NSU Florida
NOVA SOUTHEASTERN UNIVERSITY

Nova Southeastern University
NSUWorks

CEC Theses and Dissertations

College of Engineering and Computing

1998

A Simplified Faceted Approach To Information Retrieval for Reusable Software Classification

Victor Allen Nguyen
Nova Southeastern University, drnguyenusa@yahoo.com

This document is a product of extensive research conducted at the Nova Southeastern University College of Engineering and Computing. For more information on research and degree programs at the NSU College of Engineering and Computing, please click here.

Follow this and additional works at: https://nsuworks.nova.edu/gscis_etd

Part of the Computer Sciences Commons

Share Feedback About This Item

NSUWorks Citation

Victor Allen Nguyen. 1998. *A Simplified Faceted Approach To Information Retrieval for Reusable Software Classification.* Doctoral dissertation. Nova Southeastern University. Retrieved from NSUWorks, Graduate School of Computer and Information Sciences. (749)
https://nsuworks.nova.edu/gscis_etd/749.

This Dissertation is brought to you by the College of Engineering and Computing at NSUWorks. It has been accepted for inclusion in CEC Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact nsuworks@nova.edu.

A SIMPLIFIED FACETED APPROACH TO INFORMATION RETRIEVAL FOR REUSABLE SOFTWARE CLASSIFICATION

by

Victor Allen Nguyen

A Dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

School of Computer Information Systems and Sciences
Nova Southeastern University

1998

We hereby certify that this dissertation, submitted by Victor Allen Nguyen, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

S. Rollins Guild, Ph.D.
Chairperson of Dissertation Committee

6/21/99
Date

Michael Moody, Ph.D.
Member of Dissertation Committee

7/15/99
Date

Lee Leitner, Ph.D.
Member of Dissertation Committee

6/21/1999
Date

Approved:

Edward Lieblein, Ph.D.
Dean, School of Computer and Information Sciences

7-20-99
Date

School of Computer and Information Sciences
Nova Southeastern University

1998

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A Simplified Faceted Approach To Information Retrieval For Reusable Software Classification

by
Victor Allen Nguyen

June 1998

Software Reuse is widely recognized as the most promising technique presently available in reducing the cost of software production. It is the adaptation or incorporation of previously developed software components, designs or other software-related artifacts (i.e. test plans) into new software or software development regimes. Researchers and vendors are doubling their efforts and devoting their time primarily to the topic of software reuse. Most have focused on mechanisms to construct reusable software but few have focused on the problem of discovering components or designs to meet specific needs. In order for software reuse to be successful, it must be perceived to be less costly to discover a software component or related artifact to satisfy a given need than to discover one anew. As results, this study will describe a method to classify software components that meet a specified need.

Specifically, the purpose of the present research study is to provide a flexible system, comprised of a classification scheme and searcher system, entitled Guides-Search, in which processes can be retrieved by carrying out a structured dialogue with the user. The classification scheme provides both the structure of questions to be posed to the user, and the set of possible answers to each question. The model is not an attempt to replace current structures; but rather, seeks to provide a conceptual and structural method to support the improvement of software reuse methodology.

The investigation focuses on the following goals and objectives for the classification scheme and searcher system:

(1) the classification will be flexible and extensible, but usable by the searcher;
(2) the user will not be presented with a large number of questions; the user will never be required to answer a question not known to be germane to the query;

(3) the user will not be presented with a large number of possible answers to any single question; and

(4) the user will be allowed to specify an answer, even though he or she did not know exactly what question the searcher will pose to elicit that answer. (This is similar to a key word search.)

Acknowledgments

This dissertation would not have been possible without the understanding and guidance of my committee during the research process. I wish to extend my Sincere appreciation to Dr. S. Rollins Guild for his patience, encouragement and guidance in all stages of my study. I also wish to express my gratitude to the members of my academic advisory committee: Dr. Michael Moody, Dr. Lee Leitner, Dr. William M. Hartman, Dr. Diane King and the School of Computer and Information Sciences Dean's Representative - Dr. Edward Lieblein for their valuable advice and service.

I would like to express my gratitude to Professor Van Snynder, a mathematician at Jet Propulsion Laboratory. He aided my research tremendously by providing 19 pages of unfinished material relating to the subject matter which began my thesis. Although Professor Van Snyder was working two jobs, he always found the time to help. I would also like to include Professor Van Snyder's family who welcomed me into their home with undivided encouragement. They are my inspiration of the spirit of the American family.

I would like to thank Hoa V. Phan and his family for their friendship over the last 15 years and their hospitality while visiting Florida. A special thanks goes to Irene Hill, my sister-in-law, and Ms. Shaeffer for their editing talents.

Finally, I would like to express unconditional love for my family...my wife, Thao who has given many sleepless nights to encourage and motivate me and my two daughters, Ariana and Chloe, whom have been very patient.

Table of Contents

Abstract iii
Table of Contents vi
List of Figures ix

Chapter

I. Introduction 10
-- Background of the Problem 12
-- Barriers and Issues 14
-- Relevance and significance 15
1. Classification Schemes 16
2. A Guides-Search System 18
3. Users interface 18
-- Purpose of the study 20
-- Research statements to be investigated 21
-- Assumptions and Limitations 23
-- Definition of terms 24
-- Chapter Summary 27

II. Research and literature review 29
-- Capsule Description 30
-- Object-Oriented Programming 30
-- Classes 32
-- Encapsulation 34
-- Inheritance 36
-- Reuse Via Inherit 38
-- Composition 39
-- Polymorphism 40
-- Background of Mathematics Leading Toward Computing Abstraction and Reuse 42
-- Abstraction and Reuse 42
-- Functional Abstraction 45
-- Data Driven 46
-- Message Driven 46
-- Language Evolution 47
-- Research on Reuse 54
-- Evolution to measurements 54
-- Measurements 55
-- Line-of-code 56
-- Function Points (Complexity) 57

-- Model (Program Size) 58
-- Reduction of Complexity 60
-- Complexity Analysis 61
-- Hierarchical Classification 64
-- Faceted Classification 65
-- Natural Language Model 65
-- Communicating Sequential Processes (CSP) 66
-- Object-Oriented Technology in Development and Reuse 67
-- Frame and Frameworks in Reuse 67
-- Pattern Languages design in Reuse 70
-- Data Abstraction 72
-- Complexity Reduction 75
-- Background of the Theoretical Model 77
-- Relation 77
-- Frames 81
-- Propositional Logic 82
-- Constraint Satisfaction 83
-- Background of Practical Machine 84
-- Mathematical Verification by Group Theory 84
-- The Algebra of Notation 85
-- The Algebra of Sets 85
-- Relations 85
-- Functions 86
-- Euler-Venn Diagrams 86
-- Group Theory in Real World Problems 88
-- New Functions for C++ 89
-- Constraint Satisfaction and frame-Based 89
-- Expert System 89
-- Validation by I/O of FBE System 93
-- Chapter Summary 96

III. Methodology 99
-- Introduction 99
-- Specific Classification Schemes for Software Reuse 100
-- Research Methods 101
-- Research Procedures and Formats 101
-- User-Interface 103
-- Searcher Mechanism 108
-- Searcher System Roles 109
-- Retriever Functionality 115
-- Syntax and Semantics for Entity Descriptions 117
-- Description of Browser System 119
-- Link Classification Scheme 124
-- Description of Database 125
-- Package Domain 126

-- Classification Schemes 130
-- Projected Outcome 130
-- Resources To Be Used 131
-- System Measurement 133
-- Link Weight 135
-- Adequacy 136
-- Research Method to Verify Usefulness 137
-- Reliability and Validity Procedures 138
-- Chapter Summary 140

IV. **Implementation of the Guides-Search System and Evaluation** 143
-- Introduction 143
-- Environment and Characteristics 144
-- Reuse Support Information in the Guides-Search 145
-- Unconditional Rules 146
-- Conditional Rules 147
-- Global Defined Variables and Local Defined Variables 148
-- Information File Structures 154
-- The Mapping Phase 157
-- The Ordering Phase 158
-- The Searching Phase 158
-- Evaluation of the System 162
-- User Interface 164
-- Data Presentation and Analysis 168
-- Discussion of Finding 174
-- Chapter Conclusion 176

V. **Conclusions and Recommendations** 179
-- Introduction 179
-- Summary of the Study 179
-- Answers to Research Statements 182
-- Conclusions 185
-- Recommendations 187

References 190
Annotated Bibliography 205
Appendix A: Numbers of Modules from GAMS 221
Appendix B: Reusability Tally Sheet 222
Appendix C: Recording Sheet 223

List of Figures and Tables

Figure 2.1	Classes	33
Figure 2.2	Objects	34
Figure 2.3	Access Control in C++	36
Figure 2.4	GNN (Gaussian Random Numbers)	38
Figure 2.5	High Level Programming Languages Path	49
Figure 2.6	Class-Object	79
Figure 2.7	Intersection	86
Figure 2.8	Euler Venn Diagram	87
Figure 2.9	Symmetric Difference	87
Figure 3.1	General Framework for Software Reuse System	103
Figure 3.2	An Overview of the System Scheme	106
Figure 3.3	Semantic Structure to the Dialogue Menu Type	119
Figure 3.4	Composition by Merging	122
Figure 3.5	Composition Divide	122
Figure 3.6	Cyclic Graph	123
Figure 3.7	Cyclic Problems Solved	124
Figure 3.8	Relation Value Retrieval	127
Figure 3.9	Component/Property Linkage Diagram	129
Figure 4.1	Tree Structure	151
Figure 4.2	Database Tables in Category Packages	155
Figure 4.3	Ordering	158
Figure 4.4	Data Structures to Support Searcher	161
Figure 4.5	Searcher Screen	165
Figure 4.6	View Components Screen	167

Table 1	Software packages to be Used	133
Table 4.1	Query to Specific Questions	169
Table 4.2	Inteaction, Search, and Browser Times for Three Measures	172
Table 4.3	Components Classified in Real Time, By User Time, and By System Time (1,100 Functions)	172
Table 4.4	Components Classified in Real Time, By User Time, and By System Time (2,300 Functions)	173
Table 4.5	Findings for Functions Counted, Notes Found, Unique Components Identified, Number of Cyclic Found, and Number of Components	173

CHAPTER I

INTRODUCTION

The literature acknowledges that there has been a need for software sharing and reuse for quite some time (Endoso, 1992; Full Computing Reviews, 1990; Griss, 1993; Jones, 1994). The need for ways to improve the software development process has led many companies to focus on software reuse. This need was recognized in the late 1940s. The SHARE library, a repository of subprograms donated by users of IBM equipment, was one of the first attempts to address these needs.

According to the opinion of ACM Computing Reviews (Full Computing Reviews, 1990), however, routines from this library have frequently been unreliable. The Collected Algorithms of the ACM (CALGO) also have a long history, and have since about 1970 become somewhat more reliable. Commercial vendors of mathematical software libraries such as IMSL, Inc. (1987, 1989) and NAG, Ltd. (1986), because they have a vested survival interest in the quality of their product, have also become more proficient in the construction, distribution and maintenance of components of mathematical software in recent years. Still, Poulin and Werkman (1995) contend that reusable software libraries often suffer from poor interfaces, too many formal standards, requirements of high levels of training, and a high cost to build and maintain. Novak

(1995) agrees, adding that software reuse has also been inhibited by the many different ways in which equivalent data can be represented.

The need for better reuse techniques continues to be a concern (Baker& Kauffman, 1991; Biggerstaff, 1994; Chauvet, 1995;Esteva, 1995). Over the years there have been numerous articles, books, symposiums and workshops devoted to the topic of software reuse. Most of the literature deals with methods for construction. Little has been found to address the problem of discovering software components or designs that meet specific needs (Baer, 1997; James, Sangiovanni-Vincentelli, & Alberto, 1997; Poulin & Werkman, 1995; Krueger, 1992; ACM Full Computing Reviews, 1990). It has become obvious that success is only attainable if it becomes less costly to discover an existing software component or software related artifact than to develop a new one. Moreover, current research points out that the majority of reuse today involves user interface and systems-related functions (Baer, 1997; Novak, 1992).

With that in mind, there are two fundamental points which need to be addressed for reusable component systems to be successful, from both the user's point of view and the system itself. With respect to systems issues, the model must be as maintenance free as possible. Maintenance of classification must become more reliable and less tedious. But this is not an easy task (Freitag, 1995). Indeed, it requires a great deal of efforts and is a challenge to the proposed investigation which will answer to both needs as described.

The goal of the proposed thesis is to provide a methodology to classify software components in general and two mechanisms, specifically - searcher and user-interface - to use a classification developed by the methodology to discover software that meets a

specified need. In summary, the contribution of this study is a recursive methodology that provides interaction between system and programmers for finding reusable components. To achieve this goal, this researcher reviews the current problems and the complexity of reusability, as well as current methods.

Background of the Problem

The advent of the computer in the latter half of this century caused a major revolution in the processing of information. The tools used by analysts and engineers in achieving information processing objectives have continued to evolve alongside technology (Quinian & Ross, 1989). The increased use of computers, information systems concepts, and approaches gave birth to systems modeling (Blissmer, 1991; Whitten & Bentley, 1989). The technology revolution created a proliferation of software applications to meet every conceivable need. Programmers and analysts were commissioned to create new applications and, as a result, costs escalated as applications have become huge.

In order to meet the growing need to control costs and to analyze applications, a full-scale research movement gained momentum in the early 1970s, which led to the development of expert, artificial intelligence, and knowledge based systems (Klein, 1995; Turban, 1995; Van Horn, 1986). Programs that emulate human expertise in well defined problem domains were called developed: expert systems, neural nets, and fuzzy logic, among others (Frenzel, 1989; Gold & Plant, 1990; Jackson, 1992; Klinker, Linster & Yost, 1995; Plant, 1992).

According to authorities, expert systems have and continue to impact efficiency, effective, and expertise associated with applications in business and industry (Chen & Prinz, 1994; Copley, 1994; Giarratano & Riley, 1993). The problems solved with these applications in the business and engineering areas have resulted in increased efficiency and productivity with minimal time and money invested (Holden, 1992). The primary concentration of expert systems research used for information retrieval focuses on mathematical applications.

Current literature explores promising techniques to reduce and control software production costs and to improve the quality of reuse (Biggerstaff & Richter, 1987; Caldwell, 1994; Weigret & Jang, 1992). This technique is defined as the adaptation or incorporation of previously developed software components, designs, or other software-related artifacts (e.g., test plans) into new software or developmental paradigms (McClure, 1995; Schlukbier, 1995; Schrage, 1995; Tibbetts & Bernstein, 1995). Software reuse essentially catalogs engineering processes, and also identifies, reorganizes, and then reuses existing software (Krueger, 1992; Novak, Hill, Wan, & Sayrs, 1992). Some of the goals are to improve system reliability and reduce costs by using proven components (Esteva, 1995; Frakes & Pole, 1994).

For users to discover software that meets programmers' needs, IMSL (1987, 1989) and NAG (1986) provide hard-copy software component catalogs and Quick Reference guides. In every issue, ACM Transactions on Mathematical Software provide a list of algorithms published in the previous four to five years. This has been an adequate mechanism for discovery of mathematical software because of the standard terminology

that exists in the field of mathematics. For disciplines that are partially or totally non-mathematical, the situation is not so advantageous. This study relates to data-driven non-mathematical and semi-mathematical algorithms.

Barriers and Issues

Current methodology for reusable software has not been wholly successful. Many researchers address different processes (Biggerstaff, 1989; King, 1995; Price & Girardi, 1990; Redwine, 1989; Prieto-Diaz, 1987; Scheier, 1996; Shoesmith, 1996). The only discipline in which software reuse has consistently been more common than re-invention is mathematical software. Many theories suggest that software development must utilize current methodology for reuse. These methods include structured programming, abstract data types, or object oriented programming (Coad et al, 1994; Carmichael, 1994).

Semi-mathematical software reuse is common, but has been less successful for several reasons:

(1) The intellectual investment per unit of is substantially larger for mathematical software than for software in most other disciples;

(2) The background of experience or education required to construct high quality (or even some of the most simple) mathematical software is not common;

(3) Mathematics provides a framework for classification that has been standardized by several centuries of use.

The first two factors tend to increase the cost of reinvention, while the third tends to reduce the cost of discovering components. In order for software reuse to be

successful, it must be less costly to discover a software component or software related process that satisfies a given need than to develop one anew. This is the basis of pattern languages.

Relevance and Significance

Inconsistent conclusions of new production methodologies have been responsible for the lack of success in reusing existing software. Perhaps failure has been due to the lack of tools in supporting reuse during development, rather than inadequate methodologies. The significant of the proposed study is its ability to meet this need.

The investigative research also relates to the increasing need for an accurate, effective and quick search of entire databases for both routines and phrases. The present study has attempted to provide a methodology to classify software components in genre (not just mathematical software), and a mechanism to use a classification developed by the methodology to discover software that meets a specified need. Such a tool would meet the needs of non-mathematical users.

As previously explained, to aid users in discovering software needs, commercial vendors of mathematical software libraries such as IMSL (1987, 1989), NAG (1986), and MathPro (1995) provide software component catalogs and/or Quick Reference guides. They have become more proficient in constructing, distributing and maintaining components in recent years. The proposed study will help fulfill this need, which further emphasizes the significance of the investigation for semi-mathematical software.

Many discussions on software reuse focus on the mechanisms of construction . To be successful, a developer must have a large collection of useful and reliable parts and

also a mechanism for discovering components. Software reuse should not be practiced in environments where it will cost more to discover existing components than to invent them anew. In fact, this is often a major question for companies to resolve. The purpose of the investigative study will be to describe a method to classify software components and a system to use such a classification efficiently to discover software component needs. The classification and retrieval methodology will apply to software, hardware, patents, books, legal cases, and others of a related nature.

The methodology used to classify software components and the mechanisms used in classifications developed to discover existing software will be reviewed. One of the mechanisms reviewed is called a Guides-Search by this researcher. This is a system in which processes are retrieved by carrying out a structured dialogue with the use. Guides-Search is used to display the results for interactive use. A mechanism such as this is intended to be independent, but equivalent views of the same classification. They can be employed when appropriate. A review of this nature adds significance and relevance to the study. Relevance of the study is also explained in following sections.

1. Classification Schemes

A classification scheme is described as a generalization of the use of processes. It will provide both the structure of questions to be posed to the user and a set of possible answers to each question. It will consist of specifying a set of properties in each component to be classified and then refining those properties by specifying additional properties they may enjoy. This is accomplished by using binary relations of the form *entity* **relation** *value*. When the *entity* is a component or property, the *value* can be a

component, property or even an atom (a term for which no further description is provided).

A **relation** is equivalent to a process. The *value* of a relation is equivalent to the position in the dimension of "classification space" as described by that process. Consider SGEFS, for example. SGEFS is a Program Unit name defined in ANSI Fortran77. It is a subprogram for solving determinate systems of linear algebraic equations. Its name means (S)ingle precision, (GE)neral square system of linear equations, (F)actor and (S)olve. If the purpose of SGEFS is to solve a general single precision real NXN system of linear equations, SGEFS uses LINPACK subroutines SGECO and SGESL. That is, if A is an NxN real matrix and if X and B are real N-vectors, then SGEFS solves the equation A*X=B. The matrix A is first factored into upper and lower triangular matrices U and L using partial pivoting. These factors and the pivoting information are used to find the solution vector X. An approximate condition number is calculated to provide a rough estimate of the number of digits of accuracy in the computed solution. If the equation A*X=B is to be solved for more than one vector B, the factoring of A does not need to be performed again and the option to only solve (ITASK .GT. 1) will be faster for the succeeding solutions. In this case, the contents of A, LDA, N and IWORK must not have been altered by the user following factorization (ITASK=1). IND will not be changed by SGEFS in this case.

An example of the notation to specify that the value of the *functionality* for the component SGEFS of *solve* is to write the relations. *SGEFS* **has-functionality** *solve*. *Components* and *Properties* can be arranged into hierarchies. Properties can enjoy other

properties specified by relations. A relation of the form property is permitted. Suppose that the component SGEFS enjoys the relation *SGEFS* **has-operand** equation-ALD. The "equation-ALD" is the name of a property that enjoys the relations *equation-ALD* **is-a** *equation*. *Equation-ALD* **has-kind** *algebraic*. *Equation-ALD* **has-determination** *exact* and *equation-ALD* **has-linearity linear**. The relation *equation-ALD* **is-a** *equation* denotes a relation in a hierarchy, while the other relations denote properties enjoyed by the property equation-ALD. It is assumed that the **is-a** relation is known to software that uses this classification scheme. It must also be reflexive, transitive and anti-symmetric.

2. *A Guides-Search System*

A **guides-search** is a software system that enables a user to discover software needs. The interaction of searcher and user will mainly consist of alternately displaying questions known to be germane. Once a user has selected a question, the searcher will display possible answers to that questions. Questions will correspond to relations and the series of answers will correspond to the set of values of that relation. If there is a small number of retrieved components, the user may simply view them. If there is a large number, the searcher will construct a new and smaller database. The set of questions and answers will be presumably less and a new dialogue will begin.

3. *Users Interface*

Communication between the user and system will be represented through a listing of results and germane questions. Interaction consists of the system alternately displaying questions and answers to the user. Once the user has selected a question, the

system will display possible answers to that question. The questions will correspond to relations and the set of answers will have the set of values of that relation.

There are many interpretations of the meaning of non-excluded components. The proposed study will consider systems performance which has led to the adaptation of an alternative interpretation. It is important to explain that there are at least two different interpretations of "non-excluded components." One view is that one considers all answers that the user has provided. The second interpretation excludes components that do not enjoy the specified relations. This can cause the searcher to operate unacceptable or very slowly. An alternate interpretation is to allow the user at any instant to retrieve the set of components that enjoy the specified relations and to consider the set of non-excluded components to be those present as a result of the last retrieval operation. But the set of non-excluded components is not affected by questions answered since the last retrieval operation.

The later allows a searcher of somewhat better performance than the other. However, it may present an answer that is inconsistent with other questions already answered. For example, the user might have software to evaluate polynomials and to evaluate integrals in the FORTRAN programming language, but software only to evaluate polynomials in the C programming language. If the individual adopts the former interpretation of non-excluded components, a user having selected the relations "has-functionality evaluate" and "has-language C" would be presented only with the answer "polynomials" when answering the question "what is the operand?" Adopting the latter

interpretation allows both “integrals” and "polynomials" to be displayed, even though no components would be selected if "integrals" were chosen.

This researcher believes the alternative view will allow users at any moment to retrieve the set of components that enjoy specified relations. The study will also consider the presence of a set of non-excluded components which should not be affected as a result of questions answered in the last retrieval operation.

Purpose of the Study

Most discussions of software reuse focus on mechanisms to construct reusable software. For reuse to be successful, however, there must not only be a large collection of useful, reliable parts available, but also a mechanism to discover components that meet a specified need. Software reuse should not be practiced in environments where it costs more to discover components that meet a specified need than to invent them anew. The purpose of the present study is to describe a method to classify software components, and a system to use such a classification efficiently to discover software components that meet a specified need.

Specifically, the purpose of the present research study is to provide a flexible system, comprised of a classification scheme and searcher system, entitled Guides-Search, in which processes can be retrieved by carrying out a structured dialogue with the user. The classification scheme provides both the structure of questions to be posed to the user, and the set of possible answers to each question. The model is not an attempt to replace current structures; but rather, seeks to provide a conceptual and structural method to support the improvement of software reuse methodology.

The investigation focuses on the following goals and objectives for the classification scheme and searcher system:

(1) the classification will be flexible and extensible, but usable by the searcher;

(2) the user will not be presented with a large number of questions; the user will never be required to answer a question not known to be germane to the query;

(3) the user will not be presented with a large number of possible answers to any single question; and

(4) the user will be allowed to specify an answer, even though he or she did not know exactly what question the searcher will pose to elicit that answer. (This is similar to a key word search.)

Research Statements to be Investigated

The research investigation will be specifically designed to address the following:

- A comprehensive review of related literature will indicate that existing techniques are inadequate in supporting information requirements.
- There is a significant need for a new approach or method to classify software components and a system to use such a classification efficiently to discover software components that meet a specified need.
- Design of a searcher software system used to discover software needs will address the following three concerns: (1) it will allow users to retrieve the desired software without being required to answer an inordinate number of

questions; (2) it will present an adequate number of possible answers but not too many to any one question; and (3) it will not artificially restrict the performance of an expert user.

- There is a significant set of guidelines, or model, that exists to select software for reuse and thereby reduce the cost of software production as related to non-mathematical applications and systems.

The methodology developed for the proposed investigation will consist of five distinctive and sequential steps. These are described as follows:

Step 1: A review of literature will be conducted, relevant to the background of computing in terms of types of computing/problem solving, pattern languages, research on reuse measurement, complexity and analysis, and use of OO for program development. It will also focus on the background of the study's theoretical model and background of practical machine for the study's model.

Step 2: Literature review results will be recorded, analyzed, and compared to determine any inadequacies The scheme will use a guides-search engine to describe relations.

Step 3: Conclusions will be developed and summarized to answer questions based on the review of the literature and presentation of a flexible classification guides-search system.

Step 4: A simple model will be developed which will include the necessary features to classify software components and systems to utilize such classifications efficiently to discover components for specific needs.

Step 5: Recommendations will be made from the findings and summary.

Assumptions and Limitations

The researcher assumes that, from the comprehensive literature review, guidelines can be established from the literature review to assess the inadequacies of existing techniques to support information reuse needs and the development of new flexible classification schemes. It will be assumed that the results can be specified and evaluated to provide a model or set of guidelines. However, the formation of the study's results will not be a randomly conceptualized assumption. Rather, formulation of such guidelines is seen to constitute an accepted goal of many types of research investigation (Babbie, 1990; Downie & Heath, 1984).

Conclusions in the proposed research will be limited by that amount of information and data discovered in the documents, reports, research, and other related materials. Other limitations existed in using this type of technique in providing guidelines and validating findings. This appears consistent no matter what methods are used (Babbie, 1986, 1990; Fowler, 1984).

To classify processes, the present investigation focuses on the provisions of a mathematical method derived from Relation theory. It assumes that the model for a flexible classification system (generalization of the use of facets) could be developed for semi-mathematical software reuse and classification. It is believed that the overall

approach to the reusable software methodology may turn out to be the most important contribution of the research, which is to make discovery of a classification more reliable and less tedious.

It is also assumed that the proposed study has significant and relevance to complexity theory in general in that it attempts to provide a methodological tool for discovering software for reuse, and thereby reduce complexity. Complexity theory relates to the subject of the proposed study because it impacts the ability to reuse. Complexity is a realm that is difficult to define and even harder to understand because it deals with the aggregate of many simple things that can create complex forms (Goering, 1995; Kochen, 1984). Complexity theory is actually the study of how much computing is required to solve various kinds of problems, especially those related to large software systems (Devanbu, Brachman, Selfridge, & Ballard, 1991). It deals with systems as a whole. Researchers often create computer simulations of extremely intricate systems. They then use those computer programs to develop hypotheses that can later be tested with experiments. A natural measure of complexity is the entropy rate of a random process that models the problem.

Definition of Terms

A number of terms and designations are applicable to the proposed study. and have been defined for clarity.

Classification Scheme: In general terms, a classification scheme is defined as a technique for supporting information needs (Poulin & Werkman, 1995; Prieto-Diaz, 1987). Popular schemes include hierarchical and faceted classification (Biggerstaff &

Perlis, 1989). Top levels in the hierarchical scheme consist of an application domain and refinements, such as computer graphics or numerical analysis. Lower levels often represent some type of functionality such as solve equations or evaluate integrals and programming language.

Faceted classification, on the other hand, considers facets as independent views of the properties of software components. Many of the objections to a hierarchical scheme are answered by faceted classifications (Forslund, 1995; Klein, 1995).

Flexible Classification Scheme: This designation is defined in the present study as a part of the Guides-Search system, which was developed by this researcher. It is a generalization of the use of facets. A flexible classification system specifies a set of properties of each component to be classified. Properties are then classified using the same methodology.

Current literature states methodology that employs classification and retrieval works well with artifacts not related to software, such as hardware, patents, books, and legal cases, etc. (Full Computing Reviews, 1990; Klein, 1995; Tibbetts & Bernstein, 1995). It uses binary relations in the entity relation value form. Here, the entity will exist as a component or property and the value may also be a component, property, or atom (a name for which no further description is provided).

Facets: Facets are considered by Prieto-Diaz (1985) as dimensions in Cartesian space. The value collection of facets constitute the coordinates of a point in a space. They are also considered to be independent views of the properties of software

components. The properties are sufficient and necessary to include application domain, functionality, and operand.

Guides-Search: This term is defined as an approach system. It contains a classification scheme and searcher system in which artifacts can be retrieved by carrying out a structured dialogue. Guides-search is a name this researcher has coined for the research engine employed in the study, similar to the method utilized by Esteva (1995) who built a library engine and called it Snooper. For the present project, a library was built to prove the theory and has been called Guides-Search.

The retriever is a software component that retrieves all necessary files to utilize a selected component. In the simplest case, the component date base and searcher reside in the same computer and the retriever simply produces a list of file names necessary to utilize selected component. The classification scheme provides both the structure of questions to be posed to the user, and the set of possible answers to each question.

Guides-Search Interface: The searcher, as created by this researcher in a manner similar to that used by Esteva (1995), is a software system that can be used to discover software needs. Dialogs will mainly consist of alternating questions known to be germane and displaying possible answers to a selected question. Questions will correspond to relations and the set of answers to a question will consist of values to that relation.

Processes: Software processes can include design documents, source code, specifications, and test plans, among others. Any text file that is part of the software

components. The properties are sufficient and necessary to include application domain, functionality, and operand.

Guides-Search: This term is defined as an approach system. It contains a classification scheme and searcher system in which artifacts can be retrieved by carrying out a structured dialogue. Guides-search is a name this researcher has coined for the research engine employed in the study, similar to the method utilized by Esteva (1995) who built a library engine and called it Snooper. For the present project, a library was built to prove the theory and has been called Guides-Search.

The retriever is a software component that retrieves all necessary files to utilize a selected component. In the simplest case, the component date base and searcher reside in the same computer and the retriever simply produces a list of file names necessary to utilize selected component. The classification scheme provides both the structure of questions to be posed to the user, and the set of possible answers to each question.

Guides-Search Interface: The searcher, as created by this researcher in a manner similar to that used by Esteva (1995), is a software system that can be used to discover software needs. Dialogs will mainly consist of alternating questions known to be germane and displaying possible answers to a selected question. Questions will correspond to relations and the set of answers to a question will consist of values to that relation.

