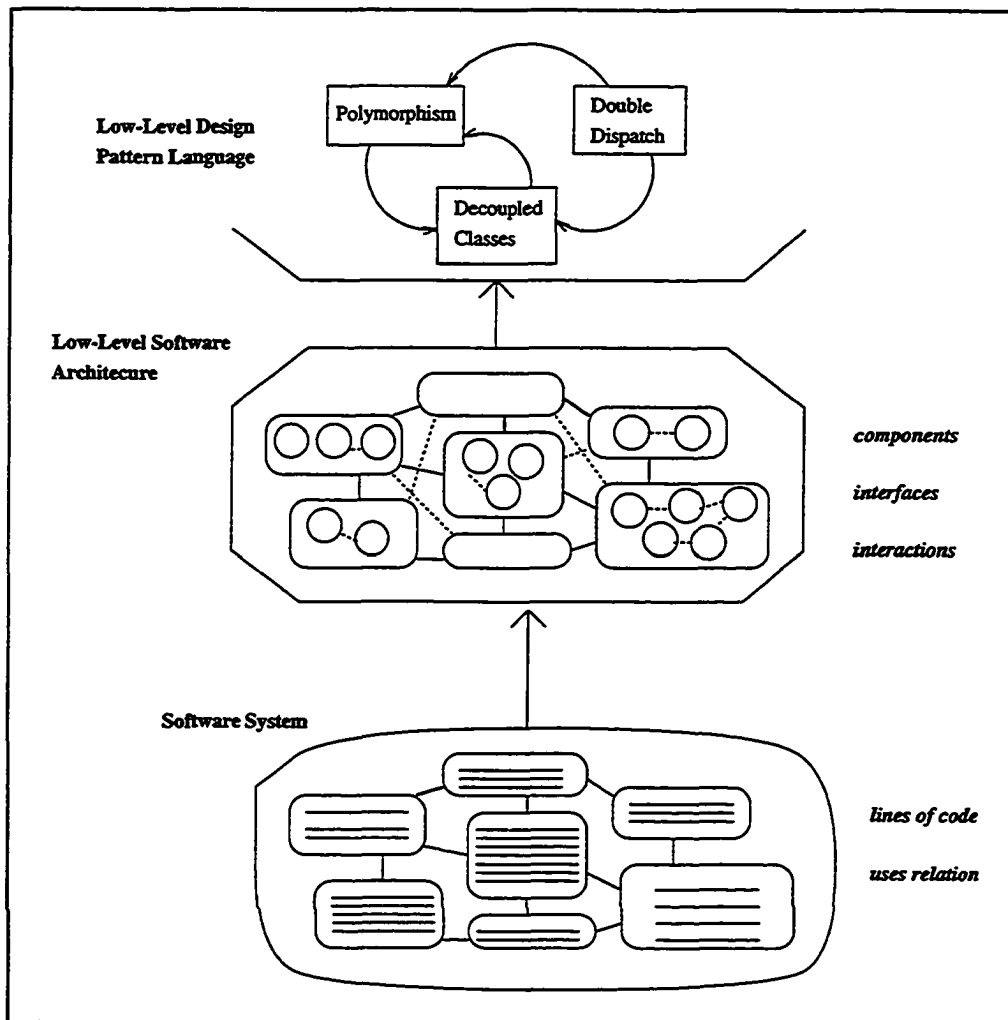


Figure 1.3: Views of a Software System

1.3.2 Design Rationale of the Low-Level Software Architecture Model

Investigation of object-oriented analysis and design revealed that that object-orientation places emphasis on relationships between components.

In a software system, all the design concepts (class, object, state, module, interaction) and relationships that are described in design documents co-exist simultaneously and are specified in terms of the constructs of the language of implementation. The simultaneous presence of multiple relationships is one of the reasons for the complex nature of object-oriented software systems. Determining the dependencies between code fragments is contingent on being able to identify the relationships between the code fragments and the dependencies that come into being as a consequence of the relationships. Code fragments may be related in more than one way and consequently there may be more than one kind of dependency between them. Moreover, different relationships may interact to create some complicated dependencies. From the maintenance point-of-view, it is the dependencies between code fragments that are of interest and not the relations themselves; ie. the design documents serve to specify how the components are related; they do not specify the ensuing dependencies and the *consequence* of a relation on the static or dynamic structure of the system. From the maintenance point of view, code fragments, or *components*, and the nature of dependencies between them are of central interest. A model that captures dependencies in a software system must therefore concentrate on determining the information content in the representation of components.

1.3.3 Using LLSA and LLDPs for Maintenance of Object-Oriented Systems

The software maintenance process model described earlier can now be modified as shown in figure 1.4. In this model, the maintainer uses the automatically extracted LLSA description of a software system to understand the low-level dependencies and relationships in the system. The LLDP and LLSA descriptions, in addition to the existing system documentation and source code, can be used to understand the system and locate the code that must be modified. The design of the change to be performed is done in the context of the existing system. The modifications may result in structural changes to the code. With the support of LLSA and LLDP, the maintainer can perform structural changes, obtain the LLSA of the changed system and compare it with the original LLSA to check for the impact the modifications may have on the structure of the system. The LLDPs aid in maintenance by documenting existing techniques which a maintainer may be unaware of.

1.4 Pulse

The information contained in the LLSA description can be gathered manually by the maintainer or it can be extracted automatically from the source code. One of the goals of reverse engineering is to automatically extract the abstract representation model from source code. Automatic extraction requires the development of non-trivial code analysis algorithms that are capable of extracting the information represented in the abstract model. A prototype software tool, *pulse*, was developed to determine the feasibility of automatically extracting the LLSA model. Pulse

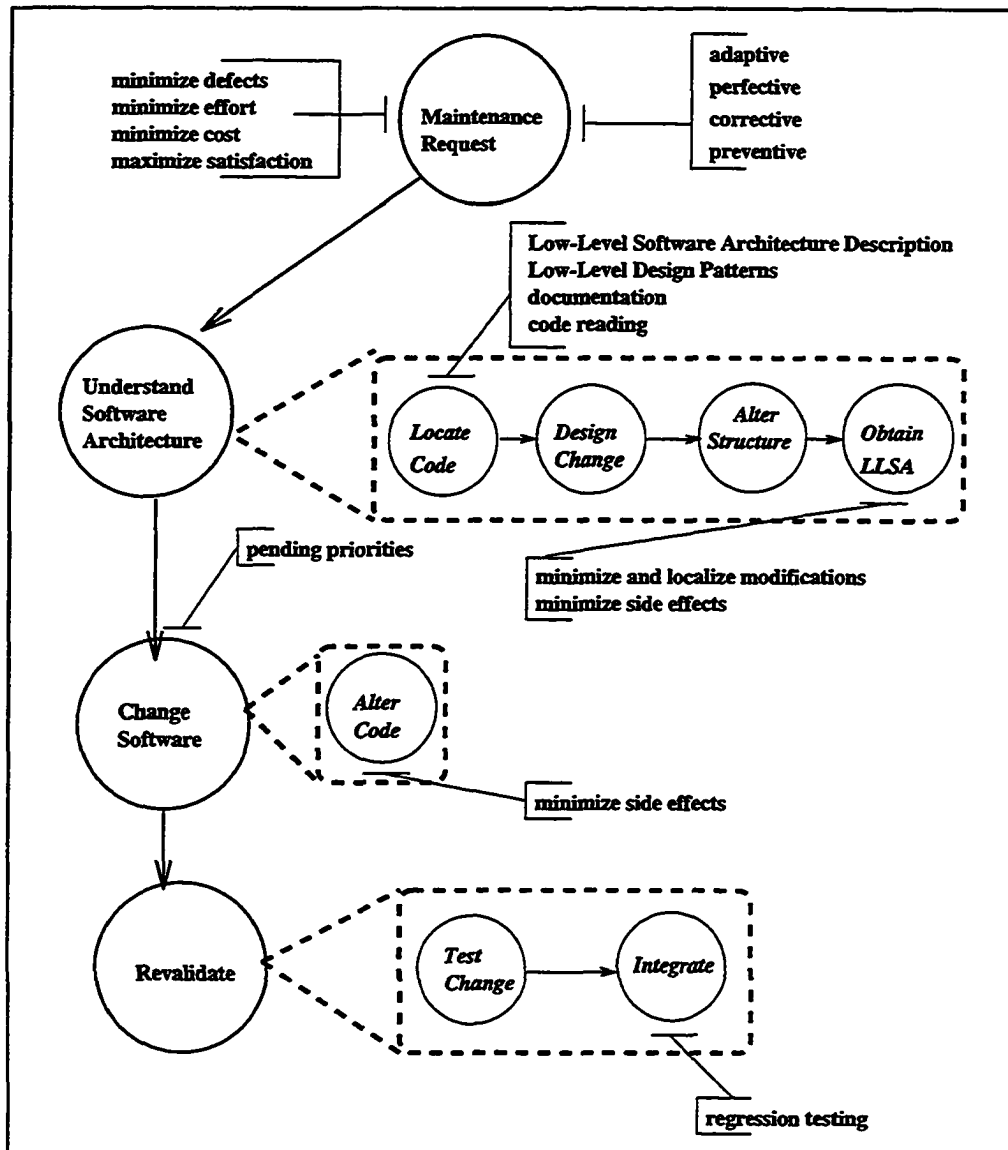


Figure 1.4: A Software Maintenance Process Model Using LLSA and LLDP

uses reverse engineering techniques in its design and implementation to identify software components and to construct the interfaces and interactions of each component. There are two phases in pulse – (i) phase I is the synthesis phase where information is collected extracted from the source code and (ii) phase II denotes the analysis phase wherein algorithms to compute the interfaces and interactions of a component are used. In addition, phase II also uses algorithms to recognize LLDPs in the system. The architecture, implementation issues and code analysis algorithms of pulse are discussed in chapter 7.

1.5 Overall Organization

Chapter 1 provided an overview of the problem, the situation in which the problem manifests itself (ie. software maintenance) and the abstract representation models LLSA and LLDPs. Justification for investigating the problem was provided by explaining the difficulty in understanding and maintaining software systems in sections 1.1.1, 1.1.2, 1.1.4 and 1.2.1. The motivating factors and background for this research are discussed in chapter 2. Chapter 3 provides an overview of related research in the areas of software maintenance in general, reverse engineering projects and approaches, issues in understanding object-oriented software, restructuring techniques and maintenance aids for object-oriented software. Chapter 4 describes the theoretical framework underlying the low-level software architecture model and defines the low-level software architecture of C++ software systems. Chapter 5 introduces and describes pattern languages, design patterns and low-level design patterns. The usefulness of both, LLSA and LLDP is explained in chapter 6. Chapter 7 explains the overall architecture of the prototype *pulse* which

reverse engineers the LLSA and recognizes instances of LLDPs in the source code. Chapter 8 lists the contributions of this research and future work for this research.

Chapter 2

Motivation and Background

The motivating factors for this work can be briefly enumerated as follows :

1. The time and effort expended by the software industry and by maintainers on software maintenance.
2. The complexities inherent in the process of understanding object-oriented software systems.
3. The unreliable nature of documentation
4. The lack of sophisticated tools and approaches that aid software maintenance.

Sections 2.1, 2.2, 2.3 and 2.4 elaborate on each of the motivating factors.

2.1 Motivation

The job of a software developer in the software industry now consists of developing software components that can be combined with other components (as opposed to developing complete programs from scratch) and more commonly of maintaining existing software. The task of a maintainer is to understand the overall structure of a software system, referred to as software architecture, and the programming logic that went into the development of the system. Studies indicate that programmers spend more than half of their time on maintenance [Sam90, GS89, GW90].

Investigations conducted by concerned software organizations revealed some interesting statistics about the role of maintenance in a software organization. The most important results were that maintenance activities accounted for 67 % of the total software life-cycle phases and that organizations spent upto 60 % of their budget on software maintenance [GJM91, Par86]. Parikh [Par86] reports the following statistics on the state of maintenance :

1. Most companies spend 50% of their budgets on maintenance.
2. Most programmers spend 50% (and in some cases 80%) of their time on maintenance.
3. The worldwide annual expenditure on maintenance is \$30 billion (Martin and McClure, 1983, p. 15).
4. This is debatable, but it is even stated that in a software life cycle, new development is only 33%, the remaining 67% is maintenance.

It is also interesting to note that the most difficult aspect of software maintenance is the phase that involves understanding the original developers' intent. Maintainers are reported to spend 50% of their time on the comprehension activity [Par86, Sam90]. The statistics and the task of grasping the overall structure of a system are compelling reasons to investigate and analyze the complexities inherent in program understanding and software maintenance in general. The growing popularity of object-orientation and C++ are the reasons we focussed on determining the complex nature of C++ systems. However, the representation models LLSA and LLDPs are applicable for object-oriented software systems developed in other object-oriented programming languages such as Smalltalk [Gol83] or Eiffel [Mey88].

2.2 Issues in Understanding Object-Oriented Systems

The object-oriented principles of polymorphism, inheritance and dynamic binding are the most common object-oriented features that make object-oriented systems difficult to understand [LMR91, WMH93, WH91, CvM93]. These principles are explained in sections 2.5.7.4, 2.5.7.3 respectively. Polymorphism, inheritance and dynamic binding are powerful features that enable a software system to be flexible, reusable, extensible and maintainable. (each software system feature is defined in section 2.5.1). However, these principles do not make the system understandable.

Wilde and Huitt [WMH93] noted that the dispersion of logic into small program fragments buried in class hierarchies of object-oriented systems is a hinderance to program understandability. The original intent of the programmer must be reconstructed by identifying and examining small program fragments distributed over classes. Furthermore, the object-oriented principles of polymorphism, inheritance and dynamic binding encourage delocalized and decoupled logic. In the presence of such mechanisms, a program understander must painstakingly trace long sequences of message requests to locate the source code that actually implements a specific functionality. Inheritance complicates the determination of the calling and dataflow dependencies in a class hierarchy. Dynamic binding and polymorphism make it practically impossible to precisely determine the actual source code being executed. Even the determination of the set of methods that may possibly be executed is difficult. Knowledge of inheritance rules is required to determine the possible set of methods. Lejter, Meyers and Reiss [LMR91] document inheritance and dynamic binding as two factors that complicate the understanding of object-oriented systems. Crocker and Mayrhauser [CvM93] include the dynamic creation/deletion of objects

and the overloading of operation names as factors in object-oriented programming languages that affect maintenance, in addition to inheritance and polymorphism.

Kung et al [KGH⁺94] address the issue of complex relationships in the class structure of object-oriented systems. Understanding the functionality of individual classes and member functions is not difficult, particularly since the code contained in them is small. Understanding the *combined functionality* of classes and member functions is very difficult. According to Kung et al [KGH⁺94] the factors that complicate maintenance of object-oriented software are (i) understanding the combined functionality of member functions (ii) understanding complex class relationships and (iii) understanding data dependencies, control dependencies and state-behavior dependencies.

2.3 Inconsistent Documentation

Motivation for reverse engineering representations from the code itself instead of relying on existing documents can be attributed to inconsistent documentation. Documentation of software design and analysis may be produced manually or automatically. Manual documents are prepared by teams consisting of technical writers and software developers who often document source code (or the design of a software system) either after the software has been developed or in parallel with the design and development of the software [And86]. Describing the design of a software system and the functionality and purpose of each software component is a daunting task. The task of documenting “live” software or software that is in the process of being developed is orders of magnitude more difficult than documenting existing software because the developers introduce changes in code very rapidly and documenting

each change is not feasible. Automatic documentation is performed by CASE tools that extract information from software and produce textual and sometimes graphical information about the software. The limitation of automatic documentation is that it does not always provide sufficient information about the software; what is more worthy of concern is that the documenting CASE tool may provide inaccurate or unreliable information about the system. Maintainability and understandability of large-scale software systems is hindered by the inaccurate record of the overall system structure and the interactions between software components of the system [YT87].

Inconsistencies in design documentation arise from the transformation of design to implementation. Two key characteristics of the transformation process from design to implementation are:

1. A refinement of the high-level design to a more realizable prototype. This refinement exposes implementation subtleties, unrealistic designer assumptions and subproblems which may be quickly designed and implemented by the implementor directly. This is a severe problem because the original design does not record these changes, leading to inconsistent documentation.
2. A loss of organizational information. Few programming languages concern themselves with organization because it is not directly useful in programming. Component organization and determining component interrelationships is an important part of the design activity. This information may be well documented in design documents, but *how* the organization and interrelationships are captured in the implementation depends on programming style, programming conventions and source code documentation. Style, conventions and

documentation are features that are not enforced by a programming language and differ from programmer to programmer and are subtly responsible for retaining design information. Part of the process of understanding code consists of being acquainted with the styles and conventions adopted by the original developer.

2.4 Maintenance Aids

The complex nature of software systems necessitates multiple views of the system to enable understanding of the software system. Dependency graphs, call graphs are representations of software that have aided maintenance in the past. These representations were designed for software systems developed using the structured paradigm. The call graph model is based on functions and the connections between functions represent the transfer of control between functions. Object oriented software has more than one component (ie. class and object in addition to function) and more than one mechanism for transferring control between components (ie. sending messages to objects and function calls). The call graph model is not very useful since it does not capture many essential aspects of object-oriented software.

Schneidewind [Sch87] provides a list of software CASE tools that are required for maintenance. These tools describe the structure of a software system. He elaborates on the different aspects of the term software structure and the need for understanding each aspect. The tools include those that represent and manage the following aspects of a software system :

1. Procedural structure, control structure, data organization, data-flow structure and input/output structure of the software system
2. Aliases of data; i.e. the different names associated with the same data
3. Multiple versions of a software system
4. Dynamic behavior of the system
5. Test cases for validation purposes
6. Low-level symbolic execution information useful for debugging

Crocker and Mayrhauser [CvM93] advocate the use of CASE strategies for the maintenance of object-oriented software. They provide a list of tools that will aid in the maintenance of software. The toolchest that they advocate consists of tools that are classified as *framework tools*, *mundane tools*, *knowledge tools* and *change tools*. The framework tools are intended to provide representational support and their function is to interface between the other kinds of tools and the software system. The intention of the mundane tools is to gather information from the software system but not to analyze the information. These tools include control-flow generators, structure chart generators, cross-reference generators, test driver generators and test coverage generators. The knowledge tools are intended to aid in understanding the object-oriented software. These tools include aids to understanding inheritance hierarchies, aiding in the creation of new abstract classes and the modification of the interfaces of classes, code browsers and code slicers. The last group of tools, change tools, are intended to aid in the actual modification of code. Automating code modification includes tools that perform consistency checks, inheritance generator in the case of programming languages that do support inheritance, test

case selector, metrics generator and ripple effect analyzer. Each tool is intended to provide a restricted view that aids in understanding some of the aspects of object-oriented software. The toolset proposed in [CvM93] was an ambitious project and therefore the authors provided an overview of what each tool was expected to do. However, the toolset is yet to be developed and to the best of our knowledge, the toolset does not exist. The need for tools and aids for maintenance activities is evident and the absence of tools which meet the needs of a maintainer adequately is one of the motivating factors for this research.

2.5 Background

This research includes concepts in software engineering, forward and reverse engineering processes, abstract representation models, principles of the object-oriented paradigm, object-oriented design and object-oriented programming languages. These concepts are developed in the following sections.

2.5.1 Software Engineering

Software engineering is a field of computer science whose concerns are the development and management of large complex software systems. In the area of software development, the ongoing effort is to discover the principles and laws that will make automatic production of reliable software a reality. The management of complex software systems deals with the problems of software maintenance and software evolution. The focus of software management is to reduce software complexity in order to facilitate software evolution. Software engineering could not be more concisely explained than in [GJM91]

Software engineering is a field of computer science that deals with the building of software systems which are so large or so complex that they are built by a team or teams of engineers. Usually, these software systems exist in multiple versions and are used for many years. During their lifetime, they undergo many changes—to fix defects, to enhance existing features, to add new features, to remove old features, or to be adapted to run in a new environment.

The goal of software engineering is to develop reliable software. The properties present in a software product determine its quality, design and longevity. A well-designed, high-quality, well-maintained software product is more likely to be accepted and used than one that is not. Terms used to describe the desirable properties of a software system are discussed below.

- **Coupling**

Coupling is a term used to refer to the degree of connectivity between modules (or components) of a software system. Connectivity between components is established by the interactions that occur between the components. Interactions between components cause dependencies between the components. High degrees of interaction between components complicate the overall structure of a software system and make it less understandable [GJM91, Pre92]. Low coupling is a desirable property in a software system.

- **Cohesion**

Cohesion denotes the interactions that occur within a module or component. A high degree of cohesion within a component is indicative of a well-designed component because every subcomponent has a clearly defined purpose and is required by the other subcomponents. A cohesive component reflects a the grouping of logically related subcomponents that interact to achieve the

overall purpose of the component [GJM91, Pre92]. High cohesion is a desirable property in a software system.

- **Flexibility**

The flexibility of a software system is a measure of the effort (in terms of time, cost and manpower) required to change an operational software system [Pre92]. A highly flexible system is one in which changes can be introduced with minimum effort and high flexibility is a desirable property for a software system to possess.

- **Reusability**

The reusability of a software system is the measure of the degree to which parts of a software system can be reused in the development of other software systems [Pre92, GJM91]. The principles of the object-oriented paradigm are particularly suited for the development of reusable software [Boo93, CAB⁺94].

- **Understandability**

The understandability of a software system is the ease with which the behavior and the structure of the system can be analysed and predicted [GJM91, Pre92, Boo93]. Properties closely related to understandability are – readability and maintainability. A readable program is one which is written to aid a human reader's understanding of the program. Typically, readable programs follow established formats and styles of programming that include meaningful names, good documenting strategies, logical decomposition and flow of control, reduced code duplication and simple, elegant code. A program that is readable and understandable program is more maintainable [GJM91, Pre92].

The significance of readability and understandability on maintainability is succinctly captured by Einbu in [Ein89] :

... most crucial problem of software engineering: how to make a program understandable. This problem is best approached from an architectural point of view, rather than from a program-engineering point of view. How should a program be composed so that its meaning becomes apparent from a reading of the program listing ?

The term *forward engineering* is used to refer to the software development aspect of software engineering whereas the concerns of *reverse engineering*, *reengineering* and *restructuring* lie in the areas of software maintenance and management.

2.5.2 Forward Engineering

Forward engineering is responsible for the traditional software development process of analysis-design-implementation where the term forward refers to the direction of the process—from requirements to implementation. Various software development tools, automated program generators, program generators generators, computer-aided software engineering (CASE) tools fall under the umbrella of forward engineering. *Program translators* [ASU86] are especially worthy of mention here, for their historical value and the impact they had on programming and software development. A translator is a complex software package whose purpose it to provide a semantics-preserving translation of a program written in a high-level language into a low-level language program. Forward engineering as defined in [CC90] is

Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system

2.5.3 Reverse Engineering

As the term suggests, reverse engineering is in the opposite direction - from implementation to requirements. Pressman [Pre92] attributes the origin of the term to the disassembling of hardware products by competitors to understand the design and properties of the hardware product. The reverse engineering process in the software field implies the analysis of source code in order to represent code at a higher and more understandable level of abstraction [Pre92]. Recovering the design of a system is also referred to as the reverse engineering process [Pre92, Big89]. In summary, reverse engineering can be described as the process of extracting and assimilating information from code in order to conclude a general property of the code. In [CC90] reverse engineering is defined as

Reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction

The focus of reverse engineering is to aid program understanding. In order to meet this single goal, reverse engineering has the following objectives [CC90].

1. To develop methods that reduce software complexity.
2. To provide multiple views of software system.
3. To recover information about the design of the system.
4. To expose unwanted system properties.
5. To synthesize higher level abstractions and to facilitate reuse.

Reverse engineering approaches discussed in chapter 3 meet the objectives listed above. Reverse engineering or extracting the design from code is an invaluable aid to maintenance because it provides important, maintenance-related information. The many advantages of reverse engineering as given in [GLG92] are:

1. identifying, documenting and classifying reusable software components.
2. salvaging the lost knowledge implemented in the code and not documented.
3. recovering design information from code and using it to implement a new version of the system.
4. generating an up-to-date documentation of the system.
5. checking consistency between the design and code and verifying that both conform to standards.
6. validating the system by detecting unplanned dynamic sequences due to errors in the initial design and/or to side effects of maintenance operations.

The term reverse engineering is typically used to refer to processes and techniques that usually extract information primarily from source code and represent the information in an abstract representation model that captures design information about the software system. Techniques that modify or change the structure of a system are classified as restructuring or reengineering techniques.

2.5.4 Restructuring

The goals of the restructuring process are different from reverse engineering. The restructuring process is the process whereby the structure of a system is changed in order to simplify the system and reorganize the components and interconnections in the components without altering the behavior of the system. Restructuring sometimes implies the replacement of a complicated subsystem by a simpler, more

abstract, cohesive component [Opd92, Cas92]. Identifying such structural replacements that do not alter the semantics or the behavior of the system in any way is one of the non-trivial tasks of the restructuring process. Restructuring deals with analyzing the structure of code and reorganizing code, without affecting the functionality of the system, in order to reduce complexity or to enhance understandability.

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics) [CC90].

2.5.5 Reengineering

Reengineering incorporates reverse engineering and restructuring. This term is employed when a system is studied and analyzed for the explicit purpose of rebuilding it anew. Reengineering involves redesigning and reimplementation. In order to do this, the original design and structure of the code is examined to analyze its advantages and drawbacks.

Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [CC90].