Processes: Software processes can include design documents, source code, specifications, and test plans, among others. Any text file that is part of the software

engineering process is defined as an artifact (Franks & Pole, 1994; Lemaire & Moore, 1994).

Relation: According to McAllister (1995), "a relationship is a modeling object with two or more roles, each of which links a specific entity to the relationship" (p. 133). A relation, however is equivalent to a facet, a dimension in Cartesian space (Prieto-Diaz, 1985). The value of a relation is equivalent to the position in the "classification space" dimension described by that facet (Prieto-Diaz, 1985). An example of this notation in specifying that the value of the functionality for the component ABCDEF is solve is to write the relation ABCDEF has-functionality solve. For the purposes of the study, a relation is equivalent to a process. The value of a relation is equivalent to the position in the dimension of "classification space" as described by that process.

Relations need not be symmetric, transitive, or reflexive, although they may exist in this form (Biggerstaff & Perlis, 1989). When several groups collaborate to classify a large collection of software or several unrelated bodies of software, it is of the utmost importance to use a common dictionary of relation names or a same-as relation to connect names in different classifications.

Chapter Summary

This chapter served as an introduction to the research investigation. It discussed the background of the study introduced the problem of concern. It was noted that the goal is to provide a methodology to classify software components in general and two mechanisms, specifically - searcher and user-interface - to use a classification developed by the methodology to discover software that meets a specified need. The contribution of

this study is a recursive methodology that provides interaction between system and programmers for finding reusable components.

Chapter 2 provides an examination of the relevant literature. The background of computing leading toward abstraction and reuse is first reviewed. Following discussion focuses on reuse in terms of measurement, complexity, and new functions for C++, among other topics. Use of object orientation (OO) for program development and reuse, the background of the study's theoretical model, and the background of practical machine for this study's model are additional concerns of the literature review. This information provides a basic foundation for the study.

Chapter 3 presents a flexible classification scheme that attempts to address the inadequacies of existing classification approaches. A detailed analysis of components and properties is undertaken. Chapter 4 concludes the present investigation. A summary is first provided, followed by answers to the study questions and conclusions based on the results. Recommendations follow.

CHAPTER II

RESEARCH AND LITERATURE REVIEW

Software reuse is a current technology whereby artifacts of the software engineering process are cataloged, reorganized, identified for reuse, and reused (Ransom & Marlin, 1995; Tracz, 1988). The goals of software reuse include improved system reliability and reduced system cost by using problem components. The reuse of software is an important aspect of controlling and reducing software costs and improving quality (Humphrey, 1990; Marlin, 1995; Prieto-Diaz, 1993). The present investigation focused on this topic.

The purpose of this chapter of the study is to present a review of the literature on the reuse of software components, design and programs. To achieve this goal, however, it is first necessary to review the background of computing leading toward abstraction and reuse. Following discussion focuses on reuse in terms of measurement, complexity, and new functions for C++, among other topics. Use of object orientation (OO) technology for program development and reuse, the background of the study's theoretical model, and the background of practical machine for this study's model are additional concerns of the literature review. This information provides a basic foundation for the study. A review of historical developments such as the evolution of artificial intelligence, expert systems, knowledge-based systems, and object-oriented technology, leads to the conclusion that there is a need to reuse components.

Capsule Description

Mathematical problem models and representational models are beyond the scope of the present study because computational problems are too complex. Perfect mathematical problems models formulate equations for any given problem. Perfect mathematical representational models captured the problem's relevant properties, such as part structures and components and transform solutions for a given problem. Because of the complexity in mechanical analysis of the equations, it seems more realistic to assume the requirements of the interactive system from software engineers to specify a specific needed. The current research devoted efforts with these ideals in mind. The model covered the areas of propositional logic, set theory, Boolean algebra, relations, Automata for process and Graphic-Matrix Theory for data representation, and isomorphism and homomorphism for verification. However, before the history and literature related to the current background of the research can be reviewed, parameters for object oriented software and its association to reuse must be established. It is only with the understanding of object-oriented programming that a model can be fully adopted.

Object-Oriented Programming

Object-Oriented Programming has been evolving rapidly as a technology that will support aspects of the reusable application (University of Liverpool, 1997). New technology concepts have moved out of the research communities into the commercial world and can be found Simula, Algol-60, Smalltalk, C++, Fortran95, and LISP, among other object-oriented programming found in today's marketplace. Each programming paradigm may have different name, but share the same spirit and common goal - to have a machine do what programmers want them to do in the way that they can described for

better maintenance, structure, complexity, power and reusability. In this manner they have a significant economical impact and influence products quantity.

There are numerous objects definitions, as noted by Bednarczyk, (1996). Object-Oriented methodology represents an approach to bring technology in line with business by providing a new way for groups to think about processes and information systems (Booch, 1994; Taylor, 1990). Construction of models expresses business concepts as real objects which include people, places, and things. Technology-based details are suppressed. This method uses object-oriented programming and case tools. Routines and procedures are considered to be objects.

However, in general it can be seen in the literature that Object-Oriented software is all about objects and related methodology. An object is a Black Box which receives and sends messages. The Black Box contains code, sequences of computer instructions, and data information upon which the instructions operate (Coad et al., 1994; Hutt, 1994). Traditionally, code and data have been kept apart. In object-oriented programming, code and data are merged into a single indivisible thing - an object. The Black-Box is mainly responsible for sending and receiving messages, where messages define the interface to the object. Object is defined via its class (Montlick, 1997), which carries out class actions, often called methods. Classes and objects are related but they are different.

Object-oriented information systems provides a different way of thinking. Learning to "object think" is, in fact, a core requirement to understanding. Reusable groups of software code can be used and reused to save time in building custom applications (Bartholomew, 1996; Anderson, 1996). In this way, applications can be adapted to a changing business or project without the requirement of changing the

underlying code. Adaptability is essentially the key. Off-the-shelf software that is built using object technology can offer incredible flexibility, Anderson (1996) explains. Applications can be easily changed because they are built using reusable, modular components.

Classes

Class is the schematic of a object with determines everything about the object. Within the Object-Oriented context, object is an instance of a class. That is, any object is unique and associated with an identifier to the base-class. Bednarczyk (1996) explains that "a class is a specification of structure (instance variables), behavior (methods), and inheritance (parents, or recursive structure and behavior) for objects."

According to Mattison and Sipolt (1995), Object-Oriented (OO) programming is streamlining the way industrial engineers are building corporate information systems. But success will not be realized until everything is treated as objects - software and hardware - and the information systems department is restructured to fit that model. In their view, the reason for the need for this approach is because the life cycle of current traditional systems no longer provides an accurate model to explain how objects are perceived, created, and delivered due to assumptions that are no longer true. One of the primary assumptions of older methodology is that most system development work involves the creation of new systems, not retrofitting old system components into a new architecture. "But it is exactly the latter that defines what the majority of corporate computer system development work will be for the next several decades" (Mattison and Sipolt, 1995, p. 53).

Consider the following example called "Bank-Account." Bank-Account has Checking-Account and Saving-Account. Thus, Checking-Account and Saving-Account are corresponding unique attributes of Bank-Account. That is, Bank-Account "has-a" Saving-Account; Bank-Account "has-a" Checking-Account. Bank-Account can be presented as a "base-class" where

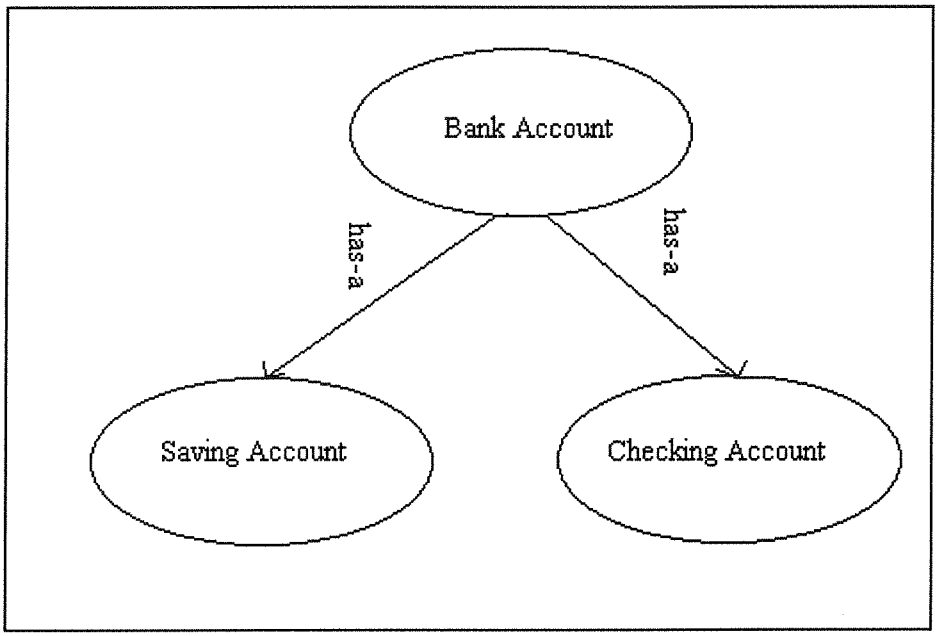


Figure 2.1: Classes

Therefore, the objects of class Bank-Account has the following forms

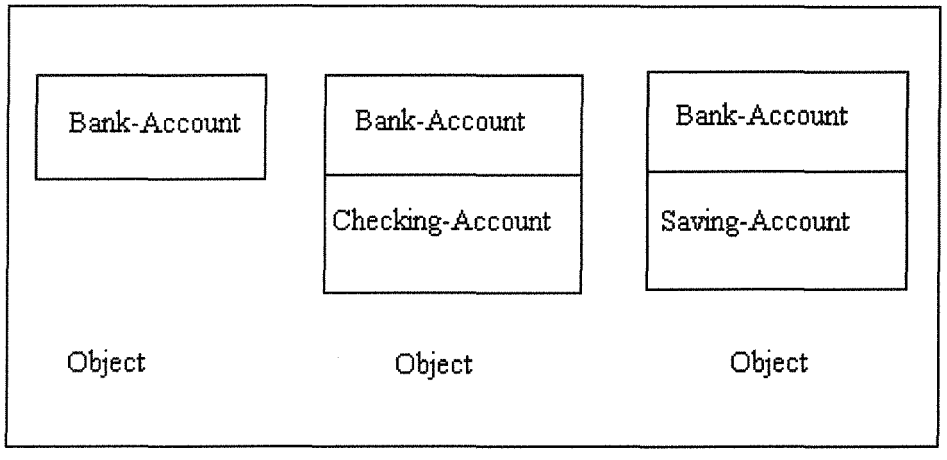


Figure 2.2: Objects

Further information on OO context may be described. Bednarczyk (1996) denotes class is an object via visa object is a class. The researcher defines class is a object true only in 1 level (called single-hierarchy) systems. In his view, it is not true in C++ because C++ classes are accessible to programs. On other hand, object is a class denoted in Bednarczyk (1996). An Object has encapsulation, inheritance, composition and polymorphism, as discussed in subsections below.

Encapsulation

The class encapsulates and protects the data from inadvertent or malicious use. It is the process that distinguishes the outside interface of the object from the internal - that is, access to the object. One does not needed to understand the internal detail of the object to receive requested information. Encapsulation also implies that the internal detail of the object could be changed without any effect to the information requested. To define a program and solve the problems, relationships to from each object can be

presented in the form called "Top-Down topological hierarchy." These principles are believed to reduce software maintenance and increased reusability.

Encapsulation consists of two features: interface and implementation. The interface feature defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. It is the unique way that the particular object tells potential clients what operations are available and how they should be invoked.

With respect to the implementation feature, it is important to understand that object implementations do not depend on how the participant objects invoke the object in question. It a black-box because it allows access without required a knowledge of how the information is implemented. Practically, besides defining the methods for the operations themselves, object implementation often allows the construction of object by using other objects or non-object facilities to make the object state persistent, to guard the object, and to implement methods. It consists of the following (as depicted below):

PRIVATE data and functions dedicated to manipulating that data.

PUBLIC functions which form the interface to access the class or objects. The different between private and public is that a Class declared member public allows everyone access. Private is only accessible inside of the class.

PROTECTED is accessible inside of the class and its inheritance classes. Inheritance is a mechanism allows sharing the commonality among classes.

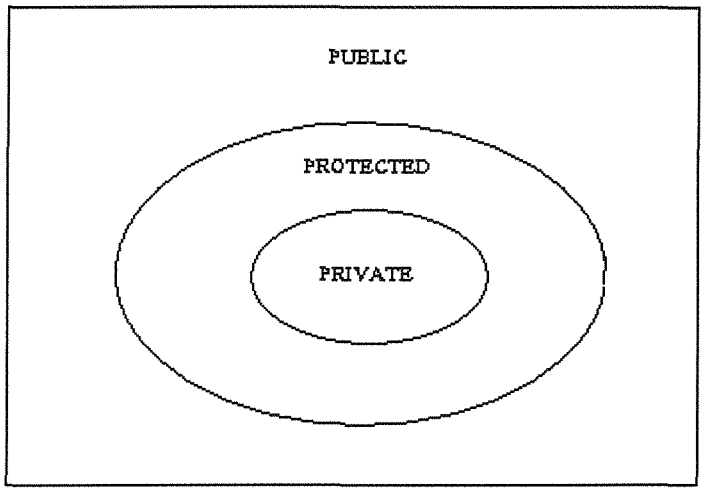


Figure 2.3: Access Control in C++

Inheritance

Inheritance is a key feature of the Object-Oriented paradigm. Inheritance is one of the ways that allow objects with similar operations and behaviors to be closely organized in the form of taxonomical hierarchies. It is a core concept that can be used to model the problem (abstraction) and code reuse in the real-world (Lea, 1993). With regard to the present thesis, abstraction and inheritance types can be emphasized by the following abstraction example:

Abstraction means to share a commonality between each class. For example, in the algorithm for generation of Gaussian random numbers (FORTRAN 77), components of the function DRANE and SRANE have the following:

```
DRANE:
        has-function Uniform-random number
        has-precision double
        has-output scalar
        has-function Exponential-random-number
        has-operand Number-R
END DRANE

SRANE:
        has-function Uniform-random number
        has-precision
                if p = s then
                        single
                else
                        double
                endif

        has-output scalar
        has-function Exponential-random-number
        has-operand Number-R
END SRANE
```

First, it can be recognized that functions DRANE and SRAND are almost identical except one has precision double. The other has single or double precision. If a function GRN (Gaussian random numbers) is carrying commonality of the DRANE and SRANE, then:

```
GRN:
        has-function Uniform-random number
        has-output scalar
        has-function Exponential-random-number
        has-operand Number-R
END GRN
```

GRN restores the commonality of DRANE and SRANE by letting function DRANE and function SRANE inherit from:

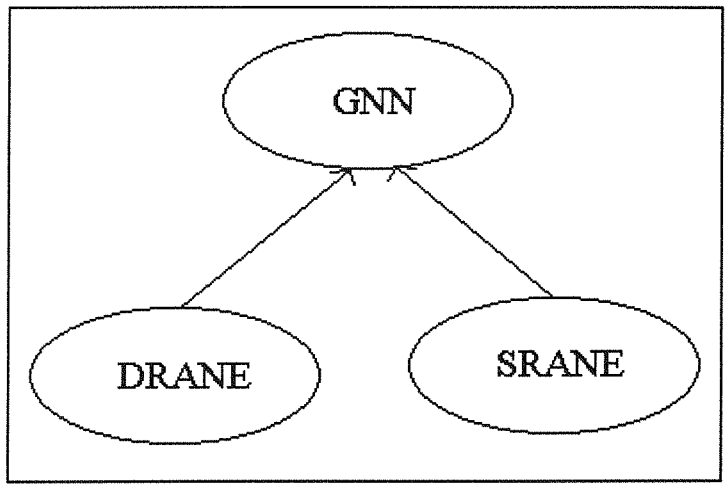


Figure 2.4: GNN (Gaussian Random Numbers)

GRN is a father or super class of the DRANE and SRANE. DRANE and SRANE inheritance to GRN. In other words, GRN is a DRANE and GRN is a SRANE. That is, Class DRANE and SRANE are subclass of its parent class GRN when all components of GRN are the components of DRANE and SRANE as well. Additionally, the subclass DRANE is constrained by one additional double precision and class SRANE has precision either single or double.

Reuse Via Inherit

A second reason for using inheritance is to avoid "done-it-twice" while allowing other to share data and functions. In the real-world of actions, the idea is to organize relevant objects into a taxonomically which goes from the general object with similar operations and behavior on the top to an increasingly divided form. In mathematics during model construction, using, for example, the function DRANE, function DRANE and function SRANE, it is important to note that both contain "has-function Uniform-random number," "has-output scalar," "has-function Exponential-random-number," and

"has-operand Number-R2" components. These were inherited from function GRN. Function DRANE has "has-precision double" defined as private component and function SRANE has "has-precision if p = s then single else double" as a private defined component. Likewise, sharing among these function can also be described as following, exemplifying the goals of Object-Oriented Programming.

Function DRANE like function SRANE
    Except has-precision double
End Function DRANE

The real advantage of inheritance is gains associated with constraints to the superclass, sometimes called a baseclass. Thus, class Q is a subclass of superclass P. Then for any features, constraints of a class P, class Q can also be used. That is, an attribute, constraint and transition network of class P can be used for class Q. But class Q is stronger than class P as instances of Q have all properties and components of class P with the addition of one or more specific features in Q (Douglas, 1993). Inheritance and polymorphism are used to represent classification in a application domain. Day (1995) explains that polymorphism is the ability to write generic code that works for families of related types. It is a name that has several meanings and implementations. These include Overloaded Function Signature, Overloaded Operators, and Virtual functions (undefined until runtime).

Composition

Object composition allows an object be used as a component part of other object. Object composition techniques bring together components parts from one object to another as needed. Once again, consider the Bank-Account example. As previously noted,

Bank-Account is composed of Checking-Account and Saving-Account. However, Bank-Account's composition has an internal structure that describes the relationships to its accounts. Perhaps a Saving-Account or a Checking-Account may not be as complex as a Bank-Account. The programmer may not want to continue de-composing the Bank-Account and risk a jail sentence.

Another example is a bitmap file. Images from another bitmap file can be "cut-and-pasted" into a piece. Commonly, the bitmap file contains the following associated compositions:

-Object ID.

-Translation X and Y coordinate of the top left corner of the bitmap canvas.

-Depth: depth of the object in the bitmap.

-Relation: associate component relation in some degree.

-Scale X and Y: Scaling dimension in horizontal and vertical.

Polymorphism

The same message can respond to differs objects. In numeric polymorphism, consider the example of SRANE above. The result returned by SRANE is the appropriate value of kind real or double, depending on whether condition p = s or not. Likewise, consider the function COSIN. Function Cos(X) can either return the result of single or double, depending on whether the parameter value X is a single or double. This generic property has a significant impact on the portable robust application which commonly can be found in ANSI Fortran 77.

As the result, Object Oriented creates high level abstraction, large scale organization, and reduction of the complexity of the inter-relations of components. It is

clear why programming paradigms have become increasingly popular over the years, especially when note is taken of successive improvement in readability, maintainability, reliability, testability, complexity, power, structure and reusability. (University of Liverpool, 1997). There are numerous major object-oriented programming languages in use today, but there are few leading commercial ones. Montlick, (1997) has listed these as C++, Smalltalk and Java. C++ appears to have and continue to maintain the largest nucleus of programmers.

In summary, the literature supports the view that reuse is widely recognized as a most promising technique presently available to reduce the cost of software development and speed development with tested components in terms of adaptation or incorporation of previously developed software components, designs, or other software-related artifacts such as test-plans into new software or software development regimes. It may also be noted that "software reuse" means exactly what the name implies - basically it infers that, if it is possible, do not develop new code, just reuse code.

Sophistication such as this is related to the techniques of Pattern Languages. Pattern Language was first discovered by Christopher Alexander. The researcher began using pattern language to describe the events and forms that appeared in towns, buildings and construction in the world at large. The significance of his work lies in its implication and emphasis on the potential for reuse. It captures common Object-Oriented concepts to solve problems and abstract them from the underlying building blocks(objects). In other words, it provides a way to share the design expertise. It also describes a solution to a problem in an environment in such a way that allows programmers to use this solution over and over many times without ever doing it the same way twice. (Alexander, 1977).

Alexander's work begins with the "problems described," follows with "discussion of the constraints forces on the problem," and proceeds to the final task where the "solution on the problem" is provided. (Brown, 1990).

Background of Mathematics Leading Toward Computing Abstraction and Reuse.

Abstraction and Reuse

Mathematics and computation must first be described in their broad sense. Mathematics started from the unique human thought involved in number counting into the development of a notation-aimed mathematics. Mathematics is a foundation, driving computing application to precision of specification and to predict and reason about properties in the application system aspects. Mathematical foundations for reuse and those for software are closely related. The relationship described further between mathematical logic and computation "will be as fruitful in the next century" as described by McCarthy (1996). Likewise, LISP syntax is based on lambda calculus (Stanford, 1996).

Like mathematics, programming language derived from the counting of numbers to geometry-coordinated calculus and analysis methods. Computer programming languages also have a long history from the well known Turing Machine to today's languages. Each has improved in development over time, especially since costs issue have increased and subsequently been recognized. To understand C++, an overview of mathematics history is required.

Mathematics has a long history which started from counting number through the Greek mathematicians period and led to the revitalization of science and mathematics.

The Greek period covered a splendid tradition of work in the exact sciences: mathematics, astronomy, and related fields. These are described in the works of Isaac Newton on Mathematics. (History of Mathematics, 1995). Before this period, perhaps from 2000 BC, earlier value notation allowed a larger number system base 60 to evolve and fractions to be represented. This development denoted the beginning of a higher power mathematics (History of Mathematics, 1995). Around 1700 BC, linear system thinking evolved, such as that evidenced by the Pythagorean triples a, b and c with a2+b2 = c2. The major Greek mathematics, according to History of Mathematics (1995), was the algebraic solution of cubic and quartic equations. This had a major impact on the psychological effect and enthusiasm for mathematical research revolution and led to co-ordinate geometry, calculus and analysis in the 18th Century. This new science was most clearly discovered by Newton's mechanics and is described by Harrison (1996) as follows:

> - Co-ordinate Geometry: Newton's study which discovered inverse problems. This find had a major impact to the science which led to the birth of algebraic methods which solve polynomial equations of degree 3 and 4.
>
> - The Calculus: Newton and Liebniz work discovered the calculus system which depend on irrational and infinitesimal numbers (infinitesimal either has zero or non-zero).
>
> - Analysis: Calculus was further developed during the 18th century period. However, there still existed a lack of logical method to until the 19th century, but this period began further analysis in mathematics activities.

Bessel's functions are next in importance. Bessel Friedrich Wilhelm, a German astronomer and mathematician, was the a first person to discover the approximate distance to a star. French philosopher and mathematician, Descartes Rene also impacted

history. He attempted to explain the entire material universe in term of mathematics and physics. There were many others brilliant mathematicians in this century. Christopher Wren, for example, was an English architect who was famous for his discovery of the method or plan for rebuilding the city on Classical lines. He used the idea of a refined and sober Baroque to fit buildings into irregular sites. However, it was the development of the Analytical machine by Charles Babbage that had the greatest impact. The first digital computing used a Jacquard punch card machine. It was then the birth of computation actually began (History of Mathematics, 1995).Of interest is the fact that Countess Lovelace was the first programmer; she was the person sponsored Babbage.

Computation is the essence of mathematical science. A machine instructed to carry out intellectual processes is the tool. There are at least three directions of mathematical research related in mathematics computing, according to Harrison (1995). These include:

1. Numerical Analysis: This is the first kind related to science and computing. Mainly used to solve by brute force problems like numerical integration. FORTRAN and C are common languages.

2. Computability: Is known as a branch of recursive theory which includes, among others: unlimited register machines, Turing machines, partial recursive functions, algorithmically unsolvable problems and diagonalization, Kleene normal form theorem, universal programs, and Rice's theorem. (Davis, 1994).

3. Formal Language Theory: This pertains to a theory of finite automata which includes functions such as: deterministic and non-deterministic finite automata and their equivalence to regular expressions, pumping lemma and Myhill-Nerode theorem, context-free grammars and languages, and the corresponding pushdown automata. (Davis, 1994)

Numerical analysis, computability and formal language theory are about using programming language design to activate concepts of real-world problems. There are two main activities described in scientific computing: theory development and numerical analysis (Peter, 1995). In theory development, the designer uses pen and paper to describe relevant properties -part structures, substructures and components -as related to the model of the object. Numerical analysis is the form to which the manual model is translated to the program where it can be simulated.

Programming language appears to be the most important tool to mathematicians and computer scientists. From the perspective of software engineering, computation is the ability to write programs that emphasize the outcome of the specified results rather than concentrate on how it should be built or written. Therefore, specified results must also be driven in a form of or determined by the perceptual characteristics of the inputs. It focuses on predictions of outcomes or goals to be achieved. As a consequence, it is an evolution which emerged into abstraction and reuse. From current research, three significant computation types leading to abstraction and reuse in recent years can be summarized: Functional abstraction, Data-driven and Message-driven. These are described in subsections below.

Functional Abstraction

Functional by definition is a technique that can be implemented among many different languages (Backus, 1978; Jagadeesan, 1991; Jarvis, 1995; Mannino, 1990). An example of functional language is Pi, square root functions. Many use these functions in daily use but do not realize this and take commonly expected results for granted. Perhaps, that was the intention from the very beginning.

Before 1954, programming was developed from machine code or assembly language. This had great error potential and was extremely labor intensive. Computing communities looked for a way that a language could be easily moved from one machine to another. That is, they sought adoption, improvement in readability, maintainability, reliability, complexity and reusability. Modules in FORTRAN were soon introduced. FORTRAN is a powerful programming language and is heavily used to perform numerical calculations. Modules have subsequently had a significant impact on structure programming as well. Davis (University of Liverpool, 1996) presented a good example of an unstructured program of 100 lines of code, which can have up to 10^158 paths. Using Modules program (function) to have structured program, approximately 100 lines of code could be placed into 4 separate functions. This reduced the paths to 10^33.

Data Driven

By definition, data driven refers to the results or output specified from the perceptual characteristics of the input (Harrington, 1995). It is commonly known as a basic estimating methodology such as analogy, factor/ratios and parametric. It also well known for the lexical decision task which can be found primarily on languages or bilingual translators.

Message Driven.

Message driven is believed to derive from the traditional method parallel computer which involved a traditional message-passing style of programming (Gursoy & Kale, 1996). With respect to the message-driven process, one processor can send one or many message to others while still running. Performance was a main technical issue and blocking was a main concern when message driven provided scheduling which prevented

the blocking processes. There are numerous researchers devoting serious effort into message-driven programming. Each has contributed to improvement of productivity of parallel programming. New features and techniques were introduced to simplify the task of development parallel applications. However, there are many complexities and techniques that may not clearly express some specific situations (Kale, 1996).

Example of message-driven is the Dagger system. Dagger was developed at the University of Illinois (Kale, 1996). The structure of Dagger extends from the Charm system which was also developed at University of Illinois. Dagger solves the complexity of Charm system blocking. The component "when-blocks" is included to enforce the blocking-condition to be satisfied before it can be scheduled for execute.

Language Evolution

No one is able to recall exactly when the history of computing began. However, it is known that the Turing Machine was a first computer language machine. Turing-Machine was developed by Alan Matheson Turing in the 1930s and used two binary number, "0" and "1." Turing proved that a machine could be used to compute a real number. Not long after the Turing machine was introduced, the Recursion-Theory was discovered. Recursion-theory helped solve a multiple of independent problems. Continued research led to the discovery of Church's lambda calculus and Posts production systems. Finite Automata theory was discovered by Kleene, Mealy, Moore and Robin in the 1950s. Context Free Gramma, Push-Down Automata theories were introduced in the 1960s by Chomsky, Oettinger and Church (Cohen, 1991).

Programming up to this point used machine code, with no indexing and limitation of memory (Rhodes). This method was complex and prone to much error. The need for

interpreters and compiler language motivated John Backus at the IBM to introduce in 1954 a high-level programming language called FORTRAN (Wilkes, 1993). Since its introduction, it has become the principal language used in the scientific community. Its numerical capabilities have marked the foot steps for many other languages to follow, especially with regard to its techniques and extensive numerical libraries which will continue to characterize the predominant infrastructure for science generations to come.

Concepts of FORTRAN include variables, expressions, statements, static arrays, condition control structures, modules (non-recursive) and directed input/output. FORTRAN continues its developments and expansion to adopted trends in technology change, mainly focused on reuse. For example, FORTRAN90, FORTRAN95 in today's market includes Object-Oriented techniques. FORTRAN was a first computer language brought us out of machine code into high-level programming. Even after many years since its introduction FORTRAN is still used extensively in science communities (University of Liverpool, 1996). Today there are numerous high-level programming languages, but in purpose of this thesis, only a few are considered - those believed to be related to the present study. A full explanation of the history of the computing and mathematics can be found in the works of UCSB at http://www.arts.ucsb.edu/HAC/his.comp and the works of John Harrison (1996).

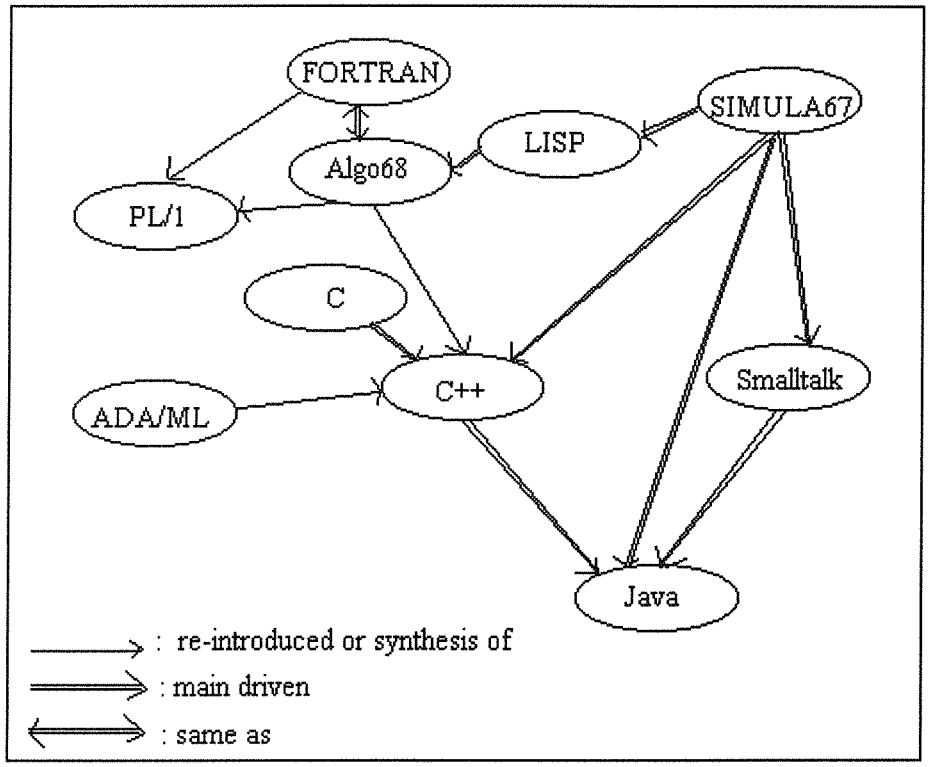


Figure 2.5: High Level Programming Languages Paths

As shown in the diagram above, from FORTRAN, the Algol language was introduced. It was based on the Recursion-Theory which McCarthy developed for the LISP language in early 1960s, together with Fritz Bauer and Joe Wegstein. The motivation to develop Algol derived from the need for a computer language that could be used by the commercial industry. FORTRAN, as noted, was developed for scientific application. The first version was Algol60 and led to the introduction of Algol68. Algol provided a language hidden structure, clear interface and data manipulation (Wilkes, 1993). Algol provided a fundamental framework and conceptual basis for programming language research for many years afterward. Although FORTRAN is geared for practically and Algol for the theoretical, they have similarities.

From the need for symbolic computation rather than numeric computation, LISP was developed by McCarthy at MIT in 1959. It became well-known through McCarthy's Recursion-Theory (The Structure of Higher Level Languages, 1997). LISP is also well known as the premier language for Artificial Intelligence language. It is primarily syntax based on lambda calculus, recursion and conditional expressions control. Researchers continue to develop LISP and expand the language to meet the most recent standards, Common LISP adopted some of the methods found in SIMULA67, such as heaps and classes structure (Matuszek, 1996).

A major issue in development needed to address programming performance. This led to the development of SIMULA67 by Nygaard and Dahl in the late 1960s. It provided inheritance concepts known for classes and prefixing. Class features included: set of procedures, data declaration,; sequence of statements, and class data type which allows the assignment of instances of class and allows data structure in the class (The Structure of Higher Level Languages, 1997). Its concept and methodology had a major impact which led to Object-Oriented Programming as seem today.

C language is commonly used by professional programmers in a UNIX environment. It was designed mainly for UNIX systems programming by Dennis Richie at Bell Laboratories in 1972. It reflects all the main features of the architecture UNIX systems, with emphasis on facilities "low-level programming" which has an impact on program performance (The Structure of Higher Level Languages, 1997). In 1970s the entire Unix operator was re-written in C language. Unix became a portable flat form able to cross from one machine to another. The term low-level-programming refers to the assembly code or close to the machine code.

It may be noted that C is very popular in computing at large. The language has dynamic memory allocation and pointers and includes some user defined structures. Modularization makes for easy maintenance as well as code development by large groups. However, it was originally designed for general purpose programming and not specifically for numerical work. It has other disadvantages. For example, it is not object oriented. It has a terse syntax (e.g., n*=1) and is not completely standardized. In addition, it contains no concise syntax for manipulating user defined data structures.

ML (MetaLanguage) came next in development (Riecke, 1996). MetaLanguage described the mechanism for declaring, raising and handling exceptions. This feature allow easy recovery from the errors. According to Bednarczyk (1996), "ML is a functional programming language with a strongly typed polymorphic type system." Unlike OO languages, ML does not allowed inheritance for polymorphism, but provides prototypical such as parametric polymorphism.

Smalltalk was developed at Xerox by Alan Kay in the 1980s (The Structure of Higher Level Languages, 1997). Bednarczyk (1996) defined Smalltalk as belonging in the group of dynamically-typed languages. It was known as a first implementation of an object-oriented language with data abstraction, inheritance and dynamic data type binding. It was designed mainly from Simula67's class concept. With dynamically-typed language, it "does not check types during assignment (and hence for parameters) and therefore provides parametric polymorphism without static constraints" (The Structure of Higher Level Languages, 1997).

Miranda was introduced by Davis A. Turner at the University of Kent also in 1980. It is a program that consists of functions and data structures represented in

recursive equations. The Miranda is a strong typed program; "list" is a high point. For example a list of operation can be presented as:

> Operation = ["++" "—", "#", ":", "!", ".."].
> Which "++" is an addition list,
> "—" is a subtraction list,
> "#" length, ":" list consing,
> "!" list subscripting and
> ".." list notation (to). FORTRAN (www.csc.liv.ac.uk/~u4sdg/his-fortran.html).

Ada was designed in 1983 in accordance with the requirements of the Department of defense (DoD). DoD called for a language with considerable expressive power covering a wide application domain, independent of any particular hardware or operating system, and able to support good software engineering and safety-critical systems. A later version is now called Ada95. This language completely supports object-oriented programming with a modern algorithmic language (Gargaro & Peterson, 1996). It has the usual control structures and the ability to define types and subprograms, modularity, data, and types. Subprograms can be packaged. That is, Ada has fully support inheritance, polymorphism and provides complexities through hierarchies packages. In addition, Ada distinguishes between public and private features of type and structure libraries access. (Church, 1991).

C++ was originally created by Strouptrup Bjarne at the Bell Laboratories in 1986. Its precursor was called "C with classes" and has been in use since 1980. Strouptrup and his team wanted to write an event-driven simulations which could be used with Simula67. Their goal was then to write a program in a shortest time possible in such a way that the program contained less code. From its original version, C++ was developed and expanded to include more features. It has been updated several times in recent years

(Stroustrup, 1996). C++ is extremely popular. It is basically similar to C, but fully object oriented. It allows overloading of operators and the code is highly reuseable. However, it also has disadvantages. It is large and difficult to learn. In addition, similar to C, it is not standardized.

Java is the newest language available today. It was just developed at the Sun Company (Sun Microsystems, 1997). With the trends of technology changing to a client/server environment, the research team at Sun Microsystems wanted to develop a portable language that only required a once time design and could run on any machines or operators. This solved the cross-platform problems that existed within the World Wide Web (WWW). It also promised the momentum for development for many years to come. It is believed that Java has the potential to eventually mark another generation in computing models (Sun Microsystems, 1997).

One report has described in great detail the advantages and disadvantages of Fortran77 versus Fortran90 and also C versus C++ (Blue Team Software Design Manual for Fortran/C/C++,). According to Sun Microsystems (1997), many languages such as Smalltalk, LISP, and Miranda have proven themselves quite powerful and are heading to the same direction as C++ - to the object-oriented paradigm. Perhaps, C++, however, has the greatest following of C programmers. This researcher believes that has been a major contribution for quick adoption. It is for this reason that more tools have been developed to support object-oriented programming.

Data-abstraction, inheritance, virtual functions and dynamic-hiding are key features for the object-oriented paradigm. C++ programming has supported these features. C++ provides data abstraction, modularity interface and implementation. For

example, the C++ data type int, together with the operators +, = , * and / allows programmers to use the feature without the need to understand how float types were implemented. C++ also supports derived class (or inheritance). It extends the notion of data-abstraction to express hierarchical relationships, that is, to express commonality among classes with the most general on the top, containing a base class to lower classes in a tree structure. Consider the example of employee in which "employee" is a base class for manager. Both employee and manager have name, age, department and salary. A manager is also a employee and is derived from employee class.

C++ also supports virtual function which helps to overcome the problems with "type." This language allows programmers to declare functions in a base-class that can also be redefined in each of derived class. Another advantage of C++ is that it supports dynamic binding. C++ allows operations to be invoked on an object without showing the actual type of the object. It only shows this at the run time. This has freed programmers from the detail of overhead.

Research on Reuse

Evolution to Measurements.

Negative economic trends in previous years eventually led to company downsizing and restructuring. Organizations have and continue to be greatly influenced by trends in techniques to control the processes of software production and by the need for quality to stay in business. Companies have learned the hard way that they must participant in reuse application to remain viable in today's economic marketplace. Authorities generally agrees that reuse application offers the potential to simplify

software development which results in costs saving, high quality and increase productivity (Jones, 1994, Comaford, 1995, Urban &Chang, 1995).

Numerous researches have contributed and continue to devote their efforts to finding new techniques in software reuse. There may be many different techniques, but all agree to the fundamental goal and definition - to recognize that application designers do not have a need to write new code, but must engage in the use of inhering and capturing commonalty tasks(Deng-Jyi & Lee, 1993). Inheriting and sharing knowledge in system design, code and others project documents are related. Design patterns reduce complexity by providing conceptual guidelines to help programmers use the proper tools for a given context. Opportunities for reuse software arise under many different circumstances of the life-cycle software development. Opportunities can be realized during measurements and specifications, design, and applications development, for example.

Measurements.

Revolution in software engineering drives software complexity. As previously indicated, there is an increasing need to control the processes of software production and quality of the product. The software metrics method has been used as a viable approach to measure a system component or process to a given attribute (IEEE, 1991). It is not only used to evaluate, determine and support reuse component, but also to facilitate in creating components for reuse (Martin, 1990). Software metrics typically involved: lines-of-code, function-points, program size. These are further described in subsections below.

Another approach is structure measures that integrate a process for identifying candidate objects in program code. This method assumes that the complexity of a

program correlates to the size of the program; then components tokens and objects tokens can be used to formalize the complexity measurement. In the view of Esteva (1995), such "structural measures are related to the data flow for procedural languages. They are concerned with assessing both the internal complexity of a component and the complexity of its relationship to the rest of the program" (p. 81). However, quality in software essentially depends on many different variables. Also, dimensional is a barrier to the software reuse community (Salamon & Wallace, no date).

Lines-of-Code

In the past, the most common measures has been based on the number lines of code. This measure does not necessary predict program complexity, however. It is a well-known fact that in the earlier days of programming, the ad-hoc" method was quite common. This resulted in similar programs carrying out similar results, but not containing the same number of lines of code. An example of this problem can be found in "Hello World" in C. One routine has more lines-of-code than another; however both carry out similar results. Example A as shown below clearly indicates that the developer had more technique productivity than the one who designed Example B.

Example A:

```
str_temp := "Hello World"
str_fr := " from "
str_id := "Joe Number II"
prinf("%s ", str_temp);
printf(" %s ", str_fr);
printf("%s", str_id)
```

Example B:

```
prinf("Hello World from Joe Number II")
```

Metrics can assist this problem by measuring data complexity in routines, logic in routines, size metrics based on lines-of-code, comments, and executable statements. Another view of using line-of-codes method with structure style may be considered. It is called the building block approach to software development. It is believed to be most important and should be considered in software reuse (Watson & McCabe, 1996).

Function Points (Complexity)

Researchers soon realized that previous methods for software engineering measurement were not accurate for estimating project costs and resources. Function-points was an alternative software metric approach developed to assist with this problem. Function Points are "derived using empirical relationships based on countable measures of software information domain and assessment of the software complexity" (Bryant, no date). Information domain includes: number of inputs, number of outputs, inquiries (combinations of inputs and outputs), number of files and number of external interfaces. (Salamon & Wallace, no date).

With function points, there may be a total count calculate to product the final function value. One report provided an example that was used to prove the productivity and costs saving of FORTRAN and MS Access when both have the same 50 Function-Points. It was proved that MS Access coding activities required five months as compared to FORTRAN, which required 24 months, Cost per-function-point FORTRAN was $2,700 UK compared to MS Access of $800.00 UK. A technical complexity adjustment (TCA) can be found from in the work of Salamon and Wallace at the United States Department of Commerce.

Model (Program Size)

It has been indicated that information system size is an important factor that contributes to system complexity. Program size has a significant impact on development efforts (Boehm, 1981). It is important to explain that relative complexity refers to a number that represents the essential characteristics of any set of metrics that may be selected to use in the software development process. Ordinary software complexity metrics simply cannot be added together to summarize the complexity of each program module. Raw metrics must first be combined into smaller uncorrelated metrics sets. These are called domain metrics (Precision Software Measurement Products, 1996).

There are certain reasonable constraints that must be observed in the computation of relative complexity. One of the most important is that no metric may be derived from any other in the set of metrics. The classic example is provided by Halstead who measured of program size in accordance with the following formula: $N = N1 + N2$ (Precision Software Measurement Products, 1996). There is no new information in the metric N. It is merely the sum of the metric N1 and N2, the total number of operands and operators.

One report explains that many software measurement tools produce a large number of software complexity metrics, but a large number are so highly related that they basically measure the same thing.

> For example, if our metric tool were to report on the total statement count and the total lines of code we would find that these metrics are strongly related to one another. If you have a program module with a large number of statements, then it will also have a large number of lines of source code as well. These two metrics are both measures of a size domain (Precision Software Measurement Products, 1996. p. 1).

The report further noted that it is important to first determine the actual number of measurement domains represented. If one measures Statement Count, Lines of Code, Halstead program size, and McCable's cyclomatic complexity, there would be but one actual measurement domain because all are simply measures of program module size. In addition, metrics used to compute relative complexity should relate to the criterion measure. "For example, if you wish to use relative complexity as a surrogate for software faults then all of the metrics that you use to compute relative complexity should relate to software faults. Relative complexity will only be as good as the metrics that make it up" (Precision Software Measurement Products, 1996, p. 2).

In summary, it requires everyone look at the subject from the same standpoint. Frenton (1991) provided an example of the problem in which measurement of human height was being considered. Should shoes be allowed? Should the measure take place from the top of the head or from the top of the hair? This variables must also be taken into consideration with respect to software reuse.

Another example can be found in the works of Esteva (1995). This author considered that the size of a given program correlated to the complexity of the program - that is, how tightly or loosely was the relationship from one component to other. This was used to determine the complexity of the program. Snooper is the name of the program (Esteva, 1995). In this work, the author has extended and formulated the results to support the representation of commonality and variability in a domain. Esteva denoted:

FOV: (F)unction (O)ccurrence (V)ector
AFO: (A)verage Function Occurrence
NO: (N)umber of Objects

OOV: (O)bject (O)ccurrence (V)ector
AOO: (A)verage (O)bject (O)ccurrence

Reduction of Complexity

Current literature has noted that programs can be developed while breaking all structure rules or not considering the efficiency factor. Such an example was "hello world," as previously described. Modules have impacted structure programming and served to reduce complexity. Davis (University of Liverpool, 1996) presented an example of an unstructured program of 100 lines of code, which can have up to 10^158 paths. Using Modules program (function) to have structured program, approximately 100 lines of code could be placed into 4 separate functions. This lowed the number of the paths to 10^33 which highly reflects cohesive modules, and thus represents an improvement.

Clearly it is possible to reduce complexity by carefully analyzing components into sub-components and applying the black-box approach. In this context, estimated size of code for a module in development phase should also be taken seriously with respect to complexity. Kumar (1996) used the "big-O notation" to generate a structure chart which presented the worst-case, average-case and the bast case complexity during modules development. With structure charts, software engineers will be able to determine the best approach to complexity in program code, modules which relate to the whole application. However, it is obvious that complexity must have a measurement to determined the complexity reduced.

The fundamental for complexity measurement is continually impacted by economic concerns as well as quality of the product. Time and space elements are also a fundamental concern of complexity measurement. Matuszek (1996) described the "time"

and "space" elements that drive complexity of the system. Time complexity is a measure of the time (or machine cycle for a digital computer) to the time execute. Space complexity is a measure of the space needed for computation. Line-of-codes, program size and function points together with structure layer are the absolutely elements to the model complexity. Detail in coupling and cohesion can be found from the lecture notes provided by Gerard Lyons at http://it-hal.ucg.ie/CAI_Tutor/func_dec/f_copcoh.htm.

Complexity Analysis

Software engineering is influenced by reuse in general and the building of reuse, specifically. Reuse at the same time is called backward-reuse; build-to-reuse is called forward-reuse (Urban & Chang, 1995). According to authors, backward-reuse can be defined as an existing set of components with their activities involved in the retrieval components mechanism. On other hand, forward-reuse involves the product that will be developed with reuse in mind. Reuse engineering encompasses both domain engineering and application engineering and is considered to be essential to institutionalization of software reuse. The term "reuse" is a conceptualization of components already build. Likewise, build-to-reuse is a conceptualization for new components such as those which can be started from the scratch. Both approaches facilitate software reuse to its maximization point. The present thesis only focuses on the domain engineering concepts and its methodology, reviewing application concepts of the greatest importance.

Domain engineering embraces the scope of the body knowledge mechanism. The need for formal domain engineering methods is apparent in large-scale application. Formal domain engineering method underscores similarities and differences among

components Activities represented in a domain model included: analysis, definition, identification and integration (Krut & Zalman, 1996; Withey, 1994). Domain engineering is defined as "The process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain" (Kang, 1990).

Team members employed at The Pacific Software Research Center (Bell, 1994) used a method which was based on domain engineering to automatic program generation from reusable components suitable to a specific program. It is important to explain that application engineering is a specific instance of a domain. By contrast, reuse-based on application engineering "studies the commonalties and difference among software-intensive systems within a functional area" (Domain Engineering: A Model-Based Approach, no date). Activities include: user requirements analysis, prototype plan and development, demo, implementation.

Key model concepts are abstraction and refinement. Many guidelines exist to perform some or part of the application engineering (U.S. Department of Defense, 1997). Feature Oriented Domain Analysis (FODA) uses the abstraction feature to create domain products from the specific applications in the domain and apply refinement methods to both refine the generic domain products and the domain products into applications.

Software engineering now exists to support software reuse. It is important to realize however, that the challenge in software reuse from reusable components focus on how to find reusable components (Frakes & Gandel, 1990). To find the reuse components, common techniques involved in software retrieval mechanism may included

systematic approaches, such as a set of keywords and index components. This approach is described in the work of Prieto-Diaz (1991), Price and Girardi (1990), and MSC (1991). Other approaches used semantic concepts like natural language in users interfaces to apply to retrieval system (Girardi & Ibrahim, 1995). Common approaches include Hierarchical Classification, Faceted Classification and Natural language model. These are described below. However, regardless of the path taken, software design must also be considered during system implementation. The absolute of software design methodologies is to product simple design that corresponds to the problems domain. The functional provides good design which involves structure aimed at the data structure that acts on them (Allworth, 1981). Methodologies structural architecture include top-down, bottom-up and Object-Oriented. In Blue team (www) explained in detail of top-down design, bottom-up design and Object-Oriented design.

Top-Down Design refers to the practice of dividing a complex software system into smaller and smaller parts, each of which are then refined independently. However, it is desirable to design each small piece to be a reusable software component that can be composed with other such components to form new applications. Thus, a side effect of a good top-down design of a system, is that the design of subsequent systems may indeed prove to be, at least in part, from the bottom-up. In practice, these two approaches are meshed.

It is also important to explain the difference between top-down and Object-Oriented design. In OO, a system is decomposed according to key abstractions in the problem domain. The problem domain is viewed as a set of autonomous entities which collaborate to perform a higher level behavior. Each "entity" in the software system is a

model of a tangible entity in the real world problem domain which displays some well-defined behavior. A most important feature of the building blocks, called objects, is that the representation of the state (data) of the object is inextricably linked with the functions that manipulate that state. Object oriented languages like C++ naturally enforce this binding. The architecture of an object-oriented system specifies the relationships between objects, of which there can be a number. These are often represented on an object diagram. Representation of object diagrams has recently been standardized in the Unified Modeling Language (UML) specification.

Hierarchical Classification

Hierarchy is best described by Li and Loew (1987) and in the Full Computing Review Classification Scheme (ACM, 1990). At the top levels one might have the application domain and refinements thereof, e.g. "computer graphics" or "numerical analysis"; lower levels might represent functionality such that "solve equations" or "evaluate integrals", "programming language", etc. However, there are a number of problems associated with this view. These can be listed as follows:

- a hierarchy is sometimes too irregular, that is, at a given depth in one branch one might discriminate a fine point of the application domain, whereas at the same depth in another branch one might discriminate the functionality;

- it is too tall, that is, the user must answer too many questions;

- some components might reasonably be classified several ways. For example a "lisp compiler" might be classified under "programming languages" and under "artificial intelligence"; and

- the hierarchy chosen by the classifier might not be the most convenient to the person attempting to retrieve a component. It might be found to be more convenient to put the application domain at the top of the hierarchy,

while another find it more convenient to put the programming language or hardware at the top of the hierarchy.

Faceted Classification

The Faceted approach has been best described and advocated by Prieto-Diaz (1987). Facets can be considered to be dimensions in a Cartesian space. The collection of values of the facets can be considered to constitute the coordinates of point in that space. Alternately, facets can be consider to be independent views of the properties of software components. In Prieto-Diaz (1987) the properties asserted to be necessary and sufficient include operand, application-domain, and functionality.

Many of the objections to a hierarchical classification are answered by a faceted one. Solderitsch (1995) described the number of limitations of faceted model. These include the following:

- reliance on a query specification and refinement approach to discover the contents of the underlying software catalog;

- lack of change ability of the underlying classification scheme as the domain evolves;

- no explicit support for supporting different user communities (e.g. managers and programmers) and different user abilities; and

- lack of a graphical view of the underlying domain model.

Natural Language Model

The Natural Language model provides a natural-language interface that allows communication to system in ordinary English sentences. Girardi and Ibrahim (1995) provide a technique for retrieval of reusable components through processing both in queries and in natural-language descriptions. This technique is believed to improve

retrieval and make it more effective. However, more researches effort needs to be directed to this area of study before any definitive conclusions can be reached.

Obviously, components libraries are only as good as the information contained therein. They must have a large number of components to reuse. The potential for reusability increases with an increase in components. For this reason, the size of components libraries is continuing to expand. There is no longer a single retrieval that is able to provide specific components requested. This is not an easy task for human interactions participants (Esteva, 1995). For the most optimal level of effectiveness of the components retrieval system, users must be knowledgeable and able to interact in the appropriate manner to provide specific components requested. Such interaction is achieved through menu dialog for users selection. Gordon (1992) provide numerous examples in this respect. Also, graphical environments can be found in the works of Citrin and McWhirter from the University of Colorado (1995).

Communicating Sequential Processes (CSP)

Current technologies undoubtedly promise a great increase in system processing. Object-Oriented allows inherence from object to other objects, which greatly contributes to the increase in processing power. However, Object Orientation inherently has also created new problems in design and implementation (Hinchey, 1995). Communicating Sequential Processes (CSP) offers a solution to solve this problem. CSP commonly used to increase the processes power system in concurrent methodology offers what has been called "process algebra". Process-algebra is defined by Glabbeek as "An algebraic approach to the study of concurrent processes. Its tools are algebraic languages for the

specification of processes and the formulation of statements about them, together with calculation for the verification of these statements." (Van Glabbeck, 1987).

CSP's concepts and methodologies can best be viewed at the following Internet address: http://heart.engr.csulb.edu/~foster/ch2-6b.asc. The Web page is produced by Foster in Language Mechanisms for Synchronization. CSP approaches included Sequential I/O, Repetitive sequential and Concurrent I/O. Foster describes CSP commands as follows.

Input command: <source process id> ? <target variable>
Output command: <destination process id> ! <expression>
Repetitive command: *[G1 -> CL1 G2 -> CL2.. Gn -> CLn]
Concurrency: [process P1's code || process P2's code || process Pn's code]
Example *[c: character; west?c -> east!c]
Said:
input from process west
output to process east
terminates when west terminates

There are various works available such as Models for Distributed-Memory Programming. Trace driven simulation has been successfully studied in research on memory design and caches performance (Smith, 1992). Decision Support Systems have been reported by D. R. Dolk and J. E. Kottemann at the following Internet address: http://www.iscs.nus.edu.sg/~yeogk/MM/biblio/journal/j000038.htm.

Object-Oriented Technology in Development and Reuse

Frame and Frameworks in Reuse

Frame concepts were introduced by Minsky(1975). Frames provide the defined structure to reduce complexity nets. Elements of this technology include: goals, key problems, problem strategies, requirements strategies, current theories, tacit knowledge,

testing procedures, implement methods design, users' interactive features, perceived substitution function and exemplary artifacts. (Minsky, 1975, 1988). In relational database systems, frame is also viewed as a records structure with database attributes. Each segment of a database is also called a frame. The structure is always hierarchical, cross-referencing link though defined relationships (RDBM concepts).

In Gentleman's (1996) R-language manual, it is suggested that the first step to follow is to frame a data frame whose components are either logical vectors, factors or numeric vectors. Data frame is a class of objects facility for the data storage which are usually used in fitting models. They are similar to matrices structure in the way that variables can be presented as a matrices columns and the observations as rows. For example, the attributes of "address" frame may have attributes such as "street", "city", "state" and "zip code" represented in columns. In R-language, Gentleman (1996) presented the data-frame in the "frame" with included attributes "syntax", "arguments", "value", "see also". For example of these attributes, consider a data-frames which was obtained from R-language (Gentleman, 1996):

[Syntax

data.frame(..., row.names=NULL, col.names=NULL,
as.is=FALSE)
as.data.frame(x)
is.data.frame(x)
row.names(data.frame.obj)
print(data.frame.obj)

Arguments

...
these arguments are of either the form value or tag=value.
Component names are created based on the tag (if present) or the deparsed
argument itself.

row.name
a character vector giving the row names for the data frame.
col.names
a character vector giving names for the variables in the data frame.
as.is
a logical value indicating whether character variables should be left ``as is'' or converted to factors.
Value
A data frame. Data frames are the fundamental data structure used by most of R's modeling software. They are tightly coupled collections of variables which share many of the properties of matrices. The main difference being that the columns of a data frame may be of differing types (numeric, factor and character).
as.data.frame
attempts to coerce its argument to be a data frame.
is.data.frame
returns TRUE if its argument is a data frame and FALSE otherwise.
See Also
read.table.]

Frameworks are reusable designs for an application scope focusing on reducing unnecessary and redundant system development through the reuse set of abstract classes or a part of class. According to Coplien & Schmidt (1995), framework provides an integrated set of domain specific functionality; frameworks exhibit an inversion of control at run-time. Essentially, a framework is a semi-complete application. Frameworks contain such elements as: Building Blocks, Abstractions and Processes reuse relative to the software reuse. Thus it becomes clear that a framework is a reusable design for an application or a part of an application that is represented by a set of abstract classes and the way these classes collaborate, as defined by (Johnson (1988). Frameworks differ from class libraries as described in Coplien and Schmidt (1995):

1. As pertains to an integrated set of domain-specific functionality, in framework, particular domains are addressed such as business data processing, GUI, databases. Class library contains such things as strings, complex numbers, dynamic arrays and bit-sets.

2. Exhibit an "inversion of control" at run time. It is a framework's responsibility to determine which methods to invoke in response to events. Events such as messages arriving, keyboard and mouse from users interaction.

3. Is a "semi-complete" application in which allows programmers complete applications by inheriting and instantiating framework components.

In Object Oriented Programming like C++, C++ is the higher-level object implemented by objects at lower levels of abstraction. An example can be found in Stroustrup (1996) for abstract frameworks such as "shape," of which "circle" and "square" can actually be used. The Lockheed Martin Tactical Defense Systems Reuse Library Framework (RLF) has successfully developed domain-specific reuse libraries with knowledge-based which are written in Ada,. The Internet address site, http://iktt.zgdv.de/ VASIE/Reports/All/10496/Objectives.html, has also developed an Object-oriented framework for vessel control systems as a part of the pilot project which determined its succeed when compared with other methodologies. These included Booch and Object Modelling Techniques (OMT).

Pattern Languages Design in Reuse

Software reuse mechanisms allow programmers to create a new pattern on top of other patterns specified. In Budinsky (1996), pattern "describes a solution to a recurring design problem in a systematic and general way." Design patterns like Object-Oriented software have promised potential techniques for software reuse - potential for software

reuse communities, specifically. It provides the solution together with guidance on how to implement. Pattern is not a code, rather a template which provides developers with guidelines for solving problems. The structure form is represented (Budinsky, 1996) in the template of Name, or the name of the pattern. It is also represented in the following:

- Intent: context of pattern

- Also Know As: other name relevant to the pattern
Force: pattern motivation

- Application: the kinds of question what/where/how

- Structure: graphical representation of the classes in the pattern

- Participants: classes or objects if any participating in the design pattern and its responsibilities

- Collaborations: description of how the participants carry out their responsibilities

- Consequences: the trade off and results

- Implementation: pitfalls and hints considering

- Sample Code: code fragments

- Related Patterns: others pattern related and patterns considering to use.

As based on a survey by Stephen Siu, there are two reuse mechanisms in reuse. Design patterns for extension are: Composition and Refinement and can be understand as a black-box and white-box, respectively. The following set of component and refinement statements are used in Siu study.

1. Composition which allows developers to create new design patterns by interconnecting an arbitrary graph of design patterns. The new pattern can, in turn, be used recursively inside another pattern. Composition is a black

box reuse because developers do not need to know the implementation of the design patterns to connect them together.

2. Refinement which allows developers to create new design patterns by specializing the structure inside existing design patterns. The behavior of the existing pattern remain unaffected in refinement. Using the mechanism, developers can substitute a node in the graph of an existing design pattern with another graph. It is a white box reuse because developers have to know the internal structure of the existing pattern to specialize it.

One research report offers Date functions written in C language and presented in refinement forms (Object Oriented Decomposition Generalization, 1995). Others use design pattern techniques. Budinsky, Finnie, Vlissides and Yu successfully developed the automatic code generation from design patterns (Budinsky, 1996). In the works of Roberts and Johnson at the University of Illinois, (st-www.cs.uiuc.edu/users/droberts/evolve.html), they describe A Pattern Language for Developing Object-Oriented Frameworks. The authors placed examples in the Pattern template itself.