2.5.6 Representations of Program Structure

A large software system is comprised of interrelated subsystems. The structure of a software system is denoted by the organization of code. The organization or structure of a system can therefore be explained in terms of the subsystems and the interrelationships between them. Software structure can be represented at different levels of abstraction. The *degree of abstraction* in a model refers to the amount of

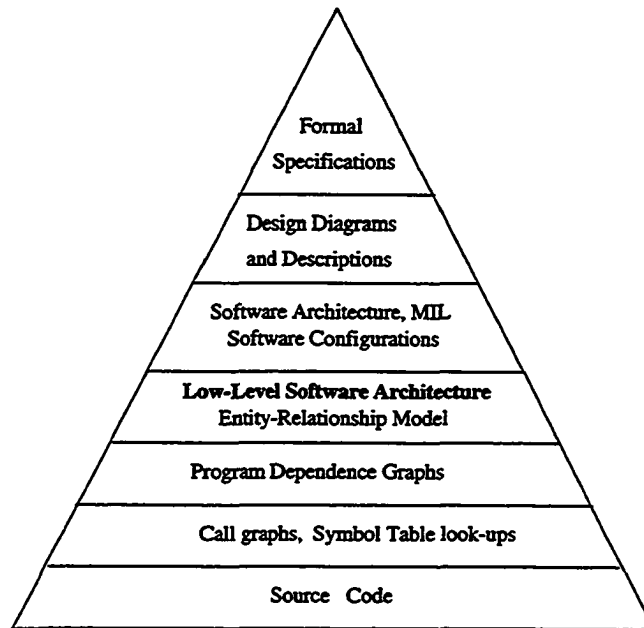


Figure 2.1: Abstract Representations of Software System Structure

detail present in the model [RC93]. Based on the degrees of abstraction, we classify software system structure as shown in figure 2.1. Different definitions of the term subsystem gives rise to different representations of structure. An encapsulated piece of code or a file can be viewed as a subsystem. A structure representation defined in terms of code fragment subsystems is more detailed and less abstract than a representation defined in terms of files. The level of abstraction required in the representation is determined by precisely analyzing how the representation will be used.

The most complete and detailed (and the least abstract) representation of a software system is source code itself. Call graph models, which represent the flow-of-control between procedures and variable-use (or symbol table look-up) models,

which provide symbol cross-referencing facilities are at the next higher level of abstraction [RC93].

A flow graph [ASU86, FOW87] abstracts multiple statements as a single basic block and models transfer of control (irrespective of precisely how the transfer was achieved in the software) as an edge between the blocks. Program dependence graphs [LC93, FOW87] represent programming constructs such as declarations, assignment statements, control statements as nodes. Edges represent different kinds of dependencies between the nodes. The information content of program dependence graphs and flow-graphs is more comprehensive than that of call-graphs and variable-use models. They are therefore placed higher in the abstraction hierarchy.

Representation models that attempt to capture overall structure and dependencies amongst higher-level programming concepts are at a higher level of abstraction than program dependence graphs. The LLSA model presented here falls in this category of representation models.

Software architecture [GS93] is a design-level description of the overall structure of a software system. At this level of abstraction, the interactions permitted in the architecture are determined by the architectural style of the system instead of the programming language used for implementation of the system. For representation models at lower levels of abstraction, the interactions are dictated by the implementation language. Software configurations [NS87] of systems also describe the arrangement of software components and their interdependencies. Module Interconnection Languages (MIL) are used to specify these configurations.

Design diagrams and descriptions represent the rationale, logic and design properties of the software system [Boo93]. Formal specifications represent the behavior, functionality and constraints of the system [Pre92].

2.5.7 Object-Oriented Concepts

The object-oriented paradigm attempts to model a problem in terms of an object model which is comprised of objects and their interactions. The construction of the object model is based on certain object-oriented principles in order to attain desirable properties. Introduction to object-oriented concepts are concisely and lucidly explained in [Pre92, Wil93]. More detailed explanations of object-oriented analysis and design, the nature of object-oriented principles, the benefits and consequences of using object-oriented techniques for software development are elaborated in [RBP⁺91, Boo93, CAB⁺94].

The central concepts of object-orientation are the notions of *object*, *class* and *messages*. An object may be an entity, an abstract concept, or a physical, tangible thing. Typically, an object has a representation associated with it, and a set of properties. Envisaging real-world entities as objects is very straightforward – a stone, a circle, a book are all examples of the object concept. Some other non-intuitive examples of objects are processes, tasks, agents fulfilling a designated role in a large organization. These are more difficult to envisage as objects because they represent actions and roles in the real world. Objects have an identity associated with them which enables a person to distinguish between similar objects and refer to each object separately. Associating an identity with an object corresponds to the notion of labeling items in order to make the activities of referring and accessing the items easier. For example, books in a library are each labelled with a unique series of letters and digits. Objects also respond to stimuli (external or internal) and this response may manifest itself in the form of changes in the attributes of the object. For example, a stone when subjected to heat expands and the volume of the stone increases. This change in attributes in response to stimuli is referred

as *behavior*. The collection of values of each attribute of an object at any given point in time is referred to as the *state* of an object. For example, the state of an expanded stone is different from the state of a stone at room temperature because the values associated with the volume and temperature attributes are different for the two stones. The definition of object adopted in this work is the one given in [Boo93] ([CAB+94] provides a similar definition):

Object : Something you can do things to. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable.

A class represents a set of similar objects [Boo93, CAB+94]. Objects may be similar in structure (representation), or behavior (properties). The class serves as a template for the common specification of the attributes of similar objects. Wilkie [Wil93] explains a class as a mechanism to describe the attributes and interface of an object. The definition of a class given in [CAB+94] is:

Class : A set of objects that share a common structure and a common behavior. A class is an abstraction, which represents the idea or general notion of a set of similar objects.

Objects are capable of sending and receiving *messages*. A message is a means of communication between objects [Pre92, Wil93, CAB+94, Boo93, RBP+91]. The request is made by a *client object* and a *server object* complies with the request by executing the requested operation. The description of the message passing mechanism as given in [Cox86] is:

An object is requested to perform one of its operations by sending it a message telling the object what to do. The receiver [object] responds to the message by first choosing the operation that implements the message name, executing this operation, and then returning control to the caller.

The four fundamental principles of the object-oriented paradigm abstraction, encapsulation, inheritance and polymorphism are described next.

2.5.7.1 Abstraction

Abstraction is the mechanism whereby the fundamental aspects of a concept are concisely expressed. Abstraction brings the core attributes to the forefront, relegating unnecessary detail to the background, emphasizing the basis and the rationale for the existence of the concept. The decision of what comprises important attributes and what comprises unimportant detail depends on what abstraction is required from the concept. Thus for example, whereas a moving car represents a means of transport to a computer scientist, to a physicist it represents an example of the application of the laws of motion and various interacting forces. The focus of the object-oriented paradigm is data abstraction. Abstraction in the object-oriented sense manifests itself in the form of *abstract classes*. An abstraction has to satisfy the conflicting constraints of characterizing representative (and thus typical) traits and distinctive traits. This definition of abstraction is concisely expressed in [Boo93].

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

2.5.7.2 Encapsulation

Encapsulation is a mechanism to group related attributes of a concept into a single unit. This association of concepts and their attributes is an important step towards organized and structured systems. Encapsulation inhibits the improper usage of the attributes and encourages decomposability and the separation of concerns [GJM91]. Since encapsulation hides the internal details of a concept, it is often referred to as *information hiding*. Class represents an encapsulated unit in

the object-oriented paradigm. [PJ92] discusses the effect of encapsulation on software structure and maintenance. The definition of encapsulation used in this work is adapted from [Boo93].

Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics.

2.5.7.3 Inheritance

Inheritance is the outstanding contribution of the object-oriented paradigm to software development and is the one principle which is not supported by the *structured programming paradigm*. Inheritance is best understood in the genealogical sense and the terminology used in the literature bears testimony to it. Inheritance is mechanism that permits the composition of classes. A *child class* inherits the structure and behavior of its *parent classes*. Examples of classes with a single parent are instances of *single inheritance* while classes having more than one parent class exhibit *multiple inheritance*. In the analysis and design phase, inheritance serves as a classification tool and helps group related classes together. Imposing a hierarchy establishes abstract classes and forces a parent-child relationship amongst classes thus clarifying the role of each class and its position in the software architecture. Designing a hierarchy is not an easy task since there are several other relationships that exist between classes that must be taken into consideration. An inheritance graph corresponds to the notion of a family tree. Ideally, the root of the inheritance graph is an abstract class representing the general concept that the classes in the graph attempt to model. This abstract class is referred to as a generalized class and its descendants are called specialized classes. Object-oriented programming languages view inheritance as a class composition mechanism that

allows code sharing, code reuse, and incremental programming. [Weg87] provides an excellent and thorough treatise on inheritance. [BGM89] provides a discussion on inheritance in combination with other principles. [WZ88] investigates the sibling relationship between classes. Inheritance as defined in [Boo93] is:

Inheritance : A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.

2.5.7.4 Polymorphism

The word polymorphism means more than one form (from poly = multi and morph = form). In the object-oriented sense, polymorphism is the mechanism whereby a programming entity can refer to objects of more than one class [Mey88]. The messages accepted and executed by the objects participating in polymorphism depend on the class of the object. Polymorphism is a very useful mechanism that makes a system flexible and extensible. For example, consider a robotic arm that can lift rectangular and spherical objects from a table. The shape of the object to be lifted determines the orientation of the clasping fingers of the the arm. Since the arm is capable of lifting two different kinds of objects, the robot arm system is exhibiting polymorphism.

Object-oriented languages support polymorphism by using *overloading* or *late-binding* techniques [CM91]. The definition of polymorphism adopted in this work is the one provided in [Boo93].

Polymorphism : A concept in type theory according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways.

2.6 Object Oriented Design

Object-oriented analysis (domain analysis) [SM89] results in the identification of objects and classes. An important activity of object-oriented design is the identification of relationships between classes and objects. This is the key step towards organization and structure within the object-oriented framework. The process of establishing relationships results in determining the functionality of classes, similarities and differences amongst classes, class interactions, the purpose of each object and object interactions.

There are three basic relationships between classes *generalization*, *aggregation* and *association* [Weg87, Boo93, CAB⁺94, CY90]. The generalization relationship, more accurately the generalization-specialization relationship, establishes a *kind-of* relationship between classes. For example, a dog is a kind-of animal and is a kind-of mammal. In this example, animal and mammal represent generalized classes whereas dog is a specialization of both. The aggregation relationship exemplifies the *part-of* relationship between classes. For example, a tail is a part-of a dog. Association is an abstract relationship that exists amongst classes that are used to model a larger concept. Association at a rather basic level may be said to capture the *uses* relationship amongst classes. For example, a dog uses a frisbee to play. Thus the unrelated classes of dog and frisbee conjure up a larger picture of an airborne dog and a flying frisbee. [Boo93] provides a comprehensive discussion on class and object relationships.

In the Booch object-oriented methodology [Boo93], design documents consist of *class diagram models*, *state transition models*, *object models*, *interaction models*. A class diagram consists of classes, the name, attributes and operations of each class

and the class relationships between classes. *Association, inheritance, has, using* are examples of class relationships. A state transition diagram specifies the states that a class (or an object of a class) can assume and event/action pairs that cause a class to change state. An event denotes a situation which triggers the associated action and this event/action pair results in the change of state of the class. The state transition diagram can be viewed as states that are connected (or related by) event/action pairs. An object diagram consists of objects and *links* between the objects. The name and attributes of each object are represented in the diagram. A link between two objects represents a concrete instance of the *class association* relation between the corresponding classes of the objects. A link establishes a bidirectional means of communication between the objects. Sequenced messages associated with a link denote the order in which operations are invoked on objects. An object diagram therefore represents the collaborations that occur between objects (in the form of messages sent to objects) to fulfill a specific system requirement. In the object diagram, the objects can be viewed as components and messages between objects represent the object collaboration relationship. A comparison of various existing object-oriented analysis and design techniques is provided in [MP92].

2.7 Summary

Motivation for research in the areas of object-oriented software maintenance and reverse engineering can be attributed to four factors – (i) cost of maintenance in terms of time and money expended by a company, (ii) complex nature of object-oriented software systems and the time spent by maintainers in understanding the overall structure of the system (iii) loss of information between the design and

implementation phases and (iv) the need for better tools and aids for software maintenance.

Understanding software structure is a basic requirement of software maintenance. Software structure can be represented and understood at different levels of abstraction. Designing an abstract representation model is an important activity in the reverse engineering process.

Class, object and message-passing between objects are key object-oriented concepts. The object-oriented principles of abstraction, encapsulation, inheritance and polymorphism make an object-oriented system flexible, extensible, reusable and maintainable. The same principles are responsible for the complex nature of object-oriented systems.

The next chapter discusses related approaches to the problems of object-oriented software maintenance and reverse engineering the design of a software system.

Chapter 3

Related Research

Research in the area of maintenance of object-oriented systems addresses issues in understandability of object-oriented systems, automatic restructuring of class hierarchies and maintenance tool support. Sections 3.1-3.9 of this chapter discuss research projects that are closely related to the the areas of program comprehension and software maintenance. Table 3.1 presents a comparison of the various approaches including the one adopted in this research. Restructuring techniques and issues are presented in section 2.5.4. An overview of issues and research in the areas of reverse engineering and software maintenance is presented in section 1.3.1.

3.1 Programmer's Apprentice

The goal of the Programmer's Apprentice research project [RW88], was to develop a theory that would logically explain the techniques that expert programmers employ in analyzing and understanding programs. An application of such a theory was perceived to be the elimination of software problems by the introduction of automatic programming. Consequently, this research would typically classify under forward engineering. However, the results, concepts and techniques that were produced from this research work proved to be useful in the area of reverse engineering. This research introduced the notion of a *cliché* [RW90, SWC93]. A cliché, as defined in [RW90, SWC93], is a commonly occurring programming structure or a common pattern that is used repeatedly.

Data structures and algorithms provide solutions to programming problems. Data structures address the issue of data organization for ease of information retrieval and storage, algorithms define precise computation methods that may require specific data structures. An experienced programmer is able to develop sophisticated data structures and algorithms (called cliché in [RW90]). Programmers test the applicability of common data structures and algorithms by using them in a variety of situations. Experienced programmers who are familiar with a large number of clichés understand programs by identifying and recognizing the clichés that were used in the development of the program.

Rich and Wills [RW90] describe a system, the *Recognizer*, that automatically detects a cliché in a program and constructs a hierarchical description of the program in terms of clichés. Initially, the Recognizer accepts a program as input and translates it into a graph-based representation. The graph-based abstract representation of a program is referred to as Plan Calculus. The Recognizer then matches the graph representations of clichés against subgraphs in the graph representation of the program. The plan calculus representation is a language-independent representation of programs that enables the recognizer to identify the structure of the cliché despite syntactic variations that may be present in the implementation of the cliché. The essence of this work is captured in the following sentence from [RW90]:

Essentially, a plan is a directed graph and cliché recognition identifies subgraphs and replaces them with more abstract operations.

The identification and representation of clichés and the automatic recognition of clichés are significant contributions to the field of reverse engineering, program comprehension, and software maintenance. The recognizer was demonstrated to work on small Common Lisp programs [RW90]. The clichés present in [RW88] are

useful for maintenance of structured software systems but not for object-oriented software systems.

3.2 Desire

The goal of the Desire (from design recovery) system was to aid program understanding for maintenance. Biggerstaff [Big89] presents a basic design recovery process method to aid maintenance and reuse. According to Biggerstaff, the design recovery process consists of the recreation of the design of a software system from an analysis and study of existing code, design documentation, personal experience and general knowledge about problem and application domains. Therefore, the automation of the design recovery process must model and incorporate these activities.

The notion of a *conceptual abstraction* is introduced and explained in [Big89]. A conceptual abstraction is an informal representation of design information for human understanding. A conceptual abstraction has two properties – (i) structural property and (ii) a semantic or associative property. The structure of a conceptual abstraction is the pattern of connections between the lower-level (code level) constructs that were used to implement the conceptual abstraction. Structural patterns enable the identification and recognition of conceptual abstractions. The semantic properties of a conceptual abstraction aid in understanding the purpose and usefulness of the abstraction. These conceptual abstractions are defined as *idioms* in [Big89]. Desire produces a dictionary containing information on functions, global variables, comments in C source code and the relationships between functions and variables. The information is presented as a hypertext document and additional

browsing support tools facilitate the examination of call-graphs and definition-use graphs of a C program.

3.3 Valhalla

The factors that complicate the understanding of object-oriented software systems are concisely formulated and expositioned in [WMH93]. Wilde and Huitt [WMH93] examine the idea of using dependency analysis to aid in understanding object-oriented software systems. Wilde et al [WHH89] present tools that exploit the concept of dependency graph to illustrate software relationships and the effect of modifications on various software components.

The notion of context-of-use is defined in [WMH93] and used to represent a method's semantics. The Valhalla object-oriented development environment advocates the use of object animation to enable understanding of complex class and object interactions. The Valhalla animator is capable of displaying message sequences between objects. The animation of message sequences is an invaluable aid to maintainers because it reveals the identity of each object, the sequence of each interaction and the nature of each interaction. However, the animation is not detected automatically by the Valhalla system; the animation sequence is provided as input to the Valhalla environment and it simply provides a graphical animation of the input. The automatic detection of message sequences is one of the difficult goals of the Valhalla project. The Valhalla environment aids in the development and understanding of C++ source code.

3.4 Demeter

Lieberherr et al [LH89, LX93, Opd92] conducted research to investigate methods in which the overall productivity of object-oriented designers and programmers could be improved. One of the outcomes of their research was the language-independent *Law of Demeter*. This law lays down guidelines for good programming styles for object-oriented software systems. The law defines the ways in which classes should depend on each other in terms of the methods that classes invoke on each other. A class is said to depend on another class if it calls a function defined in the other class. Some of the goals within the Demeter system are to reduce dependencies between classes and to ensure modular well-behaved software. Their approach defines good style and ensures it by providing techniques to transform code written in a bad style into code having the quality of good style. The law is specified for systems developed in the object-oriented programming languages C++, Common Lisp Object System (CLOS), Eiffel, Flavors and Smalltalk-80.

In [LX93] the concept of *propagation patterns* is introduced. A single propagation pattern is an abstract specification of a collection of similar object-oriented programs. Propagation patterns are a mechanism to represent programs at a higher-level than that allowed by programming languages. The abstraction available permits more adaptable software. The authors also present the concept of *growth plans* which are means of specifying, recording and testing incremental changes in a software. With the aid of propagation patterns and growth plans, smaller, adaptable, extensible and reusable software can be developed. Software development and maintainability is aided by allowing incremental development of software such that each change (or increment) in the software is recordable and testable.

3.5 OOTME

The object-oriented test model environment (OOTME) [KGH⁺93, KGH⁺94] is a graphic model that represents a reverse engineering approach for testing object-oriented software systems. The relationships between software components of an object-oriented software system are captured in three abstract representations – (i) object relation diagram (ORD) (ii) block branch diagram (BBD) (iii) object state diagrams (OSD). The ORDs of a system establish inheritance, aggregation and association relationship between classes. BBDs of a system represent the call graphs of functions and methods of a class. OSDs model the state transitions of objects. Each diagram aids in testing a specific part of the system. The ORD, BBD and OSD representations were also intended to be used by knowledge-based tools as an extension to the environment.

In [KGH⁺94] different types of code changes that can occur in an object-oriented class library are described. A *class firewall* represents the set of affected classes when changes are made in a class library or an object-oriented program. The OOTME system is a powerful and useful maintenance tool that captures class relationships and class dependencies and aids in software maintenance by automating the identification of different kinds of the code modifications and the components that are affected by the modifications.

The information content in the OOTME graphic model representation of object-oriented software system resembles the information content in the LLSA model presented in chapter 4. However, the LLSA model presents a more uniform representation of components and the LLSA model also provides for defining the interfaces of each software component in addition to the interactions between the components.

3.6 CIA, XREF/XREFDB, SAM

The C information abstraction system [CNR90] represents the structure of C programs as a relational database. The conceptual model underlying the abstract representation is an *entity-relationship* model. This model consists of five domains – file, macro, data type, global variable, function. Elements are connected by the *includes* and *refers* relationships. The abstract representation of C software systems enabled the development of at least five aspects of program structure – (i) graphical views of function call structure (ii) compilable subsystems (iii) function layering (iv) dead code detection (v) used data bindings. CIA++ [GC90] extends CIA to support the language features of C++.

The XREF/XREFDB system developed by Lejter et al [LMR91] also stores the structure of an object-oriented software system in the form of a relational database. The system provides various cross-referencing facilities such as determining the locations a function is called, the locations where a variable is defined or used, the location of the declaration/definition of a symbol. The interface of a class is also obtainable from the system.

Ketabchi explores the database approach to software maintenance in [M.A90]; this paper discusses a software analysis and maintenance system (SAMS) that integrates analysis and maintenance functions and provides facilities such as cross-referencing and configuration management.

3.7 Browser

Sametinger presents a tool for the maintenance of C++ programs [Sam90] that eases the process of navigation in the source code. Source code is represented as

“chunks of information” together with relationships that connect the chunks. The primary focus of the code browser developed by Sametinger is on classes, methods in a class and global declarations. The chunks of information and their relationships are represented as a hypertext document.

3.8 SCRUPLE

Paul and Prakash [PP94] present SCRUPLE, a system that accepts a specification of a program pattern and locates all occurrences of the pattern in source code. The program fragment to be located is specified in a specification language which they refer to as a *pattern language*. This specification language is an extension of the programming language being used.

One of the assumptions underlying the approach taken in [PP94] is that programmers hypothesize about the purpose of a code fragment and then attempt to locate the code fragment in existing code. For example, a programmer may hypothesize that the typical code for matrix multiplication involves the use of three nested iterative loops. The specification for the matrix multiplication code is input to the SCRUPLE system as is some existing source code. SCRUPLE matches the specification to the code and reports the locations of all occurrences of the code fragment. The maintainer can then proceed with code modification or replacement and the SCRUPLE system effectively reduces the time and effort a maintainer may have to spend in locating the fragment. SCRUPLE does not help a maintainer in understanding the structure of the system.

The SCRUPLE system transforms source code into a *syntax-tree* representation. The program pattern specification is transformed into a special-purpose

non-deterministic automaton (NFA). The syntax-tree representation is then fed to the non-deterministic automaton which reports successful matches between the pattern to be located and the source code being searched. The prototype system SCRUPLE is demonstrated for two languages C and PL/AS (a PL/1 variant).

3.9 PERPLEX

Multiple views of a software system aid in understanding the system [Boo93, KM94]. Kinloch and Munro [KM94] extend the work of Harrold and Malloy [HM91] and present an intermediate graph representation of C programs. The representation model is called the *combined C graph (CCG)*. The CCG representation enables the construction of multiple views such as program slices, call graph, definition-use etc.

The CCG is a dependency graph representation model that captures control-flow and data-flow dependencies in a C program. The CCG of a C program is constructed using the PERPLEX code analysis tool. This tool extracts information from C source code and stores it in the form of a Prolog fact base. The CCG is constructed from the fact base by executing Prolog queries on it.

A summary of the different approaches presented in sections 3.1-3.9 is given in Table 3.1. The first column contains the name of the project. The second column presents the significant contributions of the project by stating the abstract representation model and concepts that the project introduced. In the case of the Demeter project, the significant contribution is the Law of Demeter, which is not a representation model, but a specification of good programming styles and guidelines for the object-oriented paradigm. Most of the projects have been demonstrated to

work for software systems implemented in a particular programming language. The third column specifies the programming language of the source code for which the research has been shown to be useful. Approaches have been classified as – (i) forward engineering (ii) reverse engineering (iii) building databases and (iv) pattern specification. Each approach has an associated technique that is employed to construct the representation model. The techniques are – (i) information extraction (ii) database query and (iii) pattern matching. Thus the Programmer’s Apprentice project was a forward engineering project that constructed a plan calculus representation of a Lisp program and used pattern matching techniques to match the plan against a plan representation of a cliché.

3.10 Restructuring

Restructuring of class hierarchies to reflect simplified and organized relationships is a necessary aid to object-oriented software maintenance. Restructuring requires code modification and these modifications are performed by a maintainer. Automating class code modifications allows the prevention of human-programming errors and automatic determination of the impact of a modification. Modifications to class hierarchies can be classified as – (i) data member change (ii) method change (iii) class change (iv) class library change [KGH⁺94]. Determining the impact of a modification on the remaining classes is complicated by inheritance and other class relationships.

Casias [Cas92] provides an incremental class reorganization algorithm. The algorithm decomposes existing class relationships and restructures them by abstracting common properties into an abstract super class. Opdyke [Opd92, OJ92]

Table 3.1: Summary of Different Approaches

Project	Representation Model and Terms	Programming Language	Approach
Programmer's Apprentice	Plan Calculus, Cliché	Lisp	Forward Engineering Pattern Matching
Desire	Parse Trees, Conceptual Abstraction	C	Reverse Engineering Extraction
Demeter	Law of Demeter, Propagation Patterns	C++,CLOS, Eiffel,Flavors, Smalltalk	Forward Engineering Pattern Matching Pattern Matching
Valhalla	External Graph Object Animation	C++	Reverse Engineering Extraction
OOTME	Parse Trees, ORD,BBD,OSD	C++	Reverse Engineering Extraction
CIA	Relational Database	C	Building Databases Query
XREF	Relational Database	C	Building Databases Query
SCRUPLE	NFA, Syntax Trees	C	Pattern Specification Pattern Matching
PERPLEX	CCG, Prolog fact base	C	Reverse Engineering Extraction
pulse	LLSA, LLDP	C++	Reverse Engineering Extraction

presents algorithms to refactor class hierarchies and introduce abstract super classes, specialized subclasses and aggregate classes.

3.11 Related Research in Reverse Engineering

Software reuse is concerned with the development of software which can be used repeatedly without making any modifications; reuse is an attempt to bring pluggability to the field of software engineering. Since the code comprehension activities

performed in reuse and maintenance are similar, Basili provides three maintenance models (quick-fix, iterative-enhancement and full-reuse) and analyses them from the reuse point of view in [Bas90]. He proposes a *reuse framework* to enable the selection of appropriate maintenance models and advocates an improvement paradigm, reuse-oriented environment and automated support to support the reuse-oriented view of maintenance.

Rugaber et al [ROJ90] address the issue of design decisions by suggesting that the loss of continuity between the design and implementation phases is attributable to the lack of expressivity of design representations and the failure of design methodologies to provide a mechanism to express the constraints and conditions that guided the decision. Programming constructs and programming style guidelines are used to guide the abstracting of design from code.

Hausler et al [HPLH90] describe a function abstraction method to explain program behavior. They provide algorithms that determine the purpose (or function) of specific code fragments. These algorithms are called sequence abstraction, alternation abstraction, iteration abstraction, program slicing and pattern matching. Howden and Pak [Pak92] discuss problem domain, structural and logical abstractions and describe a method to extract functional specifications from COBOL code.

Choi and Scacchi [CS90] provide an algorithm that extracts and constructs a hierarchical design structure from source code. The design description extracted by this algorithm describes intermodule relationships in terms of the resources exchanged (this is called a resource-flow diagram—RFD) and the hierarchical relationship between system, subsystems and modules via resource-structure diagrams — RSD. The design description algorithm transforms an RFD into an RSD which

they refer to as system restructuring. The reverse engineered design is described in a module interconnection language, NuMIL.

Reconstruction of low-level design documents from code is the focus of the work of Antonini et al presented in [ABCC87]. This paper describes the information abstraction process and the low-level design documentation process employed by them. Soloway et al [SPL⁺88] propose a documentation strategy to compensate for delocalized plans. Canfora et al put forth the idea of using interactive animation techniques to support reverse engineering in [GLG92]. Colbrook and Smythe present a tool that structures code in terms of data and control flow in [CS89]. Gulla advocates the use of a software repository to store multiple versions of software and associated documentation to aid maintenance. The repository can be used by visualization techniques for displaying information computed from the different versions [Gul92]. Gillis and Wright describe a software package that reverse engineers *structure charts* and module specifications from existing FORTRAN source code in [GW90].

3.12 Summary

Though reverse engineering has made significant progress in with structured software systems, work in the field of reverse engineering of object-oriented systems is still in the nascent stages. Object-oriented software maintenance requires new technology, tools, and methods of analysis.

One of the goals of reverse engineering (see section 2.5.3) is to provide an abstract conceptual representation model of software systems [CC90, RC93]. The primary purpose of a conceptual model (or a mental model) is to aid in program comprehension. Experienced programmers understand code in terms of known concepts. An experienced programmer anticipates the presence of some common programming patterns and tries to locate these patterns in source code [PP94, RW88, RC93]. The information content in an abstract conceptual model must mimic the information content grasped by experienced programmers and the design of the model is a crucial aspect of reverse engineering projects.

The focus of maintenance tool support is to aid the actual execution of maintenance activities. The design of the representation model used by maintenance tools is driven by the facilities required by a maintainer to perform code modification; these facilities could include code navigation, location of affected components, location of specific code fragments, code modification, testing and validation of code modifications. The primary concern of maintenance tools is the facilities or the options that they must provide. A primary concern of reverse engineering techniques is to provide a representation of the system that aids a maintainer in grasping the overall architecture of the software system.

The conceptual model underlying the CIA, CIA++, XREFDB/XREF and code browser representations is the entity-relationship model and information is represented as a database of some kind. A relational-database representation is limited by the nature of the queries it can support, namely, relational queries. The architecture of a software system is better represented as a *graph* and the LLSA model that we present in chapter 4 is based on graph-theoretic concepts. A graph representation is capable of supporting many of the features of the above-mentioned maintenance

tools; in addition, a graph representation lends itself to graph algorithms such as *traversal, reachable vertices, transitive closures* [Liu85] which provide useful dependency information about the software. Graph representations are more amenable to the automatic detection of dependencies than entity-relationship models.

The OOTME system described by Kung et al in [KGH⁺94] uses graph representations for member functions and classes. The LLSA model provides a uniform component-based description of classes, objects and functions and the various interactions that are possible between these three kinds of components.

Low-level design patterns (LLDPs) represent common object-oriented structural patterns. LLDPs describe the structure of a recurring pattern as well as the semantics and usefulness of the pattern. An LLDP is a textual description of a common object-oriented strategy. LLDPs are described in chapter 5

Chapter 4

Low-Level Software Architecture of OO Systems

4.1 Introduction

Perceiving a clear picture of the overall structure and architecture of the software system is crucial to any kind of maintenance activity involving code modification, debugging or extending the system's functionality [Rom87, RC93]. This research is aimed at reducing the complexities inherent in the process of understanding and maintaining the overall structure of an object-oriented software system. A conceptual model to represent the overall structure of an object-oriented software system is described here. This conceptual model is referred to as the *low-level software architecture model (LLSA)*. The LLSA of systems developed in the object-oriented language C++ is defined in this chapter in terms of the concepts and interactions permissible by C++. C++ is a programming language whose syntactic and semantic rules are defined in [ES92].

Maintenance of large software systems requires software tools which perform a static analysis of code and use reverse engineering techniques to automatically produce useful design information [Pre92, ABCC87]. The development of an appropriate maintenance support "toolchest" (see [CvM93]) requires a careful requirements analysis of the needs of maintainers, the complexities inherent in the maintenance process and the complex nature of software systems. Polymorphism, inheritance

and dynamic binding are the most common object-oriented features that make object-oriented systems difficult to understand [CvM93, LMR91, WMH93, WH91].

Designing a conceptual model to aid the understanding of software systems is an important activity in the reverse engineering process. As stated in [RC93], the conceptual model developed must address the issues of

1. Information content
2. degree of abstraction,
3. modeling support
4. how the representation will be used

The information content and degree of abstraction of the LLSA model is provided in section 4.2. Modeling support and the usefulness of the LLSA of C++ systems are described in sections 4.4 and 4.5, respectively.

4.2 LLSA Conceptual Model

A software system is a complex entity comprised of interacting components. The basic software architecture model as defined in [AAG93, Sha94, AG94, GP94, KBAW94] is

software architecture = components + connections

A component is an architectural element or design module having an interface. The components of a software architecture may be a programming construct (such as a procedure or a function) or a group of programming constructs (such as a module or a file). Components interact with each other directly or via their interfaces.

The interface of a component provides a basis for connections between components. A collection of similar components is called the *domain* of the components.

Interactions between components connect components. Interactions may occur between components belonging to the same domain or to different domains. A function call is an example of a simple interaction.

4.2.1 Theoretical Model

Terms and notations that we use to describe the low-level software architecture of object-oriented systems are defined here. The three essential concepts in an object-oriented program are *class*, *object* and *function*. There are three domains in our model – *class*, *function* and *object* domain, denoted by C, F, O respectively.

The rationale for selecting *class*, *function* and *object* as the domains of the representation is given below.

1. A class, a function and an object are each a *higher-level programming concept*. Both, objects and variables have state, behavior and identity associated with them [Boo93]. However, the only way to change the state of a variable is by using it in lower-level constructs, like expressions or statements. The state of an object can be changed by sending it a message and requesting it to execute an operation that modifies its state; ie. state changes in a variable are accomplished by performing predefined programming language constructs whereas state changes in an object are accomplished by the messages that the object chooses to accept. These messages are defined by a programmer and not the programming language. This important distinction makes an object a higher-level concept.

2. Both class and function are encapsulating constructs. A class encapsulates data and member functions; a function encapsulates object declarations and statements. A compound statement is the only other encapsulating construct which encapsulates a series of statements; for example the body of a while loop is a compound statement. Compound statements, however, cannot appear outside the scope of a function or a member-function and therefore do not have any independent role to play in software structure.
3. Object-oriented design uses the concepts of class, object, message, function extensively. The object-oriented design of a system does not specify the computations that should be performed or how the computations must be performed; object-oriented design documents instead model components and relationships between the components.

Based on the above analysis, classes, functions and objects were designated to be the components in the LLSA model.

Each component has an internal part which is its own concern, and an external part which is the view the rest of the software has of that component. The external part of a component is what we refer to as the *interface* of the component. A component in our model can exhibit static and dynamic behavior. A component therefore has *static and dynamic interfaces*. The *static interface* of a component is the interface which can be statically determined. The *dynamic interface* is a collection of all possible interfaces that can be associated with the component dynamically.

An interaction connects two components. One of the components must initiate the interaction (the *initiator*) and the other component should be able to respond appropriately (the *recipient*) to the interaction. For example, a function f_i interacts

with another function f_j by performing a *call* to f_j . This call interaction establishes a connection (or a dependency) between the two functions.

An interaction is classified as static if both of the participants in an interaction can be statically determined on the basis of the static information available in the code. The interaction is deemed dynamic if the recipient cannot be pinpointed statically.

Notation : Let P denote an object-oriented program, C be the domain of classes in P , F the domain of functions in P and O the domain of objects in P . Let c_i , f_i , o_i represent a single class, function or object in P , respectively. We use e_i to denote a programming entity (such as class, function or object) without its interface and s_i to denote a component (ie. an entity and its interface). Let Int_s denote a static interaction and Int_d a dynamic interaction. Int denotes all interactions (static and dynamic). The low-level software architecture of a program, denoted $LLSA(P)$ is defined in terms of components and their interactions. The notations and definitions are summarized in Table 4.1.

4.2.2 Graph Representation and Views

The LLSA of a software system is defined as a directed graph. We refer to this graph as the *LLSA-graph*. A directed graph consists of nodes and directed edges [Liu85]. Nodes in the LLSA-graph symbolize software components and the directed edges denote interaction. The existence of multiple relationships in a software system necessitates definitions of different kinds of edges. Thus if there are three kinds of relationships in a system, three kinds of edges must be defined. The LLSA-graph of a system, shown in figure 4.1 (a), depicts three kinds of interactions – *control flow*, *uses*, *data flow*. The components in the system are uniformly referred

Table 4.1: Low-Level Software Architecture of Object-Oriented Systems

C	$= \{c_1, c_2, \dots, c_k\}$
F	$= \{f_1, f_2, f_3, \dots, f_l\}$
O	$= \{o_1, o_2, o_3, \dots, o_m\}$
P	$= C \cup F \cup O$
$Interface(c_i)$	$= \{static/dynamic\ interface\ of\ c_i\}$
$Interface(f_i)$	$= \{static/dynamic\ interface\ of\ f_i\}$
$Interface(o_i)$	$= \{static/dynamic\ interface\ of\ o_i\}$
$Component$	$= \{ \langle e, Interface(e) \rangle \mid e \in P \}$
$Int_s(s_i, s_j)$	$= \{all\ static\ interactions\ from\ s_i\ to\ s_j\}$
$Int_d(s_i, s_j)$	$= \{all\ dynamic\ interactions\ from\ s_i\ to\ s_j\}$
$Interactions$	$= \cup_{s_i, s_j} Int(s_i, s_j) \mid s_i, s_j \in Component$
$LLSA(P)$	$= \{ \langle s_1, s_2, Int(s_i, s_j) \rangle \mid s_1, s_2 \in Component, Int(s_i, s_j) \in Interactions \}$

to as C_1, C_2, \dots, C_7 irrespective of the domain they belong to. A directed edge between two components represents an interaction from an initiator to the recipient; for example, the edge from C_1 to C_2 conveys the information that control flows from component C_1 to component C_2 .

Graph representations (see section 1.1.3) are useful in understanding the different views of a software system. Three possible views are shown in figure 4.1 (b)-(d) – (i) *control flow graph*, (ii) *domain graph* (iii) *rooted component subgraph*. Each of these views can be constructed from the LLSA-graph. The construction and analysis of each view from the perspective of overall structure and maintenance are explained below.

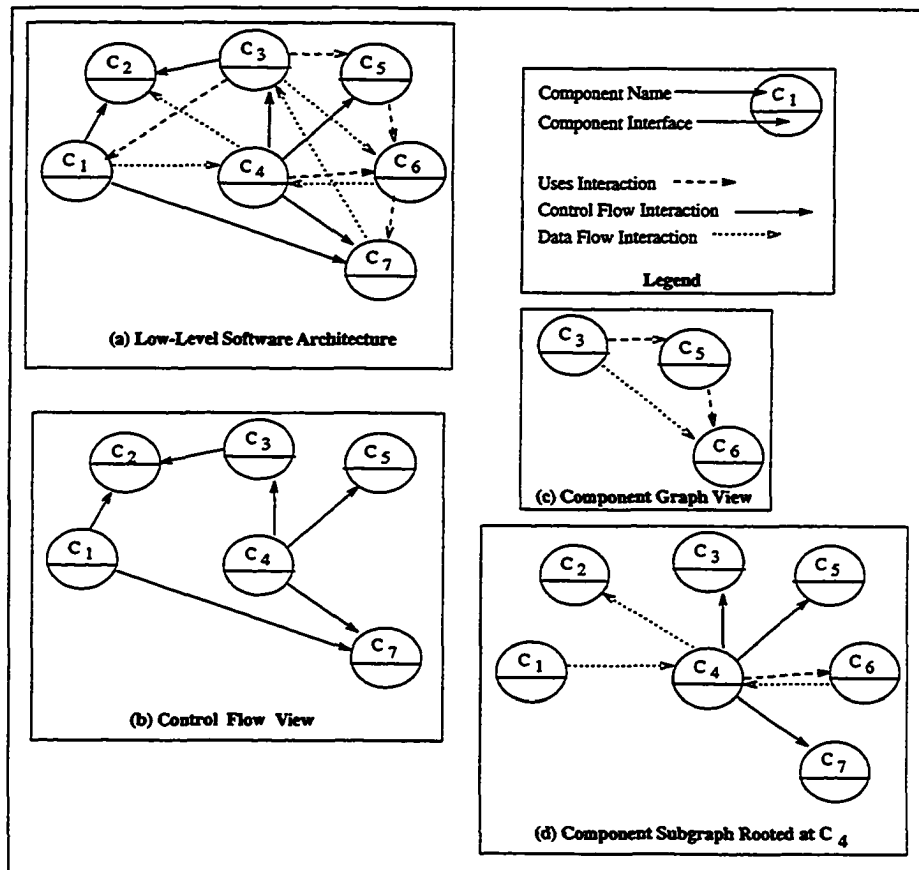


Figure 4.1: Graph Representations of a Software System

4.2.2.1 Control Flow Graph View

The control flow graph view can be obtained from the LLSA graph by performing a *transitive closure* operation on the control flow relationship (interaction). The transitive closure of a relation R results in a subgraph that consists only of components that interact with each other via the relation R. The transitive closure of the control flow relation for object-oriented systems is formally defined in table 4.2.

Notation : Let the control flow interaction between components s_i and s_j be denoted as $Int_d^{cf}(s_i, s_j)$ where the d denotes a dynamic interaction and cf denotes control flow. Let $C(Int_d^{cf})$ denote the transitive closure of $Int_d^{cf}(s_i, s_j)$. *Component* denotes the set of all components of an object-oriented program P as explained in section 4.2.1 and s_i denotes a component.

Table 4.2: Transitive Closure of A Relation

$Int_d(s_i, s_j)$	=	{ control flows from s_i to s_j }
$C(Int_d^{cf})$	=	{ s_k $s_k \in Int_d^{cf}(s_i, s_k)$ or, $s_k \in Int(s_k, s_i) \forall s_i \in Component$ }

The subgraph in figure 4.1 (b) is obtained by starting from a component, say C_1 , and including those components that C_1 interacts with via the relation *control flow*. This results in the inclusion of components C_2 and C_7 . Components C_2 and C_7 do not have a control-flow interaction with any other component and so the next component to be examined is C_3 . C_3 interacts with C_2 and is included in the view. Component C_4 is examined next. C_4 interacts with C_3, C_5, C_7 ; C_5 is the only unexamined component remaining. C_5 does not interact with any more

components. The transitive closure operation stops after all nodes have been examined. The transitive closure of the system shown in figure 4.1 (b) consists of $\{C_1, C_2, C_3, C_4, C_5, C_7\}$. Component C_6 is not in the transitive closure of the control flow interaction because there is not transfer of control to or from C_6 . A view based on any other relationship can similarly be obtained by performing a closure operation on the relationship of interest.

We shall refer to views obtained by performing a transitive closure on a relationship as a *relation closure view*. The transitive closure view could contain a subset of the total set of components in a software system. For example, component C_6 is not represented in the control flow graph view. Even if the component was included as an isolated, unconnected node in the view, its role in the software system is not evident from this view because of the absence of other relations. This view is therefore not useful from the perspective of understanding the overall architecture and the roles of different software components.

4.2.2.2 Component Domain Graph View

Since the absence of relations in the previous control flow graph view obscures the role of a component in the software system, we take an alternate approach and define another view. In this view, we restrict the graph to contain only nodes belonging to the same domain (ie. the nodes represent components belonging either to C , F , or O) and include all the interactions between these nodes. For example, we can restrict the nodes to belong to the class domain C and examine the *class subgraphs* in the system (see fig 4.1 (c)). Restricting the nodes to F or O will give the *function subgraphs* and *object subgraphs* respectively. A fairly comprehensive

picture of the overall structure can be obtained by examining the collection of all the domain graph views of the system (ie. all the layers in the system).

We refer to such a restricted-component view as the *component domain graph view* or the *layered* view of the system. The layered view provides a more comprehensive view of the structure of a software system than the relation closure view described above. A limitation of the layered view is the loss of information concerning interactions between components belonging to different domains. This limitation leads us to define the third view which does capture the software architecture of a system.

4.2.2.3 Rooted Component Subgraph View

Both, the relation closure view and the component domain view have their drawbacks since there is some information loss in both the representations. We therefore concentrate on defining a view that eliminates the information loss in the transitive closure view and the component domain graph view. The *rooted component subgraph view* captures all the information pertaining to a component that is available in the transitive closure and the component domain graph views as well as more information about the component that may not be present in the other two views. The rooted component subgraph is defined as follows (see Table 4.3).

Table 4.3: Graph-theoretic Definition of a Rooted Component Subgraph

V	$=$	$\{s_i\} \cup \{s_j \mid s_j \in Int(s_i, s_j) \vee s_j \in Int(s_j, s_i)\}$
E	$=$	$\{(s_l, s_m) \mid s_l, s_m \in V \wedge \exists Int(s_l, s_m) \ni l=i \vee m=i\}$
$RCG(s_i)$	$=$	$\{V, E\}$

Notation: Let s_i denote a component of a program P . Let $RCG(s_i)$ denote the component subgraph rooted at component s_i . The vertex set V of a rooted component subgraph $RCG(s_i)$ of component s_i consists of the component s_i and all the components that s_i directly interacts with or that interact directly with s_i . The directed edges set E of $RCG(s_i)$ consists of all the interactions between the components in V .

The rooted component subgraph view (see figure 4.1 (d)) is obtained as follows. Initially a single component is selected and included in the subgraph. This selected component is referred to as the *root component*. Next, all the components that the root component interacts with are included in the subgraph. Finally, all the components that interact with the root component are included in the subgraph.

Rooted subgraph view gives insight into all the roles of the root component since all the interactions in which the component participates are represented. Additionally, the dependency of other components on the root component is also apparent since the nodes that interact with the root component are also represented. A textual description of a rooted subgraph must contain the name of the component, the names of interactions that the component can participate in, and the names of the components that the root component interacts with. A collection of such textual documents provides the textual description of the LLSA of a system.

The collection of subgraphs rooted for every software component in the system aptly captures the LLSA of the system. The LLSA of a software system can hence be represented as a collection of rooted component subgraphs for each component in the system. Hence an equivalent definition of the LLSA of a system, in terms of the the rooted component subgraph definition is :

Table 4.4: Definition of an LLSA Graph in terms of Rooted Component Subgraphs

$$\boxed{LLSA(P) = \cup_{s_i} RCG(s_i)}$$

To summarize, the LLSA graph is a comprehensive representation of a software system that captures multiple relationships in the system and expresses the system in terms of components. Dependence graphs and layered views of software systems can be constructed from the LLSA representation; but the LLSA of a software system cannot be completely determined from the dependence graphs or layered views of a system.

4.3 Low-Level Software Architecture of C++ Programs

An abstract representation model of C++ systems, based on the conceptual LLSA model described in the previous section, is presented.

The C++ language [ES92] consists of syntactic constructs for expressing *constants*, *basic types*, *expressions*, *statements*, *declarations* and *definitions*. A basic type is a type that is directly supported by the language itself – such as integer. An object is an instance of a class, a variable is an instance of a basic type. *Class declarations* in C++ encapsulate the representation, behavior and properties of objects. *Member functions* of a class specify operations that can be performed on objects of that class. *Function definitions* manipulate objects to provide an object-oriented solution. Objects accept *messages* and comply with the request by executing the requested member function.

Name	Name of the object and its scope
Static Type	Name of the class in the declaration of this object
Dynamic Type	Names of descendant classes of the static type
Physical location	Files in which the object is declared, defined and used.
Static Interface	Nonvirtual member functions of the class associated with the object
Dynamic Interface	Virtual member functions of the class associated with the object
Dynamic Interactions	<p><i>Sends Messages To Objects:</i> Set of < object,qualified member function name >. In case of pointer objects, qualified member name is a set. The object name in the tuple is obtained from the actual parameter list of the member function in the tuple that is invoked on this object.</p>

Figure 4.2: Object Component Description Template

4.3.1 Component Description

The description template of each component is given in figures 4.3, 4.4, 4.2. The description of each component is essentially a textual representation of the rooted component subgraph view explained in section 4.2.2. The design of each description template is based on the analysis of components, their interfaces and interactions given in section 4.3. The LLSA of an object-oriented software system is represented as a collection of the textual descriptions of each component (class, function, object) in the system.

4.3.2 LLSA Components of a C++ Software System

Object Component: Each object has a name and is either a global object or a local object of some function. In the case of local objects, the name of the enclosing function is the scope of the object component. In the LLSA model,

Name	Name of the class
Physical location	Files in which the class is declared and used.
Static Interface	Qualified public member functions of the class.
Dynamic Interface	Collection of tuples < ancestor, qualified public member functions that are available for the ancestor >
Static Interactions	<ul style="list-style-type: none"> <i>Ancestor Classes</i> : Classes that this class inherits from <i>Descendant Classes</i> : Classes which inherit from this class <i>Container Classes</i> : Classes which contain this class <i>Contains Classes</i> : Classes which are contained in this class <i>Friendly With Classes</i> : Classes in which this class is a friend class <i>Friend Classes</i> : Classes which are friend classes of this class <i>Object Family</i> : Objects that are instantiations of this class <i>Calls Functions</i> : Functions that are called by member functions of this class <i>Creates Objects of Classes</i> : Classes such that the constructor function of the class is implicitly or explicitly invoked in a member function of this class <i>Created By Classes</i> : Classes that create objects of this class by invoking the constructor.
Dynamic Interactions	<ul style="list-style-type: none"> <i>Associated With Classes</i> : Classes such that this class has a pointer object of that class as a data member <i>Associate Classes</i> : Classes which are associated with this class <i>Uses Members Of Classes</i> : Classes such that a member function of this class uses a member function of the other class. <i>Used By Classes</i> : Classes that use a member function of this class

Figure 4.3: Class Component Description Template

Name	Name of the function
Physical location	Files in which the function is declared and defined.
Static Interface	Classes that occur in the parameter list
Dynamic Interface	Descendant classes of classes in the static interface
Static Interactions	
	<i>Calls Functions:</i> Functions that this function calls
	<i>Called By Functions:</i> Functions that call this function
	<i>Creates Objects Of Classes :</i> Classes such that the constructor function of the class is implicitly or explicitly invoked by this function
	<i>Used By Classes :</i> Classes whose member functions use this function
Dynamic Interactions	
	<i>Sends Messages To Objects:</i> Set of < object,qualified member function name >.
	In case of pointer objects, qualified member name is a set. The member function in the tuple is invoked on the object in the tuple.

Figure 4.4: Function Component Description Template

there are two kinds of objects – a simple object (we call this an object) and a *pointer object*. Simple objects are instances of some class. Pointer objects point to an object of a class. All other object declarations can be classified in these two broad categories. *Reference objects* are simple objects. An element of an array of objects is a simple object, an element of an array of pointers is a pointer object. Pointers to functions are not represented as objects in the LLSA model. Objects have a *static type* associated with them which is the name of the class in the declaration of the object. Pointer objects have a static and a dynamic type associated with them. The static type is the name of the class in the declaration of the pointer object and the dynamic types are the set of classes that derive from the static class type.

Class Component: There are four kinds of classes – *simple class*, *base class*, *derived class* and *abstract class*. A simple class is a standalone class that has no inheritance relationship with any other class. A base class is the term used to refer to a parent class in an inheritance relationship. Derived class is the term used to refer to child classes in an inheritance relationship. Abstract classes are classes that do not have any instances or objects. Abstract classes can only have pointer objects associated with them.

Function Component: A function has a name and a parameter list. The parameter list specifies the type of the object (ie. the basic type or a class name) and the kind of object (ie. object, pointer object, array object etc.) The name of the function and its abstract parameter list are represented in the LLSA model.