Data Abstraction

Data abstraction mechanisms are well known as important tools in software reuse with significant impacting abstract data type. (Peter, 1987, 1995). According to the author, activities of the abstracts data type (also known as user-defined types) are to provide an abstraction of one implementation per program or to describe as a single

implementation, or to allow multiple implementation per program. Two approaches in the multiple implementation abstract data types are, according to the author, data encapsulation and procedural encapsulation. Data encapsulation relies on a type system with existential types while procedural encapsulation can be applied in a polymorphic Milner/Mycroft type system with algebraic types. Such data abstraction in C++ has significant practical and easy-to-use data capability. In other words, "A type created through a module mechanism is in the most important aspects different from built-in type and enjoys support inferior to the support provided for built-in types." (Stroustrup, 1996. p18). Simple examples are Integers, Complexes, Sets, and Lists, among others. Stroustrup (1996), well known as a C++ creator, gave an example of the Arithmetic types such as rational and complex numbers.

```
class complex {
        double re, im;
public:
        complex (double r, double I) { re = r; im = i}
        complex (double r)              // float ->complex conversion
                ( re = r; im = 0; }
        friend complex operator + (complex, complex);
        friend complex operator - (complex, complex) ;      // binary
        friend complex operator - (complex);                // unary
        friend complex operator * (complex, complex);
        friend complex operator / (compelx, complex);
        //...
};
```

Classically, in this example above, the user defined type "complex" specifies the whole set of operation on a complex number, which is easy-to-use in this "complex" class. To call this function, for example:

```
void f()

{
        complex a = 2.3;
        complex b = 1/a;
        complex c = a+b*complex(1,2.3)/a
        //...
}
```

It can also be noted that the user who uses this class "complex" should not have to know the internal associated with class "complex". This method is also called a "black-box" because it does not require the user to understand or known the class "complex" implementation.

It has been generally agreed that the reasons for user-defined-types include the following:

1. Programmers can work directly with so called real-world objects of that type. Rather than from the traditional lower-level types language. This certainly yields more natural solutions.

2. Better design and document modules mechanism.

3. Provides hidden components and encapsulation variables which leads fewer global variables.

4. Reuse easy and simplifies program verification.

Various Object-Oriented Language offer this classic data abstraction such as Ada, Clu, and GLISP, among others. GLISP provides data abstraction facility with hierarchical inheritance of properties and object centered programming (Novak, 1983).

Complexity Reduction

Software is often complex, especially when the application is for a large module level. Abstraction reduces the apparent complexity of an implementation in a way that presents only the most relevant component and hides all others. However, because each human mind thinks differently, views differ from one to other. No one user-interface can be suitable to all. This is also clearly reflected in programming. According to Strouptrip (1996), the problems with abstract data type is that there is no way of adapting it to new uses within a program without modify its definition. This can lead to several problems such as inflexibility, prone error. For example, for the purpose of use in a graphics system, consider the type "shape." Define "shape" like:

```
enum kind { circle, triangle, square };
class shape {
        point center;
        color col;
        kind k;
public:
        point where();          {return center;}
        void move (point to)    {center = to; draw();}
        void draw();
        void rotate(int);
};
```

and the function draw can be defined:

```
void shape::draw()
{
        switch (k) {
                // draw a circle
                break;
                //draw a triangle
                break;
                //draw a square
                break;
        }
}
```

This is known as a trouble-maker because the function draw() requires the user to know all the its elements of "shape." If a shape like a triangle is needed, this function then needs to be modified, thereby creating great potential for error. Unfortunately, not anyone can go in and modify this function; it required access authorization. This represents yet another problem.

However it is possible to reduce complexity. A common way is to divide into sub-systems. Sub-systems in turn can be divided into their sub-systems until further division can not be performed. This is often called the hierarchical decomposition of system (Verstraete, 1997). It is important to note that there are several decomposition classes (Object Oriented Decomposition Generalization, 1995). These are written in C language such as class decomposition of constructor c_time(), constructor function c_time(int, int), constructure function c_time(c_time&), destructor function c_time::add(int), ~c_time().

In summary, it can be seen that there are many motivational factors to use Object-Oriented concepts in software engineering programs. The following is a list of some of the many reasons why programmers use OO concepts.

- Improvement of trace-ability.
- Reduction of integration problems.
- Improvement of the impact of process and product.
- Need to keep to a minimum objectification and de-objectification.
- Hiding of information.
- Abstraction of data.
- Encapsulation.
- Concurrency.

Background of the Theoretical Model

Looking back from the counting number in Greek period to structure languages and more recently, Artificial Intelligence, researchers have stressed the role of general computing mechanisms. Many of the goals of computing languages such as Fortran, C, Lisp, Ada are the same as those of languages in general: to provide a mechanism notation useful for humans and machine to understand and hopefully a method of expressing notation in words for greater understanding. One view is that the collection of various notational mechanism can be "strict enough in its syntax, and on the other hand, rich enough in its semantics..." (Barabashev, 1995, p. 1). Semantics of a languages tells the user what a sentence means. Syntax of a languages tells the user how a sentence in a language is put together in a sentence or formula.

Relation

The term relation here refers to its common use in the computer science and mathematics literatures. Given a set (x1, x2, x3...xn), R is a relation on these n-tuples if for each element of x is in R. R is say to have degree n, degree 1 often called "unary", 2 called binary, 3 called "ternary" and degree n called "n-ary". For example, the n-ary relation has the following properties (Codd, 1970):

- Each row-X represented an n-tuple of R
- The ordering of row is immaterial
- All rows are unique
- All column are partially conveyed by labeling its name corresponding domain

- The ordering of column is corresponding to the ordering of x1, x2, ...xn which R defined.

Consider the relation of degree called "sale" which reflects the products-solve of "product_id" to specified "cust-id" in specified "quantity" (a relation of 3 degree)

sales (cust-id, product-id, quantity)

1, 2, 5
2, 3, 1
2, 4, 6

The Artificial Intelligence Applications Institute(1996) defins data relation as:

> A relation is a set of tuples that represents a relationship among objects in the universe of discourse. Each tuple is a finite, ordered sequence (i.e., list) of objects. A relation is also an object itself, namely, the set of tuples. Tuples are also entities in the universe of discourse, and can be represented as individual objects, but they are not equal to their symbol-level representation as lists.

In context, there is a definition of "set" theory which needs to be explored. Sets are the most commonly used in mathematics and known as a building-block. The elements in set include a "sub-set", "union", "intersection". The most used "sets" and "empty-sets" in mathematics are "natural numbers," "integer," "rational," and "real numbers." For example, if N, I, R, 0 are natural-numbers, then Integer, real and empty-set is denoted in the order in which they represented. Then

N = {0, 1, 2, 3...n} = natural numbers
I = {..., -2, -1, 0, 1, 2,...} = integer
R = {1/3, n/ q} = real number
0 = {0} = empty set

Likewise, consider the same in the definition of a function. Function is a correspondence between two sets. Each first element corresponds to exactly one and only one second

element. If A and B be two sets, a function f said has relation between A and B such that for each a $\in$ A there is one and only one associate of b where each b $\in$ B. Function often denotes y = f(x) indicating the relation { (x, f(x)) }. Therefore, the set of A is called "domain" and set of B is called the "range."

By convention, consider relations via visa classes. As defined, equivalence relations are one of the most useful kind of relation besides functions. Functions are also relations called equivalence relations. A relation is an equivalence relation if it has three properties: reflexive, symmetric, and transitive. Reflexive as defined in KSL (Stanford, no date) "Relation R is reflexive if R(x,x) for all x in the domain of R., Relation R is symmetric if R(x,y) implies R(y,x), and Relation R is transitive if R(x,y) and R(y,z) implies R(x,z)." With respect to class equivalence to the Objects (OO context as described), classes are objects and objects are classes, as denoted by Bednarczyk (1996). Class also has-a objects. The relation "is-a" and "has-a" represented in the dimension space one look at.

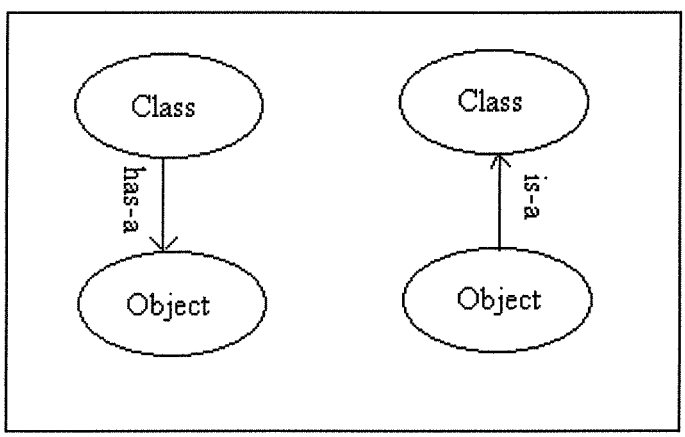


Figure 2.6: Class-Object

Class provides inheritance. Inheritance as defined means that a new class can be derived from existing class (types) - that is, super-class - sub-classes concepts. Sub-classes inherit all attributes and operators of super-class and additional attributes and operators as the additional instance of super-classes. This mechanism id commonly found in Object-Oriented Concepts and that is all Object-Oriented above.

The inheritance ("is-a") relationship is significant and impacts the design and implementation of an application systems. Such concepts provide application to avoid redundant information. For example, a properties real-number in mathematics contains in "number:" Real-number is-a Number. It must also be considered that has type real. This will be further discussed in paragraphs below. Likewise, Fortran77 is-a Fortran.

With respect to types, most of relations preserve the subtype relation (KSL, no date). It also called the "constituency" and allows the propagation of information. The relation "has-a" is also considered as playing a significant roles in the object-oriented paradigm. In the ACL project, the example was given of a "wall" which has "window" and "door." If the "wall" needed to be moved, then all its objects need also to be informed. The preserving-subtype in KSL is such that:

> A headword presented N. Automobile has-part motor can be presented as:
> N. automobile.
> {{has-part}} motor.

Likewise, the relation "isnota" (is-not-a) is used in KSL to prevent inheritance properties when the hierarchy is not correct. Consider the example in the KSL case as "salary" with "benefits" which is also a part of salary. "Salary" usually money, but if there is a clearly stated "benefits are not money," KSL uses the following notation:

{{has-part (salary)}} {{!:not-has-subtype (money)}} benefits.

This example can be seen completely and in greater detail at its Web site. The following are some of the relations defined and used in KSL listing for the study purpose.

In KSL, hierarchy/class include:

F1: is_a, has_subtype, has_subtype_x, has_subtype_d
F2: has_domain, domain_of
F3: isnota means do not continue chain of inheritance from higher categories.
Physical related:
part_of, has_part, partition_of, has_partition...

Frames

Frame is well known as a conventions support in object-centered knowledge representation. To reduce the complexity such as number of link or path in semantic nets, frame allows objects to have knowledge by themselves (Moledor). Each frame represents a set (or class, entity, slot). Consider properties defined for class "Elephant" (Cawsey, 1996).

Elephant

subclass:	Mammal
* color:	grey
* size:	large

The frame "Elephant" has attributes or slots of "color" and "size" where these slots have value "grey" and "large" respectably. Frame sometimes can also be viewed as a record data structures in database system (Cawsey, 1996). In other words, the record named "elephant" has fields: color and size and field color has value "grey" and field "size" has value "large." However, as the author points out, frame with the additional is

supported inheritance. In this case, frame "elephant" has inherited from the parent-frame "Mammal."

Other examples can be cited. Consider the following Standard Generalized Markup Language (SGML) example, presented here for the study perspective:

Property definition:

RCS name: "PROLOG", application name: "", full name: ""

belongs to class: "sgmldoc"
specification document: "SGML", clause: "71001"
datatype: "NODELIST"
allowed value classes: "DOCTPDCL LKTPDCL COMDCL PI SSEP"
allowed class names: ""
node relation: "SUBNODE"
lexical type: ""
string normalization rule: ""
verify type: ""

Where name, belongs to class, data type, allowed class names, node relation, lexical type, string normalization rule and verify type are elements contained in the defined frame in SGML system.

Propositional Logic

Propositional logic is an algebra term for reasoning about the truth of logical expressions. Where a logic is concerned only with sentence connectives, it is called a propositional logic. In natural languages, words whose primary role is truth functions often have other roles as well. This is one of many ways in which natural languages fail to be ideal for some logical or technical purposes. In the natural language such connectives as "and," "or," "not," and "implies" are constraints that predicate logic. When A and B are two sentences, "and" is consist in conjoining two sentence (A and B)

which formed true if and only if sentence A hold true and sentence B also hold true (Jones, 1995). "Or" is a disjunction formed true if either A or B is formed true or both true.

Also in First Order Predicate Logic: (A -> B) -> (B -> A), there exists x A(x) -> (For all) x A(x), therefore (A -> C) AND (B -> C) -> (A -> B). In Russell and Norvig (p. 167), the authors described a very simple logic:

BNF grammar

sentence -> literal | complex sentence
literal -> atomic sentence | NOT atomic sentence
atomic sentence -> TRUE | FALSE | P | Q | . . .
complex sentence -> (sentence) | sentence connective sentence | NOT sentence
connective -> AND | OR | | =>

According to Jones (1995), there are certain constructions in natural languages which have the following features: they are sentential operators; they operate on one or more complete sentences to give a new sentence; they are truth functional operators; the truth of the resulting sentence can be determined knowing only the truth values of the sentences from which it was constructed. The most well known, and probably the simplest of propositional logic is known as classical or boolean, in which it is assumed that all propositions have a definite truth value; a proposition is either true or it is false.

Constraint Satisfaction

A constraint satisfaction problem is concerned only a local consistency conditions instead of corresponding to an optimal path. One definition that has been provided in the literature is that "A constraint satisfaction problem is one in which a series of constraints

is imposed on a set of discrete variables. The task is to find a set of values for all the variables that satisfies all the constraints simultaneously" (Indiana University, no date). For example, in a crossword puzzle game, the search mechanism concerns only the words cross to each other which have the same letter in the location where they cross. In context, constraint satisfaction represents relationships among variables which constraint structure and consist of node and arcs. Node represents a variables or constraint and arc represents the relationship among variables and the constraints.

Many problems can be solved when a constraint satisfaction is applied to problem concepts. MathSoft poses numerous problems together with the solutions which can be stated by constraint satisfaction concepts. Detailed information in this regard can be found at the Internet address, http://www.mathsoft.com/puzzle.html. Graphical user-interface such as 3-D Playing Cards can be found at http://www.unitedvisions.com/3dcards/.

Background of Practical Machine

Mathematical Verification by Group Theory

Group Theory is not only central to the mathematical of use, but also provides useful application in other areas as well. This is true because the natural algebraic structure which defines a group is natural and familiar with the concept of a correspondence and transformations of a physical system. When one begins to place rules to the set, a richer algebraic structure is created. Before the review can focus on mathematics by group theory, however, it is first necessary to provide an outline of basic

definitions and notations. This is intended to refresh the memory regarding algebra notations, algebra sets, relations and functions.

The Algebra of Notations

Set:

$A = \{1, 2, 3\}$ said, 1, 2, and 3 are elements or members of set A. Therefore, $1 \in A$ said, 1 is a element of set A. Likewise $2 \in A$, $3 \in A$

Subset:

$A \subseteq B$ said, A is a subset of, or is contained in a set B if for each $x \in A$ and $x \in B$.

Empty:

Set is empty or null which denoted $\varnothing$

Union:

$A \cup B$ defined, $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

Intersection:

$A \cap B$ defined, $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$

Invert:

$A \sim B$ defined, elements of set A which are not in B

Cancel (also called symmetric difference)

$A \Delta B$ defined, the symmetric difference of two sets A and B such that $A \Delta B = (A \cup B) \sim (A \cap B)$ also implies, $(A \sim B) \cup (B \sim A)$

The Algebra of Sets

The following rules are straight forwardly from the algebra definitions

$A \cup (B \cup C) = (A \cup B) \cup C$, likewise $A \cap (B \cap C) = (A \cap B) \cap C$.
$A \cup B = B \cup A$, likewise $A \cap B = B \cap A$.
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, likewise $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
$A \cup A = A$, likewise $A \cap A = A$
$A \subseteq C$ and $B \subseteq C$ imply $A \cup B \subseteq C$
$A \cup \varnothing = A$, likewise $A \cap \varnothing = \varnothing$

Relations:

An equivalence relation has three properties:

1. aRa for each $a \in A$, called reflexivity.
2. If aRb then also bRa, called symmetry

3. If aRb and bRc, then also aRc, called transitivity.

Functions:

Let f be a function on X to Y. Therefore f: X -> Y and designate by f(x), for x -> X.
A constant function f: X -> Y for some fixed $y \in Y$, $f(x) = y$ for all $x \in X$
A identity function on X, a: X-> X is $a(x) = x$ for all $x \in X$
one-to-one function. The function f: X->Y if f(X) = Y and
$x1 \neq x2$ implies $f(x1) \neq f(x2)$

"G" is a notion function: "G" is commonly used as a notation of a group finite and infinite objects A, B and C which implies the combination and product algebraic. That is, A(BC) = (AB)C likewise, if A, B are elements of set then the product of A and B = AB is also an element of set.

Euler-Venn Diagrams

In dealing with the real-world problems, specially commonly used in the set theory, Euler-Venn Diagrams are frequently helpful to picture relations between the sets. For example, the following Euler Venn diagram, shows $A \cap B$ as:

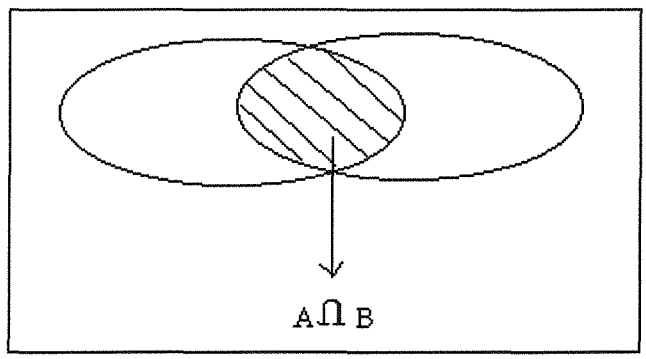


Figure 2.7: Intersection

and the following Euler Venn diagram, shows $A \cup B$ as

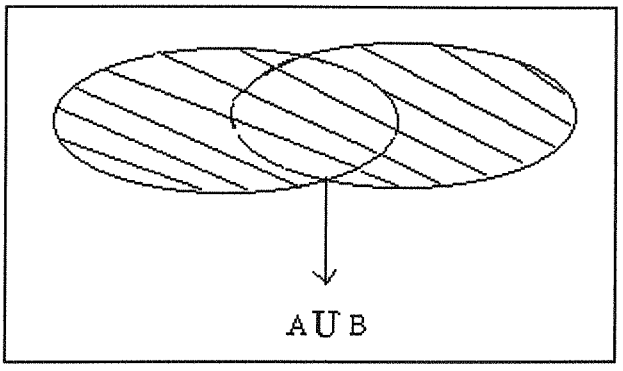


Figure 2.8: Euler Venn Diagram

Cancel (symmetric difference) of set A and B:

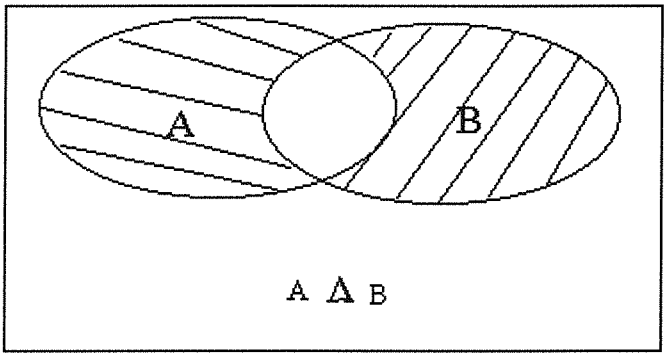


Figure 2.9: Symmetric Difference

Therefore, it holds true that $A \cap B = (A \cup B) \sim (A \Delta B)$.

A quick view of the problems in Group Theory and their solution are also in order. Consider a subgroups problem in Group Theory as presented by Dixon (1967). Problem 1.38 states that, if A, B, and C are subgroups of a group G, and $A \subseteq C$, then $AB \cap C = A (B \cap C)$.

The solution in Dixon (1967, p. 80). Is as follows:

Let $ac \in A(B \cap C)$ where $a \in A$ and $c \in B \cap C$.
Then, $ac \in AB$, and $ac \in aC = C$
Therefore, $A(B \cap C) \subseteq AB \cap C$. On other hand,
if $ab \in A(B \cap C$, where $a \in A$ and $b \in B$,

Then $b \in a^{-1}C = C$ and so

$ab \in A(B \cap C)$.

Thus, $AB \cap C \subseteq A(B \cap C)$

Therefore, implies $A = A(A \cap K) = A(B \cap K) = AK \cap B = BK \cap B = B$ also hold true.

Group Theory in Real World Problems.

Perhaps no concept is more central to group theory than the Automatic structure. Automatic concept is based on the computer scientists' version of finite state automata. According to Cohen (1991), Finite State Automata (FSA) can briefly be described as a collection three things (1) A finite set of states, a start state and a final state, (2) A alphabet $\sum$ of possible input letters which automaton reads a word one letter at a time and (3) a finite set of transitions for its operation or recording states. FSA uses an algorithmic method to determine if a given language and said words have particular properties by examining its elements. The automaton reads one element in this finite state one at a time and it recognizes only that element the automaton in at that time. That element is also used to determine the next state into which the automaton goes. The state may be an accept state, or not. If it is a accept state, the word that led to this word has a property. If, after all elements have been visited and it has not reach to the final state, that word is rejected.

Keeping this word problem in mind, the automatic structure can provide a significant impact to efficiency when dealing with a large lists of group. In addition, it requires only a small amount of memory because it does not require that all elements be retained in memory (Sander, 1994).

New Functions for C++

It is important to first explain that each object (entity) in Object-Oriented design is a specific instance of a more general construct known as a class. A class is a template for a set of objects that have a common structure and functionality. Object oriented languages have representations of relationships between classes which allow new classes to inherit structure and behaviors from previously defined classes. Such a feature promotes a great deal of code module reuse and extendibility.

For this reason, among others, there is increasing motivation on the part of programmers to use Object-Oriented concepts, data abstraction methods, and inheritance methods to solve problems and to provide ease of maintenance. Functions in C++ allow programmers to implement readable modules, reuse predefined and tested functions, and simplify the programming task. According to Stroustrup (1996), new functions for C++ also allow programmers to:

- Identify static data members and member functions, classes features;
- Use and override assignment operators;
- Copy constructors and convert classes; and
- Allow inheritance, multiple inheritance and polymorphism in a program.

Constraint Satisfaction and Frame-Based Expert System

Artificial intelligence, expert, and knowledge-based systems made a first step forward in assisting to reorder information in such a way as to begin to simulate the basic foundations of the way complex problem-solving occurs (Biondo, 1990; Buchanan &

Shortliffe, 1985). Programming languages designed in the past were used for the procedural manipulation of data, but the solution of complex problems by people frequently involve the use of symbolic and very abstract approaches. These are not well suited for a procedural programming language. In the late 1960s this need provided the impetus for concerted effort into the development of artificial intelligence, expert, and knowledge based systems (Giarratano & Riley, 1993; Jackson, 1992). It was hoped at that time that analysts would be able to create "thinking" machines (Frenzel, 1989). This designation remains popular today even though the technology never moved forward to the point of realizing this goal.

Expert computer systems or knowledge-based systems are computer programs that analyze data in a way that, if performed by an individual, would be considered intelligent (Frenzel, 1989). They are characterized by symbolic logic, rather than just numerical calculation and an explicit knowledge base that is understandable to an expert in that area of that particular knowledge. In addition, they have the ability to explain conclusions with concepts that are meaningful to the user.

Expert systems allow inferences to be drawn on encapsulated knowledge. This type of system is characterized by its method of logical deduction from stored data, in accordance with rules independent of the program while conducting the search strategy. There are three basic components of an expert system. These include: a knowledge base, an inference engine, and a user interface (Giarratano & Riley, 1993; O'Keefe & Rune, 1993). These programs embody the modeling of information at higher levels of abstraction and are easier to develop and maintain.

Jackson (1992) has noted that user interfaces are the means of users to communicate with the system. Current expert systems use a pseudonatural dialogue through graphical user-interfaces to communicate. According to the literature, current and near future research is moving in the direction of development of full natural-language interfaces which use a syntax that is close to the user's native language are largely a future development (Frenzel, 1989; Turban, 1995).

Like a database, the knowledge base stores information, or facts. Different than a database, the knowledge base also holds rules for manipulating and interpreting the data (Klinker, Linster & Yost, 1995). Rule-based programming is at the heart of knowledge-based and expert systems (Goble, 1989; Jackson, 1992). It is one of the most commonly used techniques. Rules are used to represent heuristics which specify a set of actions that need to be performed for a given situation. This knowledge is in the form of factual statements, frames, or classes.

As described previously, experts were initially developed in LISP and Prolog. The methodologies commonly used in an expert system are Rule-based and Frame-based methods. Rule-based methods are mainly in the IF-THEN statement with an associated confidence factor. For example, IF N is a set of numbers contains -2, -1, 0, 1, 2 THEN N is called a natural numbers. This IF THEN statement is widely used in higher level languages such as Fortran, C, C++. The conditions and conclusions of the rules consist of object/attribute/value triples. The "if" portion of a rule is essentially a series of patterns which specify the facts, or data, which cause the rule to be applicable. The "if" portion of a rule could be perceived of as the "whenever" portion, since pattern matching will always occur whenever changes are made to facts. In expert systems, pattern matching

occurs as a result of the process of matching facts to patterns. In this way the expert system tool provides a mechanism which automatically matches facts against patterns and determines which rules are applicable. The mechanism is commonly called the inference engine (King, 1993). The parts of the whole of an expert system can best be viewed from a flow chart perspective.

The function of the inference engine is to perform logical inferences on the data held in the knowledge base. The inference engine component of the system tends to be a conventional program that is written in an imperative language. However, it is the inferencing process whereby a controlled search strategy is used to draw information from a knowledge base, in accordance with a set of rules held within that portion, that makes an expert system unique. One technique that is utilized by the inference engine is forward-chaining. It reaches a conclusion directly from the user's data. When necessary, the program requests the provision of supplementary information. Another way of performing logical inferences is through the technique of backward-chaining, which begins with a hypothesis, or conclusion, and works backwards, using the data to either prove or disprove it (Klein, 1995). According to Plant (1992), more sophisticated expert systems can combine these techniques.

Rule-based methods offer applications of expert systems to commercial industries (Bell, no date; Motorola, 1995;Pesky, no date) and has been widely used in object-oriented context (Frohn, 1994; Gehani, 1994). However, frame-based knowledge has been recognized as a most useful approach in data modeling (Genesis Database Model) in for two reasons (1) can represent infinite data, and (2) supports a flexible solution domain in a reasonable alternative solution to a problem. A frame based knowledge

representation is the same as a semantic network such that information is arranged in the network hierarchies. Grant's knowledge base (Cohen, 1987) was an example of a frame-based representation in a semantic network. Grant's knowledge base also highly cross-indexed which provides high performance in finding resources for a given proposal. In the Genesis database model, information is arranged as a network of hierarchies. Each sub-frame inherits a characteristics from its parent-frames. It also has data structure that provides inheritance, a reasonable alternative to the solution if the specific data instance is not available.

However, inheritance is complex and sophisticated when there is multiple frame to sub-frames and a child-frame to multiple parent-frame (sometimes called cyclic graph). There are various mechanisms for solving this kind of problem. But it is frame-based concepts that are well recognized for solutions by distinguishing between default and define values, and by allowing users to make slots (first class citizens), giving the slot particular properties by writing a frame-based definition, such as "has-part," "is-a," and "is-part," among others. Ongoing research between Stanford University and the United States Army with the investment over $2,700K per year in the development of the structures for concrete reinforcement (PD13,) proves that the frame-based method is viable. For this reason it has and will continue to receive increased attention in research communities and industries.

Validation by I/O of FBE System

Function-Based Encryption (FBE) is accomplished by a specialized mathematical function (as a hash function, for example) and an entity called a Secondary Function Set

(SFS) to manipulate data in a complex manner. Input and output are coordinated. Input forms or convert letters calculate successive outputs as a secret key in such a way that no single input letter is encrypted twice in the same way though a word, text or application. The same algorithm is then used to convert ciphertext back to its original plaintext (often called decryption). Data only can be read (cipher) by using exactly the same key used to encipher it. In this section, description mode is used to describe the function FBE and its elements, rather than symbolic notation which create more harm than good.

The algorithm is described by Hanink (1997). FBE transforms the input letters (plaintext) with a 4-byte random initialization vector into an un-plaintext (ciphertext) output feedback mode, according to the formV2(x) = G(x)+[S(x-4)V2(x-4)V(x-4) + S(x-3)V2(x-3)V(x-3) + S(x-2)V2(x-2)V(x-2) + S(x-1)V2(x-1)V(x-1)] + V(x) Modulo 256. G(x) is a secret key, V2 is the ciphertext, V is the input vector, and "x" is the current position letter to be encoded. S denotes the SFS values, V(x-4) to V(x-1), the last four encrypted ciphertext values. According to Hanink, what allows the result to remain secret is the correlate values substitution S(x-4)V2(x-4)V(x-4) with V2 and V. This substitution is also responsible for the output characteristics (offer called a cipher feedback) which ensure that each ciphertext value also corresponds to the previous plantext.

In this study, the focus is on the input and output process. As described, the secret key ensures that there are no ciphertext products used the same way twice, even the same plaintext in words, text or application. The algorithm may access many times over the finite output state in the working memory. The internal computational overflow in the output block is amenable to description. It is important to realize that the absolute values of x in function G(x) described above will get larger when x values increase. The concern

here is that when the algorithm is applied too many times to a message in order to encrypt it, it will generate and exceed computational overflow limitations. To prevent this, it has been suggested by Hanick (1997) that the program place an upper bound (divide into subset, subgroup). Pre-computing is also needed on the output values according to the "x" position. However, by employing this technique, the coordinated between value stored in x and the position of x's character no longer exists. This changes the definition of x which helps to control or prevent the overflow limitation in working memory. In Terlouw (1997), the author describes the output set and subsets of the function "GDSOUT" and subroutines "GDSCPA" and "GDSCSA". These functions reflect to this problem area.

There are much needed research efforts taking place at the present time in this area. Numerous researchers have devoted efforts toward improving the FBE algorithm, such as Skipjack (Brickell, Denning, Kent, Maher, & Tuchman, 1993), Xmath (1996), and The Autonomous Machine Learning Laboratory (AUTON) (no date). The objective of Skipjack was to provide a mechanism whereby persons outside the government could evaluate the strength of the classified encryption algorithm used in the escrowed encryption devices and publicly report their findings. Skipjack was but one component of a large, complex system.