4.3.3 Component Interface

Class Interface: A class declaration in a C++ program consists of data members and member functions (collectively referred to as members). Members are segregated into three sections – *private*, *protected*, *public*. Public members can be accessed by any other component in the program. Access to protected members of a class is limited to derived classes. Private members cannot be accessed by any other component and are restricted to the class itself. Data members of a class specify the representation of the objects of that class. C++ places no restriction on whether or not data members should be private or public. However, it is considered good practice to “hide” the representation of an object to ensure encapsulation and usually data members are private to a class. Data members of a class do not play any *direct* significant role

in the overall structure of the system. Therefore we define the interface of a class solely in terms of the member functions of the class and ignore data members. The interface of a class is defined to be the collection of all member functions of the class. The textual description of a class component contains the *qualified* name of a member function. The qualified name of a member function is the name of the class in which the member function is defined and the name of the member function itself.

C++ supports *dynamic binding* via *virtual member functions* of a class. The actual virtual member function that is bound to an object depends on the dynamic class type of the object. Non-virtual member functions are statically bound to an object depending on the static type of the object. Determination of the static/dynamic nature of a member function is possible from an examination of class declarations. For derived classes, determining the nature of a member function may require an examination of all the classes in the inheritance hierarchy associated with the derived class.

Objects of a derived class exhibit the characteristics of the derived class and the characteristics of all the ancestors of the derived class. If we view a class as defining a type, then the type associated with the object is the derived class type and the ancestor class type. The member functions that can be invoked on the object depend on whether we view the object as having the type of the derived class or the type of the ancestor classes. Pointer objects can dynamically assume the type of the ancestor classes or the derived class type. Hence, the static type of a pointer object is the derived class itself, whereas its dynamic type is the set of ancestors of the derived class. This

leads us to define the static interface and dynamic interface of a class in terms of the static and dynamic types of a pointer object of the class.

1. *Static interface of a class* : The static interface of a class c_i is the collection of all the member functions defined in the class and the member functions of ancestor classes that can be accessed via this class.
2. *Dynamic interface of a class* : The dynamic interface of class c_i is a set of tuples $\langle c_j, \text{dynamic - interface of } c_j \rangle$ where c_j is an ancestor of c_i . The dynamic interface of c_j in c_i is the set of methods that can be invoked on a pointer object of type c_i when viewed as an object of type c_j . If c_i has no ancestors, then its static and dynamic interfaces are identical.

Function Interface: We define the interface of a function to be its *abstract parameter list*. The abstract parameter list consists of the types (class names) of each parameter and the kind of parameter (object or pointer object). This represents the static interface of the function.

Since C++ allows class conversion between a derived class and its ancestor classes, it is possible for multiple types to be associated with a pointer object parameter. The types that can be associated dynamically with a parameter are the descendant class types of the static class type.

1. *Static interface of a function* : The collection of tuples $\langle c_i, \text{object/pointer object} \rangle$ where c_i is the name of a class.
2. *Dynamic interface of a function* : The dynamic interface of a function is set of tuples $\langle c_j, \text{object/pointer object} \rangle$ where c_j is a descendant of a

class c_i in the static interface of the function. The dynamic interface is identical to the static interface if there are no pointer object parameters in the static interface of the function.

Object Interface: The interface of an object is the set of messages it can accept.

These messages may statically or dynamically bound to an object. Statically bound messages constitute the static interface of an object and dynamically bound messages constitute the dynamic interface of the object.

1. *Static interface of an object* : The static interface of an object is the collection of methods that are statically bound to it; ie. the non-virtual member functions in the static class type of the object.
2. *Dynamic interface of an object* : The dynamic interface of an object is the collection of methods that are dynamically bound to it; ie. the virtual member functions in the dynamic class type of the object.

4.3.4 Component Interactions

An interaction is viewed symmetrically so that information about the interaction is represented in the initiator and the recipient components. Each interaction is represented as a collection of the names of components with which a component interacts.

Class Interactions: There are three class relationships that are used in object-oriented design – (i) *inheritance*, (ii) *aggregation*, (iii) *association*. Inheritance is directly supported in C++; aggregation and association are simulated by using language features.

Inheritance establishes a parent-child dependency between classes. Changes in ancestor classes may affect the dynamic interface of child classes. Changes in derived classes can affect the behavior of functions that interact with the ancestor classes. Inheritance is represented as the *ancestor/descendant interactions* in the LLSA model of C++ systems.

Aggregation establishes containment relationship between classes; ie. a class contains another class or is contained by another class. Aggregation is not directly supported by C++. It can be easily modeled by including an object as a data member in the container class. The class type of the contained object indicates the class that is contained within the container class. Aggregation creates dependencies between classes that may be in different inheritance hierarchies. Changes in a contained class, its ancestors or its descendants may affect the class that contains it. This relationship is represented as a *contains/contained-by* interaction in the textual description of the class component.

Association permits dynamic interaction between classes. Dynamic properties in software are realized via the concept of indirection or *delayed binding*. Pointer objects are used to implement dynamic associations in object-oriented software systems. We classify the presence of a pointer object in a class as an indication of an association connection between the classes and represent it as a dynamic interaction. The association relationship we have defined for the LLSA of C++ systems models a dependency between classes belonging to distinct inheritance hierarchies. It is represented as the *associates/associated with* interaction.

Member functions of a class can use the members of another class directly via the *friend feature* of C++. This feature is a convenience provided by C++ to allow classes to use the private restricted part of another class. The friend construct connects classes that belong to separate inheritance hierarchies and hence *friends/friendly with* are class interactions that are included for representation.

To a lesser extent, classes also interact with functions and objects. The invocation of a function in the body of a member function represents class-function interaction. This interaction is referred to as the *calls functions* interaction. Information regarding the collection of objects associated with a class is useful in determining the effect of a change in the class on the objects. This implicit class-object interaction is represented as the *object family* interaction.

Member functions can create objects and send messages to objects. These capabilities of member functions are represented as interactions between the classes that contain the member functions and the classes of the objects that the member functions operate upon. This gives rise to two kinds of interactions *creates objects/created by* and *uses members/used by*. As a result of the analysis, the following static and dynamic interactions are defined for a class component.

1. *Static Interactions:* The following interactions can be precisely determined by performing a static analysis of the code – *ancestors/descendants, containers/contains, friends/friendly with, object family, uses functions and creates objects/created by*.

2. *Dynamic Interactions* For the following interactions, the precise class with which a class component interacts cannot be determined statically. The possible *set of classes* with which a class interacts is determinable and is represented in the textual description. The dynamic interactions for a class are *associates/associated with* and *uses/used by*.

Function Interactions: Functions interact with one another by performing a *function call*. Each function call results in the transfer of control from the caller function to the called function. Function calls can be statically determined and are classified as static interactions. This interaction is represented as the *call/called by* interaction.

Creation of objects within functions can be viewed as a function-class interaction. This interaction establishes inter-component dependencies and is recorded as the *creates objects of class* interaction.

Functions *send messages to objects*. In the case of an object, the *sends messages* relationship is static because the member function (both, virtual and non-virtual) that is invoked can be statically determined. For pointer objects, the type of the pointer object and the actual member function that is bound to the object are both determined dynamically. The *sends messages* interaction is classified as a dynamic interaction because of the dynamic nature of method-binding to pointer objects. The static and dynamic interactions that a function component is capable of exhibiting are :

1. *Static Interactions* The collection of functions that a function invokes *called functions* and the set of functions that call a function *caller functions*.

2. *Dynamic Interactions* The *sends messages to objects* interaction is represented as a collection of tuples. The tuple $\langle \textit{object}, \textit{qualified member name} \rangle$ represents a member function that is statically bound to an object. For pointer objects, the tuple represents the complete set of *possible* member functions that can be bound.

Object Interactions: Object-oriented design models its solution in terms of messages exchanged between objects. There is no direct construct in C++ that supports message exchange between objects. Objects can only receive messages. However, objects can interact with each other indirectly. This indirect interaction is achieved by passing an object as a parameter to a message that is sent to another object. The static and dynamic types of an object can be viewed as implicit object-class interactions.

There is no direct interaction between objects and functions. But there are dependencies between objects and functions. Functions manipulate objects by sending messages to them. Therefore, objects can be modified and acted upon by functions. Due to the aliasing problem it is difficult to determine the functions that modify an object if the object occurs outside the context of the function. We therefore do not represent any interaction between an object and a function and rely on the interaction between a function and an object for producing meaningful information. An object component participates in the following static and dynamic interactions:

1. *Static Interactions* There are no static interactions initiated by objects.

2. *Dynamic Interactions* A dynamic interaction between two objects o_i and o_j is represented as a tuple $\langle o_j, \text{member function} \rangle$. The class c_i of object o_i has a *member function* which has a parameter of type c_j , where c_j is the class of object o_j . The collection of such tuples *sends messages to objects* partially represents information about messages exchanged between objects. This is a partial representation because objects can exchange messages in other more indirect ways.

4.4 Representational Support of LLSA

The LLSA text description of each component represents the micro-architecture associated with the component. The information provides a direct lead to the components that may be investigated next (such as function main).

The inclusion of the names of interacting components in the LLSA description of a component is a significant contribution towards code navigation and code comprehension. An examination of a class declaration reveals the purpose of the class and the services it provides. Determining the protocol to be followed when using the class requires the examination of at least one object of that class. The messages that are sent to the object and the order in which the messages are sent establish the protocol that must be followed. Information about this protocol is available in functions that send messages. The LLSA associated with an object that appears in the object family of a class LLSA reveals the location (file and function) of the object. The LLSA of the function reveals the messages sent to the object.

As explained in section 4.2.2, various *views* can be constructed over the LLSA model to highlight certain aspects of software structure. This view capability is additional modeling support provided by the LLSA model.

The model also lends itself to further abstraction. Clustering techniques can be used to group related components together into a subsystem; relationships can then be defined over the subsystems to give rise to an abstract design-level software architecture representation.

4.5 How will LLSA be used ?

The LLSA model can be used to perform structure-preserving maintenance. Different types of code changes in a class are defined in [KGH⁺94]. The impact of a code change in a component can be expressed in terms of the change in the static/dynamic interfaces of the components and the static/dynamic interactions of the component. If the LLSA obtained after the change does not match the LLSA of the original unchanged component, then the structure of the system has been affected and the affected components can be directly determined by comparing the interactions of the changed component with the interactions of the unchanged components.

4.6 Summary

An abstract representation model for object-oriented software systems that depicts the components and interrelationships between the components was defined. The information content in the LLSA model is comprehensive and rich in structural information. Views of software systems can be easily defined and determined from

the LLSA representation. The LLSA model abstracts structure-significant aspects of object-oriented software systems and ignores statement-level computational detail. The behavior of the system, in terms of its LLSA representation, is expressed in terms of the static and dynamic interactions between the components. The effect of code changes to a software system can be expressed in terms of the effect on individual components, their interfaces and their interactions. Structure-preserving maintenance is therefore aided by the LLSA representation which can show the effect of code modifications on the overall structure of the system.

Chapter 5

Low-Level Design Patterns

5.1 Introduction

Notionally, a pattern is a recurring structure, a *leit-motif*. A pattern has a name, elements and relationships between elements. To illustrate, consider two simple geometric patterns, a *star* and a *hexagon* (see figure 5.1).

The first geometric pattern (figure 5.1.a) has the name *star*. It has six vertices. Each vertex is connected to two other vertices. The structural rule governing the connections is that adjacent vertices cannot be connected. The second geometric pattern (figure 5.1.b) has the name *hexagon*. It has six vertices. Each vertex is connected to two other vertices. The structural rule governing the connections is that adjacent vertices must be connected.

It is interesting to note that the two geometric patterns – *star* and *hexagon* have the *same number of elements* (namely, vertices and edges) and some *similar properties* (each vertex is connected to two other vertices) yet very *different structures*. The structure of a pattern can therefore be informally defined as the rules governing the connection between elements. The rules may specify the kind of connection that is permitted between the elements and the nature of the connection. Structural rules are designed to imbue the pattern with desirable properties. For instance, the star pattern can be viewed as being composed of triangles or of lines and vertices whereas the hexagon can only be viewed as being composed of lines

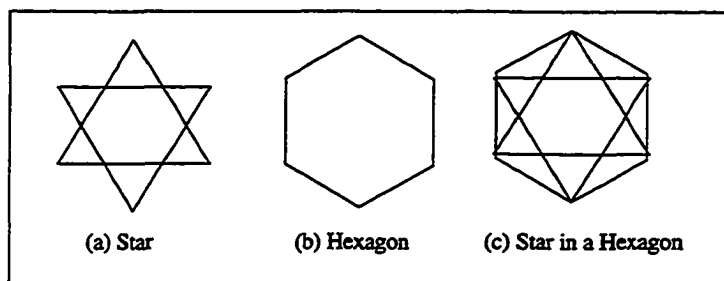


Figure 5.1: Geometric Patterns

and vertices. The star pattern thus has the desirable property of two possible methods of construction. The structure of a pattern is a significant characteristic that allows one to identify and classify patterns as well as reason about the structural properties of the pattern.

Patterns can be composed to form more complex patterns (see figure 5.1.c). Composability has two significant consequences – (i) a complex and large structure can be incrementally constructed by composing simpler and smaller patterns (ii) a large complex structure can be understood by decomposing it into smaller and simpler structures that are easier to understand and analyze.

5.2 Pattern Languages

Patterns as used in the object-oriented community has its origins in Christopher Alexander's work in architectural patterns [Ale77, Ale79, Lea94, Coa92]. Alexander et al [Ale77] define a *pattern language* to be an ordered collection of *patterns*. A *pattern* describes a commonly occurring problem, the solution to the problem and the patterns with which this pattern is connected. Connections between patterns establish an order between the patterns and connections also provide each pattern

with a *context*. Larger patterns are more completely described by smaller patterns. Smaller patterns are useful in the context of the larger patterns. Requiring a pattern to specify and justify its existence in the context of other existing patterns forces the pattern language formulator to design solutions that fit coherently and elegantly with each other.

The solution in a pattern describes the bare essential characteristics that any solution to the problem must have but leaves ample room for variation. The variations permitted are such that the core of the solution is not affected by them. For example, the solution in a pattern description of the human hand would specify the *bone structure* of the hand, the positioning of each bone with respect to the others, the number of joints necessary for flexibility and the degrees of movement for each finger. The length of each bone, the precise angle between the bones the shape of each bone are all factors that can be varied. Therefore, comparison at the level of bone structure leads one to say that one human hand is very much like another; but at a finer and more detailed level of comparison, there are plenty of distinctive features in human hands that distinguish them from one another and allow one to say that no two hands are alike. The solution in a pattern can therefore be described as an abstract, general solution whose goal is to identify the invariant or unchanging aspects of any solution to the problem. It may not always be possible to find such a core solution in which case the solution in the pattern represents one of a possible set of solutions.

In essence, a pattern language provides a series of patterns of varying levels of abstraction, each of which can be selected and combined with other patterns to provide a single coherent solution to a large problem. The basic concepts underlying

the theory and definition of Alexander's patterns have been found to be useful and applicable in the design of software systems too.

Patterns occur in various phases of software development, and a program can be viewed as being composed of a hierarchy of patterns, where each higher level represents a more abstract view of the program. The set of patterns that make up a program are variously referred to as a pattern language of the program, a framework of the program, or the architecture of the program.

5.3 Design Patterns

Design patterns [GHJV93, GHJV94] capture solutions to commonly occurring problems in object-oriented design. A design pattern is an informal textual description of the problem, its context, the solution and the consequences of applying the design pattern [GHJV94]. The textual description of a design pattern follows a format specified in a design pattern template (see [GHJV93, GHJV94] for the design pattern template). The template is a comprehensive and informative representation of the problem addressed by the designers, the nature of the solution and its motivation, the applicability and consequences of the solution and the participants in the solution. The relationships between the participants as well as the nature of the relationships are captured in the form of *structure* and *collaborations* between the participants. A design pattern is a succinct and comprehensive representation of design information which can be viewed as a microarchitecture or a small subsystem. Design patterns may be used in conjunction with one another to solve a larger object-oriented design problem. Idioms [Cop92] are language-specific patterns.

Gamma et al have devised a catalog of design patterns that address a variety of object-oriented design problems. The solutions proposed in the design patterns presented in [GHJV94] themselves use some common techniques and object-oriented principles to lend the properties of extensibility and reusability to the patterns. One of the techniques used is the strategy of *decoupling* components in order to reduce dependencies amongst them and create a flexible extensible solution (see Abstract Factory (87) in [GHJV94]). While decoupling is easily understood as a useful design concept, it must be implementable in an object-oriented language in order to retain the benefits of a decoupled design. Low-level design patterns capture such low-level programming techniques that recur in the design pattern descriptions.

A design pattern describes the problem, the constraints that a solution must fulfill and the solution to the problem. The solution is specified so that all the elements and the interrelationships that are necessary for the solution are fixed (ie. the structure of the solution is specified). Identifying or recognizing the occurrence of a design pattern in a software system is an activity that can be completed if (i) the design pattern is known (ii) the structure of the design pattern is distinctive and unique (iii) the elements are identifiable. The structure of a design pattern is often quite complex; moreover, structures of different design patterns share some common substructure. This substructure can be viewed as a recurring pattern across design patterns. Low-level design patterns correspond to common object-oriented techniques that have distinctive structures. From the software architecture representation viewpoint, a design pattern has the following useful characteristics.

- The description of a design pattern lists the participating classes and objects in the **Participants** section. This information is useful in determining the components that are likely to be affected if one of the participants is modified.

- The **Diagram** (**Diagram** has been replaced with **Structure** in [GHJV94]) section describes the static structure connecting the participants. This is useful in determining the nature of the dependencies and relationships between the participants.
- The **Collaborations** section explains the order in which the participants interact and the precise nature of the interactions. This section is therefore useful in understanding why the participants are structured in a particular way.
- The design pattern description partially captures inter-design-pattern relationships in the **See also** section by listing the names of patterns that a design pattern can combine with.

A set of patterns of a program provides insight into the logic and design rationale underlying a software system. A pattern language of a software system also describes the overall architecture of the system.

5.4 Low-Level Design Patterns

We define a *low-level design pattern (LLDP)* to be an informal textual representation of a common object-oriented strategy occurring in a software system. LLDPs describe the structure of a recurring strategy as well as the structure, semantics and usefulness of the strategy. Three sets of LLDPs – *polymorphism*, *decoupling* and *messages* are presented. The LLSA abstract model provides a view of software systems that captures the dependency relationships between code, and the nature of the dependencies. LLDPs provide the reasons that the dependencies must exist and be preserved.

The connection between LLSA interactions and the structure of an LLDP is explained in section 5.4.1, a description of the textual template designed for defining LLDPs is given in section 5.4.4. The definitions of the polymorphism LLDPs, decoupling LLDPs and messages LLDPs are provided in sections 5.4.5, 5.4.6, and 5.4.7 respectively.

5.4.1 Low-Level Design Pattern Structures

As explained in chapter 4, the LLSA textual description of a component of a system depicts those component relations that can be completely determined from an examination of the source code and a thorough understanding of the syntax and semantics of the programming language used for the implementation of the source code; ie. for each interaction described in the LLSA model, there exists a corresponding syntactic construct (or group of constructs) in an object-oriented programming language. The syntactic constructs of an object-oriented programming language that correspond to LLSA interactions are referred to as *fundamental patterns of interactions* in this work. Fundamental patterns for the C++ programming language are described in section 7.4.1.

LLDPs are at a higher level of abstraction than the LLSA model (see figure 1.3 in chapter 1). The structure of an LLDP is expressed in terms of the LLSA interactions. The relationship between the structure of an LLDP, LLSA interactions, fundamental patterns of interactions and programming constructs is shown in fig 5.2. The structure of an LLDP is defined in terms of one or more LLSA interactions, each LLSA interaction is defined in terms of one or more fundamental patterns of interactions, each fundamental pattern of interaction is defined in terms of one or more programming construct. There are two kinds of structures an LLDP

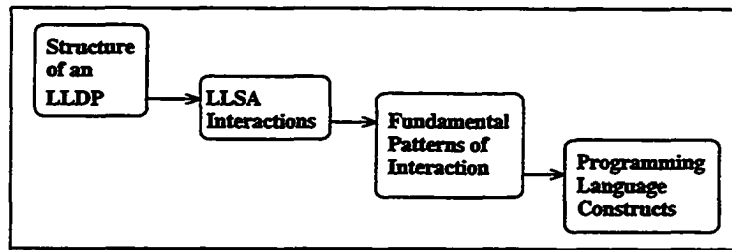


Figure 5.2: Relationship between LLDPs, LLSA, Fundamental Patterns and Language Constructs

can exhibit – (i) structure that spans the LLSA descriptions of two or more components and thus exposes hidden dependencies and (ii) structure that is confined to the interactions within the LLSA description of a single component. Both these structures are shown in figure 5.3.

5.4.2 LLDP Structure that Exposes Hidden Dependencies

Consider the LLSA graph of program P in figure 4.1 of chapter 4. Let us examine the rooted component subgraph views of component C_1 and C_5 ($RCG(C_1)$ and $RCG(C_5)$ are shown in figure 5.3 (a) and (b) respectively). The LLSA of component C_1 does not show any interaction with C_5 and vice-versa. Thus, from the LLSA representation model, it appears that there is no dependency between components C_1 and C_5 . However, an examination of the complete LLSA graph in figure 4.1 reveals an *indirect dependency* between C_1 and C_5 via the components C_3 and C_4 . In essence, it is possible for an object-oriented strategy to organize the components C_1, C_3, C_4, C_5 in such a way that there is no *direct syntactic dependency* between C_1 and C_5 and yet if C_1 is modified then C_5 is likely to be affected by the modification. Such an object-oriented strategy (LLDP) whose structure is distributed over the LLSA component descriptions of more than one component is said to incur

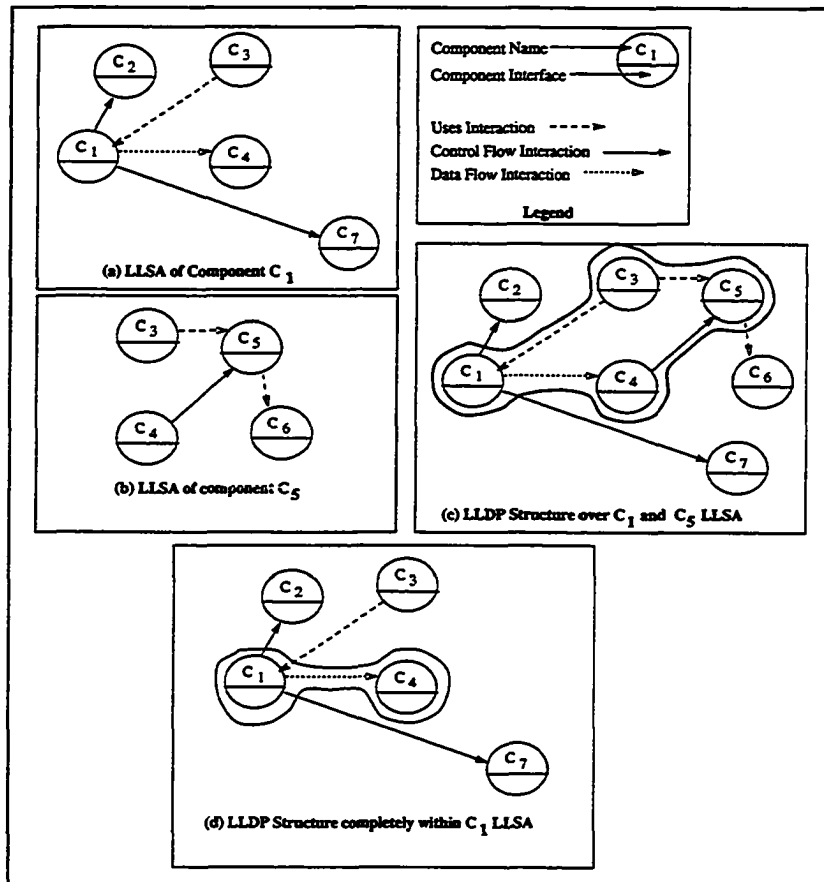


Figure 5.3: Low-Level Design Patterns Defined Over LLSA

hidden dependencies. One such example is the *polymorphism LLDP* which incurs dependencies between classes, objects and functions.

5.4.3 LLDP Structure Embedded in the LLSA Description of a Component

The structure of an LLDP can be completely embedded in the LLSA description of a single component; ie. it is possible for the structure of an LLDP to correspond to a single LLSA interaction. In such cases, the difference between an interaction and an LLDP may appear trivial but in essence the benefit of the LLDP over the interaction is that the LLDP description documents the probable *purpose* of the interaction. Understanding the rationale or purpose behind the particular organization of code is necessary in maintenance. For example, the structure of the *decoupled classes LLDP* corresponds to the *associates class interaction*.

5.4.4 Low-Level Design Pattern Template

In this section we describe the *low-level design pattern template*. The template serves as a mechanism for expressing those aspects of a technique that are useful from the maintenance perspective. This template is given in figure 5.4.

A low-level design pattern is comprised of *Name*, *Intent*, *Elements*, *Collaborations*, *Example*, *Benefits*, *Changes*, and *Consequences*. The *Name* and *Intent* sections together describe what the technique does. If the applicability of the technique is not obvious, then a situation where the technique may prove to be useful is provided in the *Intent* section. The *Elements* section of an LLDP is comprised of the components of the LLSA model. The *Collaborations* entry in the template describes the structure of the LLDP and the nature in which LLSA components

Name	Name of the low-level design pattern
Intent	Purpose of the pattern (document properties)
Elements	Entities that play a significant role in this technique.
Collaborations	Description of how the elements cooperate to achieve the objective; also give the sequence of interactions.
Example	The actual code implementing this technique
Benefits	The desirable properties of this LLDP
Changes	List of modifications on this LLDP and their impact on the elements
Consequences	An analysis of some of the negative aspects of the LLDP that complicate code understanding from the maintenance perspective.