The problems still remains, however, especially the need for a generically encryption function which will provide security to any given circumstance. There are promising research studies devoting efforts into this area, such as the work of the Terlouw (1997), Soar, Cellular Automata (Gutowitz, 1996) and Data Encryption Standard, which is most well known system (DES, 1994) for its secure reliability.

Chapter Summary

The purpose of this chapter was to present a review of the literature on the reuse of software components, design and programs. To achieve this goal it was first necessary to review the background of computing leading toward abstraction and reuse. Discussion focused on object-orientation programming with respect to classes, encapsulation, and inheritance, and reuse via inheritance in terms of composition and polymorphism.

The background of mathematics leading toward computing abstraction and reuse was the subject of the second major section. Abstraction and reuse were discussed as relevant to functional abstraction, data driven and message drive. Language evolution was also reviewed. It was noted that no one was able to recall exactly when the history of computing began, but it is known that the Turing Machine was a first computer language machine to be developed. The review briefly outlined evolutionary developments of FORTRAN, Algol, Lisp, C, ML , Miranda, Ada and C++.

Research on reuse was the concern of the third major section. Evolution to measurement was traced first, followed by a review of measurements in terms of lines of code, function points (complexity), and model (program size). Reduction of complexity was the focus of the next subsection. The literature agreed that it is possible to reduce complexity by carefully analyzing components into sub-components and applying the black-box approach. The topic of complexity analysis was reviewed next. Hierarchical classification, faceted classification, and the natural language model were explained. The last subject of this portion of the review centered on communicating sequential processes.

The fourth major section of the review was concerned with object-oriented technology in development and reuse. In the review of frame and frameworks in reuse, it

was noted that the frame concepts were first introduced by Minsky in 1975. Frames provide the defined structure needed to reduce complexity nets. With respect to relational database systems, frame is viewed as a records structure with a database attributes. Each segment of a database is also called a frame. The structure is always hierarchical, cross-referencing link though defined relationships.

It was explained in the following subsection focused on pattern languages design in reuse, that pattern describes a solution to a recurring design problem in a systematic and general way. Design patterns like object-oriented software have promised potential techniques for software reuse. Pattern is not a code, rather a template which provides developers with guidelines for solving problems. The subjects of data abstraction and complexity reduction were also reviewed. It was clear that there were many motivational factors to using object-oriented concepts: trace-ability improvement, reduction of integration problems, improvement of process and product, need to keep to a minimum objectification and de-objectification, ability to hide information, abstraction of data, encapsulation, and concurrency, among others.

Background of the theoretical model was next subject of review. The review included explanations of relations, frames, propositional logic, and constraint satisfaction. The sixth major section dealt with the background of practical machine. Mathematical verification by group theory was discussed first. This included a review of the algebra of notations, the algebra of sets, relations, functions, and Euler-Venn diagrams. An examination of group theory in real world problems, and new functions for C++ followed. According to the literature, new functions for C++ will allow programmers to identify static data members and member functions, classes features; use and override

assignment operators; copy constructors and convert classes; and apply inheritance, multiple inheritance and polymorphism in a program. Final subsections focused on constraint satisfaction, frame-based expert systems, and validation by I/O of FBE systems. The components of expert systems were described in detail. It was noted that expert systems allow inferences to be drawn on encapsulated knowledge. This type of system is characterized by its method of logical deduction from stored data in accordance with rules independent of the program while conducting the search strategy. Like a database, the knowledge base stores information, or facts. It also holds rules for manipulating and interpreting the data, unlike a database.

The section concluded with a review of Function Based Encryption (FBE) systems which use a specialized mathematical function and a Secondary Function Set (SFS) to manipulate data in a complex manner. Input and output are coordinated. Input forms or convert letters calculate successive outputs as a secret key. No single input letter is encrypted twice in the same way though a word, text or application. The same algorithm is then used to convert ciphertext back. FBE transforms input letters (plaintext) with a 4-byte random initialization vector into an un-plaintext (ciphertext) output feedback mode. In the algorithm, G(x) is a secret key. What allows the result to remain secret is the correlate values substitution S(x-4)V2(x-4)V(x-4) with V2 and V. This was important to this researcher because the present study focuses on the input and output process. The review concluded that many researchers are currently devoting efforts toward improving the FBE algorithm.

CHAPTER III

METHODOLOGY

Introduction

The purpose of the proposed study is to describe a method to classify software components and a system to use such a classification efficiently to discover software components that meet a specified need. Specifically, the purpose is to provide a flexible system, comprised of a classification scheme and searcher system, entitled Guides-Search, in which artifacts can be retrieved by carrying out a structured dialogue with the user. The classification scheme provides both the structures of questions to be posed to the user, as well as the set of possible answers to each question. This classification and retrieval methodology applies well to artifacts that are not related to software, such as hardware, patents, books, and legal cases, among others. The model is not an attempt to replace current structures; but rather, seeks to provide a conceptual and structural method to support improvement of software reuse methodology.

The purpose of this chapter is to describe the methodology that will be used for classification purposes and for verifying the effectiveness of the scheme and searcher system. Following portions of the chapter are devoted to this purpose. The first two subsections discuss specific classification schemes for software reuse and study research methods and formats which include explanations of user interface, searcher function, searcher-system roles, and relations used by searcher system. Following subsections describe the browser system, database, projected outcome, resources to be used, and

system measurement. The next portion of the chapter discusses the second type of methodology employed by the study to evaluate the usefulness of this approach. Included are descriptions of the software environment and procedures. A final summary section is also provided.

Specific Classification Schemes for Software Reuse

The review of the literature indicated that for software reuse to be successful, it should not be practiced in environments where it will cost more to discover components than to invent them anew. In addition, there are critical factors which software reuse systems development must take into account in designs and developments. These can be described as follows:

- The classification scheme should include the following attributes: flexibility, extensibility, and ease of use.
- A user should not be presented with a large number of questions or be required to answer any questions known to be germane to query.
- A user should not be given a large number of possible answers to any one single question.
- A user should be allowed to specify an answer not knowing exactly what question the searcher posed to elicit that answer.

In this research, a model of software reuse which will satisfy these critical factors is explored. The methods and procedures used in this study will be discussed in detail in this chapter. The methods used to determine reliability and validity of this study are

discussed in various chapter sections. It is also important to note that, throughout this chapter, the word "user" or "users" refers to system users. In addition, a system users can include software engineers, programmers, managers, or persons in similar positions who uses this system for any reason.

Research Methods

A new method for software reuse as a framework for retrieval systems was briefly described in Chapter 1. It derived from mathematical concepts. While its features provides for improving current software reuse problems, the methodology presented is believed to be compatible to all engineering disciplines. This includes hardware, patents, books, and legal cases, among others. The ad-hoc concept model is used as a guideline for description and will be discussed further in the next section.

The second methodology of the present study focuses on the measurement of components reuse and effectiveness to determine usability. Research procedures and formats using the second methodology are also discussed in greater detail in a later section of this chapter (see section entitled, Research Method to Verify Usefulness).

Research Procedures and Formats

There are various techniques that can found in the current literature for presenting software reuse components. These include: an indexing scheme (Maarek, 1991), keyword-based systems (Mili et al, 1993) and knowledge-based systems (Fischer, 1992; Smith, 1992). Indexing systems use indexing languages to place selected resources into

groups or sets. Keyword-based systems are more domain-independent than other schemes. Knowledge-based systems include semantic-nets and AI technologies.

In the present research, the system can be viewed as one that incorporates a combination of all of these features. The combination of hierarchies model lays over domain model. As pertains to the user-interface, the directed-graph is employed where nodes represent the reusable components(objects) and arcs representing the relationship among objects through classification schemes actions. The specific problems can be specified within the user-interactive mechanism which performs recursively with search and browse capability. Menu-based or windows is used in dialog mapping the input-output for specific components retrieval. Hypertext provides links features among objects or resources on a frame-based according to their relations. At any given stage, users can simply view components stored in the working memory or continue to simplify more specific problems through coordination among the user-interface, retrieval mechanism and browser mechanism.

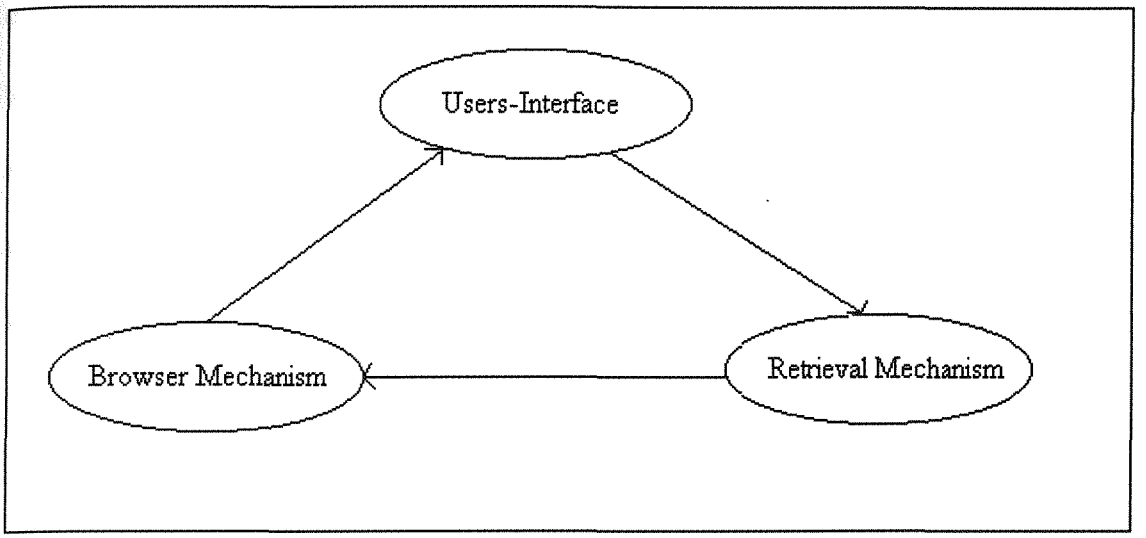


Figure 3.1: General Framework for Software Reuse System.

To support the specific retrieval components, a formal feature describes properties and attributes of reusable components organized through the finite sets of classification schemes known as domain-knowledge. Classification schemes are careful built and stored in the knowledge-base. The database provides a link mechanism to classification schemes. It can be described as a visual storage in which local and global are considered as the same level links. Such components are found in the local machine, WWW respectively. Figure 3.1 provides a flowchart diagram of the general framework for a software reuse system. As indicated, there are three major components. These include a user interface, retrieval mechanism, and browser mechanism.

User-Interface

A successful system goes beyond basic concepts in its definition of user friendliness. However, designing the best user-interface system requires reaching even further. This proposed research effort is devoted to the mechanism for increasing the

efficiency of user-interaction with the system. Obviously, almost all systems include the user interface mechanism. According to the literature, Graphical User Interfaces (GUI) are quickly moving forward to become the most pervasive interface for desktop applications because they are easy to use, as evidence through today's Microsoft Windows Systems (Yazici, Muthuswamy, & Vila, 1994). Although GUI software can be found on DOS and UNIX operating systems, it is a standard feature of Microsoft. Graphic representation of the association between elements of knowledge helps users build their representation of the problem (Heeren, & Collis, 1992). Mental pictures with context instructions improves performance. The picture display helps users to store the relationships between the system variables in working memory, thus functioning as a memory aid (Yazici, Muthuswamy, & Vila, 1994). According to Heeren and Collis (1992), graphical overviews should be organized according to the contextual structure of plans, as they unfold to test various hypotheses in succession as the decision progress.

GUI display interactive objects such as Icons, Buttons, List-boxes, Combo-boxes through its input and output (I/O) mapping concepts. Input-Output mapping concepts should be in perspective from the users point of view, not from that of the application designers (Johnson, 1993). In other words, an information space can be tailored to convey general properties. For example in graphical views, information space is a resolution constraint. An information view can be changed as the user moves through resolution. Objects of interest in the user's view may be moved closer while others of non-interest can be moved further away from immediate attention. The present study is

no exception and takes this into consideration. The user-interface mechanism and features are discussed as follows.

The user-interface mechanism basically represents results in a tree structure dialogue using relations. That is, interaction of users mainly consist of alternately displaying questions known to be germane. For example, users may select "mathematics functions" to be specified. The following paragraphs describe the subset of questions users may choose.

An answer typically would be a property in which a question corresponds to relations and a set of answers to a question which consists of the set of values of that relation. Take, for example, a simple sentence search which asks:

"What is the function"?

This sentence consists of an "is-a" relation and the property defined "function." The initial display result on the properties have the relation "is-a" such as

Legendre functions

Inverse cumulative distribution function

Bessel Function First kind

Parabolic cylinder functions

Probability density functions

Sparsity functions

Time series analysis

Trigonometric functions

Weber functions ...

In other words, the user is allowed to specify a "word" and ask for the relations for which that word is a value (or sub-string of the value, entity names equal to the keyword or which the key word is a sub-string) to be displayed. (An overview of the system scheme is presented in Figure 3.2) Thus, for example, a user might simply specify the key word "spline," and thereby initiate a query to select:

$\{x \mid x \text{ has-operand spline}\} \cup$

$\{x \mid x \text{ has-result spline}\} \cup$

$\{x \mid x \text{ has-operand 2D-cubic-B splines}\}\ldots$

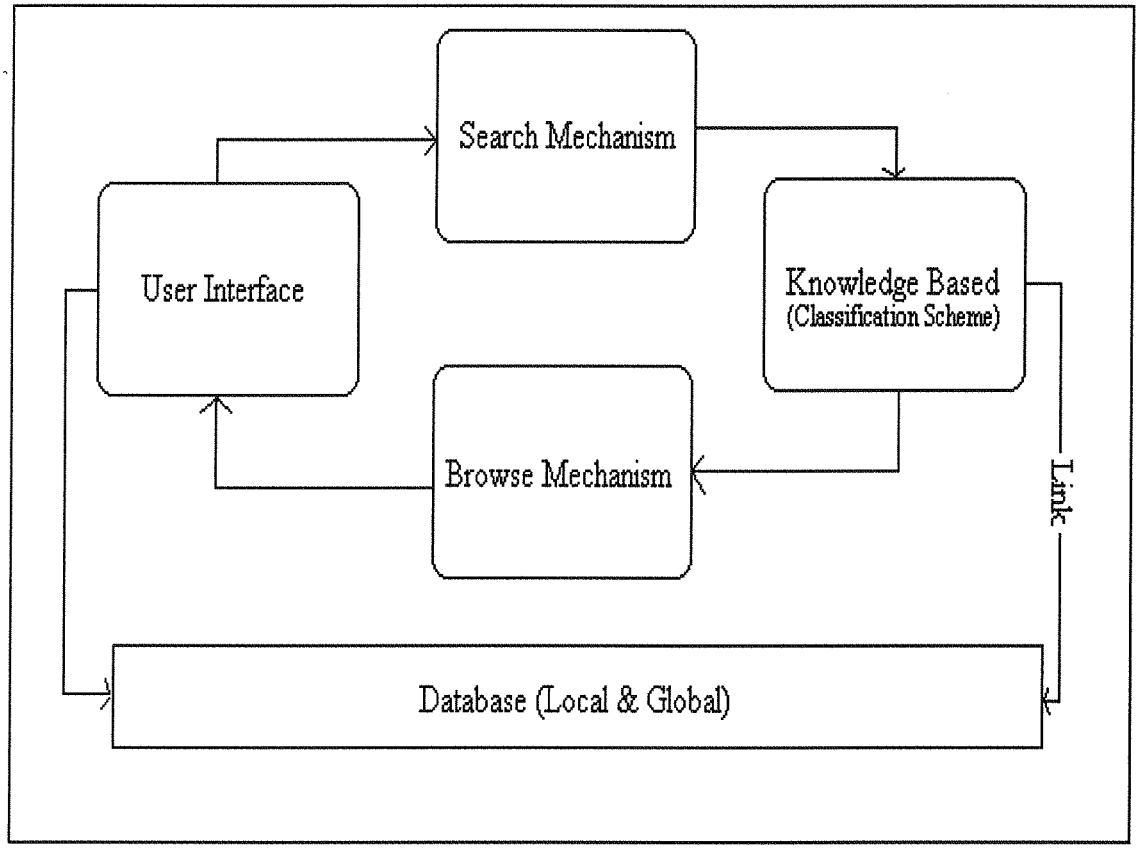


Figure 3.2: An Overview of the System Scheme

However, if the keyword is a value enjoyed by the is-a relation, the user will be allowed to carry out a dialogue to discriminate the value or if the user declines to differentiate the request for *equation*, all of the entities in the set {x | x is-a equation} should be selected. Equivalently, all entities not in the set should be excluded, respectively. For example, if the user denotes V' as a possible candidate for components found per query, and x and y denote properties or components it represents, then using 'is-a' relation:

$$V' = \{x \mid x \in V_r \bigcup V'_r\} \bigcup \{y \mid x \in V_r \bigcup V'_r \wedge x \text{ is-a } y\} - \{x \mid x \in V_r \bigcup V'_r \wedge x \text{ is-a } y\}$$

Given these user-interactive concepts, the research presents an alternatively resolved multiple relationship (cyclic) of problems. A components may have multiple parents through a means often called partitioning. Speaking in Dor. Dilemma to Disjunction, partitioning is characterized from its relations by the user-interactive derived specified problems. The Dor meta-rule is used to show links between a disjunction and the conditionals that the user might choose. For example the user might simply ask a system for a "function." The sets components are returned such as has-mathematical-functions and has-Statistical-functions - that is, {x | x has-mathematical-functions} $\cap$ {x | x has-Statistical-functions}. Having chosen x, x has-mathematical-functions and negative binomial distribution. Normal distribution would be excluded. In addition, the user should also be allowed to type a prefix of an equation or answer. Ideally this would

allow a regular expression with display showing the set of questions or answers of which the typed expression is a prefix (example (x and y)). Also, the user should be allowed to change the value selected for any relation or to indicate the relation to be excluded for which a value has been selected. Association with relations are described. Propositional logic such as negation "not x" and disjunction (x or y) are also allowed. Input and output mapping can totally clear out in the working memory at any given stage. When the "Cancel" button is triggered, it brings users to the start-selection state.

Searcher Mechanism.

According to the literature and previously described, a retrieval mechanism is crucial in software reuse systems (Girardi & Ibrahim, 1995; Novak, Hill, Wan, & Sayrs, 1992; Solderitsch, 1995). It is not that it just provides fast and easy identification of reusable components in the library, but also permits browsing relevant components that it enjoys through relation form (a r b). "A" is denoted as a component sharing or holding some functionality and "b" is denoted as either components or properties through its relation defined "r".

In context, the searcher is a software system that a user employs to discover the specified software. Interaction of the searcher with a user mainly consists of alternately displaying questions known to be germane. Once a question/answer has been selected, it displays possible answers to that question. Questions correspond to relations. The set of answers to a question that the user chooses to answer consists of the set of values of that relation. To help users discover alternative components that can meet their needs, the

role of the searcher system must first be described. Role components include the following:

- To assist users in extracting components specified.
- To provide a similarity candidate, potential components prior to the components specified.
- To narrow a relations by using recursive methods.

The role and responsibility of the searcher mechanism is to retrieve relevant components specified by the user. The retrieval mechanism mainly communicates with relations existing classification schemes repositories to obtain all possible candidates related to the request. This mechanism is mainly responsible for matching the users input to the existing classification schemes via the relations. This research uses the "is-a" relation to collect components and properties in the form of [a **r** b] relation - that is, to represent a classification by the collection of triples, where a triple is a record in the database, and each member of each triple is a field in a record. For efficiency, the relation **"r"** will be sorted first, followed by components "*b*" in the relation form "property relation component." Thus, it is presented as a form (a r b) in which a precedes b in lexicographic order. By using lexicographic order, the searcher mechanism forces every components-relation-properties form in a canonical form in which there are no duplicated classification schemes.

a. Searcher System Roles

In order to archive its roles, this study will need to examine the following critical points: how to avoid giving too many answers to a single question; how to avoid asking

too many questions for a single answer; how to get users to specify answers not knowing exactly what question the searcher posed; and how to broaden the performance of expert users. These topics are addressed in separate sections below.

How to avoid presenting the user with too many possible answers to any question: This example will compute the set of values V_r that a germane relation **r** enjoys. It is most ideal when restricted to non-excluded components, that is:

$$V_r = \bigcup_{\text{a not excluded}} \{x \mid a\ \mathbf{r}\ x\}.$$

consider only components in $\{x \mid x$ **has-application-domain** *numerical-analysis*$\}$; it is unlikely that the value of another language such as value *lisp* will remain for relation **has-programming-language** (*lisp* should not be in V_r **has-programming-language**). This will prevent the user from asking a question for which there is only one or no answer. Moreover, it must be assumed that V_r is very large. To avoid exposing the entire set, the classifier might have arranged V_r values into hierarchies. This is accomplished by using the **is-a** relation which considers the top of a hierarchy or parent node. For example, suppose the *SVECP* **has-operand** *vector* and *SGEFS* **has-operand** *equation-ALD*. The property *vector* does not employ any **is-a** relations. It employs the relation *equation-ALD* **is-a** *equation*. The algorithm illustrated below makes an obvious point which is to avoid asking a question for which there is only one possible answer available:

While $\exists z \in \vee'r \ni x$ **is-a** $z \wedge |S| = 1$
where $S = \{t \mid t$ **is-a** $z\}$
{replace $\vee'r$ by $(\vee'r - z) \cup S$}
End while.

This amounts to replacing the top of a hierarchy with its child if there is only one child. For example, one might have software for solving algebraic and differential equations in the FORTRAN programming language, but software only to solve algebraic equations in the C programming language. A user having selected **has-functionality** *solve* and **has-language** FORTRAN would find the answer equation when answering the question "what is the operand?"

Thus, the FORTRAN software for solving algebraic equations employs the relation **has-operand** *equation-ALD*, while the FORTRAN software for solving differential equations employs the relation **has-operand** *equation-ODIN2*. The *equation-ALD* and *equation-ODIN2* employ the relation **is-a** equation. A user who selects *has-language* C instead of *has-language* FORTRAN would find *equation-ALD* in the set of possible answers to the question "what is the operand?". This would occur because the *equation-ALD* is the only property that is the value of the operand relation for a non-excluded component, and that employs the relation **is-a** equation. Therefore, it should not ask the user "what kind of equation?" because the only possible answer to the question is *equation-ALD*.

If the user chooses a value $z \in \{y \mid x$ **is-a** $y\}$, the interface then exposes one of two subsidiary windows depending on $N = \bigcap_{X \in S} (N_X)$, the intersection of the sets of

relations employed by the property in S. In this case N = {is-a}, the searcher simply exposes a window containing S. For example, if the user selects *equation* as the value of the **has-operand** relation and the only **is-a** relation is employed by *equation-ALD* and all other properties in E = {x | x **is-a** *equation*} (even though some members of E enjoy other relations), the subsidiary window should contain *equation-ALD* and all other values in E. In the examples above, one saw that the property *equation-ALD* also enjoys the relation *equation-ALD* **has-kind** *algebraic*. In fact, every property x that employs the relation x **is-a** equation also employs the relation x **has-kind** k. Having specified a value k for the has-kind relation, the set

$$S' = \{x \mid x \text{ is-a equation} \wedge x \text{ has-kind } k\} \subseteq S$$
$$\text{therefore } |S| \leq |S|$$

Having restricted attention to S', there may be relations that apply to every member of S' but did not apply to every member of S. For this reason users should be asked to specify values for them. Other than using the **is-a** relation in reverse, the same mechanism that was used above can be applied to avoid asking the user questions that are not known to be germane.

If the user has selected a property value z such that $S \neq 0$, that user need not select one x from S. The effect of declining to select is that the system behaves as though every element of $S^{+}=\{x \mid x\ is\text{-}a^{+}\ z\}$ were selected. Thus a user might specify *equation* as the value of the **has-operand** relation and decline to differentiate further the kind of equation about which one is interested.

How to avoid asking too many questions: The proposed study will look into the solicitation of answers to questions known to be germane. The searcher will ask the user to provide values only for relations that can distinguish between components that have not yet been excluded. (It can also be noted that this method also solves the semantic problems often encountered in the so-called cyclic link). After the values are received, construct for each component x a set $N_x = \{r | x \ \mathbf{r} \ y\}$ which will consist of names of relations that component x employs. The set N_x for a component x from a library of the mathematical software might include **has-linearity** and **has-precision** but would probably not include **has-data-model**. At any given stage in the query, the searcher displays only the set of relation names that are in $N = \bigcap_{x \text{ not excluded}} (N_x)$. The intersection of all sets N_x for components x that have not been excluded, the searcher displays a set of relation names that apply to every component. Therefore, this initial set should be computed a priori. Thus, the user initially might be asked to specify values for the **has-application-domain** and **has-programming-language** relations, but s/he will not be asked about the precision of floating point calculations nor whether a data model is hierarchical or relational. For example, after specifying the *application-domain* **is** *database systems*, every non-excluded component might employ the relation **has-data-model**.

A user may have knowledge of key words which relate to the problem, but might not know the relations. If queries are directed by key words (allowing users to specify a word and ask the relations for its value), a sub-string or entity name will correspond to

that key word. This will provide a better server system. When the set is non-empty, the user can be allowed to stipulate relations having a value given by that key word. For example, a user might simply specify the key word "spline". This will initiate a query to select {x | *x* **has-operand** *spline*} ∪ {x | *x* **has-result** *spline*} ∪ {x | *x* **has-operand** *2D-cubic-B-splines*}. A key word is a value that can be employed by the is-a relation. When a user declines to differentiate a request for an equation, all of the entities in the set {x | *x* **is-a** *equation*} will be selected. All remaining entities not in the set {x | *x* **is-a** *equation*} will be eliminated.

An alternative choice could be that the user selects a property value such that $z \neq 0$. This would not necessitate selecting one x from S. There is a declining effect selected through every element of S = {x | *x* **is-a** *z*}. Thus, the user may specify an equation as the value of the has-operand relation and choose to decline to differentiate the alternative equation.

An expert user may know a set of relations germane to query. A value of any relation in the catalog will be available at any given moment as requested by the user. Initially, a list of names of relations will be displayed. Once the name of a relation is selected, the value of that relation will appear. For example, if choosing to start with "what is the function," the selected answer would be *Bessel functions of the first kind.* Because the value of a relation will be selected, the previous screen will be displayed again showing the value selected for each relation. In addition to using a pointing device, the user will be allowed to type the prefix of a question or answer which is a regular

expression. The display will show the set of questions or answers that contain the typed prefix.

When sufficient information is specified from the typed prefix, a unique determination will be made from the question entered. The user will be able to change the value selected for any relation or to indicate a relation for which a value has been selected or to not have a value selected. The user may choose to provide values for any, all or none of the relations. Declining to specify a value for a relation is equivalent to specifying all possible values. An example would be a user interacting by selecting $\{x \mid x$ **has-functionality** *solve*$\} \cap \{x \mid x$ **has-operand** *equation-ALD*$\}$.

b. Retriever Functionality

The retriever is a software component that retrieves all of the files necessary to use a selected component. In the simplest case, the component database and searcher reside in the same computer and the retriever might simply produce a list of the names of the files necessary to use the selected component. In another more complex case, the component might be distributed across several computers. This case would require specific tools such as TCP/IP, FTP, and modem. However, for the purpose in this study, the Word Wide Web (WWW) is used to present other components not found in the local machine.

The needs for relations "is-in", "needs", and "part-of" are of use to a retriever. The relation "is-in" specifies the file in which a component is contained. The second relation is the "needs" relation, which indicates other components which are necessary to use the specified component. The third is the "part-of" relation, which specifies that one

component is part of other component, the latter being a composite component in which one might not require all components in order to use one of them, such as the library of mathematical software. An example can be found in the Guide to Available Mathematical Software (GAMS). Let us consider the components available in the AMOS. That is,

```
Package AMOS
        has-function CBESY
        has-function CAIRY...
        is-in http://math.nist.gov/cgi-bin/gams-serve/list-modules-in-package
                /AMOS.html
End Package AMOS
```

Likewise, considering a component Zero of a Univariate Function in ANSI FORTRAN77

```
Component SZERO
        has-operand univariate-function
        has-function zero-find
        has-datatype real
        has-precision single
        is-in LocalFile C:\Public\Math\Fortran77\SZERO.FOR
        needs (R1MACH ERMSG)
End Component SZERO
```

where R1MACH ERMSG is an external references. (FORTRAN77, Fortner Research LLC).

To retrieve the set of files necessary to use a component x, one might naively believe it is sufficient to compute the set of components c_x, needed by x or that are a part of x, that is, $c_x = \{x\} \cup \{t \mid x \textbf{ needs } t \vee t \textbf{ part-of } x\}$ and retrieve the set of files $f_x = \{f \mid$

c **is-in** f $\wedge$ c $\in c_x\}$ in which these components are contained. However, the file might contain more than one function and that function might not be present in the same file. To construct the correct set, let Φ denote a set of files and x denote a set of components. The auxiliary functions can be described in the following way:

c(f): Φ -> x = $U_g \in$ f {t | t is-in g}; is a set of components in the set of files

n(c): x -> x = $U_z \in$ c {t | z **needs** t $\vee$ t **part-of** z} is the set of components needed or part of the set components

f(c):x -> Φ = $U_z \in$ c {f | z **is-in** f} is the set of files containing a set of components.

The set of files needed is then the least solution of the equation
f'_x = f({x}) $\cup$ f(n(c(f'_x))) which can be solved by fixed-point iteration:
Let f'_x := f (x{x}) which initially contains only one element

repeat let f'_x := $f'_x \cup f$ (n(c(f'_x))) **until** nothing is added to f'_x

c. Syntax and Semantics for Entity Descriptions

The collection of relations that describes an entity could be specified by enumerating all the relations the entity enjoys. However in the real world, components often enjoys several relations and specifying the entity name in every one is repetitive. An alternative way to solve this problem is to group into a block the relations that characterize an entity. Let us considering in the example of the property equation-ALD:

Property equation-ALD
    is-a equation
    has-kind algebraic
    has-determination exact
    has-linearity linear
end equation-ALD

A component block describes an object to be classified or the properties of a collection of objects. As described, a property might be an atom. An atomic property enjoys only exactly the relations specified in "is-a" relation. An example that *numerical-analysis* **is-a** *computing* and *numerical-analysis* **is-a** *mathematics*.