Figure 5.4: Low-Level Design Pattern Template

and their interactions co-operate to achieve the intent of the LLDP. *Example* gives actual code illustrating the technique. The *Benefits* section attempts to explain the rationale underlying the technique and the benefits of using the technique. The benefits of a technique may be illustrated by comparing the technique with alternative strategies and comparing the merits and demerits of the strategies. *Changes* lists valid syntactic and semantic modifications that can be made and the effect of the changes on the technique. The *Consequences* section presents an analysis of the technique from the program comprehension aspect.

The *Name*, *Intent*, *Elements* and *Consequences* sections of the LLDP are language-independent aspects of the LLDP. The *Collaborations*, *Example* and *Changes* sections of the LLDP template are necessarily language-dependent. The techniques described in sections 5.4.5-5.4.7 are expounded and explained in detail (including

examples illustrating their applicability) in [GHJV94, Str91, ES92, Cop92, Ste93, Mey88, Gol83].

5.4.5 Polymorphism

An overview of the concept of polymorphism is provided in section 2.5.7.4. This section defines three different kinds of polymorphism in the context of object-oriented programming. The LLDPs are presented in figures 5.5, 5.6, 5.7.

5.4.5.1 Ad-hoc Polymorphism

The simplest kind of polymorphism is *ad-hoc polymorphism*. Ad-hoc polymorphism is defined and explained in [Weg87, Jon87, CM91, Boo93]. Ad-hoc polymorphism is the mechanism whereby the meaning (or the semantics) associated with an expression depend on the context that the expression appears in. Typically, the term *ad-hoc polymorphism* is used to denote *overloaded* language constructs. A classic example of overloading is the different programming language semantics associated with the operator '+'. In the programming language FORTRAN, operator '+' is overloaded to mean integer addition, floating-point addition, double floating addition or complex addition [Mac87]. In C++, the operator '++' is overloaded to mean post-increment and pre-increment depending on whether the operator follows and expression or precedes an expression respectively. Some programming languages allow the programmer to overload operators; examples of such programming languages are C++ and Ada [Seb93]. C++ and Ada allow one or more functions to have the same name. Consider the problem of writing a program that requires a subroutine for sorting strings and another for sorting integers. The basic sorting strategy used in the two sorting procedures is likely to be the same; two subroutines

are required because the data to be sorted is of different types – strings and integers. In such a situation, overloading procedure names is a useful technique for grouping related subroutines by giving them the same name. The subroutines are differentiated by the types of the parameters that they accept. The number of parameters, the types of each parameter of a function and the order in which the types appear form the *signature* of the subroutine. The signature of each overloaded subroutine can also include the return type of the subroutine and is unique.

When a compiler encounters a function call to an overloaded function, it resolves the call by examining the types of the parameters in the call with the signatures of each overloaded function and determining the best possible match between the call and the overloaded functions. A program reader must perform the compiler's resolution process when he or she attempts to understand code that includes a call to an overloaded function. For the human reader this is a time-consuming and difficult job that entails remembering type conversion rules, and the compiler resolution algorithm. The advantages of this technique, from the programming perspective are listed in the benefits section of figure 5.5. The effect this LLDP has on program understanding are listed in the consequences section.

Modifications that can adversely affect the structure of this LLDP are simple and avoidable. The modifications are explained in terms of the example shown in figure 5.5. Function f_1 calls function f_2 where function f_2 is an overloaded function. Suppose a maintenance request requires the code in function f_1 to be modified and one of the changes is a change in the type of a parameter that is sent to f_2 . Changing the type could result in the invocation of a different overloaded f_2 than the original one, leading to an undetected change in the behavior of f_1 and an introduction of

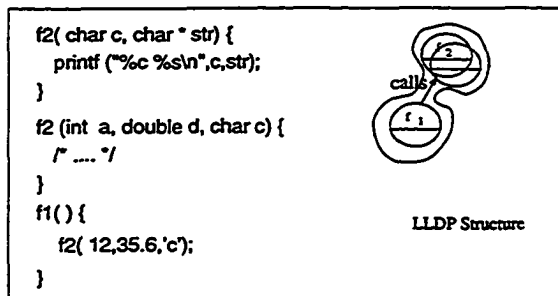
Name Overloading Procedure Names

Intent To simulate polymorphism in a statically typed language

Elements Functions f_1 and f_2

Collaborations Function f_1 calls f_2 ; function f_2 is overloaded.

Example



Benefits (i) Programmer does not have to think of new names (ii) Compiler automatically determines which overloaded function to invoke (iii) Allows procedures with similar functionality to share the same name thus aiding in readability (iv) Overloading can be used to produce the same result for parameters of different types.

Changes (i) Parameter type change changes the signature of an overloaded function (ii) Function call must be carefully coded (iii) Errors can be introduced if a parameter is misplaced or missing altogether

Consequences (i) It is difficult for a program reader to determine which function is invoked (ii) Behavior of caller can depend on the function that is invoked (iii) Ensuring that the overloaded functions have similar functionality is difficult; involves a detailed analysis of each overloaded function

Figure 5.5: Polymorphism LLDP-1

a bug. Similarly, changes in the types of the parameters of one of the overloaded f_2 functions could result in a change in the behavior of f_1 .

5.4.5.2 Polymorphism And Reuse

Consider a situation where a class has been defined and is commonly used. For example, the class implementing the concept of string, say *class String* is commonly used and string operations such as *strcmp* for comparing strings, *strlen* for obtaining the length of a string, are defined and encapsulated in the class. Now, a class must be defined that has a string as one of its data members. Let us refer to the new class as *class Person*. Class *Person* has a data member *Name* which stores the name of a person as a string of characters. One of the operations class *Person* must provide is a comparison of different objects to determine if two people have the same name. In such a situation, instead of coding the logic of *strcmp* and providing an operation for class *Person*, a more general and better solution is to treat *Person* objects as *String* objects and use all *String* operations on *Person* objects. Such a solution is feasible in C++ where conversion between objects of different classes is possible by overloading a special member function of a class – the constructor, or by providing an explicit conversion operator. In essence, class *String* can provide a constructor to convert *Person* objects to strings, or class *Person* can provide a conversion operator which converts *Person* objects to *String* objects. The structure, benefits, changes and consequences of using this LLDP are listed in figure 5.6.

5.4.5.3 Polymorphism Using Inheritance and Dynamic Binding

Polymorphism as explained in section 2.5.7.4 is very easily and directly supported in Smalltalk [Gol83]. Eiffel and C++ use rely on inheritance and dynamic

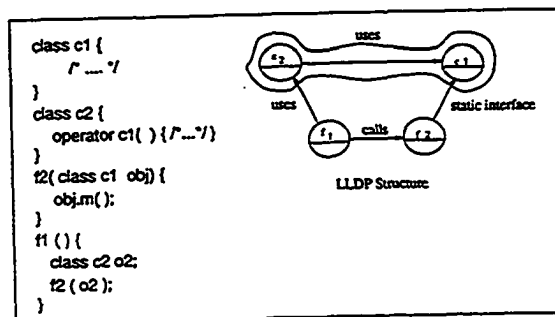
Name Reusing a polymorphic function

Intent To make an existing piece of code work on a new type with minimum modifications.

Elements classes c_1 , c_2 ,

Collaborations Class c_1 allows objects of class c_2 to be converted into objects of class c_1 .

Example



Benefits (i) The operations provided by class c_1 can be used on objects of class c_2 . (ii) The use of this technique reduces code duplication across classes (iii) The functions that accept and manipulate objects of class c_1 can do the same with objects of class c_2 . The functions do not have to be modified or changed in any way. Hence, the functions that operate on objects of class c_1 are reused. (iv) the class c_2 is in a different class hierarchy than c_1 .

Changes (i) Any changes made to operations of class c_1 affect the behavior of objects of class c_2 . One of the most drastic code change that can adversely affect objects of class c_2 is a change in the representation of class c_1 objects. The effect of such a change can be mollified if the conversion function is also modified to accommodate the new representation of c_1 objects.

Consequences (i) The use of this technique effectively allows an objects to have aliases. For example, a Person object, due to this technique, has a String alias. (ii) Employing this technique requires a detailed understanding of the type conversion rules of the language. (iii) understanding it requires an understanding of how the compiler actually implements and supports constructors.

Figure 5.6: Polymorphism LLDP-2

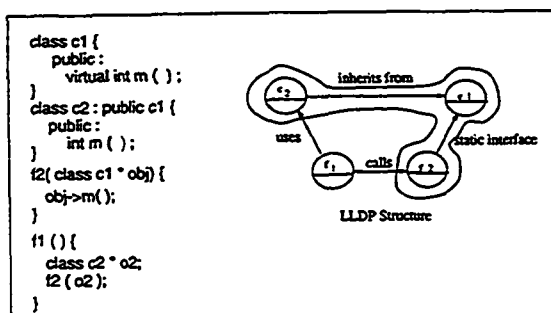
Name Polymorphism in OOP

Intent To implement a polymorphic function in C++

Elements classes c_1, c_2 , functions f_1, f_2 , object o_1, o_2

Collaborations function f_1 accepts object o_1 of type c_1 . function f_1 sends a message to o_1 . class c_2 inherits from class c_1 and accepts messages specified in c_1 's interface. Function f_2 invokes function f_1 passing object o_2 as the parameter. o_2 is an object of class c_2 .

Example



Benefits (i) The polymorphic function f_1 can accept objects of any class that derives from class c_1 . (ii) Reusing function f_1 is a simple matter of defining a new class, say class c_n that inherits from class c_1 and class c_n redefines method m .

Changes (i) Dynamic binding can be lost by the removal of a keyword thus introducing a bug and changing the behavior of the system. (ii) Because of the limitations in the language, f_1 must accept pointer objects to exploit dynamic binding. (iii) If f_1 is modified to accept objects of class c_2 instead of c_1 , the function will be restricted to accept objects of classes that derive from class c_2 instead of class c_1 .

Consequences (i) For derived classes that use multiple inheritance, determining the actual method that is invoked is a difficult task that involves examining all the classes in the inheritance hierarchy and using the semantic rules of the implementation language to resolve the method.

Figure 5.7: Polymorphism LLDP-3

binding to support polymorphism. Consider a function, *Display* whose purpose is to draw an object of any shape on the screen; ie. function *Display* should be polymorphic and should be able to accept objects of any shape. This function takes in an object of *class Shape* and sends the message *draw* to the object and the object draws itself on the screen. In order to implement such a function in C++ or Eiffel, the program must make proper use of inheritance and dynamic binding. Class *Shape* represents a general abstract class, and class *circle* and class *square* derive from class *shape*. Class *Shape* includes the draw operation in its interface but the actual implementation (or definition) of the draw operation is deferred to classes that derive from class *Shape* by using a special keyword (*virtual* in C++ and *rename* in Eiffel). Hence classes *circle* and *square* each define the draw operation to draw their characteristic shapes. Type conversions between derived and base classes in C++ and Eiffel allow objects of class *circle* and class *square* to be treated as objects of class *Shape*. Hence when *circle* or *square* objects are passed to function *Display*, the draw function invoked on these objects is determined by the type of the object due to dynamic binding. This LLDP is shown in figure 5.7.

5.4.6 Decoupling

As explained in chapters 1 and 2, a software system exhibiting low coupling is considered to have a good design. This section discusses LLDPs that demonstrate when decoupling of components leads to a flexible design and the benefits and drawbacks of decoupling. The decoupling LLDPs are presented in figures 5.8, 5.9, 5.10.

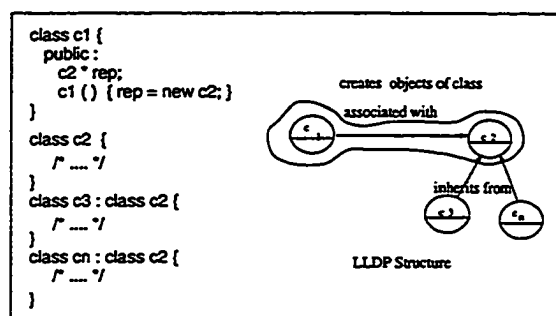
Name Decoupling A Class From its Representation

Intent To allow flexible data-structure organizations

Elements Classes c_1, c_2

Collaborations Class c_1 is associated with c_2 and c_1 creates objects of c_2 . For every object of class c_1 , an object of class c_2 (or any class that derives from c_2) is created. Representation is a private data member.

Example



Benefits (i) Customizable data representation (ii) Objects of class c_1 are not restricted to a single representation; instead they can choose from the representations offered by class c_2 and its descendant classes.

Changes (i) If the representation is changed from a pointer object to a simple object, this technique is destroyed. (ii) In the interests of encapsulation and information hiding, the representation must be private to class c_1 . (iii) The representation is typically created by a method of class c_1 , but it is possible for it to be created outside class c_1 . Creating the representation in class c_1 is a logical design strategy.

Consequences (i) The presence of this technique complicates code understanding because it is not easy to determine the representation associated with an object of class c_1 . (ii) After understanding the concept of class as being a specification for the common structure and behavior of similar objects, this technique forces one to accept that similar objects can have different structures! (iii) Due to this technique the behavior of an object will be partially determined by its representation. Understanding the behavior of objects and consequently the system is further complicated by the use of this technique.

Figure 5.8: Decoupling LLDP-1

5.4.6.1 Decoupling a Class from its Representation

Consider a situation where three possible data structures are being tested to determine their effectiveness. One way to perform such a test would be to implement three classes each with their own data structure representations and test each class. A better and more general approach would be to decouple a class from its representation and create three objects such that each object had its own data structure representation. Decoupling the representation of a class from the class is referred to as *object composition* in [GHJV94].

This technique involves classes c_1 , and c_2 such that class c_1 contains a pointer object of class c_2 . Whenever an object of class c_1 is created, a corresponding object of class c_2 or descendant classes of c_2 are created. This enables objects of class c_1 to have dynamic and different representations.

5.4.6.2 Decoupling for Flexible Design

Object oriented analysis and design often establish an association relationship (see sections 1.1.2 and 2.6 for an explanation of association) between classes. As a consequence of this relationship between classes, objects of the classes can connect with each other dynamically. Consider a software system that is designed to handle exceptions. For each kind of exception there is an associated exception handler. Every time an exception occurs, the appropriate exception handler is invoked to cater to the exception. An object-oriented design of such a system would have a class *Exception* and another class *ExceptionHandler*. Assume there are three kinds of exceptions; correspondingly there are three kinds of exception handlers. There can be any number of exception *objects* in this system; but exactly three exception handler objects. Each exception object will be associated with its corresponding

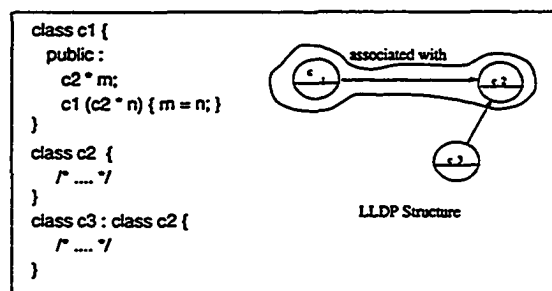
Name Associative Classes

Intent To support the dynamic association of objects

Elements Class c_1, c_2

Collaborations class c_1 *associates with* c_2

Example



Benefits (i) Ensures a decoupled and flexible design of classes, allowing weakly related classes to be associated with another. (ii) Allows objects to connect with each other at run-time. (iii) This technique allows a system to be easily extended because the classes are decoupled.

Changes (i) If the data member is changed from a pointer object to a simple object, this technique is destroyed. (ii) In the interests of encapsulation and information hiding, the representation must be private to class c_1 .

Consequences (i) It is possible for more than one object of class c_1 to be associated with a single object of class c_2 . The object of class c_2 can therefore be manipulated via any of the objects of c_1 . This complicated code understanding because it is difficult to predict the behavior of the object of class c_2 . (ii) The destruction of the object of class c_2 is an event that must be broadcast to all objects that are associated with it so that they realize that the services of the destroyed object are not available. (iii) Understanding the design rationale underlying some of the association relationships between classes is a non-trivial task if the association relationships are different from the ones that the program reader imagines they should be.

Figure 5.9: Decoupling LLDP-2

exception handler object. The creation of the `ExceptionHandler` objects occurs independent of the creation of `Exception` objects. The structure of this LLDP (shown in figure 5.9) appears to be similar to the structure of the Decoupling LLDP-1 shown in figure 5.8. The difference between these two techniques is that there is only one interaction between class c_1 and class c_2 , namely, the *associates with* interaction whereas in the previous technique, there are two relationships between the classes – *associates with*, *creates objects of class*.

5.4.6.3 Decoupling a Function from a Class

Functions in object-oriented languages can accept objects. The interface of the class of the object determines the messages that can be invoked on the object. An object-oriented strategy that allows flexible, extensible and decoupled code to be developed is to design a general class (also called an abstract class) whose interface reflects the most general services that may be expected from objects of that class or objects of classes that derive from it. Functions are then defined to accept objects of the general class so that the function components are decoupled from specific classes and restricted to interact only with the most general classes. Such a strategy ensures that changes in class design do not affect functions. Changes in the interface of the general class will affect functions. This LLDP is shown in fig 5.10.

5.4.7 Messages

Message passing is a significant feature of the object-oriented paradigm. Object-oriented analysis and design methodologies have developed notations to represent message passing between objects and classes. The message passing model supported

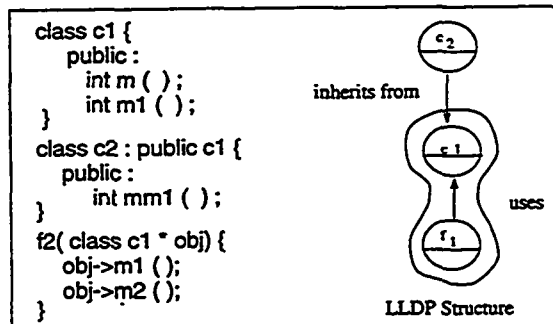
Name Decoupling a function from child classes

Intent To reduce dependencies and enhance low-coupling

Elements Function f_1 , class c_1

Collaborations At run-time, o_2 is an object of class c_2 . f_1 accepts a pointer to class c_1 as a parameter. *function sends message to o_2* . The method that is actually executed as a result of the message is in class c_2 and not in class c_1 .

Example



Benefits (i) Function f_1 deals with class c_1 only; ie. the messages that f_1 can send are specified in the interface of class c_1 . (ii) Function f_1 is decoupled from the classes that actually receive and act upon the messages.

Changes (i) If Function f_1 's parameter type is changed from c_1 to c_2 , function f_1 becomes less general. Such a change may make the additional services of class c_2 available to f_1 , but this comes at the cost of f_1 being restricted to accept objects of class c_2 and classes that derive from c_2 whereas previously f_1 was capable of accepting objects of class c_1 and classes that derive from c_1 which is a larger hierarchy than the one rooted at class c_2 .

Consequences (i) This technique enforces the rigid interface of the general class c_1 on functions and obscures the specific class that the function actually interacts with. Once again, determining the actual class and method that is used requires understanding the semantic rules of the implementation language. (ii) The general interface of class c_1 is often a union of the interfaces of the classes that derive from it. In the case of a badly designed class hierarchy this results in a cumbersome and complex interface. Understanding the class, the interface and the purpose of each method becomes a difficult task.

Figure 5.10: Decoupling LLDP-3

by object-oriented programming languages such as C++, Eiffel and Smalltalk is more primitive than the message passing models used in the analysis and design stages. Consequently, programming techniques bridge the gap between the more powerful object-oriented design message passing model and the primitive implementation language message passing model. The LLDPs in figs 5.11, 5.12 and 5.13 describe some of these techniques.

5.4.7.1 Messages Between an Object and a SubObject

The simplest message exchange between objects is the communication that is possible between the object of a parent and an object of a child class since they share attributes and data members. A message is effectively communicated to a parent object by changing the contents of a shared attribute; i.e. the normal mechanism of invoking operations on an object is bypassed and the shared attribute is directly modified. There are several uses for this simple technique that is directly supported in Smalltalk, C++ and Eiffel. The structure of this LLDP is simple because the technique is directly supported by most implementation languages. This simple message-LLDP is shown in fig 5.11.

5.4.7.2 Messages Between Objects of a Class

Objects of the same class can communicate with each other in a special way in C++. The *static data member* feature of C++ allows objects of a class to share data members. An object can therefore read and write information to the shared data member that can be accessed and used by other objects of the same class. This LLDP is shown in fig 5.12.

Name Communication between parent and child objects

Intent To let the child get control of the parent object

Elements object o_1

Collaborations none


Example

```

class c1 {
  public :
    int data_mem;
}
class c2 : public c1 {
  public :
    int method () { data_mem = 10; }
}
f2 () {
  c2 o1;
  o1.method ( );
}

```

LLDP Structure



Benefits (i) An object with multiple roles can communicate via the parent object
(ii) This technique can be used for debugging purposes to keep track of the state changes in the object.

Changes none

Consequences (i) This technique complicates program understanding because the location of the modified data member requires a search through the complete inheritance hierarchy. Once all definitions of the data member are located in the hierarchy, the resolution algorithm of the implementation language must be used to determine precisely which data member is modified.

Figure 5.11: Messages LLDP-1

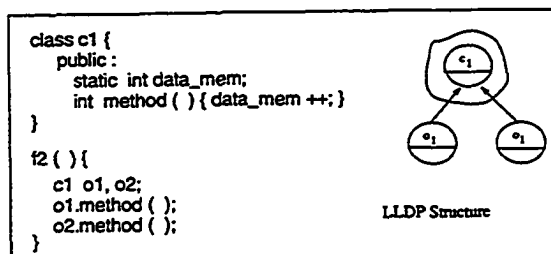
Name Communication between objects belonging to the same class

Intent Broadcast to similar objects

Elements Classes c_1 , objects o_1, o_2

Collaborations c_1 has a static data member and o_1, o_2 are objects of class c_1 .

Example



Benefits (i) Broadcasting an event by changing the contents of the shared data members. (ii) Since data is shared between objects, this technique helps in saving space. (iii)

Changes (i) The keyword static must be associated with the data members. In the absence of this keyword, the data members are no longer shared between the objects and changes made to the data members are local to an object.

Consequences none

Figure 5.12: Messages LLDP-2

5.4.7.3 Messages Between Objects of Different Classes

There is no direct syntactic construct for supporting message-passing between objects. Objects are capable of receiving messages but not dispatching messages. In order to establish communication between objects, a message must be sent to one of the objects, o_1 , with the second object, o_2 as a parameter. The message received by o_1 must in turn send a message to o_2 with o_1 as a parameter. The LLDP is shown in figure 5.13. This technique is referred to as *double-dispatch* in [GHJV94] because the message is accepted by two objects. In *multiple dispatch*, [GHJV94] multiple objects accept the message. The multiple-dispatch (and hence double-dispatch) technique is directly supported in the object-oriented language CLOS (Common Lisp Object System). Smalltalk and C++ directly support the single-dispatch technique where a single object accepts a message. This LLDP describes how double-dispatch can be simulated in programming languages that have only single-dispatch.

5.5 Summary

The notions of pattern languages, design patterns and idioms in the context of object-orientation were introduced in this chapter. A low-level design pattern (LLDP) is defined to represent an object-oriented technique. The relationship between the structure of an LLDP and LLSA interactions is explained and defined in section 5.4.1. The three sets of LLDPs – polymorphism, decoupling and messages are defined are commonly used object-oriented strategies that enable a software system to have the desirable design properties of being flexible, extensible, reusable and decoupled. The significance and usefulness of LLDPs from the maintenance point of view is examined in the next chapter.

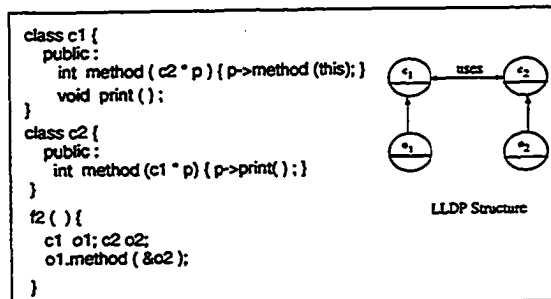
Name Message passing between objects (double dispatch)

Intent To allow communication between objects

Elements Objects o_1, o_2 , class c_1, c_2 , function f_1

Collaborations f_1 sends a message to object o_1 ; the message includes o_2 as a parameter. The invoked message sends a message to object o_2 , passing object o_1 as a parameter.

Example



Benefits (i) This technique models communication between objects and is therefore useful in the translation of design diagrams to implementation code if the diagrams depict communicating objects. (ii) This technique can be extended to support two-way communication between objects. (iii) This technique can be generalized to include multiple participating objects.

Changes (i) If the message to the parametric object is removed from the method of class c_1 , this technique is destroyed. (ii) Establishing a friendship relationship between the classes enables the classes to use more member functions in each other's interfaces, thereby enhancing the usefulness of this technique.

Consequences Very difficult to understand this elaborate scheme which is doing something really simple.

Figure 5.13: Messages LLDP-3

Chapter 6

Using LLSA and LLDP for Maintaining OO Systems

The purpose of the representation models presented in chapters 4 and 5 is to aid in understanding object-oriented software systems from the maintenance perspective. This chapter addresses issues that complicate the process of understanding object-oriented programs and expounds on how LLSA and LLDPs serve as an aid in understanding object-oriented programs. The usefulness of LLSA has been briefly touched upon in sections 1.3.3 and 4.5. The direct and indirect benefits of the LLSA and LLDP models are explained in detail in sections 6.1 and 6.2 respectively.

6.1 LLSA as an Aid for Software Maintenance

The various ways in which the LLSA representation model aids in understanding the structural aspects of an object-oriented system from the maintenance perspective is explored in this section. The usefulness of the LLSA model with respect to code organization and the static and dynamic structures in a system are addressed in section 6.1.1. Sections 6.1.2 and 6.1.3 discuss the applicability of LLSA in the areas of understanding the role of a component and code navigation respectively.

6.1.1 Using LLSA for Understanding the Structure

As explained in 1.1.2, the structure of a software system has two aspects – organization and relationships. The LLSA component descriptions aid in understanding the logical and physical organization of code as well as some of the static and dynamic structure of the system.

6.1.1.1 Logical and Physical Organization

The physical organization (or allocation) of code is apparent from an examination of the multiple files of a software system. The physical organization of code provides a rudimentary view of the structure of the system. An implementor is faced with the difficult task of physically organizing code such that a combination of compilation dependencies, uses dependencies, and logical dependencies between components is accurately reflected by the physical allocation of code. The dependency relationships often conflict with one another and eventually the implementor is constrained to satisfactorily represent a subset of the dependency relationships in the physical organization.

For example, an implementor may decide to group functions *A*, *B* in the same file because function *A* calls function *B*. In this case, the physical organization is modelling a calling relationship by allocating a file for each function and the functions it calls. Alternatively, another implementor may decide to make use of the separate compilation feature available in object-oriented languages and separate the two functions in order to minimize compilation dependencies. In the second organization, the allocation of code reflects compilation dependencies (ie. code fragment *A*, *B* are allocated to the same file if code modifications to *A* require *B* to be recompiled). In essence, the physical organization of code is not a useful

aid in understanding the logical structure and dependencies of a system. In the absence of a suitable abstract representation model, a maintainer is constrained to examine and understand the components of the software system by using the physical organization as an indicator of logical organization of code.

The LLSA abstract representation model overcomes the shortcomings of physical structure by presenting the logical structure of the system in terms of interacting logical components that correspond directly to object-oriented design concepts, such as class, function and object. Moreover, the explicit listing of logical relationships between the components, as described in each LLSA textual description template, aids the maintainer in grouping and understanding logically related components. The LLSA model provides a maintainer with a view of the logical structure of the system which enables the maintainer to obtain a clear picture of logical dependencies between components. The LLSA model therefore overcomes the rigidity imposed by physical organization and displays the embedded low-level logical design of a system. The LLSA of a software system lists the name of each component, classifies it as a function, a class or an object, lists the modules in which the component is used, and the names of the components that a component interacts with. The logical organization in an object-oriented system can be inferred by grouping components related by the LLSA interactions.

6.1.1.2 Static and Dynamic Structure

There are many static and dynamic substructures (or views) in an object-oriented software system. The class inheritance hierarchies, class diagrams modelling part-of or associative relationships, call-graphs modelling the call structure of a system, are all examples of static substructures in an object-oriented system.

Each of these static substructures enhance the understanding of the logical, object-oriented decomposition and design inherent in the system.

The automatic construction of call-graphs and class relationship graphs from source code is a non-trivial task and their contributions to maintenance have been stated in sections 1.1.1, 3.6, 3.7. The LLSA representation model permits the automatic construction of several views of software structure. Because the information content in the model is rich in terms of object-oriented relationships, several views can be provided by the simple means of performing a transitive closure operation (transitive closure is defined in section 4.2.2.1) on an LLSA interaction. Each static substructures can be constructed by performing a transitive closure operation on the appropriate LLSA interaction. Table 6.1 lists the substructures and classifies them as providing a static or dynamic view of the code. The last column provides the name of the LLSA interaction over which a transitive closure must be performed.

The call-graph structure of an object-oriented system can be constructed by performing the transitive closure operation over the *Calls Function* interaction in the LLSA description of function components. Some of the views require transitive closures of more than one interaction. For example, to obtain the class diagram, the transitive closure operation must be performed over several LLSA interactions of the class component.

The class structures depict the static structure and associations between components of an object-oriented software system. Objects and the messages sent to messages truly reveal the dynamic behavior of an object-oriented system. The objects that are present in a system are difficult to identify, especially objects that have a short-lived dynamic existence. The object creation view of object-oriented

Table 6.1: View Construction from LLSA Representation

View	Components	Classification	LLSA Interaction
Call Graph	Functions	Static	Transitive closure of <i>Calls Functions</i> in Function Component
Inheritance Hierarchy	Classes	Static	Transitive closure of <i>Descendant Classes</i> in Class Component
Object Creation	Classes, Functions	Dynamic	Transitive closures of <i>Creates Objects of Classes</i> in Class & Function Components
Class Diagram	Classes	Static	Transitive closures of <i>Container Classes, Friend Classes, Associated With Classes, Uses Members of Classes</i> of the Class Component

systems pinpoints the creator of all objects that may come into existence dynamically. This object creation view is an invaluable aid for maintenance because it gives some insight into the dynamic aspects of the system and the class organization of the system.

Some other views that can be constructed are views obtained by simply clustering similar components together and examining each cluster of like components separately. This is similar to the layered view of a software system discussed in chapter 4.

6.1.2 Using an LLSA Description to Understand a Component

Understanding the purpose and role of a component of a system is a necessary subactivity in the process of understanding the structure and behavior of a system.

The purpose of a component can be partially gleaned by a thorough examination of the actual code associated with the component. If the component makes use of other components, then a thorough understanding of the component under consideration requires an examination of the associated components as well. Locating and identifying the associated components in the case of object-oriented software systems is a non-trivial task especially in the presence of features such as overloading and dynamic binding.

The LLSA textual description of a component corresponds to the rooted subgraph view of the component. The rooted subgraph has the following interesting characteristics :

1. There are arrows directed away from the root component (we shall refer to these arrows as *outward* arrows). The outward arrows represent the components that the root component uses or depends upon in some way. An examination of the components used by the root component may be required by maintainer in order to gain a clearer picture of the functionality of the root component. The information represented by the outward arrows is also useful in determining the effect that modifications on these components has on the root component.
2. There are arrows directed into the root component (we shall refer to these arrows as *inward* arrows). The inward arrows represent the components that use or depend on the root component in some way. An examination of the components that use the root component establish a *context-of-use* (a term defined in [WMH93]) for the root component. The context-of-use of a component gives insight into the role the component plays in the overall structure of

the software system. The information represented by the inward arrows is also useful in determining the components that will be affected by modifications on the root component.

6.1.3 Code Navigation

The classification of code fragments as function, class and object components as well as the LLSA description associated with each component helps in code understanding and code navigation. The LLSA description of a component lists the names of interacting components. This inclusion of the names of components provides the maintainer with a means of navigating by simple name-lookup techniques. Essentially, a maintainer reading an LLSA component description can use the name of associated components to examine the LLSA component description of the associated component. The navigation facilities of the LLSA representation model are more general than those provided by browsers (see section 3.7) because the LLSA model allows navigation between different kinds of components whereas browsers typically restrict themselves to one kind of component, either class or function.

6.2 LLDP as an Aid for Software Maintenance

The LLSA model aids in understanding the logical structure of an object-oriented software system. The logical structure of a system aids in understanding the design of the system. Another important aspect of understanding the design of a system is understanding the reason *why* components relate to each other. The process of understanding a software system includes understanding the original requirements and the analysis and design that went into the creation of the software

solution. As a result, a maintainer often develops a mental model of what the design of the software system should be and how the components should interact [RW88, RW90, RC93]. The maintainer's mental model may be quite different from the actual existing design of the software system. One of the difficult tasks for a maintainer, from a psychological viewpoint, is to discard the design that he/she would like the system to have and accept the existing design. Accepting the existing design becomes easier if the benefits and the rationale of the strategies and techniques employed in the existing design are explained to the maintainer. The purpose served by LLDPs is to document existing object-oriented strategies, primarily so that a maintainer can build his/her awareness of these techniques, and secondarily so that the maintainer can evaluate and improve upon the techniques as well as develop more techniques.

Often, a maintainer must analyse code to evaluate it from the perspectives of extensibility, reusability, flexibility, modularity in order to make decisions regarding maintenance requests. Such evaluations require the maintainer to understand and evaluate the design of the system. In order to perform a meaningful evaluation, a maintainer should be aware of commonly used programming techniques that are used to transform the design of a software system to actual code. In essence, the evaluation and analysis of a software system requires programming experience. The LLDPs presented in chapter 5 concisely explain object-oriented techniques, their benefits and consequences. The information content of an LLDP is useful for evaluating the usefulness and applicability of the technique.

It is impossible to predict the rationale behind every inter-component relationship. However, with the help of LLDPs, a maintainer is able to understand and

identify certain commonly occurring composition of relationships between components. LLDPs such as the polymorphism LLDPs, explain the reason why and how class, function and object components cooperate in order to achieve certain desirable properties in a software system. LLDPs therefore enable a maintainer to understand groups of components. The difference between the LLSA model and the LLDP model is that in the LLDP model, a group of components is viewed collectively as a single unit, whereas a component is a single unit in the LLSA model. Effectively, LLDPs abstract a group of components from the LLSA model and this is the reason that the LLDP model is depicted as a model containing a higher degree of abstraction than the LLSA model (see figure 1.3).

6.2.1 Using LLSA and LLDP in Code Modifications

Subsequent to the processes of understanding, analysing and evaluating source code and the maintenance request, a maintainer may decide to actually modify code. The various subactivities involved in code modification and the ways in which LLSA and LLDPs aid each subactivity are examined below. Code modification is classified as *(i) addition of new code, (ii) deletion of existing code, (iii) modification of the structure of existing code such that the original logic and meaning is preserved*. A maintainer must determine whether code needs to be added, modified or deleted. The performance of a maintenance request may often involve all three. The subactivities of code addition, deletion and modification separately are each described below.

1. *Addition of Code* : The addition of new code is a decision that is arrived at after the maintainer has ascertained that none of the existing code can be reused for the purposes of satisfying the maintenance request. The addition

of new code requires an evaluation of the best strategy to be employed by the maintainer such that the logical distribution of functionality in the system and the functionality of individual components are preserved. The LLSA serves as a means to compare the logical structure of the system before the addition of code and after the addition of code. The LLDPs serve to check if the structure of a technique in the existing code is broken as a result of adding new code.

A maintainer must also determine which components of software are affected by the addition and which components must be recompiled in order to integrate the new code into the system; the LLSA descriptions aid the maintainer in determining this information. The LLSA of the original system and the system with new code can be compared to determine if the static or dynamic interfaces of components have changed or if the static and dynamic interactions of a component have changed. Any change in the LLSA description of a component indicates that the behavior of the component has changed. The maintainer can examine the LLSA description to determine if the change has occurred in a desirable or undesirable way.

Lastly, the maintainer may use one or more LLDPs in the new code; ie. the new code to be added may incorporate an existing technique.

2. *Deletion of Code* : Code is deleted if it is determined to be unused, redundant or if a maintainer determines a cheaper and more effective solution. Deletion of code usually spawns compilation discrepancies and requires a careful readjustment of code that depends on the deleted code. In the case of unused code, deletion can be performed easily. In the case of components that depend on

the code to be deleted, the LLSA model is useful in determining the components that should be modified and recompiled because of the deletion of code. LLDPs can be used to determine if the deletion results in the deformation of a technique in the code.

3. *Modification of Code* : Code modification is an activity that spans changes such as simple name replacement in code (ie. renaming a variable or a function) to reordering of code for purposes of enhancing readability or to optimize the code. The purpose of this activity is to enhance the system's capabilities and properties without affecting its functionality. Code modification can easily be viewed as an activity comprised of the deletion of code and then the replacement of the deleted code by new code that has the required modifications. The ways in which LLSA and LLDP can be used in code modification are exactly the same as the ways in which they are used in the addition and deletion of code.

6.3 Summary

This chapter demonstrates the many uses of both the abstraction models, LLSA and LLDPs from the maintenance perspective. The LLSA model serves as a useful aid in modelling and understanding the structure of an object oriented system. The LLSA component textual description contains information that enables the maintainer to understand the role, the functionality and the dependencies of the component. LLDPs aid in providing the maintainer with some insight into the rationale behind the low-level relationships between the components. LLDPs also serve to enlighten the novice or unaware maintainer about some of the existing

techniques in the field. Apart from aiding the understanding process, LLSA and LLDPs can also be used in the actual modification of code. The next chapter discusses the reverse engineering processes consisting of the extraction of the LLSA of an object-oriented system and the identification of LLDPs in the system.

Chapter 7

Reverse Engineering LLDPs

As explained in section 2.5.3, the focus of the reverse engineering process is to aid program understanding. In order to meet this objective, a reverse engineering effort must address the two central issues in reverse engineering, namely, knowledge representation and automated extraction of the knowledge representation model. The two issues will be referred to as (i) the representation problem and (ii) the automation problem [CC90, Big89, RC93].

In [RC93], Rugaber and Clayton distinguish between *mental model* and *representation*. This distinction is useful in understanding the relationship between the LLSA model and the LLDP model presented in chapters 4 and 5 respectively. A mental model represents the informal insights, knowledge and comprehension processes that a programmer employs in understanding code. The objective of a mental model is to explicitly represent the comprehension activities of a programmer [RC93]. In contrast, the focus of a representation model is the design of the results of the comprehension activities. In essence, the concerns of the mental model are the explicit representation of mental processes, whereas the concerns of the representation model are the representation of the results of the mental processes. The LLDP representation model is a representation of the informal strategies and techniques used by programmers to develop and understand object-oriented software.

The objective of the second issue, the automation problem, is to ensure that the representation model designed as part of a reverse engineering process is automatically extractable from source code [Pre92]. Program slicing [GL91], code analysis and dependency analysis [JOR92, LC93, LR92, WHH89] techniques and algorithms are an outcome of the automation effort in a reverse engineering project. The term *code analysis* will be used in a generic sense to collectively refer to program slicing, code analysis or dependency analysis.

Code analysis algorithms are designed by investigating the information content of the representation model, investigating the syntax and semantics of the programming language used in the implementation of the software system, and analysing the correspondence between the two. The correspondence between the information required by the abstract representation model (the LLSA and LLDP models for example) and the implementation language of the software system (C++ for example) reveals the information that can be correctly inferred and automatically obtained from source code. The next step in devising code analysis algorithms consists of devising strategies and methods to establish the correspondence between the two kinds of information. One of the outcomes of the code analysis algorithm development phase is the identification of the information content of the representation model that can be automatically extracted and that which cannot. The code analysis phase therefore serves to modulate the information content and the level of abstraction in the representation model.

Both, the extraction of components and the abstract representation of information are non-trivial aspects of reverse engineering. Chapters 4 and 5 defined two representation models of object-oriented software systems. This chapter discusses the issues involved in the automatic extraction and analysis of information from

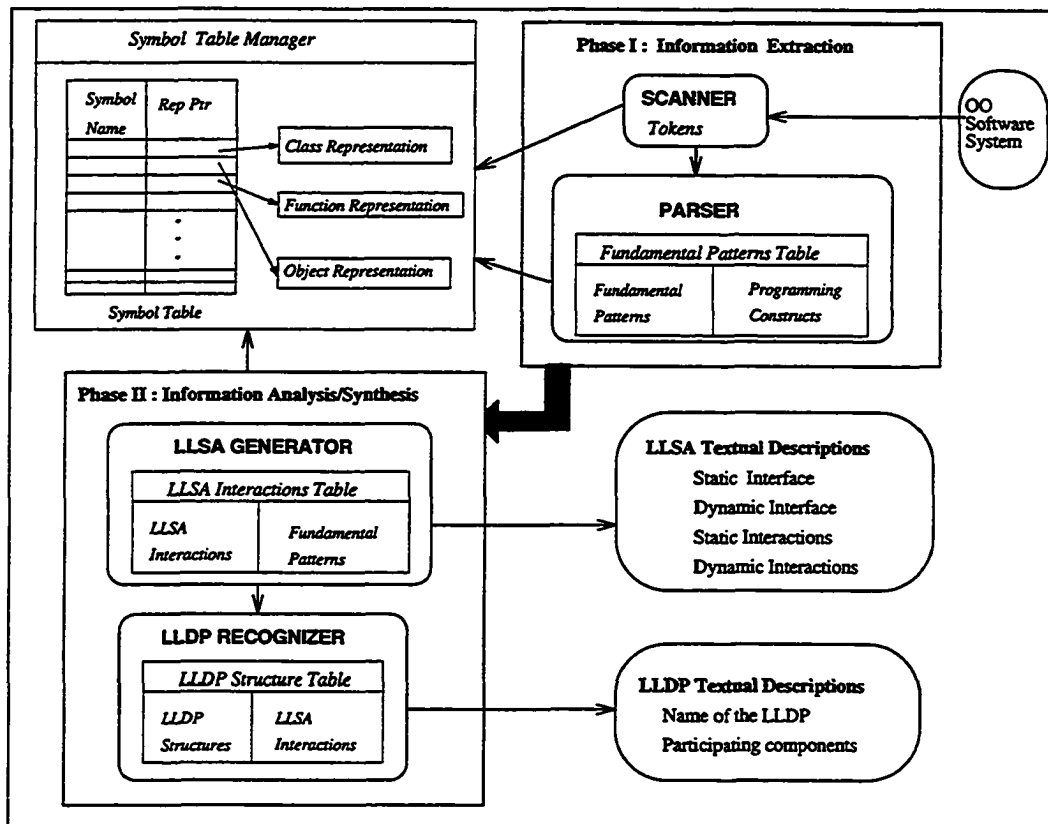


Figure 7.1: Overall Architecture of pulse

an object-oriented software system. A prototype software tool, *pulse*, developed as part of this research work, extracts the LLSA of a system and recognizes LLDAPs in the system. The architecture, subcomponents, algorithms and data structures in *pulse* are described in the following sections.

7.1 Overall Architecture of pulse

The reverse engineering process (see figure 7.1) employed in the design of *pulse* consists of two phases. In phase I, information is extracted from source code, and stored in a symbol table. In phase II, the LLSA textual description of each

component is generated, and the components that collaborate to implement an LLDP are identified. In phase I, the source code is scanned and parsed and the information obtained from these two activities is stored in the symbol table. The parser recognizes certain syntactic constructs and updates the appropriate symbol table entries and the data structures associated with the symbol table entry with the information recognized. This correspondence of syntactic constructs and associated information is described in the form of a table, called the *fundamental patterns correspondence table*. This table is useful for understanding the parser and its actions. The scanner and parser are necessarily language-dependent components of pulse.

The purpose of the analyser and synthesizer is to compute the static and dynamic interfaces of each component identified by the extractor, to establish the static and dynamic interactions between the components and to recognize LLDPs in the code. The activities comprised of the generation of LLSA descriptions and LLDP recognition are performed after all the source files have been scanned and parsed and all relevant information has been extracted. The second phase is decoupled from the first phase. The LLSA generator computes the static and dynamic interfaces of each component, in some cases using algorithms similar to the semantic-checking algorithms of a compiler. In order to generate the static and dynamic interactions, the LLSA generator uses an *LLSA interactions correspondence table*. The LLDP recognizer uses the information contained in the interface of a component and a *LLDP structure correspondence table* to recognize the structure of an LLDP. The components that actually collaborate with each other to implement the technique are also recognized by the LLDP recognizer. The LLSA generator and LLDP recognizer are largely language-independent.

The symbol table is an important data structure that stores information about symbols identified in the source code. This data structure is explained in section 7.2. The correspondence tables serve to document the rules used in pulse for the extraction, analysis and synthesis of information. Each table is described in the relevant section.

The output of pulse consists of LLSA textual descriptions of each component and the names of LLDPs along with participating components. A separate document consisting of all the LLDPs and their descriptions is provided to the maintainer.

7.2 Symbol Table Organization

The data structure organization of the symbol table is shown in figure 7.2. The symbol table is a simple array of symbol entries. Each entry in the table has three fields. The first field stores the name of the symbol, the second field the nature of the symbol (ie. function, class, object). The information associated and stored with a symbol depends on the nature of the symbol. For example, a function has a parameter list associated with it, whereas a class or an object symbol does not. Due to the varying nature of information associated with each symbol, the third field in the entry, called a representation pointer, is dynamically allocated to store information relevant to the symbol. The third field is called the representation field. The representation information stored with a class component consists of lists of the names of different kinds of symbols. For example, the field *Objects* stores the names of objects that are instances of this class and the field *Bases* stores the names of base classes. In some cases, the list is more complex. For example, the information

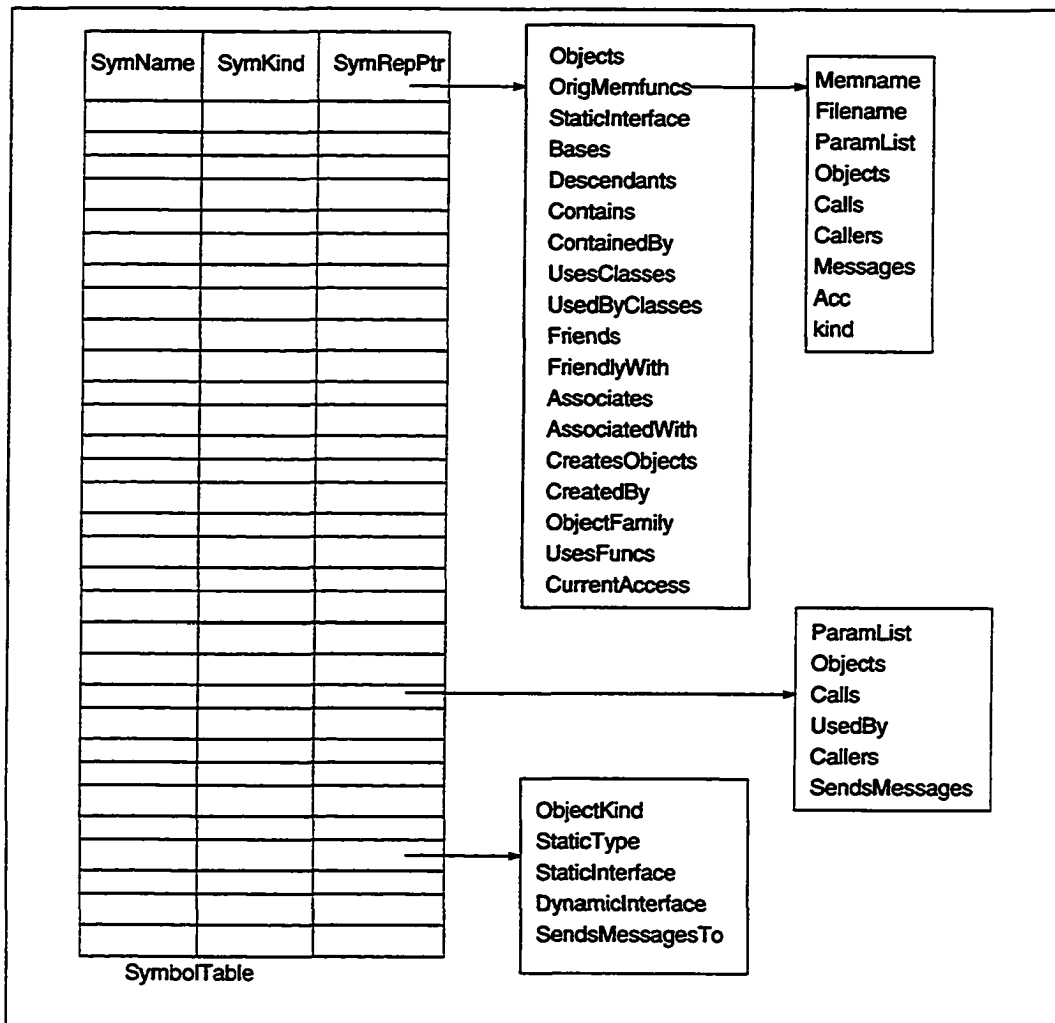


Figure 7.2: Symbol Table Data Structure

associated with a single member function consists of the name of member function, the name of the file in which it appears and the parameter list associated with the member function. Each item in the list of member functions has information pertaining to a member function.

7.3 Scanner

The scanner is the component that actually reads in the characters from the source code and forms tokens by using the rules specified by the language in which the source code was implemented. A token is a sequence of characters delimited from other characters by certain syntactic rules. The purpose of the scanner is to correctly recognize tokens in the input. The scanner used for pulse was automatically generated by the scanner generator software tool *lex*. Lex [LS86] accepts a specification of tokens and generates a finite deterministic automaton that recognizes the specified tokens. The specification of C++ tokens is available in [ES92]. The tokens formed by the scanner are used by the parser.

The scanner is responsible for identifying and adding identifiers in the source code to the symbol table. The scanner maintains context-sensitive information to determine the scope that is started by a syntactic construct. Scope information is used by the parser to update the appropriate data structure. The scanner is quite primitive and does not recognize comments or preprocessor directives.

7.4 Parser

The parser component of pulse recognizes syntactic constructs according to the syntax rules of C++. The sequence of tokens supplied by the scanner are matched

against syntax rules, the sequence is accepted as a valid construct or rejected, and an error is reported. The parser itself is generated by the parser generator tool YACC, which is an acronym for yet another compiler compiler. The generator accepts a grammar specification as input and produces a push down automata based bottom-up left recursive (henceforth abbreviated as LR) parser [Joh86]. The YACC tool used for the implementation of pulse was an adapted version of Berkeley YACC. This adapted version has a special debugging feature which was invaluable in the implementation of pulse. This feature is particularly useful in designing the actions that enable the incremental extraction of information.