Entities frequently enjoy similar relations. Components classified can be simplified into a group of relevant. The mechanism allows an entity is "like" another entity. Such that

```
Property equation-ALO like equation-ALD
        except has-determination over-determined
End property-ALO
```

where the syntax of a default block is

```
default (component | type-name? Property)
        (relation-name relation-value)*
                ((type-name? Property |  Component)
                        entity-name ("("(g(","g)*)?")")
                (like entity-name)?
                ((except | needs)? (relation-name relation-value))*)*
end default
```

It is also important to note that the language (inner default block elements) is defined by ()*, indicating that the language could contains the null word (Automata theory). Also, it should be noted that a block is generic if it includes a list of names or generic parameters in parentheses after the block name. It may be used to collect relations enjoyed by several similar components or properties without enforcing an order in which users must indicated questions about properties. The mechanism can be defined in a frame-based as described in the previous chapter.

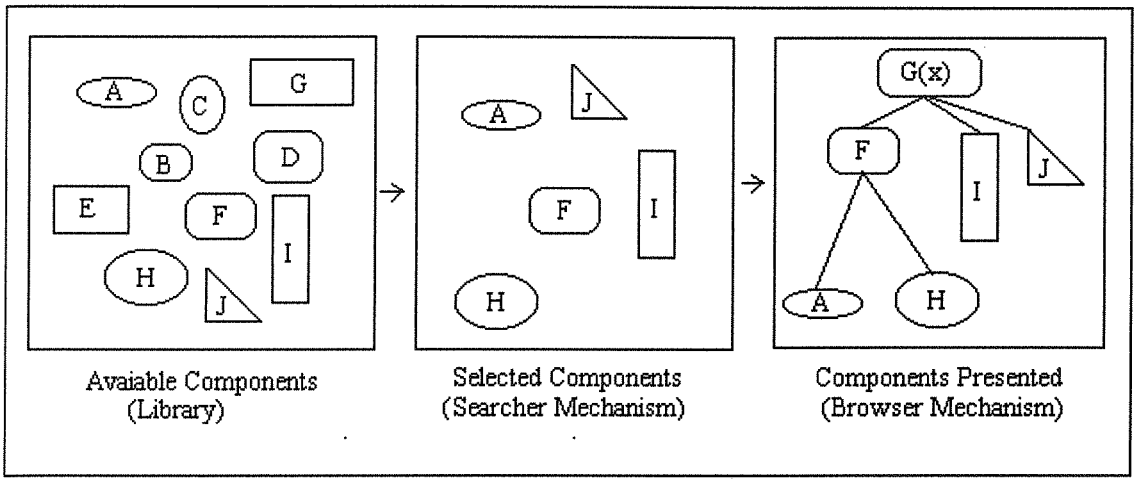


Figure 3.3: Semantic Structure to the Dialogue Menu Type

The relationship of the Semantic Structure to the Dialogue Menu Type is illustrated in Figure 3.3. In the new semantic of classification and inheritance, a component default block indicates that all relevant components enjoy the relations and values specified in the block until another component default block is brought to the current working memory. This allows objects (higher level object, groups level) constructor to achieve higher abstraction. That is, any specification of relations within a component block replaces the values of corresponding relations inherited from the default block. An example of GNN is described in Chapter 2 of this study.

Description of Browser System

The algorithm known to be a key player of software reuse systems and its features is significantly impacted on the research in the way that it allows a specified problem to be presented. The key concept in this case is the communications of a concurrent graph, called nodes. Each node is representing the relationship R over the set of relevant or

neighbor components. The mapping of nodes is transparent through users interaction with the system. Therefore, it is possible to provide functionality to the dynamically composition divide or merge nodes. These components can briefly be denoted in terms of the "has-a" relationship, as described in the previous chapter. Additional, there is also denoted in term of an inheritance "is-a" relationship. Katzenelson (1992) defined such composition in terms of type-graphs and can be summarized in terms of composition merge and composition divide as follows:

The group G levels to be constructed is based on the values of the most general relation according to a specified priority (Salton, 1989). It is assumed that the relations can be ordered lexicographically once described in type-graphs. This assumption relates to the uniqueness of the labeling operands (or types). Priority form such that **priorities** (relation-names) where relation-names one might specify "has-application-domain", "has-functionality", "has-operand", or "has-language", "has-package", etc.

These relations may generate a large number of distinct displays possible in interacting with the users. This poses an important difficulty. To avoid exposing the entire set, the classified might have arranged values G into hierarchies by using the "is-a" relation. As described in the previous chapter, that is only the top of a hierarchy, i.e., the relation values in the G' as an alternative solution of the equation

$$G' = \{x \mid x \in G \cup G'\} \cup \{y \mid x \in G \cup G' \wedge x \text{ is-a } y\} - \{x \mid x \in G \cup G' \wedge x \text{ is-a } y\}$$ should initially be exposed.

Consider a function SVECP. (SVECP has-operand vector) and (SVECP has-operand equation-ALD). The property vector does not enjoy any "is-a" relations, although it may enjoy others. Likewise, the property equation-ALD enjoys the relation (equation-ALD is-a equation). Thus when soliciting a value for the "has-operand," the set of possible values should initially include vector and equation, but not equation-ALD.

Likewise, let $A_{rb} = \{a \mid a\ \mathbf{r}\ b\}$ then values (V) would be

$$V_{rv,s} = \{ w \mid a \in A_{rv} \wedge a\ \mathbf{s}\ w \} - \{w \mid a \in A_{rv} \wedge a\ \mathbf{s}\ w \wedge w\ is\text{-}a^{+}\ z\} \cup \{z \mid a \in A_{rv} \wedge a\ \mathbf{s}\ w \wedge w\ \mathbf{is\text{-}a}\ z\}$$

if a selection of the value x $\in V_{rv,s}$ for the relation **s** is subsequently made, the only values displayed for the relation **t** would be values in the set:

$$V_{rv,sx,t} = \{y \mid a \in A_{rv} \cap A_{sx} \wedge a\ \mathbf{t}\ y \wedge y\ is\text{-}a^{+}\ z\} - \{y \mid a \in A_{rv} \cap A_{sx} \wedge a\ \mathbf{t}\ y\} \cup \{z \mid a \in A_{rv} \cap A_{sx} \wedge a\ \mathbf{t}\ y \wedge y\ is\text{-}a^{+}\ z\}$$

Thus, the total number displayed in the browser is the sum of the numbers of components that employ values of relations that contribute to the classification browser (See Figure 3.3). This led this researcher to consider in this case that relations value refers to a duplicated nodes already exist in the working memory, as indicated in Figure 3.4 which presents a flow diagram of composition by merging.

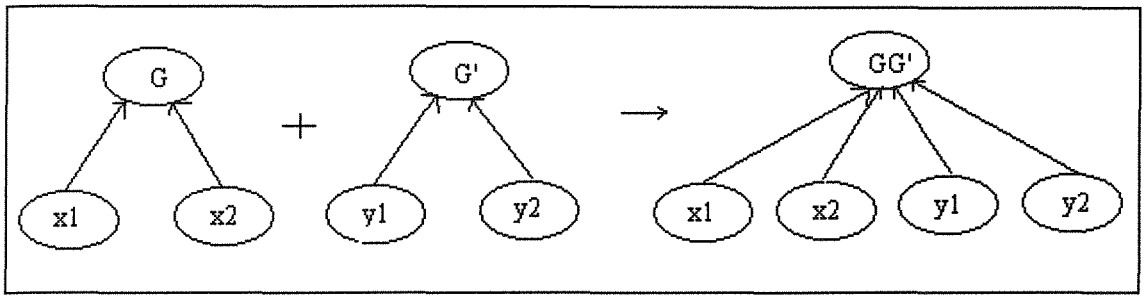


Figure 3.4: Composition by Merging

Whenever a relations value refers to an existing node, that relation-value-node will be eliminated and all edges will be redirected to the corresponding referred node. That is, given a set G where G = {x | x ∈ X} and a set G' = {y | y ∈ Y}, for each x(i)with an edge (x -> X), there is a y(i) with an edge (y -> Y), and partitioning instance of the relation (or number of common features) X(r) = Y(r). Therefore, G = G'. The end of the results is that, G' will be eliminated and leave G with whole components are the sum of x and y. G = {x, y | x ∈ X and y ∈ X} by composition merge. On other hand in the composition divide, the new relation-node can be created, called "N". Composition divide is diagrammed visually in Figure 3.5.

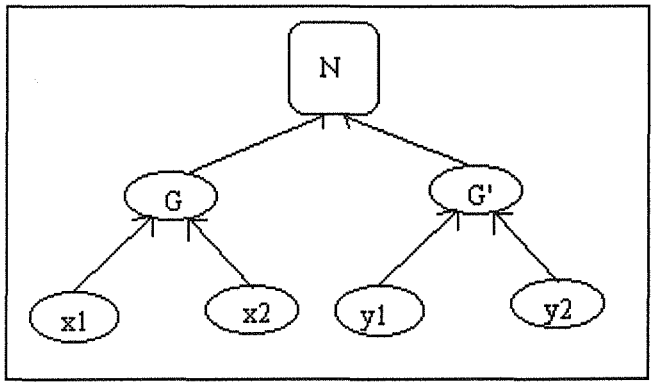


Figure 3.5: Composition Divide

By using a one-level-deep traversal in hierarchy and a substantial interactive part with users, X or Y can be automatically determined. This raises the issue of how this algorithm handles the case of a cyclic graph in which a component might be determined by multiple nodes or by multiple parents. This issue is illustrated in the cyclic graph presented in Figure 3.6.

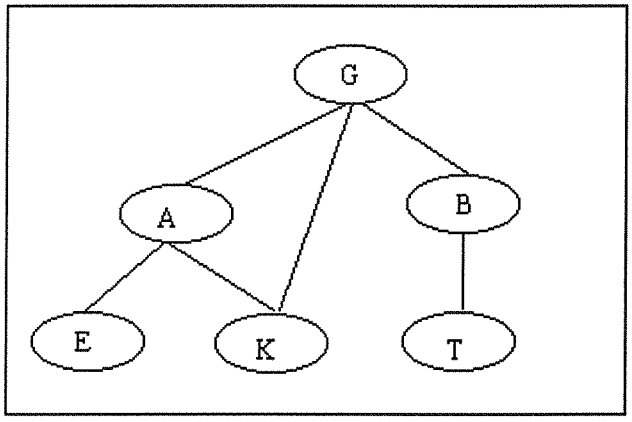


Figure 3.6: Cyclic Graph

In Figure 3.6, G could be denoted as "a computing language" which has-components in nodes A, K and B. For example, A "lisp" compiler (node K) might be classified under *programming-languages* (node G) and also under *artificial intelligence* (AI) in node A. That is, AI is a programming language, a LISP compiler. LISP compiler is also referred to as an AI. The cycle can be eliminated as users participate.

The relation defined set of components, as indicated in Figure 3.6 can be presented as; G = (A, K, B), A = (E, K). By applying an ordered depth-first search (assuming that the height of a node has been defined, that is h(GA) = 1, h(GE) = 2, h(AK) = 1, h(GK) = 1, etc.), it is guaranteed that the unique set of elements can be defined. In

this case, G and A (GA) = (E, K, K, B) → = (E, K, B), as illustrated in Figure 3.7 which offers a solution to cyclic problems.

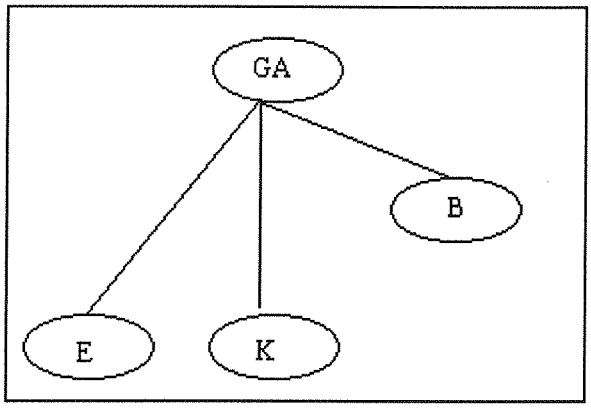


Figure 3.7: Cyclic Problems Solved

It can also be noted that the original graph can be recalled by applying the composition laws as has been described above. If GA is a current state, questions correspond to a G or A. Upon selection of "A" by using the height h(v) as defined in the tgraph, it redraws A and goes back to its original as showed in Figure 3.6. E is closer to A than to G, likewise K is closer to A than to G, as previously described previously. In any state, however, users can reset and the search will start from the beginning or users can continue on with smaller possible components and in this manner move closer to the specified problem.

Link Classification Scheme

In addition to the classification schemes described, the system supports the users in defining relations link from classification schemes to their documents. In this context, documents refer to a collection of source-code posted as a file on the World Wide Web,

or in the local machine. However for efficiently search, classification-schemes are carefully defined and stored in the local machine. Classification schemes also provide the location of components, as described in the relation "is-in" above.

The advantage of technologies as seen today can also be noted, commonly found through the WWW. Perhaps within the near future there may be possible for a real-efficient search (a hope!) an algorithm that will be able to provide a mechanism which can travel into all world personal computers and all networks to obtain a component that has been specified. If this should eventually come about, there will be no need to be concerned with the data space requirements on local machines for the building of classification-schemes.

Natural languages mechanism has at the present time become one of the most popular areas in which much research is currently devoted. These efforts continue to bring more promise to this hope. It appears possible that someday the hope will be realized. At that time, users will only need to obtain a front-end mechanism and a perfect-natural-language mechanism for sharing whole word problems, together with their respective solutions. However, considerably more effort needs to be exerted in this direction before this hope can become a reality. In the mean time, it is important to discuss the database.

Description of Database

This research will maintain a "virtual library" database. The local database contained on the PC and global databases will provide the classification of software

mainly through the World Wide Web (WWW). Thus, for consistency in the present study, the local database and the global database are to be considered at the same level. The researcher does not maintain a complete repository; but rather provides indexing to other resources through the WWW.

However, there are database tables to represent the classification schemes which consist of a data structure to the values of the relations in the (a r b) form. For the purpose of this research, the study mainly focuses on the resources obtained from GAMS. This relationship is illustrated in Figure 3.6.

a. Package Domain

A Package Domain can be presented in the form of tables in the relation database. In GAMS, the package-Domain table contains:

Database: GAMS

ID	Char	// TOMS, NAG, NAPACK, etc...
Domain	Char	// On-Site as Called in GAMS
Package-Des	Char	// Package Description
Location	Char	// URLs (www), File C:\...
Language	Char	// Fortran77...
Reference	Char	// Ex: ACM Vol3, no:2, (Oct, 1997)
Developer	Char	// Authors
Distributor	Char	// NETLIB

Such a record contains:

ID: ITPACK
Domain: NETLIB
Domain: A collection of subroutine packages solving large sparse systems of linear algebra

Location: http://math.nist.gov/cgi-bin/gams-serve/list-package-components/
ITPACK.html
Language: Fortran
Reference: ACM TOMS 8 (1982)
Developer: University of Texas at Austin
Distributor: NETLIB

ID field and Domain field are used to represent the primary key because there may be cases in the same package that exist in different locations. Such a package is IMSLM, in GRANTA, CAMSUN, TIBER, etc. Likewise, sometime at the same location different packages contain the same components. For example, Class-O, Symbolic Computation at CAMSUN location, there are existing packages such that FORMAT, MACSYMA, MAPLE, MATHEMATICA, TOMS, etc., containing symbolic-Computation relevance components.

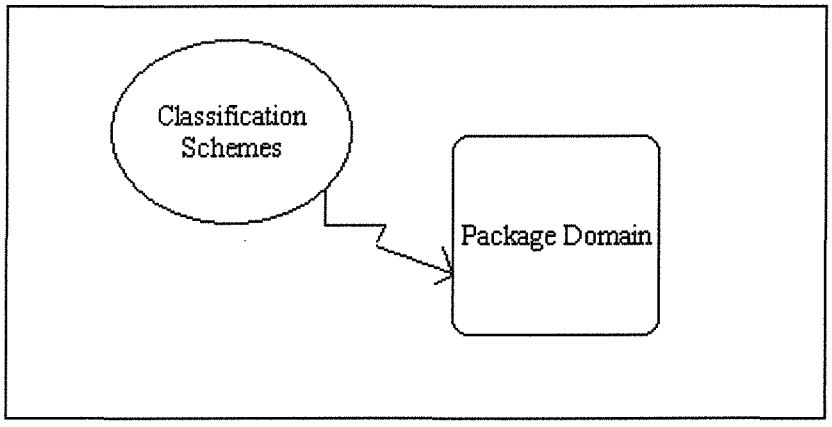


Figure 3.8: Relation Value Retrieval

Each package contains one or many components. They are presented in GAMS as a module and assigned to an unique index number in each package or its retrieval. For

the purpose of this research, for the elements (modules, procedures, etc.), each package will be represented in the database format of:

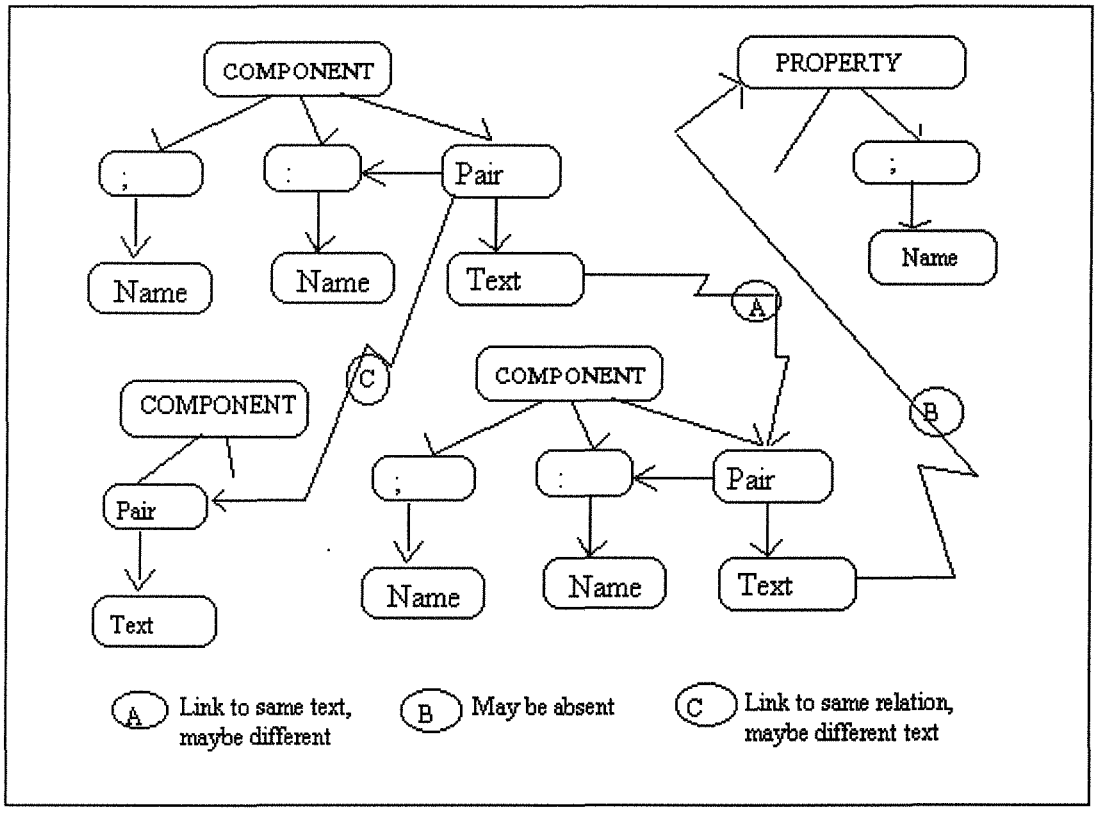
Domain-Elements

ID	Char	// Unique DOMAIN-ID identification
Domain	Char	//
GAMS-Index	Char	// 746-in package TOMS to the problem:
		// Class-O Symbolic Computation
GAMS-Des	Char	// Modules description or title
Location	Char	// http://math.nist.gov/cgi-bin/gams-serve/list-module-components/TOMS/746/13033.html
Catalog:	Char	// Problems/solution. For example: Class-O Symbolic Computation.
		// Also defined as Problems-Domain

Fields: ID, Domain and GRAMS-index is defined as a primary key in this table. The following is an example of a record:

ID: TOMS
Domain: NETLIB
GAMS-Index: 746
GAMS-Des: PCOMP: An automatic differentiation package
Location: http://math.nist.gov/cgi-bin/gams-serve/list-module-components/TOMS/
746/13033.html
Catalog: Symbolic Computation

Figure 3.9: Component/Property Linkage Diagram



b. Classification Schemes

As described above, classification schemes will be presented in the form of (a r b) -that is (components **relations** properties). The database table consists of the following:

ID	Char	// an unique identification (ID-DOMAIN) defined
Domain	Char	// component name
Relations	Char	// has-..., is-a, like, except, etc...
Property	Char	// Uniform-random-number, etc...

For example, in ANSI Fortran77 (Fortner, 1995). Function SRANU is used to compute a uniform random number and can explored in the format of the relation database such as:

ID: RANU
Domain: Fortner
Relations: has-function
Property: Uniform-random-number

Likewise, Component RANU has-operand none,

Component RANU has-precision if p=s then single else double fi
Component RANU has-output scalar

Projected Outcome

A flexible classification scheme is believed by this researcher to be the significant contribution of the study and one of the projected outcomes. This scheme can be defined as an interaction among users to specify users' needs which acknowledge the value of the different levels of expertise among users. The contributions of the scheme can be characterized in the following manner. It is:

1. a mechanism that allows recursive interaction between users and systems;
2. a mechanism that allows users to specify problems without too many questions/answers;
3. a mechanism that allows the expansion of classification schemes; and
4. a model that makes no attempt to replace the current structure. Instead, it seeks to provide a conceptual and structural method to support the improvement of software reuse designs.

A second projected outcome for the study is agreement between the data obtained in this study from this evaluator with that derived from the Guides Search system. It will be explained in later sections that this researcher will apply human intervention to determine the amount of time necessary to retrieve components and rate specified components of three or four identified systems according to three criteria. It is critical that a system responds in a reasonable time frame. It was hoped that human subject evaluation of the same components derived from application of the Guides Search system would agree with respect to reusability and efficiency, thereby validating the reliability, validity, and usefulness of the searcher system approach.

Resources To Be Used

As noted, the present study focuses on provisions of a simplified, faceted approach to information retrieval for reusable software classification. The purpose was to describe a method to classify software components and a system to utilize such a

classification efficiently for discovering software component needs. To succeed, a prototype library system will need to be designed and developed in conjunction with simplified classification schemes. Microsoft Visual Foxpro v5.0, running on the Pentium-133 with 2.5 GHD and 16MRAM, could be used to build the prototype system.

Additionally, there are software packages from well-known companies that can be used - Fortner Research LLC, GAMS on the WWW. It is also important to point out that the research goal is to provide a link to all available components or modules in this system in all different language such as C, VBASIC, VC++, LISP, Smalltalk, and others of a similar nature. It could be possible if there are modules available in the public domain and if time permits. However, for the purpose of this research study, some packages from GAMS need to be selected and used for link purposes. The indicated numbers of modules from GAMS are provided by Boisvert who was personally contacted (Appendix A, 1997). These can be, but are not limited to, incorporation as a part of the prototype library. A breakdown of pertinent information regarding these packages is as follows:

System or Package*	Number of Components/Modules	Language
FORTRAN77	450	FORTRAN
GAMS-NAG	2148	FORTRAN
GAMS-IMSLS	625	FORTRAN
GAMS-CMLIB	739	FORTRAN

GAMS-DATAPAC	169	FORTRAN
GAMS-IMSLM	1049	FORTRAN
GAMS-IMSLS	752	FORTRAN

Table 1: Software Packages to be Used. *(Appendix A)

System Measurement

As stated in the current literature, in order for repositories be useful they must have a large number of components in all different areas to support the developer (Henninger, 1996; Esteva, 1995). However, when there are many components available, it is no longer possible for a single retrieval to find the specific components a user needs. According to the current literature, this is an open problem and it is acknowledged that there are many necessary tools currently available.

Given this challenge, the researcher introduced a flexible classification scheme which acknowledges the value of expertise differences in users and coordinates that value by allowing interaction between users and the system to locate specific components that are specified by the user. With that in mind, reliability measures will be demonstrated as pertains to the complexity for reusability. Within the flexible methods for reusable components, the measures of the effectiveness will also be evaluated by this researcher during the evolution of the interactive and retrieval components.

This study confirms that reusability is related to many variables. These variables range from program size to each component's attributes and the expertise level of the user to the capabilities of software engineering. There are many things surrounding each

system which need to be considered in its measurement. Taking all of these variables into consideration is hard to do, according to Esteva (1995). In this research, notice must be taken that the collection of values of the facets can be considered to constitute the coordinates of a point in space. Therefore, the importance of how tightly or loosely each component is bounded to another in its relations must be considered. This researcher asserts that these bound variables and their components attributes are also correlated with the level of human expertise.

As previously explained, the software reuse system mainly focuses on three mechanism: user-interface, retrieval mechanism and browser mechanism. Each has its own responsibilities to the outcome related to the efficiently and effectively to the system. Their distinguish tasks can simply be summarized in the following manner. The retrieval mechanism is responsible for identifying reusable components or relevant components that a user has specified. The browser mechanism is responsible for the organization of identified components in such as way that components can be closely linked to their respective groups. The user-interface mechanism is utilized to present the answers or questions interactively to the retrieval and browser mechanisms which in turn, present the results to the users.

The idea is to start with the set of values of the relations which is identified as the infinite components relevant to the one specified. Upon identification of the relevant components, the algorithm then generates components in the successful group as described in the graph. The node of components may changed during the process. This

depends on the values of relations each new component enjoys. That is, the algorithm may repeatedly be called and process per unbound components specified.

To generate the new unbound components path, the algorithm stores the previous node and its baseline paths in working memory and changes the current state by allowing the new component to be rejoined as when a new process begin. Subject to the constrains of the relevant from the values of relations corresponding to the current node, a node might be terminated from its parent node or it becomes a parent node itself. Regardless of the current components set, user-interaction is required for a specific identification. Relevance to the nodes can be determined by the shortest path (region 1, following the Euler formula) which make up the set the components enjoy. This method can simply be measured in the link-weight and is described in the next section.

Link Weight

As explained in the previous section, the purpose for the complexity of the reusability of the classification scheme is to divide the information space into many small pieces and solve one piece at the time. This approach has led to the measurement of the closeness between each component and its relation. The link weight is also a good indicator of the number of questions or answers in the user's interaction. The simplest concept that can be used to explain this is the hierarchy form. The relation is at the root {*}, denoted as a parent node of a classification scheme. If an arc is drawn from each child to the parent node, this concept can be measured by measuring the distance and computing the minimal value for the closeness formula.

To compute the distance between each component, let U_x be a set of an unbounded component of S, denote Q_{ij} a subset of S in the information space where d_{ij} denotes the shortest path as described above from all set a_i to a_j components in a given search.

$$S == \sum_1^n \{a_{x,y} \mid a_{x,y} \in Q_y\} \wedge \{x \mid x \in U\}$$

where y = 1 to n and "= =" denotes nearly or equal.

Applying this concept to compute the shortest path in S can be denoted:

$$\Omega == d_{ij} (Q \cap U)$$

Adequacy

In order to evaluate the efficiency of the selected relevant components, the quality attribute to be used in the system is defined as an adequacy measurement. This structure measures the consists of high level set in term of relevant percentages. As described in the link weight above, the high level set is subject to the constrains of the relevant from the values of relations corresponding to the current node. Nodes might be terminated from its parent node or it is a parent node itself. If the graph has no edges cross (sometimes called as a planar graph), it is then left to the user's determination (X or Y). However, if there are existing paths draw from each high level set, then the computation of percentage components relevant to each components set would be as follows:

$$\text{Adequacy}(C, G) = \left[1 - \frac{\text{Number of actually selected components}}{\text{Total number of relevant components identified}}\right] * 100$$

where C is a candidate component and G is a denoted high-level set or parent nodes or a group.

Research Method to Verify Usefulness

It was previously explained that the proposed research will describe the complexity for reusability of the system using complexity and structural measures. This research describes linkages among components, adequacy, and finally, issues of measurement, control, and maintenance. However, more is needed to evaluate the system. This research also proposes an evaluation of the system, as derived this researcher who will produce an ad-hoc report describing amount of time taken to understand components, the reusability or non-reusability of system components, and system procedures. Using Snooper (Esteva, 1995) which contains critical features as a basis for usefulness evaluation, the proposed approach will deal directly with practicality, reusability, and understandability, respectively. In this manner, results from the searcher system will be benchmarked against this researcher in terms of recognizing reusable components. In this manner, the usefulness of the searcher system approach will be validated and confirmed.

Reliability and Validity Procedures

Kochen (1984) stated that time has a major impact on human decision: "The value of an expert's time or that of a user's time is considered to be far more valuable than that of communication channels, computer memory or CPU time" (p. 354). Within a minute, sometimes even seconds, the human mind can change focus from one problem to another. This aspect of the human mind is extremely important to consider when a user needs an item of information. It is critical that the system responds in a reasonable time frame or the user's mind will change focus to some other concern.

Bearing this in mind, this researcher will pose a number of questions concerning the components that will be selected for evaluation. Components of three to four systems will be selected for evaluation. The effective number of selected components will be identified for each system. This researcher will record the amount of time it takes to retrieve the selected components for each system in order to determine if that amount of time is reasonable and compares favorably to the Guides Search system. In this way the reliability and validity of the Guides Search system can be ascertained. It is important to explain that reliability applies to a measure when similar results are obtained across situations. Reliability always refers to consistency throughout two or more measurements. Broadly defined, reliability is the degree to which measures are free from error and therefore yield consistent results (Daniel & Terrell, 1995; Devore, 1991; Zikmund, 1991). For example, ordinal-level measures and reliable if they consistently rank order variables in the same manner. The test-retest method such as the one that will

be conducted in the proposed study involves evaluating the same variables at two separate times which therefore tests for stability.

This researcher will rate library components using a Reusability Tally Sheet (Appendix B). Responses of this researcher will be compared to those derived from classification by Guides-Search, the proposed scheme and searcher system, as reusable or non-reusable and thereby ascertain validity of the Guides-Search system. According to the literature, reliability, although necessary for validity, is not in itself sufficient (Babbie, 1990; Daniel & Terrell, 1995). Validity addresses the problem of whether a system produces what it is supposed to produce and how valid the system is for the decision that users will make during its use. In other words, the question to decide is for what decisions this system is valid. For this reason, the following three criteria will be used by this researcher:

1. Whether or not there are too many answers to a question;
2. Whether or not there are too many questions; and
3. Whether or not a shortcut has been provided.