The specification rules specify the structure of syntactic constructs. YACC allows actions to be associated with rules. This option is useful in the incremental extraction and storage of information and construction of data structures. The activities of the parser component of pulse are divided into the following categories:

- *Maintaining Context (Scope)*: A significant amount of information that pulse extracts is *context-sensitive*; ie. the nature of the information depends on the context in which it appears. To illustrate, the expression *obj* → *method*(), when appearing in the body of a function, is interpreted to mean that the function sends a message to an object. The same expression, when appearing in a class, is interpreted to mean that the class uses the interface of another class. Due to the context-sensitivity of the information, the parser must keep track of the context in which a construct is parsed. The parser makes use of variables *CurrScope* and *PrevScope* to identify the current and previous scopes. There are five possible scopes *CLASS-SCOPE*, *FUNC-SCOPE*, *MEM-FUNC-SCOPE*, *GLOBAL-SCOPE*, *FILE-SCOPE*. The file *enums.h* contains an enumerated type that specifies the different scopes.

- *Incrementally Extracting Information* : The information that is required for the construction of the LLSA descriptions is dispersed over the source code and often over different syntactic constructs. One of the difficult aspects of the parser component is identifying the grammar rules with which actions must be associated in order to construct intermediate data structures. This aspect of identifying significant grammar rules was aided by the DEBUG option available in YACC. The debug option creates a derivation tree for every construct that is parsed. The derivation tree enables one to identify which grammar rule is being used by the parser in the derivation of the construct. The information extraction and storage action can therefore be appropriately associated with the help of the derivation trees output by the parser.
- *Building Lists* : A significant aspect of the parser is building lists of symbolic names and attaching them to the correct symbol entry on the basis of the current scope. Due to the bottom-up nature of the parser, the contents of the list are available before the nature of the list can be determined. For example, the symbol A could represent a parameter in a function call (in which case it is part of a *parameter list*) or it could represent the name of an object in a declaration statement (in which case it is a part of an object list). Because the information regarding the nature of the list is available after the list is constructed, the parser is forced to build generic symbol lists and each item in the generic list is populated with information after the nature of the list is determined.
- *Building Expression Trees* : An expression tree is a data structure that stores an operation (such as the addition operation or the $- >$ operation) and its

operands. There are four expressions that are of interest to pulse – a function call, a dot-expression (ie. *obj.member()*), an arrow-expression (ie. *obj->member()*) and an expression using the allocation operator *new*. One of the operands for each of these operations is a parameter list of actual parameters. A function symbol entry (or a member function symbol entry) has a list of expression trees associated with it.

- *Elision of information* : The grammar specification of pulse is a general specification that is capable of recognizing every syntactic construct of C++. However, the LLSA model represents an abstraction of the system, wherein certain constructs are required and others are not. The parser in pulse recognizes all the syntactic constructs in the source files and also constructs data structures with them. The parser collects information which is not required by the LLSA. This information must be discarded by the parser, which means that the data structure must be destroyed when it is identified as useless information.

7.4.1 Fundamental Patterns and Programming Constructs

Fundamental patterns represent information that can be directly gathered from source code. A fundamental pattern set [SC95] is defined to be the set of all interactions between two components that are permissible and directly supported by the implementation language. For example, a function call is an interaction between two function components that is directly supported by most programming languages. The associates relationship between classes is an example of an interaction between classes that is not directly supported by object-oriented programming languages. Hence, a function call is represented as a fundamental pattern whereas the associates relationship is not a fundamental pattern.

The fundamental patterns correspondence table describes a fundamental pattern and its associated programming constructs. This table essentially shows the context-sensitive information that is gathered by the parser and is useful in determining the grammar rules with which actions should be associated. If more than one fundamental pattern is associated with the same construct(s) then they are grouped together in the same row.

Table 7.1: Fundamental Patterns and Programming Constructs

Fundamental Pattern	Abstract C++ Syntax of Constructs
A is an ancestor of B B is a descendant of A	class B : A { ... };
A contains B B is contained in A	class A { class B obj ; };
A is friendly with B	class B { friend class A; };
C is a friend of A	class A { friend class C; };
O is an object of A	class A { ... } O; or class A O;
A calls function F	A::method() { F (); ... }
F calls P P is called by F	int F () { P (); ... }
F creates O of class A	int F () { ... O = new A; }
F sends message to O	int F () { ... O->method(); } or int F () { ... O.method(); }

7.5 LLSA Generator

The collection of LLSA component text descriptions describe the low-level software architecture of the source code. The task of the LLSA generator consists of collecting information pertaining to a single component and generating the LLSA

textual description of the component. In order to do this, the generator computes the static and dynamic interfaces of a component and determines the components that interact with each other. The information contained in an LLSA text description is either inferred directly from the fundamental pattern set or computed by using the fundamental pattern set and additional rules. The LLSA correspondence table has four columns. The first column denotes the LLSA interaction that is being determined, the second the fundamental patterns that are used in determining it, the third column classifies the interaction as being computed or directly inferable from the fundamental patterns, and the last column gives a short description of the algorithm used to determine the interaction.

The generation of the LLSA description of each type of component – class, function and object – is individually addressed in sections 7.5.1, 7.5.2, 7.5.3 respectively.

7.5.1 Class Component LLSA

The name of a class component and its physical locations are directly obtained from the symbol table. The static and dynamic interfaces of a class component are computed by using algorithms that are closely related to semantic analysis algorithms employed by a compiler.

7.5.1.1 Static Interface

The symbol table entry associated with a class stores the names of all the members of the class. If the class is a derived class, then the members of all ancestor classes are also a part of the derived class. Computing the union of the interfaces of ancestor classes and the interface of the derived requires the *ambiguity resolution*

algorithm [SC94], which was developed as part of this research. The ambiguity resolution algorithm can be used by compilers for object-oriented languages to resolve name conflicts that arise as a result of attributes in ancestor and derived classes. A name conflict occurs when the attributes in ancestor and derived classes have the same name. The conflict is resolved by using semantic rules that the implementation language specifies for the name conflict problem. The resolution algorithm also locates the definition of a member.

The ambiguity resolution algorithm uses three data structures – *dominates set*, *base search order list*, and *discard list*. The *dominates set* establishes a dominance relationship between the classes in an inheritance hierarchy. The *base search order list* denotes the order in which base classes must be searched to locate a member. The *discard list* is used to record the classes that have been visited in a previous step. Associated with the ambiguity resolution algorithm are two data structure algorithms for the construction of the *dominates set* and the *base search order list* – *Create_Dom_Set*, *Create_Base_Search_Order*. The ambiguity algorithm establishes a dominance relationship between classes and constructs a *base search order list* on the basis of the dominance relationship. The *base search order list* thus constructed makes the algorithm more efficient in certain situations. The complete description of the ambiguity resolution algorithm and its associated data structures and algorithms can be found in [SC94].

The algorithms for the construction of various lists in the symbol table (see section 7.4) in phase I of pulse use algorithms similar to the *Create_Dom_Set* and *Create_Base_Search_Order* algorithms. The algorithm to compute the static interface of a class component uses conflict resolution semantic rules and inheritance rules of the implementation language. In particular, for the extraction of the LLSA

of C++ systems, the static interface algorithm uses a technique similar to that used by the ambiguity resolution algorithm. The static interface construction algorithm for a class component is described below. The LLSA generator invokes this algorithm if the current symbol table entry that it encounters is a class symbol. In essence, the static interface algorithm performs a restricted union of the interfaces of the ancestor classes and the interface of the derived class. This algorithm is presented in figure 7.3.

7.5.1.2 Dynamic Interface

The algorithm for computing the dynamic interface performs an intersection of the static interfaces of the ancestor classes with the interface of the derived class. This algorithm assumes that the static interfaces of all classes are available in the symbol table. If the static interface is not yet computed, the static interface algorithm is invoked to create it. C represents a generic class whose dynamic interface is computed by this algorithm. This algorithm is presented in figure 7.4.

The class interactions that are computed on the basis of fundamental patterns are given in table 7.2. This correspondence table establishes the relationship between LLSA interactions and fundamental patterns. The fundamental pattern set described in section 7.4.1 is used by the parser to extract information and eventually store it in data structures. The LLSA-fundamental correspondence tables documents the LLSA interactions that can be inferred directly from the information associated with fundamental patterns. Some of the LLSA interactions are computed using other information available in the symbol table. These LLSA interactions are listed separately. The class LLSA interactions that are heuristically inferred are given in table 7.3.

Let C represent the class, S represent its static interface, QML its list of qualified names of member functions, AL the list of names of ancestor classes and $QMLA$ the list of qualified names of member functions of an ancestor class. MF denotes a member function.

Input: Class C , Ancestor List AL , Member Function List QML

Output: Static Interface of Class

Algorithm:

1. $S \leftarrow QML$
2. For each class $A \in AL$ do
For each MF in $QMLA$ of A do
 - (a) if $MF \notin S$ then {
 - (b) Use C++ semantic rules to determine if MF is accessible from class C
 - (c) if MF is accessible from C then
 $S \leftarrow S \cup MF$
}

Figure 7.3: Algorithm to Compute the Static Interface of a Class

Let **C** represent the class, **S** represent its static interface, **D** its dynamic interface, **AL** the list of names of ancestor classes, **A** an ancestor class, **SA** and **DA** the static and dynamic interfaces of an ancestor class **A** in class **C**. **MF** is a member function

Input: Class **C**, Ancestor List **AL**, static interface **S** of class **C**, static interface **SA** of each ancestor class **A**

Output: Dynamic Interface of Class

Algorithm:

1. $D \leftarrow \langle C, S \rangle$
2. For each ancestor class **A** in **AL** do
 - (a) $DA \leftarrow SA \cap S$
 - (b) For each **MF** in **DA** do
 - if **MF** is not dynamically bound then
prefix **MF** with **A**
 - if **MF** is dynamically bound then
copy the qualified name of **MF** from **S**
 - (c) $D \leftarrow \langle A, DA \rangle$

Figure 7.4: Algorithm to Compute the Dynamic Interface of a Class

Table 7.2: Class LLSA Interactions and Fundamental Patterns

LLSA Class Interaction	Fundamental Patterns
Ancestor Classes	A is an ancestor of B
Descendant Classes	B is a descendant of A
Contains Classes	A contains B
Container Classes	B is contained in A
Friendly With Classes	A is friendly with B
Friendly Classes	C is a friend of A
Object Family	O is an object of A
Calls Functions	A calls function F
	F sends message to O

7.5.2 Function Component LLSA

The static interface of a function corresponds to its formal parameter list. This list is constructed in phase I and the LLSA generator does not have to compute the static interface of a function. The algorithm to compute the dynamic interface of the function is given in figure 7.5. The correspondence table for function LLSA interactions and fundamental patterns is given in table 7.4.

7.5.3 Object Component LLSA

The static interface of an object is constructed by examining the static interface of the static type class of the object and selecting those member functions that are not dynamically bound. The dynamic interface is built using more complex rules. The algorithm is presented in fig 7.6. There are no fundamental patterns corresponding to object LLSA interactions.

The computation of the *Sends Messages to Objects* interaction between objects is based on the following heuristic– if a message sent to an object contains another

Let **F** represent the class, **S** represent its static interface, **D** its dynamic interface, **PL** the formal parameter list of **F**, **CL** the actual parameter list passed to **F** in a function call, **PO** a parameter that is a pointer object and **C** the static class type of **PO**, **DL** the descendant list of **C**.

Input: Function **F**, static interface of **S** of **F**

Output: Dynamic interface of **D** of **F**

Algorithm:

1. For each **PO** in **PL** do
 - (a) Search Symbol table for the entry storing **C**
 - (b) If the symbol table entry was found then
create the tuple **<PO,DL>**
 - (c) **D** \leftarrow **<PO,DL>**

Figure 7.5: Algorithm to Compute the Dynamic Interface of a Function

Let O represent an object, SO represent a simple object and PO a pointer object. Let S represent the static interface of O and D its dynamic interface. Let C represent the static class type of O and DL the descendant class list of C . CD represents a descendant class in DL . SC represents the static interface of class C and SCD the static interface of a descendant class CD . MF is a member function. TL represents a temporary list data structure.

Input: Object O , static type class C of O , static interface of C , descendant list DL of C .

Output: Dynamic interface of O

Algorithm:

1. Determine if O is a simple object (SO) or a pointer object (PO).
 2. $D \leftarrow \{ \}$
 3. If O is a simple object (SO) then {
 - For each MF in S of C do
 - if MF is dynamically bound then
 - $D \leftarrow D \cup MF$
 - }
 4. If O is a pointer object (PO) then {
 - For each CD in DL do
 - $TL \leftarrow SCD \cap S$
 - if TL is not empty then {
 - For each MF in TL do
 - if MF is dynamically bound then
 - $D \leftarrow D \cup MF$
 - }
-

Figure 7.6: Algorithm to Compute the Dynamic Interface of an Object

Table 7.3: Class LLSA Interactions Heuristics

LLSA Class Interaction	Heuristics
Associate With Classes (A is associated with B)	class A contains a pointer object of class B
Associate Classes (B is an associate of A)	class B contains a pointer object of class A
Creates Objects of Class (A creates B objects)	Constructor of A creates objects of class B
Created By (A objects created by B)	Constructor of B creates objects of class A
Uses Members (A uses B)	Member functions of A use members of class B
Used By (A is used by B)	Member functions of B use members of class A

Table 7.4: Function LLSA Interactions and Fundamental Patterns

LLSA Function Interaction	Fundamental Patterns
Calls Functions	F calls P
Called By Functions	P is called by F
Creates Objects of Classes	F creates O of class A
Sends Messages To Objects	F sends message to O
Used By Classes	A calls F

object as a parameter, then it is likely that the receiving object will send a message to the object sent as an actual parameter.

7.6 LLDP Recognizer

The LLDP recognizer module performs the function of detecting LLDPs in an object oriented system. The recognizer requires information collected in the synthesis phase as well as the information generated by the LLSA generator. These

information requirements of the LLDP force it to be the last module that is executed in pulse.

The recognizer has two primary functions – (i) detecting an LLDP by recognizing its structure (ii) identification of the actual components that participate in an LLDP. The automation of both these tasks is discussed in more detail in this section. The identification of the names of actual components that participate in LLDPs with simple structures (ie. the LLDP structure is entirely contained within the LLSA representation of a component) is directly obtained from the LLSA descriptions of the component. For LLDPs with complicated structures, an identification algorithm must visit multiple components to detect the complete structure. The LLDP recognizer algorithm is given in figure 7.7. The recognizer visits every single component in the symbol table and attempts to detect the LLDP structures that may be associated with the component.

The LLDP recognizer uses the correspondence tables and algorithms given in sections 7.6.1, 7.6.2 and 7.6.3 to identify the structure of an LLDP. Sections 7.6.1, 7.6.2 and 7.6.3 discuss the correspondence between LLSA interactions and the polymorphism, decoupling and message LLDP structures, respectively. The correspondence table documents the interactions that are used in the algorithms to recognize the structures.

7.6.1 Identification of the Polymorphism LLDPs

The LLSA interactions and component interfaces that are required for the identification of the structures of polymorphism LLDPs are given in table 7.5. The LLDP recognizer examines the calls functions list and the called by function list of a function component and determines if a called function is overloaded. If the called

Let **O** represent an object, **C** a class and **F** a function. **SO**, **SC**, **SF** represent the static interfaces of an object, class and function component respectively. **DO**, **DC**, **DF** represent the dynamic interfaces of an object, class and function respectively.

Input: Symbol Table

Output: LLDP names and the participating components in the LLDPs

Algorithm:

1. For each symbol **S** in the symbol table do {
 - (a) If **S** is an object then
Check for message LLDP-1, LLDP-2,LLDP-3
 - (b) If **S** is a class then
Check for decoupling LLDP-1, LLDP-2, Check for polymorphism LLDP-2
 - (c) If **S** is a function then
Check for polymorphism LLDP-1, LLDP-3
Check for decoupling LLDP-3}

Figure 7.7: LLDP Recognizer Algorithm

Table 7.5: Polymorphism LLDPs and LLSA Interactions

LLDP	LLSA Interactions and Component Interfaces
Polymorphism LLDP-1	Calls Functions, Called by Functions
Polymorphism LLDP-2	Uses Members, Used By Classes
Polymorphism LLDP-3	Calls Function, Called by Functions, Ancestor Classes, Descendant Classes, Sends Messages to Objects, Dynamic interface of Class

function is overloaded, the LLDP recognizer reports the identification of polymorphism LLDP-1. For the identification of polymorphism LLDP-2, the recognizer uses the information contained in the uses members and used by classes lists to detect the presence of member functions that permit type conversion between the classes. If type conversion member functions are found, the recognizer reports the identification of polymorphism LLDP-2. The recognizer uses information in the dynamic interface of a class to detect dynamically bound member functions in order to recognize the structure of polymorphism LLDP-3. The algorithm to identify the LLDP structures and the participants is given in fig 7.8. Since the algorithm in figure 7.8 uses a function component as a starting point, it recognizes polymorphism LLDP-1 and LLDP-2 and decoupling LLDP-3. Polymorphism LLDP-2 is recognized by the algorithm in 7.9 which uses a class component as a starting point.

7.6.2 Identification of the Decoupling LLDPs

The LLDP recognizer uses the LLSA interactions given in table 7.6 to recognize the decoupling LLDPs. Decoupling LLDP-1 is identified by the recognizer if two classes associate with each other and one of the classes creates objects of its

Let F represent a function, CL the list of functions F calls, CBL the list of functions that call F , P a pointer object parameter of F , C the static class type of P , DL the descendant list of C . UL and UBL represent the list of classes whose member functions are used by C and the list of classes that use C 's member functions respectively. Let CF represent a called function and CBF a function that calls F . Let A represent an actual pointer object parameter in a function call. Let M represent a message to an object

Input: Symbol table entry of function F .

Output: Polymorphism LLDP-1 or LLDP-2 or Decoupling LLDP-3

Algorithm:

1. Identification of Polymorphism LLDP-1

- (a) For each called function CF in the CL of F do
 - if CF is overloaded then
 - report the identification of Polymorphism LLDP-1, participants F and CF

2. Identification of Polymorphism LLDP-3

- (a) If F has a P then {
- (b) For each function CBF in the CBL of F do
 - if A in CBF has static class type D and $D \in DL$ of C then
 - report the identification of Polymorphism LLDP-2, participants F , D and C

3. Identification of Decoupling LLDP-3

- (a) For each P of F do
 - For each M sent by F to P do
 - For each D in the DL of C of P do
 - if $M \in$ dynamic interface of D then
 - report the identification of Decoupling LLDP-3 and participants F, C, D
-

Figure 7.8: Algorithm to Recognize Polymorphism LLDPs

Table 7.6: Decoupling LLDPs and LLSA Interactions

LLDP	LLSA Interactions and Component Interfaces
Decoupling LLDP-1	Associate Classes, Associated With Classes, Creates Objects of Classes
Decoupling LLDP-2	Associate Classes, Associated With Classes
Decoupling LLDP-3	Calls Function, Called by Functions, Ancestor Classes, Descendant Classes, Sends Messages to Objects, Dynamic interface of Class

associate class. If the two classes associate with each other but the creates objects of classes relationship does not exist between them, the recognizer reports the identification of LLDP-2. The interactions used to identify decoupling LLDP-3 are similar to the interactions used for the identification to polymorphism LLDP-3. To identify decoupling LLDP-3, the recognizer determines the dynamically bound member functions from the dynamic interface of a class and checks to see if the function sends a dynamically bound message to the object.

7.6.3 Identification of the Message LLDPs

Table 7.7: Message LLDPs and LLSA Interactions

LLDP	LLSA Interactions and Component Interfaces
Message LLDP-1	None
Message LLDP-2	None
Message LLDP-3	Sends Messages to Objects, Uses Members of Class

Let **C** represent a class, **CA** and **CU** represent classes in the associates with and uses members of classes lists of **C** respectively.

Input: Class **C**

Output: Identification of Decoupling LLDP-1, LLDP-2 and Polymorphism LLDP-2

Algorithm:

1. Identification of Decoupling LLDP-1

- (a) For each class **CA** in the associates with classes list of **C** do
if **CA** appears in the creates objects of class list then
report the identification of Decoupling LLDP-1, participants **C,CA**

2. Identification of Decoupling LLDP-2

- (a) For each class **CA** in the associates with classes list of **C** do
if **CA** does not appear in the creates objects of class list then
report the identification of Decoupling LLDP-2, participants **C,CA**

3. Identification of Polymorphism LLDP-2

- (a) For each class **CU** in the uses members of classes list of **C** do
if **CU** contains a conversion member function for class **C** then
report the identification of Polymorphism LLDP-2, participants **C,CA**
-

Figure 7.9: Algorithm to Recognize Decoupling LLDPs

Let **O** represent an object, **C** its static class type, **S** represent its static interface and **D** its dynamic interface. Let **MF** represent a member function and **P** a parameter of **MF**.

Input: Function **F**

Output: Identification of Message LLDP-3

Algorithm:

1. For each object **O** in the sends messages to objects list of **F** do
 - For each **MF** associated with **O** in the sends messages list of **F** do
 - if there is a **P** in **MF** then
 - if **MF** sends a message to **P** then
 - report the identification of Message LLDP-3 with participants **O** and **OA**, where **OA** is determined from the symbol table entry of **F**.
-

Figure 7.10: Algorithm to Recognize Message LLDPs

The identification of message LLDP-1 and LLDP-2 does not require LLSA interaction information. The information contained in the sends messages to objects and the uses members interactions is used in the identification of message LLDP-3.

7.7 Sample Session Using pulse

Pulse views a set of files containing source code as a software system. Typically, for C++ software, there are two kinds of files – header files containing declarations of objects and classes, and code files that contain the actual code that implements member functions and functions. Header files have a .h extension and code files have a .C extension. Pulse accepts both kinds of files. The input to pulse consists of the names of all the files in the software system. Pulse scans and parses each file in the

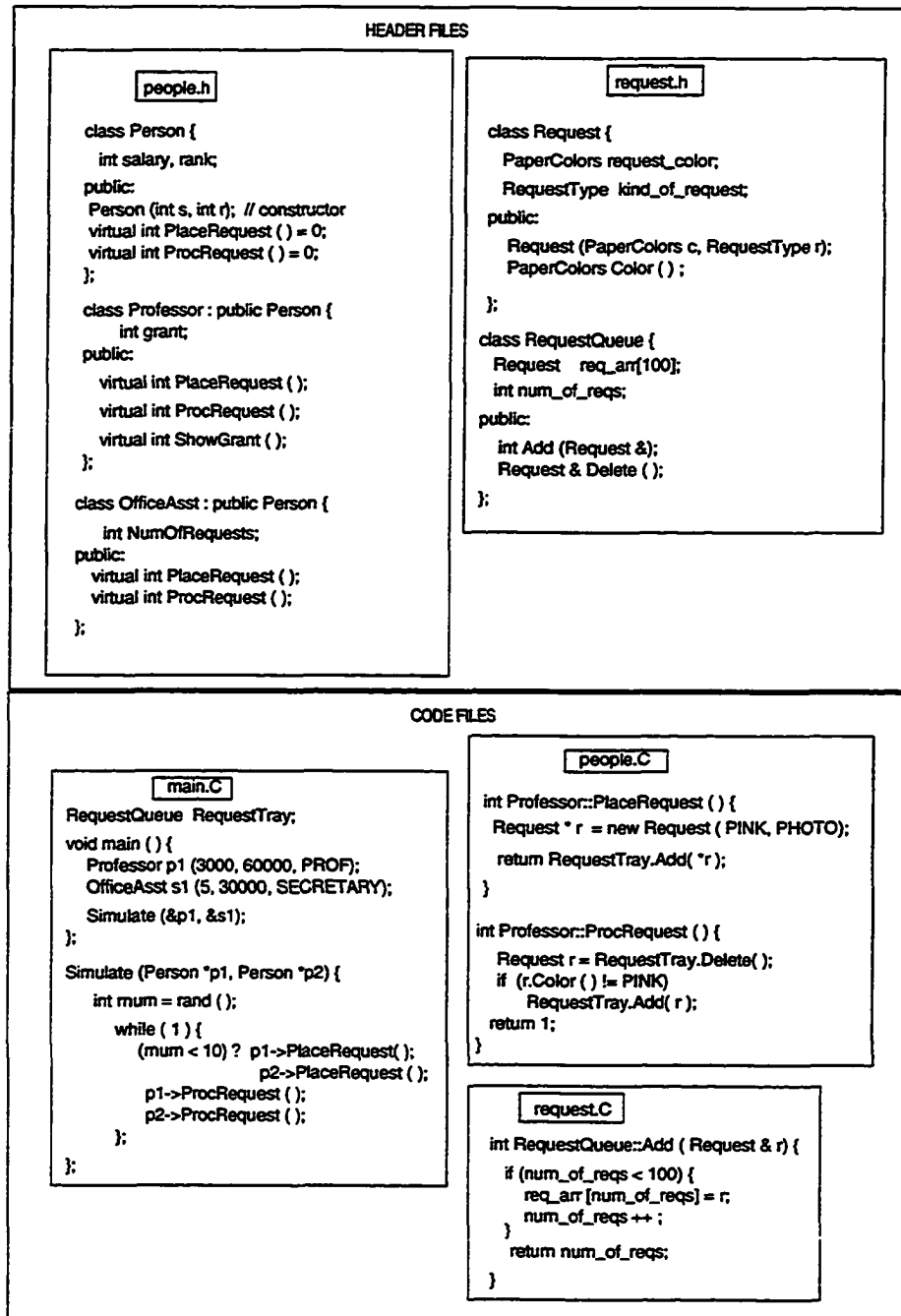


Figure 7.11: A C++ Software System

order specified by the user. The output produced by pulse consists of three ascii files - Class.LLSA, Functions.LLSA, Objects.LLSA File Class.LLSA contains the LLSA textual descriptions of all classes in the system. Similarly, files Functions.LLSA and Objects.LLSA contain the LLSA textual descriptions of functions and objects respectively. The file System.LLDP contains the name of an LLDP and the names of participating components. The LLDPs themselves are available as a reference document.