In summary, results that are derived from the scheme and searcher system in terms of recognizing reusable components in a timely and efficient manner will be compared against those found by this researcher. Results obtained from the Guides-Search system will be noted on a Recording Sheet (Appendix C) prepared by this researcher especially for documenting the information.

Chapter Summary

The purpose of this chapter was to describe the methodology of the study that will be used for classification purposes and for verifying the effectiveness of the scheme and search system, called the Guides Search. It was important to explain that the model is not an attempt to replace current structures; rather, it seeks to provide a conceptual and structural method to support improvement of software reuse methodology. It was noted that the methodology will employ two types of analysis:

> (1) identification of system components and classification of reusability or non-reusability by Guides-Search, the proposed scheme and searcher system; and
>
> (2) this researcher's evaluation of the same components in order to determine reusability or non-reusability and thus the reliability validity, and usefulness of the searcher system approach.

Research methods were first described, including formats and procedures incorporated in the proposed scheme and searcher system. Various techniques exist for presenting software reuse components. These include an indexing scheme, keyword-based systems, and knowledge-based systems It was explained that the Guides Search system incorporates a combination of all of these features.

Attributes of the classification scheme were also noted. It should, for example, include flexibility, extensibility and ease of use. A user should not be presented with a large number of questions nor be required to answer any questions known to be germane

to query. A user should not be given a large number of possible answers to any one single question nor be allowed to specify an answer not knowing exactly what question the searcher posed to elicit that answer.

Descriptions of the browser system, database, projected outcome, and resources to be used were presented in following sections. It was noted that this research will maintain a virtual library database. The local database contained on the PC and global databases will provide the classification of software mainly through the World Wide Web (WWW). For consistency in the study, local and global databases are to be considered at the same level.

It was also explained that a complete repository will not be maintained. The researcher provides indexing to other resources through the Web. However, database tables will exist to represent the classification schemes which consist of a data structure to the values of the relations. Also, to succeed, a prototype library system will need to be designed and developed in conjunction with simplified classification schemes. Microsoft Visual Foxpro v5.0, running on the Pentium-133 with 2.5 GHD and 16MRAM, could be used to build the prototype system. Additionally, there are software packages from well-known companies that can be used.

System measurement was the focus of the next portion of the chapter. For repositories to be useful, they must have a large number of components in all different areas to support the developer. But when too many are available, it is no longer possible for a single retrieval to find the specific components that a user needs. Given this challenge, the researcher introduces a flexible classification scheme that acknowledges

the value of expertise differences in users and coordinates that value by allowing interaction between users and system to locate precise components that are specified by the user. In addition, this study confirms that reusability is related to many variables. But how tightly or how loosely components are bound to others in their relations must be considered. This researcher asserts that bound variables and their component attributes are also correlated with the level of human expertise.

In the next section, this researcher's procedures for human intervention evaluation were described. According to Esteva (1995): "It is important to understand that even the most successful identification system will require human intervention when evaluating components for reusability" (p. 84). It was for this reason that this researcher desired to rate and evaluate components and compare his rating to those of Guides-Search to determine differences or similarities.

For evaluation purposes, three to four systems will be selected by this researcher. Size and number of components will be identified for each system. This researcher will evaluate each in accordance with criteria established by three critical features of the system. Results derived from the scheme and searcher system will be compared in terms of recognizing reusable components.

CHAPTER IV

IMPLEMENTATION OF THE GUIDES-SEARCH SYSTEM AND EVALUATION

Introduction

The first three chapters of the present research introduced the subject of concern and the study problem, reviewed the literature pertinent to the theoretical foundations and major variables of the searcher system, and described the methodology employed to implement the Guides Search system, as well as to collect the data. Included were four research statements to be investigated. It was noted that the purpose of the present research study was to provide a flexible system, comprised of a classification scheme and searcher system, entitled Guides-Search, in which processes can be retrieved by carrying out a structured dialogue with the user. The present study focused on the input and output process.

The purpose of this portion of the study is to present the implementation of the Guides System, analyze, and report the findings. The first section focuses on a description of the overall strategy of the implementation and the design method that was used. Details that comprise a basic understanding of the system were discussed in previous sections of the study. The searcher mechanism and browser system, for example, was described in depth in the third chapter. The concern at this point is to provide crucial details and some of the major particulars.

In the next section, research support information in the Guides-Search is presented. Global defined variables and local defined variables are explained. In the third

portion of the chapter, information file structures are detailed. Aspects of the Guides-Search System structure includes subsystem, topologically mapping relationships between components, and coupling. The subject of concern in the following section was evaluation of the system and verification of usefulness. It is here that tests that were used to verify usefulness are explained and the resulting data presented. Included are descriptive statistics of test results. Tables are provided for this purpose. A final section concludes the presentation and analysis.

Environment and Characteristics

The Guide-Search System was implemented using Visual Basic, MS SQL Sever 6.5 Evaluation version, and IIS 4.0 Beta version to manipulate the requests/problems-solution. The object-oriented method was applied because the characteristics of the Guide-Search are basically hierarchical. The Guide-Search is provided according to the fundamentals of object-oriented in the following manner:

1. Emphasis is on structuring a system around the relations objects manipulates.
2. Objects are described as instances of abstract data relations. The system knows from an interactive rather than system representation of such aspects as keywords.
3. The basic module unit describes a set of possible components of the same abstract data type or its relations.
4. Finally, structure reflects the relations in the form property-relation-components which provided the inheritance relations.

It is also important to mention that encapsulation is another significant feature of the Guide-Search system. As described, Guide-Search allows users to interact with the system to traverse any child node for specific problems-solution, once the user has entered the specified information.

Overloading may also be applied to the Guide-Search. Take, for example, methods to browse a set of relations' components. Each set's relation component in different phases may be different components. Levels of its relations that are presented in tree nodes are also considered. In order to browse its specified components, level of nodes may or may not be written in chunks. If the data can be fit into memory, the value of relations can be used to retrieve all the data in one operation. If their related components are in large, organized components, then they must written in chunks. In chunks, relations of the components are presented in directed graph form, as found in variety words-lexicon searches rather than a tree graph which knows the size needed and helps the system with respect to performance.

Reuse Support Information in the Guide-Search

In order to implement the Guide-Search, there are generic program developed to provide set of rules. They exist in addition to the reuse itself. Rules are basically categorized into two kinds: (a) those indicating properties/components to be inherited; and (b) those indicating properties/components to be rejected. Components are primary presented through users for classifying items into categories that are based on common characteristics. Rules, on the other hand, specify characteristics but allow the classification of components into more than one location in the scheme through its

relations. The components link activates the search in the classification scheme, which in turn contains search links throughout its components.

Unconditional rules

Regardless of any constraint and condition, this methodology is used in a leveraged manner. It is used for retrieval and adaptation in an ad-hoc fashion, browsing taxonomies and faceted views of reusable system components. These components belong or are related to the candidate properties defined in the classification scheme of the relations "is-a," "has," "is-in", and "like." That is, the classification scheme allows many relations for distinguishing components and can be copied directly with no tailoring in the browser mechanism. Rule are interpreted in pseudo code in the following manner:

```
At <components/properties>
        /* Establish the link of <components/properties> */
        For each component link to node
                Established components
        Loop
End unconditional rule
```

It is also important to discuss the components to be established in the Guide-Search system. These include:

```
Component SEI like EI(s) end SEI
Component DEI like EI(s) end DEI

Property Elliptic-integral
        is-a special
end Elliptic-integral

Property Elliptic-integral-first-kind
        is-a Elliptic-integral
end Elliptic-integral-first-kind
```

```
Property Elliptic-integral-second-kind
        is-a Elliptic-integral
End Elliptic-integral-second-kind

Property Elliptic-integral-third-kind
        is-a Elliptic-integral
end Elliptic-integral-third-kind

Property Number-R
        is-a Number
        has datatype Real
end Number-R
```

The key concept of this approach is to translate the most relevant components, based on the classification scheme that is defined and translated in the natural way without any conditions.

Conditional Rules

The condition-rules are established to interpreted components that by themselves can not be executed or complete. They must depend on other properties - that is, components that must inheritance properties from other. The pseudo-code and example components described in this catalog are designed using recursive algorithm. It may simply be described in the following manner:

```
At <components/properties>
        /* Establish the link of <components/properties> */
        For each component link to node
                Established components
                        If  inherited from others
                                Inherit Except, Part-of, need, etc...
                        End-if
        Loop
End conditional rule
```

The condition which depends on all the possible inheritance components can be specified. These are included from the classification scheme and are specified by the users from the accept/reject feature. Components to be established such that

```
Low-level component INITDS
        like INITS
        except has-precision double
        is-in C:\file:INITDS
                needs DERM1
end INITDS

Property Number-I
        like Number-R
        except has-datatype Integer
end Number-I

Property Number-C
        like Number-R
        except has-datatype Complex
end Number-C

Low-level component CSEVL
        has-precision single
        is-in C:\File:CSEVL
        needs (SERM1, IERM1)
end CSEVL
```

Global Defined Variables and Local Defined Variables

In order to implement the Guide-Search, global defined variables and local-defined variables must be included. Global defined variables are stored in the cursor tables and shared by all the selection components. It is assumed that there is enough memory space to store these components. In other case, one can simply defined a cursor table and manages on disk if needed. The concept in this case is that, for any given query, the components will be checked from the cursor table based on the most relevant and flexible components for reuse. They are organized in the frame-base. In this study,

related-class" was simply described in the table where inner-join query was used to retrieve available components in the cursor tables before going out to physical database.

The global variables defined in this case were used in the present study during the life of the search components. While global defined variables components continue to be stored on the cursor table until users reset or re-query, local variables defined use only during the group selection. During each group selection, components are captured from the relevant properties related to the relation and mapping against global variables defined. Where components are found in the global defined, it may simply be dropped. When components are not found in the global defined, the system then captures into global defined for later reuse in the form of design decisions. However, the local variables defined task differs from the global defined. It builds from scratch for each new group or node that is found.

Components that are created from the scratch or by an abstraction process from the specific global variables defined are used to help the browser mechanism reduce over-processing which in turn helps system performance. The idea here is to transform the solution if found in the global area into potential solution problems components associated with particular components that are specified by the user. In other words, global variables defined can be described as an index to the domain-problems. Local variables defined can be presented in terms of specific problems/solution specified by the user. The more information users specify, the deeper in detail or closer to exact problems can be found. In other words, in this level the available assets are specific components or components that can specified close to exact problems one has specified. For the Guide-

Search, local variables defined reference to when users first select "equation." Results found in Fortran77 and GAMS package software include:

- Kind
- Linearity
- Determinacy
- Constraints

Upon users selected "kind" equation, the critical query carries out the results and contains:

Has Kind Differential
Has Kind Integral
Has Kind Difference...

Components "Function has kind integral" can then be defined as a global and be placed in the cursor table with its relevant components referenced in GAMS.

Bessel integral
Complete elliptic integral
Error integral
Exponential integrals
Mathematical Functions
Sine integral
Volterra integral...

These components also reference its location in such away that Bessel integral "is-in" GAMS at C10f . The C10f is a location where component document stored. This location could be in C:\ , networks, or on the internet at a specified www (World Wide Web) location.

Figure 1 on the following page presents this structure in a tree. Assume that nodes A, B, C and D are among selected components. If node B was chosen, the query then carries out the combination AB to perform the search. That is, at node A, A is in query. At node B, AB are used in the respective query. AT node C, ABC are in the

query and so forth. Level components are derived from the overall Guide-Search system. In order to facilitate reuse, the components available for selection should meet the classification scheme that has been defined. Without such consideration, many potential components for reuse could not be eliminated from reuse because they could not fit in any component classes. In this situation, rule base functionality helps to capture and allocate one or more component classes elements as well as alternative and optional constraints.

The present study developed a search option that allows the user to be more specific and precise with regard to his or her problem-solutions query. It may be described in the following manner.

1. The Boolean operators includes conjunctive terms included: AND, OR operators. "AND" operator which requires that all components must be presented. In using an "OR" operator, it is simply required that at least one component be satisfied in the respective request.

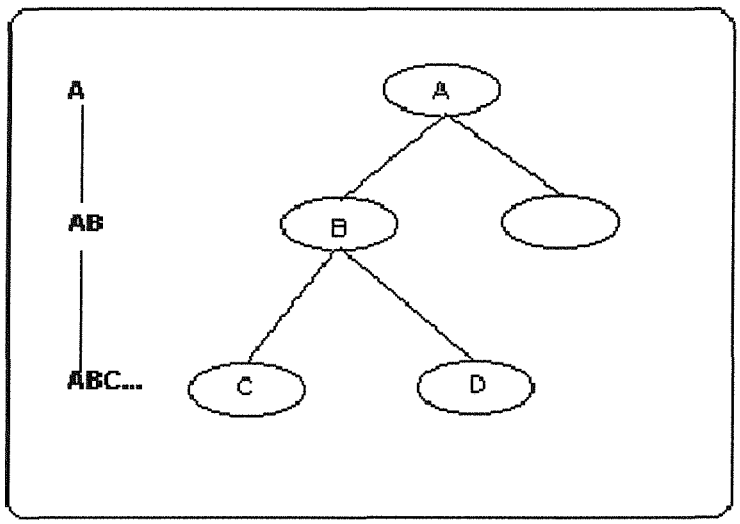


Figure 4.1: Tree Structure

Components

Component AND Component
Component OR Component
Component AND Component AND Component
Component OR Component OR Component
Component AND Component OR Component (process in format of A * B + C)
Component OR Component AND Component (process in format of A + B * C)

Wildcard search

Example if a Fortran77 files search: was performed:

Function has-operand equation-ODIN2
AND
Function has-precision double
Results
Function SIVA
Function IVAS...

2. A wide-card search is also allowed. In this case, users can search for entity database or include all components from the current states. In the screens following, one can simply search without specified any constraints.
3. Block constraints allow queries to automatically be reconstructed. In other words, inner levels within block search will be based on the term or condition which are relevant to the block. For instance, as results when searching for "what is functions has determine" when components in GAMS-package are specified:

- Linear equations

When the constraints "has-least squares solution" is specified, this is the following results

DBOCLS Solves the general linearly constrained linear least squares problems.

DBOLS Solves linear least squares problems with simple bounds on the variables.

LPDP Solves least projected distance problem.

LSEI Solves linearly constrained least squares problem with equality and inequality constraints.

WNNLS Solves linearly constrained non-negative least squares problem.

Block constraint is a most important feature of the Guide-Search System. Block constraint is used to present a measurable strength or weakness of gathering components in the system. Blocks represent only a small percentage of available components in the database. The corresponding blocks are given in the format of outer loop and inner loop routines. The algorithm is given as follows:

Component "c" of a given block B with a node N is defined by c(B(N)) and is defined recursively:

(a) Initial Block

(b) Fetch relevance components into the current state.

(c) Node defined. If the node N' corresponded to N and has N's properties, but N has no immediate descendant, then N' is started as a new block.

(d) Otherwise, for each immediate descendant i of N, set h(i) and edges ordered according to the order of the immediate descendants in the block B.

Block size is gathered from the following data collection sizes: 450 components, 1100 components and 750 components. Normalization percentage for any given components given by:

$$C(G) = [1-C/G] * 100$$

Note also that components are presented through a frame-based - that is, a component that a user does not qualify or specify in the search. These components are built in classification scheme so that it provides the opportunities for users to retrieve all components associated, even those that the user has not specified. It is important to explain that component presented in frame-based do not effect the mechanisms search. This occurs primarily because they are mostly presented through the relation "is-a" and "need."

Information File Structures.

The following discussion provides a description of the various aspects of the Guide-Search System Structures. These include:

(a) The subsystem, which further decomposed components in large system;

(b) topologically mapping of the relationships between components; and

(c) coupling, which analyzes the binding strength between components.

In the Guide-Search system, a Relation Database Management System (RDMS) is used because it has a strong mathematical basis. It is important to explain that RDMS allows users to demand a solution from specified problems. A major feature of the system is to express each components in the class which uses SQL statements. Only particular

critical tables used in Guide-Search System are described in the current analysis. Figure 4.1 illustrates database tables as found in category packages.

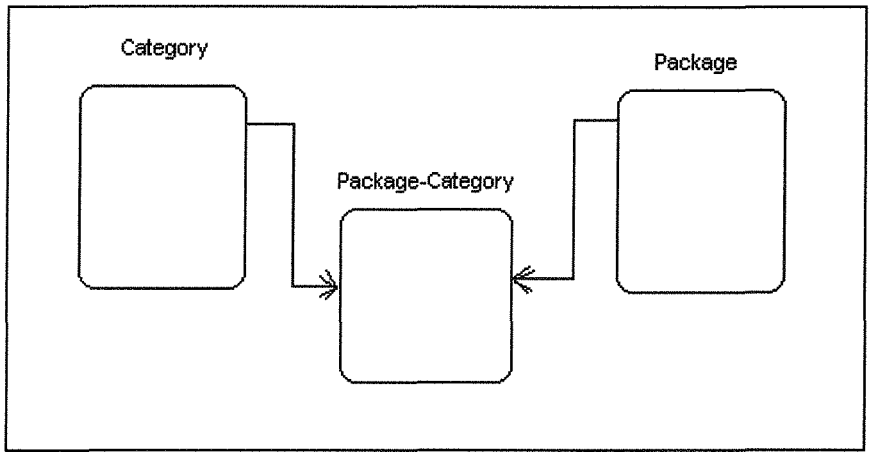


Figure 4.2: Database Tables in Category Packages

Additional portions from the Package-Domain, Package-Elements, as defined in Chapter 3, are category tables. Components include:

(a) Arithmetics
(b) Number theory
(c) Elementary and special functions
(d) Linear algebra
(e) Interpolation...

Categories are used to reference mathematical routine within a specific category defined in GAMS, Mlab, and others. A single letter presents a key-field each category. It is used to reference a mathematical routine within mathematical database.

Package table components include:

GAMS	http://gams.nist.gov
AMS	http://e-math.ams.org
MathPro	http://sashimi.wwa.com/math/MathPro.html
Mlab	http://software-guide.com/cdprod1/swhrec/011/708.shtml
Fortran77...	C:\Fortran77

Package tables are used to present each major Mathematical Providers and their locations.

The Category-Package-Table is used to hold multiple relationships between Category and Package tables and to present a "decision tree." Itis also used to hold all subclasses defined in GAMS, Mlab, MathPro and a wide variety of others applications. It contains a sequence number which is used as a unique constraint and in the present case is called: Seq-ID, Package_ID (Parent) and its categories (items).

The table used for this purpose can be presented as follows:

Seq-ID	Parent	Items
1	GAMS	A
2	GAMS	B
3	MLab	A
4	MathPro	A
5	1	Integer
6	1	Rational
7	1	Real
8	1	Complex...

Seq-ID is used to define the distinct components in each package which is associated to its category. In GAMS, single letter used to reference mathematical routines included A thru Z. Where each major category is divided into its sub-categories such that A represented "Arithmetic, error analysis," a "1" is used to identify "A." In turn, Arithmetic, the error analysis category, is divided into many sub-categories which have the following sub-categories:

Integer has Key-ID 5 and belongs to parent denoted "1"
Rational has Key-ID 6 and belongs to parent denoted "1" and so on.

The same is true for the Components tables (component-relations-property). Elements stored within this table include:

SIVA	has-function	solve
SIVA	has-operand	equation-ODIN2
SIVA	has-output	per-step time-intervals G-stop
SIVA	has-precision	double
SIVA	is-in	C:\Fortran77\SIVA
SIVA	needs	R1MACH
SIVA	needs	SASUM
SIVA	like	IVAS...

The present study applied G-node to describe how each individual classes linked together. Links specifications included both data and direction. This architecture is used to join a specific existing component in term of relationships between each class to another from the mapping, ordering, and searching phases. These phases are subsequently described in individual subsections listed below.

The Mapping Phase

The method used in mapping is to travel through all edges using depth-first methodology of all components in G as an algorithm:

- Split components into a directory part, called x and the rest called y.
- Move components into cursor table where all the pattern rules one of whole targets match x or y.
- If any rule can not be applied, remove components from the list.

In using this technique, it was found filtering technique was best.

The Ordering Phase

In the Ordering Phase, components are presented in the lexicon ordered. Edges are labeled by a letter which represents the relevance to components. Each component in the lexicon that is ordered corresponds to the inheritance from the G-node. For two components sharing an initial path from their initial G-node end ending, if applied, in an "is-a" relation, it is called a terminate node. The height of the node that is called v, is denoted h(v) and is defined with respect to the ordering of the G-node. At each node, components that are not called will be assigned the number 0, and at each subsequent ordering, the assigned number is increase by a value of 1. In other words, the components inheritance by level 0 will appear at level 1; components inheritance from level 0 and level 1 will appear at level 2, and so forth, as indicated in Figure 3.

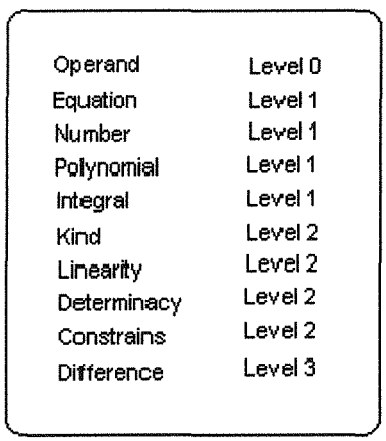


Figure 4.3: Ordering

The Searching Phase

Automated functions to support extract components from its query to generate components to the users is accomplished in the Searching Phase. There is also a case

where there is a set of components sharing an alternate. It the presented to users for more specific selection. Upon selection by the user, the set of components that is eliminated will not be used for further consideration until a new search is undertaken. Thus, this activity is further decomposed into lower-level functions.

Functions provide a set of analysis capabilities for Guide-Search system. This approach also avoids the duplication of components. The Guide-Search system also allows dynamic set through SQL functions. They can be outlined in the following manner. To decide the set of relations to display, the entries for values of "r" in the symbol table for the A(rb) data structure include the number of components in which that value of "r" is used. The algorithm is described as follows:

- The symbol table is sorted into descending order according to this number, For a given value for this number it sorted into alphabetic order according to the value of "r".
- If there exist values of "r," initial only those "r" values.

-Otherwise, the system allows users to view all selected components, only if there are no existing value of "r."

- When the user selects a value of "r", the associated values of "b" by using "r" to access the first elements of the parse representation of the (r b) matrix for any given value of "r."

The list then traverse in the direction of constant "r" and increasing "b"

- When is value "b" selected, column (r b) matrix to traverse until the given value of "r" is discovered.

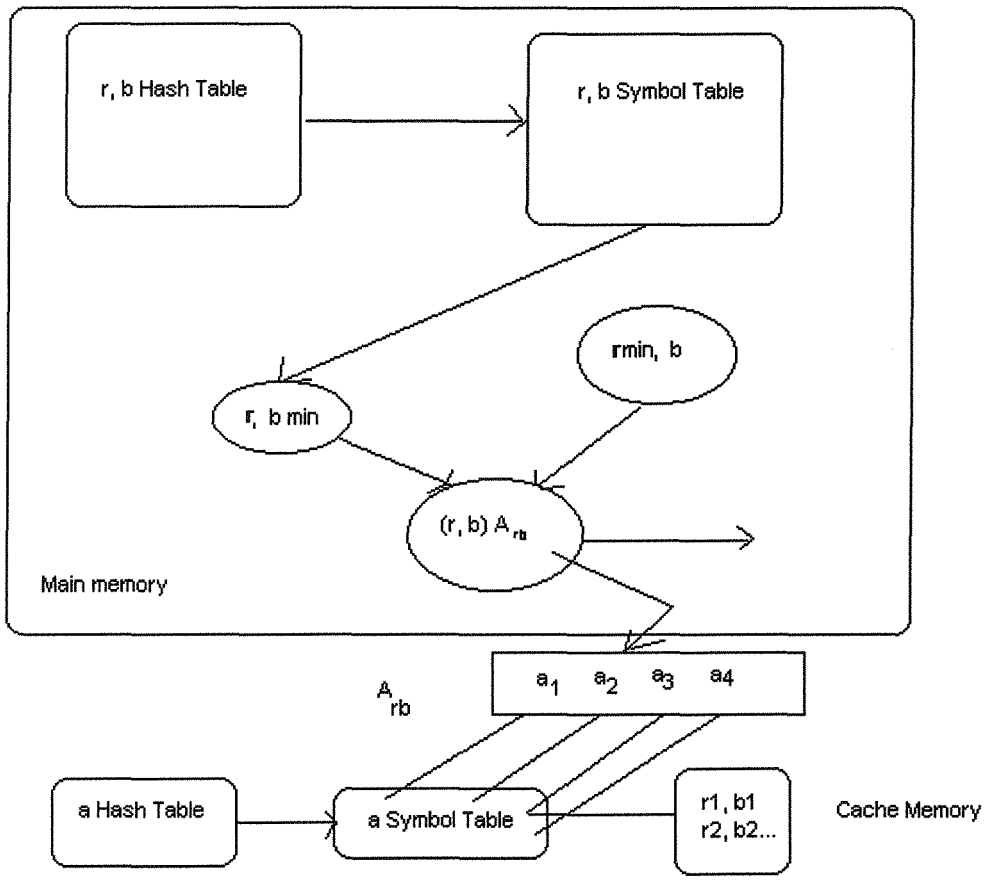
- To reduce the amount of data transfer between cursor tables and physiscal tables. Filtering techniques was applied.
- The fragment of the orginal RB(a) data structure that corresponds to the retrieved components constitute a new instance of a RB(a) data structure. Since, the RB(a) data structure contains the (r b) pairs that correspond to each a.

Rationals are provide to facilitate components selection among reusable components. For example, the first 10 selection components are used to display back to users while the rest continue to be retrieved in the backend.

Building the Browser Components

After the components are elected, they undergo construction in the organization structure from Searcher-mechanism. Once the components are determined,

Figure 4.4: Data Structures to Support Searcher



they are ready to populated with information to users. The components selected for users are based on characteristics of their respective classes. The rules based are used to compare and ensure their best fit.

It is important to eliminate a duplicate node. The present study first attempt was to define type equality of the properties associated with the corresponding nodes and the order components associated to its relations. However, because components are in space direction, the study only considered backward chaining, which points to one level direction to each relations-components relationship.

Evaluation of the System

It was noted in the literature, that in order for repositories be useful, they must have a large number of components in all different areas to support the developer (Henninger, 1996; Esteva, 1995). However, when there are many components available, it is no longer possible for a single retrieval to find the specific components a user needs. Given this challenge, this researcher introduced a flexible classification scheme which acknowledges the value of expertise differences in users and coordinates that value by allowing interaction between users and the system for the purpose of locating specific components that are specified by the user.

It was previously explained that the purpose of the present study was to describe the complexity for reusability of the system using complexity and structural measures. Although the research described linkages among components, adequacy, and issues of measurement, control, and maintenance, more was needed to evaluate the system. For this reason an evaluation was proposed as derived from this researcher in the form of ad-hoc

reports describing the amount of time it was taken to understand components, the reusability or non-reusability of system components, and system procedures. Using Snooper (Esteva, 1995) which contains critical features as a basis for usefulness evaluation, the current approach dealt directly with practicality, reusability, and understandability of the system.

This researcher first posed a number of questions concerning the components to be selected for evaluation. Components of three to four systems were selected for evaluation. The effective numbers of selected components were identified for each system. The amount of time it took to retrieve the selected components for each system was recorded in order to determine if that amount of time was reasonable and compared favorably to the Guides Search system. In this way the reliability and validity of the Guides Search system could ascertained.

Results from the search system were benchmarked by this researcher in terms of recognizing reusable components, thus validating and confirming the usefulness of the searcher system approach. It is important to explain that there are three common steps to evaluating a system. These were used in the present analysis for evaluative purposes. The first step included posting a problem which related to the user-interface. Utilization of the Searcher mechanism to gather available components was the second step. The third and final step was to initiate the browser mechanism. This provided a mechanism for the user to specify his or her needs. These three functions provide most users with the tools needed to gather problem solutions.

User Interface

Before the data can analyzed, it is first necessary to describe the user interface that was employed in the present study. It was previously noted that a successful system goes beyond basic concepts in its definition of user friendliness. Designing the best user-interface system is also a requirement for success. The user-interface mechanism represents results in a tree structure dialogue using relations and can be used to generate graphical information. Graphics indicate in picture form the relationships among components. In the current development, the study used the combobox, listbox, and buttons command. Components were automatically selected from Browser mechanism. Figure 5 reveals the fan-ins and fan-outs of each component that was selected. The listbox was used to display selected components. Consider the following example when the user is asking for Function has Operand Number-R. The results included list in the listbox, as indicated in the following figure define:

Component ERF
Component Inverse-Hyperbolic
Component SASINH
Component DASINH...

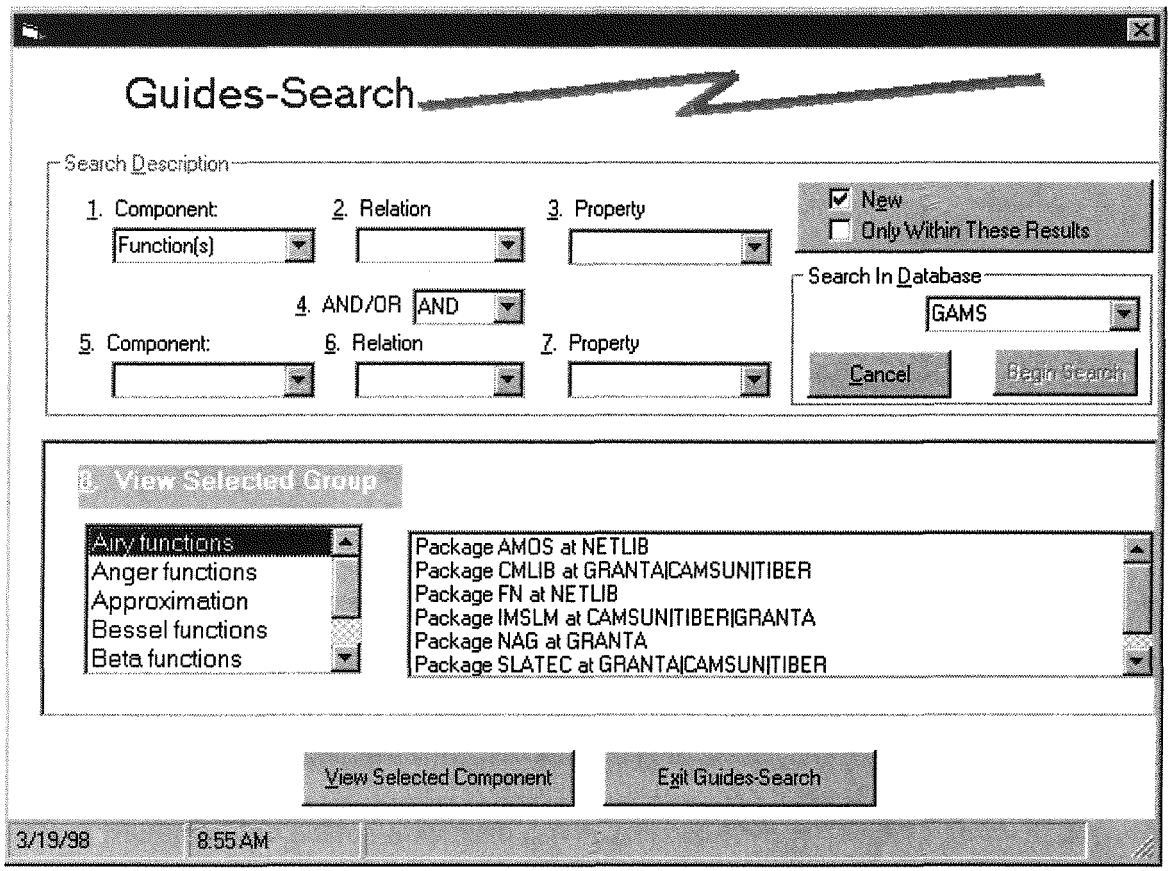
Components included:

Function
Operand
Output...