Sample C++ source code is shown in figure 7.11 and the LLSA textual descriptions generated by pulse for this source code by pulse is shown in below. The complete input source code and the LLSA generated for the input source code is given in Appendix A. The LLDPs recognized in this input are also listed in Appendix A.

7.8 Summary

This chapter provided the overall architecture of the prototype tool pulse, the code analysis algorithms necessary for the generation of the LLSA descriptions and the recognition of LLDPs and the design and implementation details relevant to the implementation of pulse. The design and development of the code analysis algorithms is a significant contribution towards the reverse engineering of low-level design patterns.

Chapter 8

Conclusion

This research was undertaken to study the issues that complicate the process of understanding object-oriented software systems and to investigate and use reverse engineering techniques to aid in program comprehension and software maintenance. The contributions of this research are summarised in section 8.1. The extensions and future work possible in this research are explored in section 8.2.

8.1 Contributions

The LLSA representation is an abstract representation model that uses concepts from the areas of software architecture and graph theory. The LLSA model can be viewed as a graph consisting of nodes and edges, where the nodes correspond to the notion of an LLSA component and the edges correspond to an LLSA interaction. The LLSA model is more informative than a graph model because an LLSA component is defined to have an interface whereas a node in a graph is not similarly defined. The definition of a component provided in the software architecture model served as a useful abstraction mechanism in the definition of the LLSA component. The interface of an LLSA component is classified as static or dynamic. The interfaces of a component are useful in understanding the static and dynamic behavior of the component. Defining nodes to have an interface is an idea that was borrowed from the area of software architecture.

The analysis and design underlying the LLSA textual description of a component utilized graph-theoretic concepts, and the notion of a rooted component subgraph, as defined in this research, was identified as being ideal for describing each LLSA component. Algorithms for performing depth-first traversal, breadth-first traversal, determining connected components, determining the transitive closure of a component, determining the minimum spanning tree of a connected graph (see [Liu85, MT81] for an introduction to concepts and algorithms in graph theory) already exist. The applicability and usefulness of graph algorithms is also well-known. By modelling the abstract representation along graph-theoretic lines, the LLSA model lends itself to graph algorithms. In essence, the LLSA model is designed to reap the benefits of the areas of software architecture and graph theory.

In the theoretical LLSA model, a component is permitted to interact with any other component and each interaction is symmetrically represented in the textual descriptions of both the participating components. In practice, the interactions permitted between components are determined by the implementation language. An interesting feature of the theoretical model is that the number of interactions in the model are determined by the number of components in the model; ie. if there are N components in the model, there are N^2 kinds of interactions. The higher the number of interactions, the more complex the representation. An obvious measure to reduce the complexity of the LLSA model is to choose the components carefully and minimize the number of components in the model.

The practical LLSA model defined for C++ software systems consists of three components and hence there are nine kinds of interactions. An analysis of the practical issues revealed that each kind of interaction could be refined to depict a more appropriate interaction. For example, from the theoretical point of view,

there is only one class-to-class kind of interaction between two class components. However, from the practical point of view, it is useful to distinguish between a *inherits – from* interaction between classes and a *part – of* interaction between classes. Hence, the theoretical model provides some basis for judging the potential complexity of the practical LLSA model, but the measure is not very reliable. The contributions arising from the LLSA model are listed below.

- The low-level software representation model for object-oriented systems is a significant and original contribution of this research because it uses concepts from software architecture and graph theory whereas most other abstract representation models are based on the entity-relationship model. The low-level software architecture model supports multiple views of software. A view can be a call-graph or a dependency graph. These views are easily obtained from the LLSA model by performing a transitive closure operation on the appropriate LLSA interactions. In order to obtain information regarding dependencies among software components from an entity-relationship model, the program database must be correctly queried. The dependency information thus obtained is subject to the relations stored in the database and the nature of the query. In contrast, the LLSA model provides a single general method of obtaining dependency information – the transitive closure of an LLSA interaction.
- A significant contribution of this research is the collection of code analysis algorithms that enable the LLSA model of a C++ system to be reverse engineered. The code analysis algorithms presented in chapter 7 are a unique

contribution of this reverse engineering effort that demonstrate the feasibility of automatically extracting the LLSA and LLDP models. The ambiguity resolution algorithm and the algorithms to compute the static and dynamic interfaces of a class are particularly significant because each algorithm solves a complex data analysis problem. The prototype, pulse, uses these algorithms to extract the LLSA and LLDPs of C++ systems.

- The definitions of static and dynamic interfaces of a component enable a better view of the structural and behavioral aspects of each component and eventually of the software system. The static interface of a component provided information about the static behavior of the component and the dynamic interface provides information about the potential roles the component is capable of dynamically playing. The definition of a component with static and dynamic interfaces is an original and significant contribution of this research.
- The notion of interactions was borrowed from the software architecture model. The classification of interactions as static and dynamic, in addition to the static and dynamic interfaces of a component, aids in obtaining a view of the static and dynamic structure of a software system in a unique way.
- The practicality and feasibility of the LLSA model was demonstrated by defining the LLSA model of C++ systems. The information content of the LLSA model of C++ system is useful in various activities of maintenance that include understanding the logical and physical organization of code and the static and dynamic aspects of a system.

- The LLSA representation of an object-oriented system provides comprehensive information to the maintainer. The automatic extraction of the information reduces maintenance effort, ensures the validity of the information and eliminates the maintainer's need to apply complex semantic rules to obtain the information.
- The LLSA model supports graph-theoretic algorithms. Views of software that use graph-theoretic concepts and techniques can be easily derived from the LLSA representation of the software system. In particular, the transitive closure operation can be used to obtain views of the software that focus on a single LLSA interaction. For example, the call graph of a software system can be obtained by performing a transitive closure operation on the *calls functions, called by functions* LLSA interaction of the function component. The class inheritance graph can be obtained from the *ancestor, descendant* interactions. A particularly useful view that is afforded by the LLSA model is the object creation graph. This graph can be constructed over the *creates objects* LLSA interaction of classes. The representation models of CIA++, XREFDB/XREF and code browsers (see chapter 3) do not support the object creation relationship. This is a significant advantage of the LLSA model.
- The LLSA model lends itself to further abstraction; for the next level of abstraction, a clustering operation can be used to group related components to form a component at the higher level of abstraction, and interactions at the higher level of abstraction can be a combination of LLSA interactions. This property, whereby further abstraction can be achieved, is a significant aspect of the LLSA model.

- LLSA is a language-independent representation model. The decision to define software components in terms of object-oriented *concepts* as opposed to object-oriented programming language *constructs* contributes to the language-independence of the LLSA model. The generality achieved as a consequence of language-independence makes the LLSA model a useful representation model for more than one object-oriented software systems.

The design patterns [GHJV93, GHJV94] approach for representing the design and architecture of an object-oriented system has been an influential factor in the design of the LLDP template. The collection of design patterns in [GHJV94] provides a wide variety of carefully analysed design solutions to commonly occurring problems and the authors of the collection expect the design patterns to be commonly used in object-oriented software development. Understanding a design pattern is a non-trivial task. Identifying a pattern in a software system is also a difficult task. From the maintenance perspective, the automatic identification and classification of a design pattern from object-oriented software system would be extremely useful. This research work makes a contribution towards the reverse engineering of design patterns in the form of reverse engineering LLDPs. The design of an LLDP template was influenced by the design pattern template. The information content of an LLDP was influenced by two major considerations – (i) determining what information would be useful from the maintenance perspective and (ii) determining the feasibility of automatically extracting and correctly inferring the information. The contributions arising from the LLDP model are listed below.

- The design of the LLDP template is an original contribution of this research. The LLDP template was designed by investigating and understanding the

Design Pattern template (see [GHJV93, GHJV94]) and adapting it to the needs of a software maintainer. An LLDP is an abstract representation of low-level object-oriented design. Nine LLDPs were presented in this research. Each LLDP represents a common object-oriented strategy. Several literature sources on object-oriented programming describe these strategies. However, providing the structure of the strategy such that the strategy can be identified and recognized is an original contribution of this work.

- An analysis of the benefits and consequences of a strategy from the point of view of desirable properties associated with the strategy and from the point of view of understandability is a contribution towards aiding maintenance.
- The automatic recognition of LLDPs was made possible by representing the structure of an LLDP in terms of LLSA components and interactions. This is a significant research contribution towards reverse engineering because it demonstrates the correlation between techniques and programming constructs. The correlation was established by defining an LLDP structure in terms of LLSA components and interactions which in turn are defined by programming constructs.

8.2 Extensions and Future Work

The extensions for this work include :

- The design information that is recovered is low-level. A maintainer often requires low-level information. However, to address the general problem of understandability, this work can be extended to extract design information at a higher-level of abstraction.

- The prototype implementation, pulse, is could be adaptapted to other object-oriented programming languages. Despite the decoupling between phase I and phase II, there are dependencies between the two phases that make it difficult to change the scanner and the parser without affecting the LLSA generator and LLDP generator.
- The LLSA model can be enriched with more low-level information to aid in debugging activities.
- The prototype pulse can be extended with transitive closure operations so that views of a software system can be an added feature.
- The C++ LLSA model can be extended by analysing the effect of other features such as overloaded operators, templates, exception handling on the understandability and complexity of object-oriented systems.
- Developing a formal specification language to specify the LLSA of software systems implemented in a specific programming language would be an invaluable extension of this work.
- The possibility of correlating the structure of design patterns with the structure of LLDPs is a possible area of future research. Establishing such a correlation would aid in the reverse engineering of design patterns.
- The design of a maintenance methodology that uses the LLSA model for code comprehension and code modification is a possible area of future research.
- An area worthy of investigation and research is the possibility of automating code modifications to reduce programmer errors.

- The applicability of LLSA and LLDP in the area of reusability is envisioned as a future research area.

Bibliography

- [AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. *Software Engineering Notes*, 18(5):9–20, December 1993.
- [ABCC87] P. Antonini, P. Benedusi, G. Cantone, and A. Cimitile. Maintenance and reverse engineering : Low-level design documents production and improvement. In *IEEE Conference on Software Maintenance*, pages 91–100. IEEE Computer Society Press, 1987.
- [AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, 1994.
- [Ale77] Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977.
- [Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.
- [And86] Stephen J. Andriole. *Software Validation Verification Testing and Documentation*. Petrocelli Books, 1986.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Tools and Techniques*. Addison-Wesley, 2 edition, 1986.
- [Bas90] Victor R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE SOFTWARE*, pages 19–25, January 1990.
- [BGM89] Gordon S. Blair, John J. Gallagher, and Javad Malik. Genericity vs inheritance vs delegation vs conformance vs ... *Journal of Object Oriented Programming*, September-October 1989.
- [Big89] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, pages 36–49, July 1989.
- [Boo93] Grady Booch. *Object-Oriented Design with Applications*. Benjamin Cummings, 2 edition, 1993.
- [CAB+94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development The Fusion Method*. Prentice Hall Object-Oriented Series, 1994.

- [Cas92] Eduardo Casais. An incremental class reorganization approach. In *ECOOP '92 - Object Oriented Programming, LNCS 615*, pages 114–231. Springer-Verlag, 1992.
- [CC90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE SOFTWARE*, pages 13–17, January 1990.
- [CM91] Injun Choi and Michael V. Mannino. Graph interpretation of methods: A unifying framework for polymorphism in object-oriented programming. In *OOPS Messenger*, pages 38–54, 1991.
- [CNR90] Yih-Farn Chen, Michael Y. Nishimoto, and C.V. Ramamoorthy. The c information abstraction system. *IEEE Transactions on Software Engineering*, pages 325–334, March 1990.
- [Coa92] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–158, September 1992.
- [Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. John Wiley and Sons, 1992.
- [Cox86] B. J. Cox. *Object-Oriented Programming*. Addison-Wesley, 1986.
- [CS89] A. Colbrook and C. Smythe. The retrospective introduction of abstraction into software. In *IEEE Conference on Software Maintenance 1989*, pages 166–173. IEEE Computer Society Press, 1989.
- [CS90] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *IEEE SOFTWARE*, pages 66–71, January 1990.
- [CvM93] Ronald T. Crocker and Anneliese von Mayrhauser. Maintenance support needs for object-oriented software. In *Proceedings of COMPSAC-93*, pages 63–69. IEEE Computer Society Press, 1993.
- [CY90] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Yourdon Press Computing Series, 1990.
- [DeM78] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Inc., 1978.
- [Ein89] John Einbu. *A Program Architecture for Improved Maintenance in Software Engineering*. John Wiley and Sons, 1989.
- [ES92] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1992.

- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.
- [GC90] J. E. Grass and Y. F. Chen. The c++ information abstractor. In *Usenix C++ Conference Proceedings*. Usenix Association, 1990.
- [GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93 - Object Oriented Programming, LNCS 707*, pages 406–419. Springer - Verlag, 1993.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.
- [GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions of Software Engineering*, 17(8), August 1991.
- [GLG92] G.Canfora, L.Sansone, and G.Visaggio. Data flow diagrams: Reverse engineering production and animation. In *IEEE Conference on Software Maintenance 1992*, pages 366–375. IEEE Computer Society Press, 1992.
- [Gol83] Adele Goldberg. *Smalltalk-80 : The Language and Its Implementation*. Addison-Wesley, 1983.
- [GP94] David Garlan and Dewayne Perry. Software architecture: Practice, potential, and pitfalls. In *16th International Conference on Software Engineering*, pages 363–364. IEEE Computer Society Press, 1994.
- [GS89] Virginia R. Gibson and James A. Senn. System structure and software maintenance. *Communications of the ACM*, 32(3):347–358, March 1989.
- [GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–39. World Scientific Publishing Company, 1993.

- [Gul92] Bjorn Gulla. Improved maintenance support by multi-version visualizations. In *IEEE Conference on Software Maintenance 1992*, pages 376–383. IEEE Computer Society Press, 1992.
- [GW90] Keith D. Gillis and David G. Wright. Improving software maintenance using system-level reverse engineering. In *IEEE Conference on Software Maintenance*, pages 84–90. IEEE Computer Society Press, 1990.
- [HM91] M. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. In *IEEE Conference on Software Maintenance 1991*. IEEE Computer Society Press, 1991.
- [HPLH90] Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner. Using function abstraction to understand program behavior. *IEEE SOFTWARE*, pages 55–63, January 1990.
- [Joh86] Stephen C. Johnson. Yacc: Yet another compiler compiler. *UNIX Programmer's Supplementary Documents, Volume 1 (PS1) Pages PS1:15,1-33*, April 1986.
- [Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science, 1987.
- [JOR92] Bret Johnson, Steve Ornburn, and Spencer Rugaber. A quick tools strategy for program analysis and software maintenance. In *IEEE Conference on Software Maintenance 1992*, pages 206–213. IEEE Computer Society Press, 1992.
- [KBAW94] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. Saam: A method for analyzing the properties of software architectures. In *16th International Conference on Software Engineering*, pages 81–90. IEEE Computer Society Press, 1994.
- [KGH⁺93] D. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Design recovery for software testing of object-oriented programs. In *Working Conference on Reverse Engineering 1993*, pages 202–211. IEEE Computer Society Press, 1993.
- [KGH⁺94] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object-oriented software maintenance. In *IEEE Conference on Software Maintenance 1994*, pages 202–211. IEEE Computer Society Press, 1994.

- [KM94] David A. Kinloch and Malcolm Munro. Understanding c programs using the combined c graph representation. In *IEEE Conference on Software Maintenance 1994*, pages 172–180. IEEE Computer Society Press, 1994.
- [LC93] Panos E. Livadas and Stephen Croll. System dependence graph construction for recursive programs. In *Proceedings of the Reverse Engineering Conference*, pages 414–420. IEEE Computer Society Press, 1993.
- [Lea94] Doug Lea. Christopher alexander: An introduction for object-oriented designers. *Software Engineering Notes*, pages 1–8, January 1994.
- [LH89] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.
- [Liu85] C. L. Liu. *Elements of Discrete Mathematics*. McGraw-Hill Computer Science Series, 1985.
- [LMR91] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. In *IEEE Conference on Software Maintenance 1991*, pages 171–178. IEEE Computer Society Press, 1991.
- [LR92] Panos E. Livadas and Prabal K. Roy. Program dependence analysis. In *IEEE Conference on Software Maintenance 1992*, pages 356–365. IEEE Computer Society Press, 1992.
- [LS86] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. *UNIX Programmer's Supplementary Documents, Volume 1 (PS1) Pages PS1:16,1-13*, April 1986.
- [LX93] Karl J. Lieberherr and Cun Xiao. Object-oriented software evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.
- [M.A90] M.A.Ketabchi. An object-oriented integrated software analysis and maintenance. In *IEEE Conference on Software Maintenance 1990*, pages 60–62. IEEE Computer Society Press, 1990.
- [Mac87] Bruce J. MacLennan. *Principles of Programming Languages*. Holt, Rinehart and Winston, 2 edition, 1987.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1 edition, 1988.

- [MP92] David E. Monarchi and Gretchen I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9), September 1992.
- [MT81] M.N.S.Swamy and K. Thulasiraman. *Graphs, Networks, and Algorithms*. John Wiley and Sons, 1981.
- [NS87] K. Narayanaswamy and Walt Scacchi. Maintaining configurations of evolving software systems. *IEEE Transactions of Software Engineering*, SE-13(3), March 1987.
- [OJ92] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *SOOPPA-92*, pages 1–8, 1992.
- [Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. Phd Thesis, University of Illinois at Urbana-Champaign, 1992.
- [Pak92] Howden W.E.and Suehee Pak. Problem domain, structural and logical abstractions in reverse engineering. In *IEEE Conference on Software Maintenance 1992*, pages 214–224. IEEE Computer Society Press, 1992.
- [Par86] Girish Parikh. *Handbook of Software Maintenance*. John Wiley & Sons, 1986.
- [PJ92] Meilir Page-Jones. Comparing techniques by means of encapsulation and connascence. *Communications of the ACM*, 35(9):147–151, September 1992.
- [PP94] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.
- [Pre92] Roger Pressman. *Software Engineering, A Practitioner's Approach*. McGraw-Hill, Inc., 3 edition, 1992.
- [RBP⁺91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [RC93] Spencer Rugaber and Richard Clayton. The representation problem in reverse maintenance. In *Working Conference on Reverse Engineering 1993*, pages 8–16. IEEE Computer Society Press, 1993.
- [ROJ90] Spencer Rugaber, Stephen B. Ornburn, and Richard J. Leblanc Jr. Recognizing design decisions in programs. *IEEE SOFTWARE*, pages 46–54, January 1990.

- [Rom87] H. Dieter Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions to Software Engineering*, SE-13(3), March 1987.
- [RW88] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *COMPUTER*, pages 11–25, November 1988.
- [RW90] Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE SOFTWARE*, pages 82–89, January 1990.
- [Sam90] Johannes Sametinger. A tool for the maintenance of c programs. In *IEEE Conference on Software Maintenance 1990*, pages 54–59. IEEE Computer Society Press, 1990.
- [SC94] Chandra Shrivastava and Doris L. Carver. An ambiguity resolution algorithm. *Object-Oriented Methodologies and Systems, Lecture Notes in Computer Science*, 258, 1994.
- [SC95] Chandra Shrivastava and Doris L. Carver. Extracting fundamental patterns from c code. *Submitted to 1995 IEEE Aerospace Applications Conference*, 1995.
- [Sch87] Norman F. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, March 1987.
- [Seb93] Robert W. Sebesta. *Concepts of Programming Languages*. Benjamin Cummings, 2 edition, 1993.
- [Sha94] Mary Shaw. Patterns for software architectures. In *First Annual Conference on Pattern Languages of Programs*, pages 1–7, 1994.
- [SM89] Sally Shlaer and Stephen J. Mellor. An object-oriented approach to domain analysis. *Software Engineering Notes*, July 1989.
- [SPL+88] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.
- [Sta94] Ryan Stansifer. *The Study of Programming Languages*. Prentice Hall, 1994.
- [Ste93] Al Stevens. *teach yourself ... C++*. MIS: press, 3 edition, 1993.

- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2 edition, 1991.
- [SWC93] Peter G. Selfridge, Richard C. Waters, and Elliot J. Chikofsky. Challenges to the field of reverse engineering. In *Working Conference on Reverse Engineering 1993*, pages 144–150. IEEE Computer Society Press, 1993.
- [Weg87] Peter Wegner. Object-oriented classification. *Research Directions in Object-Oriented Programming*, 1987.
- [WH91] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. In *IEEE Conference on Software Maintenance*, pages 162–170. IEEE Computer Society Press, 1991, 1991.
- [WHH89] Norman Wilde, Ross Huitt, and Scott Huitt. Dependency analysis tools:reusable components for software maintenance. In *IEEE Conference on Software Maintenance 1989*, pages 126–131. IEEE Computer Society Press, 1989.
- [Wil93] George Wilkie. *Object-Oriented Software Engineering The Professional Developer's Guide*. Addison-Wesley, 1993.
- [WMH93] Norman Wilde, Paul Matthews, and Ross Huitt. Maintaining object-oriented software. *IEEE Software, Vol 10 Number 1, January*, pages 75–80, 1993.
- [WZ88] Peter Wegner and Stan Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP '88, LNCS 322*. Springer-Verlag, 1988.
- [YT87] S. S. Yau and J. J. Tsai. Knowledge representation of software component interconnection information for large-scale software modifications. *IEEE Transactions of Software Engineering*, SE-13(3):355–361, March 1987.

Appendix

Session Listing

```
=====  
Class Component 0  
=====
```

```
[Name] : Person
```

```
[Physical Location] : ../test-suite/peoplereq.h.pulse
```

```
[Static Interface]:
```

```
Person::Person ()
```

```
Person::placeRequest ()
```

```
Person::procRequest ()
```

```
[Static Interactions]
```

```
Descendant Classes :
```

```
-----
```

```
class Professor
```

```
class OfficeAsst
```

```
[Dynamic Interactions]
```

```
=====
```

Class Component 1

=====

[Name] : Professor**[Physical Location] : ../test-suite/peoplereq.h.pulse****[Static Interface]:****Person::Person ()****Professor::Professor ()****Professor::placeRequest ()****Professor::procRequest ()****Professor::showRequest ()****[Dynamic Interface]:****Class Person****Person::Person ()****Person::placeRequest ()****Person::procRequest ()****[Static Interactions]****Ancestor Classes :**

<Person,Public>

[Dynamic Interactions]

=====

Class Component 2

=====

[Name] : OfficeAsst

[Physical Location] : ../test-suite/peoplereq.h.pulse

[Static Interface]:

OfficeAsst::OfficeAsst ()

OfficeAsst::placeRequest ()

OfficeAsst::procRequest ()

Person::Person ()

[Dynamic Interface]:

Class Person

Person::Person ()

Person::placeRequest ()

Person::procRequest ()

[Static Interactions]

Ancestor Classes :

<Person,Public>

[Dynamic Interactions]

=====

Class Component 3

=====

[Name] : Request

[Physical Location] : ../test-suite/peoplereq.h.pulse

[Static Interface]:

Request::Request ()

Request::color ()

Request::Request ()

[Static Interactions]

[Dynamic Interactions]

=====

Class Component 4

=====

[Name] : RequestQueue

[Physical Location] : ../test-suite/peoplereq.h.pulse

[Static Interface]:

RequestQueue::RequestQueue ()

RequestQueue::add ()

RequestQueue::remove ()

[Static Interactions]

[Dynamic Interactions]

=====

Object Component 5

=====

[Name] : requestTray

[Physical Location] : ../test-suite/main.C.pulse

=====

Function Component 6

=====

[Name] : simulate

[Physical Location] : ../test-suite/main.C.pulse

[Static Interface]

<p1,Person> <p2,Person>

[Dynamic Interface]

[Static Interactions] :

=====

Function Component 7

=====

[Name] : main

[Physical Location] : ../test-suite/main.C.pulse

[Static Interactions] :

Polymorphism LLDP 3 Identified

Participants : simulate, Person

Polymorphism LLDP 3 Identified

Participants : simulate, Person

Vita

Chandra Shrivastava was born in Hyderabad, India on October 27, 1966. She completed the bachelor of science degree in Physics at Fergusson College, Pune, India, in June 1987. She received the master of science degree in Computer Science from Poona University, India, in 1989. From August 1989, to May 1991, she served as an instructor, project co-investigator and admissions incharge at Poona University, Computer Science Department. She joined the doctoral program offered by the Computer Science Department, Louisiana State University in June 1991 and will be awarded her doctor of philosophy degree in May of 1996.


DOCTORAL EXAMINATION AND DISSERTATION REPORT

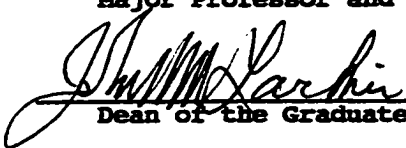
Candidate: Chandra Shrivastava

Major Field: Computer Science

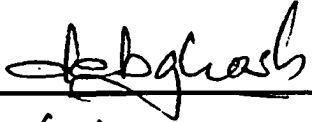
Title of Dissertation: Reverse Engineering Low-Level Design
Patterns from Object-Oriented Code

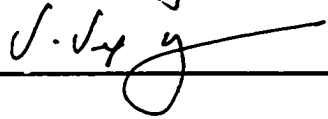
Approved:



Major Professor and Chairman

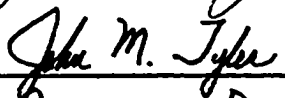

Dean of the Graduate School

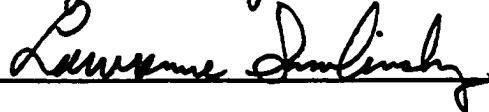
EXAMINING COMMITTEE:











Date of Examination:

November 14, 1995