Relations included:

has-function
has-operand
has-precision

Figure 4.5: Searcher Screen



has-output
is-in
like
except...

Properties included:

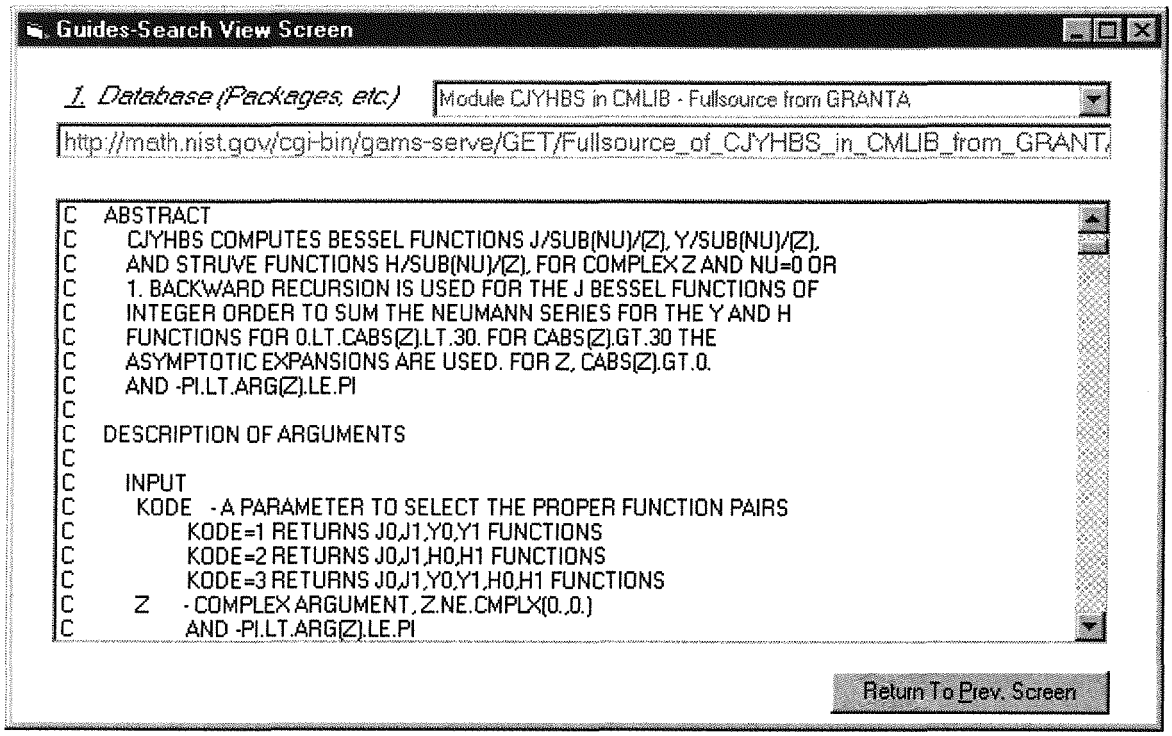
Solve
Equation-ODIN2
Per-step time-interval G-stop
Double
Single
C:\fortran77...

Logical AND or OR but not both.

A search description displays the actual query will be used (see Figure 6). Consider the following example when the user performs selects in the Component-Combobox "Function," selects in the Relations-Combobox a value "has-operand," and selects in the Properties-Combobox a value "equation-ODIN2." When the Search-Button is clicked, the system first reads a query-statement in search-Textbox and performs its search routines. The Search-Statement can be reached such as "Select * from Components-Table where properties = 'equation-ODIN2.'"

Results carried out from the query are described as follows. It first involved the function Mapping and Ordering phases, as described in previous sections. This placed the results-elements in a set , together with all their associated relevance components and nodes. Results were then presented back to the user. In the majority of cases, the results are presented back to the user in the Listbox. The user simply views them. In other cases, however, results are dependent upon the number of nodes. When the query provides results larger than three levels of nodes, the system brings back to the user not just the

Figure 4.6: View Components Screen



results placed in the Listbox, but also results in a pop-up window form which identifies more specific options. At the time the user is required to provide more specific values. The system uses the Filter-Technique in the Database Functions to filter results. Each time it sends the results back to the Mapping and Ordering phases and presents back to the user increasingly more specific solutions. For any given task, the user has a variety of option. The user can re-select, back, clear results, or simply specify a keyword to be matched on the current results.

Data Presentation and Analysis

As a test result, it was found that even for a first result returned to users, a question was generated for more specific components. It was found that the system terminated a great number of components from which the user would ordinarily have to browse for the necessary results. Consider the following query example:

Function has-function solve

The results that were returned included 152 functions out of a possible 2,300 functions included in the database. When the system responded to a more specific inclusion, "has-language Fortran77," the results significantly changed. Specifically, the system returned with 21 functions for the "has-function-solve" and "has-language Fortran77."

Results were recorded for three specific questions: What is a function; What is an Operand; and What is the language. These data are presented in Table 4.1. Percentage decreases for each question are included. As indicated, the first result for the question as to what is a function produced 2,300. The second, however, was reduced to 301 (a 76.9

Table 4.1: Query Results to Specific Questions

Question	First Results	Second Results	Third Results
What is a function?	2,300	301	3
Percentage Decrease		76.9	99.7
What is an Operand?	1,012	37	1
Percentage Decrease		96.4	99.9
What is the language?	2,300	450	6
Percentage Decrease		80.4	99.6

percent decrease). The third result reduced the number retrieved to 3. Similar results were obtained for the second and third question. For the question as to what was an operand, the first result produced 1,012 responses. A decrease of 96.4 percent was realized for the second result, which was then reduced to one. For the question as to what is the language, the first results produced 2,300 responses, which were reduced to 450 for the second result. The third time, only 6 were retrieved.

It is important to explain that relations were represented as a directed graph rather than a tree for the purpose of reducing size. For any given component, different finite state were recognized. These are described as follows:

- Nodes in the graph are the states of the finite-state machine;
- Edges of the components are the transitions of the machine; and
- Terminal nodes are the accepting states.

In other words, each relation is represented as a tree node where each corresponding components that share the same properties are joined. Thus, a path is created from an initial components to the newly found components.

In the present study, queries were run on the program database SQL-Server 6.5 with Visual Basic which was operating in a Window environment. The Searcher retrieved from the 450 functions database where the browser mechanism presented 132 nodes for users selection from a 2,300 functions database. Once components were selected, the view function then obtained the "file" or location and attributes of the selected function. This was followed by a retrieval of the text from the source files. The

time stamp on each function was used to provide useful information for evaluation, specifically in term of performance issues. Units were measured in mini-seconds.

Tables 4.2, 4.3, 4.4, and 4.5 on the following pages present the information gathered from testing by this researcher to verify the usefulness of the Guides Searcher system. Testing data was collected in real time, from testing by users, and from testing using the Guides Searcher System. All data are presented in mini-seconds, which was the unit of measurement as previously noted.

Table 4.2 presents the data regarding the number of times the user must be interactive, the searcher time, and the browser time. The data are presented on 450 functions for three measures. These included: Real Time, User Time, and System Time. As indicated in the table, in real time the user was required to be interactive 120 times, while the user interacted 54 times and the system interacted 89 times. Similar information is contained in the next two tables, but for 1,100 functions and for 2,400 functions.

Table 4.5 presents the findings for the number of functions counted (450), the nodes presented and found, the number of unique components identified, the number of cyclic found, and the number of regular components. Data is separated in subsequent tables by the number of functions assessed. These included: 450, 1,100, and 2,300. As indicated, 130 nodes were presented and found for 450 functions. This number increased to 176 for 2,300 functions. A total of 1,873 components were identified for 450 functions.

A significant increase was realized for 2,300 functions. Specifically, 8,920 components were found. Unique component identification also increased with the

increase in the number of functions. For 450 functions, 210 unique components were found, but for

Table 4.2:

Interaction, Search, and Browser Times for Three Measures (450 Functions)

Measure	Number Times User Must be Interactive (in Mini-seconds)	Searcher Time (in Mini-seconds)	Browser Time (in Mini-seconds)
Real Time	120	15	145
User Time	54	9	52
System Time	89	21	69

Table 4.3

Components Classified in Real Time, By User Time, and By System Time (1100 Functions)

Measure	Number Times User Must be Interactive (in Mini-seconds)	Searcher Time (in Mini-seconds)	Browser Time (in Mini-seconds)
Real Time	230	21	290
User Time	150	19	180
System Time	210	30	150

Table 4.4

Components Classified in Real Time, By User Time, and By System Time (2,300 Functions)

Measure	Number Times User Must be Interactive (in Mini-seconds)	Searcher Time (in Mini-seconds)	Browser Time (in Mini-seconds)
Real Time	250	27	310
User Time	230	21	195
System Time	250	30	180

Table 4.5

Findings for Functions Counted, Nodes Found, Unique Components Identified, Number of Cyclic Found, and Number of Components

No. Functions Counted	Nodes Presented and Found	No. Unique components	No. Cyclic Found	Number of Components
450	130	210	36	1,873
1,100	145	325	29	4,717
2,300	176	392	17	8,920

2,300 functions, a total of 392 unique components derived. This represented a 176.6 percent increase.

Discussion of Findings

From the findings it became clear that there was a significant advantage to using the Guides Searcher system. Evaluation of the system, although run on a very small scale by this researcher, provided preliminary results that verified the usefulness of the model in general and the system, specifically. As previously noted, the search for ways to improve the software development process has led many organizations to pursue the substantial benefits available through software reuse. According to the literature previously reviewed, design reuse is emerging as a powerful and essential tool for dealing with increasing complexity. As noted by Yoelle, Maarek, Berry and Kaiser (1991), among other authorities, software reuse is widely believed to be a promising means for improving software productivity and reliability. However, it is only through application of searcher systems such as the one developed in the present study that the benefits can be realized.

The presentation and analysis clearly indicated that there was direct correlation between system performance and search critical. That is, the package with the most options yielded the most complexity and therefore it represented the worst case in term of system performance. An alternate way that was suggested was to break a query into numerous times search rather than place all in a single search statement.

When taking performance issues in consideration, the user was able to create a new node for each search. All nodes that were found were placed with their

corresponding relations which were found in memory. Thus, a cursor table in double link-list layer was built. Each node that was found was then stored in this cursor table. Therefore, for any given node and its corresponding components, it need only be found once. For each search, the study first performed all possible components related to a given relation. A check was performed to determine if those relations corresponded to components already found in the cursor table. If this was true, components corresponding to each relations-node with their structure (presented in tree) were gathered. Another factor is also crucial to system performance - that of terminate nodes. When relations have only one component that corresponds to the relation, results are masked as a terminate node. Thus there are no further requirements for a search.

The present findings have significant and relevance to complexity theory in general in that the searcher system provided a valid methodological tool for discovering software for reuse, and thereby reduce complexity. Complexity theory related to the subject of the study because it impacts the ability to reuse. Complexity has been and continues to be a realm that is difficult to define and even harder to understand because it deals with the aggregate of many simple things that can create complex forms (Goering, 1995; Kochen, 1984). Complexity theory is actually the study of how much computing is required to solve various kinds of problems, especially those related to large software systems (Devanbu, Brachman, Selfridge, & Ballard, 1991). It deals with systems as a whole. Researchers often create computer simulations of extremely intricate systems, then use those computer programs to develop hypotheses that can later be tested with experiments. A natural measure of complexity is the entropy rate of a random process

that models the problem. Reduction of complexity was the focus on the study. The literature agrees that it is possible to decrease complexity by carefully analyzing components into sub-components and applying the black-box approach. This is the approach this researcher used in developing the Guides Searcher system and its usefulness in this respect was verified.

It was previous noted in the literature that, for software reuse to be successful, there are critical factors which software reuse systems development must take into account in designs and developments. These were described as follows:

(1) The classification scheme should include the following attributes: flexibility, extensibility, and ease of use;

(2) Users should not be presented with a large number of questions or be required to answer any questions known to be germane to query;

(3) Users should not be given a large number of possible answers to any one single question; and

(4) Users should be allowed to specify an answer not knowing exactly what question the searcher posed to elicit that answer.

In the present research, a model of software reuse which would satisfy these factors was explored. Findings also verified the fact that such a model could be developed and its usefulness verified.

Chapter Conclusion

The purpose of this portion of the study was to present the implementation of the Guides System, analyze, and report the findings. The first section focused on a

description of the overall strategy of the implementation and the design method that was used. The concern was provide crucial details and some of the major particulars. In the next section, research support information in the Guides-Search was presented. Global defined variables and local defined variables were explained. In the third portion of the chapter, information file structures were detailed, followed by an evaluation of the system and verification of usefulness. It is here that tests that were used to verify usefulness were explained and the resulting data presented. Included were descriptive statistics of test results.

To classify processes, the present investigation focused on the provision of a mathematical method derived from Relation theory. It assumed that the model for a flexible classification system (generalization of the use of facets) could be developed for semi-mathematical software reuse and classification. It was believed that the overall approach to the reusable software methodology may turn out to be the most important contribution of the research, which is to make discovery of a classification more reliable and less tedious. This researcher believes this goal has been achieved. The model was developed which appeared to be more reliable and less tedious. Its usefulness was verified through a small, mini-test.

From the review of literature presented in previous portions of this study, it became that technology is moving closer to reality in natural language translaters. A natural language technique can be applied to gather a classification scheme automatically. Libraries only need to keep all mathematics functions in the document format and the system will provide the associated scheme.

The need for a dictionary table was found to be critical for the components search. In other words, a dictionary of all relevant terms related to mathematics functions is needed. The present study continued to develop the system throughout the thesis. The goal was to have a semantic and syntactic search related to natural language which could automatically build the classification scheme. Although findings in the present study verified the usefulness of the Guides Search system, when the new classification scheme is introduced to the real-world system, it will first be required to pass a major audit. A future goal is to place the Guide-Search system on a public www, where users can enjoy its many benefits. Another goal is to develop an updated version to provide visualization levels of components as, for example, in 3D graph format.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

Introduction

Previous portions of the research presented modular components of the study. This chapter combines previous modules into a unified whole, summarizing the research, discussing the model that was implemented and the data that have been presented, drawing conclusions from the data analysis and literature review, and providing recommendations. Recommendations focus on suggestions for future investigative studies of a similar nature, as well as on areas of concern deemed important in the light of the findings of this study. The following subsections provide this information.

Summary of the Study

Most discussions of software reuse focus on mechanisms to construct reusable software. For reuse to be successful, however, there must not only be a large collection of useful, reliable parts available, but also a mechanism to discover components that meet a specified need. Software reuse should not be practiced in environments where it costs more to discover components that meet a specified need than to invent them anew. The purpose of the present study was to describe a method to classify software components. Of secondary, but equal importance, was to develop a system to use such a classification efficiently to discover software components that meet specified needs.

Specifically, the purpose of the present research study was to provide a flexible system, comprised of a classification scheme and searcher system, entitled Guides-Search, in which processes can be retrieved by carrying out a structured dialogue with

the user. The classification scheme provides both the structure of questions to be posed to the user, and the set of possible answers to each question. The model did not attempt to replace current structures. Rather, it sought to provide a conceptual and structural method to support the improvement of software reuse methodology.

The study focused on the following goals and objectives for the classification scheme and searcher system: (1) The classification must be flexible and extensible, but usable by the searcher; (2) users cannot be presented with a large number of questions; the user cannot be required to answer a question not known to be germane to the query; (3) users cannot be presented with a large number of possible answers to any single question; and (4) users are allowed to specify an answer, even though the user does not know exactly what question the searcher will pose to elicit that answer. (This is similar to a key word search.)

The literature pertinent to the background of computing was reviewed, followed by an examination of reuse of software components, design, and programs. It was explained that design patterns - templates that provide developers with guidelines for solving problems - like object-oriented software have promised potential techniques for software reuse. Data abstraction and complexity reduction were also reviewed. It was clear that there were many motivational factors to using object-oriented concepts such as trace-ability improvement, reduction of integration problems, improvement of process and product, ability to hid information, abstraction of data, encapsulation, and concurrency.

Also reviewed were relations, frames, propositional logic, and constraint satisfaction. Included were explanations of the algebra of notations, the algebra of sets,

regulations, functions, and Euler-Venn diagrams. The components of expert systems were described in detail. This type of system is characterized by its method of logical deduction from stored data in accordance with rules independent of the program while conducting the search strategy. Current expert systems use a pseudo-natural dialogue through graphical user-interfaces to communicate. Current and future research is moving in the direction of development of full natural-language interfaces which use a syntax that is close to the user's native language.

The review was concluded with an examination of Function Based Encryption (FBE) systems which use a specialized mathematical function and a secondary function set to manipulate date in a complex manner. This was important because the present study focused on the input and output process.

The methodology that was used for classification purposes and for verifying the effectiveness of the scheme and searcher system was described in detail. Explanations were provided of the user interface for system communication purposes, the searcher function and mechanism, searcher-system roles, the database, and relations used by the searcher system. It was noted that, in addition to the classification schemes described, the system supports users in defining the relations link from the classification schemes to their documents. In this context, documents referred to a collection of source-code posted as a file on the World Wide Web or in the local machine. For efficient searching, however, classification-schemes in the present study were defined and stored in the local machine.

The overall strategy of the implementation and design method that was used for the Guides Searcher system was also described in depth. Included were reuse support

information in the Guides Search, such as unconditional and conditional rules, globally defined and locally defined variables, and information file structures (ordering and searching phases). It was first explained that the object-oriented method was applied because the characteristics of the Guides Search were basically hierarchical. The Guides Search was provided in accordance to the fundamentals of object orientation: emphasis was on structuring the system around the relations objects manipulations; the system gained knowledge from an interactive rather than a system representation of such aspects as keywords; and structure reflected the relations in the form of property-relation-components which provided the inheritance relations.

Answers to Research Statements

Four research statements were outlined at the beginning of the research. Each statement is reiterated below. Each is followed by an answer as derived from the review and implementation of the model.

1. A comprehensive review of related literature will indicate that existing techniques are inadequate in supporting information requirements.

The review of literature indicated that existing techniques are currently inadequate in supporting information requirements. Baker and Kauffman (1991), for example, concluded that few companies know what programs are in their current inventory; even less have solid productivity measurement systems in place to monitor systems development efforts in supporting information requirements. Booch (1994) suggested that, to overcome the problem of inadequacy, the discipline of object-oriented technology will soon give rise to a marketplace of reusable software components that can be

assembled into robust and scaleable software solutions. According to Due (1995), techniques designed to promote code reuse are sound; the problem has been with implementation and support. In this respect, Maarek, Berry, and Kaiser (1991) commented, "Although software reuse presents clear advantages for programmer productivity and code reliability, it is not practiced enough. One of the reasons...is the lack of software libraries that facilitate the actual locating and understanding of reusable components" (p. 800).

Poulin and Werkman (1995) agreed, adding that reusable software libraries suffer from poor interfaces, too many formal standards, high levels of training required for their use, and a high cost to build and maintain. Their study used a structured abstract of reusable components. Structured abstracts provided them with a natural, easy to use way for developers to search for components, quickly assess the component for us, and submit components to the reusable software library.

2. There is a significant need for a new approach or method to classify software components and a system to use such a classification efficiently to discover software components that meet a specified need.

Review of the literature clearly documented that there was a significant need for a new approach or method to classify software components and a system to use such a classification efficiently (Brian, 1992; Chauvet, 1995; Freitag (1994); Novak, 1991; Novak, Member, Hill, Wan, & Sayrs, 1992; Prieto-Diaz, 1987, 1991; Ray, 1992). The reuse of software as an important aspect of controlling and reducing software costs and improving quality has also been documented in the literature (Humphrey, 1990; Marlin,

1995; Prieto-Diaz, 1993). According to Novak (1991), a significant barrier to the reuse of software has been the rigid interface presented by a subroutine. For nontrivial data structures, it is unlikely that the existing form of the data of an application will match the requirements of a separately written subroutine.

A new flexible approach was introduced and implemented in the present study. It was called Guides Search system. The system's ability to discover and identify components that met a specified need was verified though testing. By using the Guides Search, processes were retrieved by carrying out a structured dialogue. The classification scheme provided both the structure of questions to be posed to the user and the set of possible answers to each question. In this manner the Guides system provided a conceptual and structural method to support the improvement of software reuse methodology.

3. Design of a searcher software system used to discover software needs will address the following three concerns: (a) it will allow users to retrieve the desired software without being required to answer an inordinate number of questions; (b) it will present an adequate number of possible answers but not too many to any one question; and (c) it will not artificially restrict the performance of an expert user.

Implementation of the Guides Searcher software system addressed each of the concerns listed above. For example, it allowed this user to retrieve the desired components without being required to answer an inordinate number of questions. It

presented an adequate number of possible answers and did not restrict performance. Testing verified its usefulness, applicability and time saving capabilities.

4. There is a significant set of guidelines, or model, that exists to select software for reuse and thereby reduce the cost of software production as related to non-mathematical applications and systems.

Implementation and mini-testing for usefulness produced results from the study to verify that the Guides Searcher system had merit and could serve as a practical mechanism for effectively identifying and classifying reusable components from existing software libraries. Thus, the model that was presented and implemented in the present study can serve as a set of guidelines to select components for reuse and thus reduce software production costs as related to non-mathematical applications and systems.

Conclusions

On the basis of the literature review, implementation and analysis of the Guides Searcher system, and findings from the implementation, this research study reached the following conclusions:

1. While it is too early to claim a major success, the results of the present study are encouraging enough to support the idea that this particular approach for identification of reusable components is a valid one. The classification was flexible and extensible, but usable by the searcher. The model for a flexible classification system (generalization of the use of facets) was successfully developed for semi-mathematical software reuse

and classification. However, it is important to note that even the most successful model and identification system will still require human intervention when performing evaluation for the reusability of components.

2. The study concluded that the Guides Searcher system approach has merit and can serve as a practical mechanism for effectively identifying reusable components from existing software libraries.
3. The study also concluded that the present overall approach to the reusable software methodology was an important contribution of the research, which was to make discovery of a classification more reliable and less tedious. Also, the user interface allowed views to be created quickly and easily. This appears to be an efficient and practical technique. The Guides Searcher system, through its user-interface, is self-documenting and allows vies to be created quickly and easily.
4. In addition, it was concluded that the study had significance and relevance to complexity theory in general in that it provided a methodological tool for discovering software for reuse and thereby reduce complexity. It was noted in the literature that complexity is a realm that is difficult to define and even harder to understand because it deals with the aggregate of many simple things that can create complex forms. Software is often complex, but abstraction such as that employed in the Guides Searcher system reduces the apparent complexity in a way that presents only the most relevant component and hides all others. Still, no one user-interface (the

means of users to communicate with the system) can be suitable to all. This is clearly reflected in programming. It may be concluded, then, that the present investigation has made a contribution to research specifically focused on reducing complexity.

5. Finally, it was concluded that the present study supported the method utilized by Esteva (1995) who built a library engine and called it Snooper. Esteva considered that the size of a given program correlated to the complexity of the program that is, how tightly or loosely was the relationship from one component to another. Snooper was thus used to determine the complexity of the program. Similar to the method employed by Esteva, the Guides Search contains a classification scheme and searcher system in which artifacts can be retrieved by carrying out a structured dialogue. The name Guides Search was coined by this researcher for the research engine employed in the present study.

Recommendations

In an effort to apply the findings of the study, specific recommendations have been formulated, as based on the findings and conclusions of the present investigation. These recommendations are as follows:

1. The study recommends that future research, in an effort to support the findings of the Guides Searcher system usefulness, conduct follow-up studies, but on a broader scale as regards sample size and number of components to discover and classify. A research investigation that included more knowledgeable persons for testing purposes would almost

certainly yield greater insight and perhaps an even closer convergence with the findings of the present research. A research study that would include a greater number of knowledgeable testers would serve to validate the findings of this study and provide additional and substantial support to the growing body of empirical evidence on the importance and the need for developing systems such as the Guides Search. Esteva commented that steps would be taken to continue the development of Snooper. The same should occur for the Guides Searcher system.

2. It is also recommended that the Guide-Search system be placed on a public World Wide Web location in the future, where users can enjoy its many benefits. Another recommendation is for future research to develop an updated version to provide visualization levels of components as, for example, in 3D graph format.
3. Also, it is recommended that replication of the presents study should logically be made at intervals in the future in an effort to empirically verify the usefulness and applicability of the Guides Searcher system and was developed and implemented in the present study. The system described and implemented in the present investigation has been proven to be useful, but additional work remains to be done. For this reason, this researcher recommends increased usage and development of the Guides Searcher system because it is eminent and financially necessary for companies in order to remain economically viable in today's competitive corporate world. Spiraling costs associated with programming in the

current business environment far outweigh the costs of developing new approaches such as the one presented in the current research study.

References.

Adamczyk, J., & Moldauer, T. (September, 1995). *Trading Off: Inheritance vs. Reuse. Object Magazine,*

Alexander, Christopher. (1977). A Pattern Language: *Towns, Building, Construction.* Oxford University Press, New Yord, 1977

Allworth, S.T. (1981). *Introduction to real-time Software Design.* Macmillan, New York. QA76.54 A44

Anthes, Gary H. (1993). Software reuse plans bring paybacks. Computerworld v27, n49 (Dec 6, 1993):73.

Attila, Gursoy and Kale, Laxmikant V., (1996). *Simulating Message-Driven Programs.* University of Illinois. (charm.cs.uiuc.edu/papers/SimulatorICPP96.www)

Automated Information Mining of Large Software Collections for the Extraction of Reusable Code. NAS5-38035, July 21, 1994.

Auton (no date). www.cs.cmu.edu/~AUTON/doc

Babbie, E. (1990). *Survey research methods.* Belmont, CA: Wadsworth Publishing Company.

Babbie, E. (1986). *The practice of social research.* Belmont, CA: Wadsworth Publishing Company.

Backus, J., (1978). *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.* ACM (v21, n8). p.613

Baer, Tony. (1997). *The politics of reuse.* (software code reuse) Software Magazine v17, n1 (Jan, 1997).

Baldwin-Morgan, A. A. (1994). The Impact of an Expert System for Audit Planning: Evidence from a Case Study. *International Journal of Applied Expert Systems*, 2(3), 101-106.

Banker, Rajiv D.; Kauffman, Robert J. and Zweig, Dani. *Repository Evaluation of Software Reuse.* IEEE Transactions on Software Engineering, Vol. 19, No. 4, April 1993, 379-389.

Barabashev (no date). *Mathematical Notational Systems and the Visual Representation of Metaphysical Ideas*. http://www.seas.gwu.edu/seas/institutes/nel/ vladi. html

Baum, D. (1995). *The Right Tools for Coding Business Rules. Datamation*, v.42, n.4, 36-38.

Baun, John D. (1991). *Elements of pointset topology*. Dover, New York, 1991

Bednarczyk, Marek A. (no date). Gdansk Division of the Institute of Computer Science, Polish Academy of Sciences. http://www.ipipan.gda.pl/~marek/

Bell, F. Bellegarde; et al (1994). Pacific Software Research Center. Oregon Graduate Institute of Science & Technology. TRI-Ada '94 Proceedings, ACM Press, November, 1994, pages 396-404.

Bererd, Edward V., (no date*). Motivation for an Object-Oriented Approach to Software Engineering*. http://www.toa.com/pub/net_articals/motivation_article.txt

Biondo, S. J. (1990). *Fundamentals of Expert Systems Technology*: Principles and Concepts. Norwood, NJ: Ablex Publishers.

Biggerstaff, T. J., and Perlis, A. J. (1989). *Software Reusability.* New York, NY: ACM Press.

Biggerstaff, T. J.; and Richter, C. (1987*). Reusability Framework, Assessment, and Directions*, IEEE Software 41-49.

Blissmer, R. H. (1991). *Introducing computers: Concepts, systems and applications*. New York: Wiley and Sons.

Boehm, B.W (1981). *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

Brown, Kyle (no date*) Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk.* www2.ncsu.edu/eos/info/tasug/krown/thesis2.htm

Buchanan, B. G., & E. H. Shortliffe (1985). *Rule-Based Expert Systems*. Reading, MA: Addison-Wesley.

Budinsky, F.J; Finnie, M.A.; Vlissdes, J.M and Yu, P.S (1996*). Automatic code generation from design patterns*. IBM Systems Journal. V35, n2 (1996). http://www.almaden.ibm.com/journal/sj/budin

Barabashev (no date). *Mathematical Notational Systems and the Visual Representation of Metaphysical Ideas*. http://www.seas.gwu.edu/seas/institutes/nel/ vladi. html

Baum, D. (1995). *The Right Tools for Coding Business Rules. Datamation*, v.42, n.4, 36-38.

Baun, John D. (1991). *Elements of pointset topology*. Dover, New York, 1991

Bednarczyk, Marek A. (no date). Gdansk Division of the Institute of Computer Science, Polish Academy of Sciences. http://www.ipipan.gda.pl/~marek/

Bell, F. Bellegarde; et al (1994). Pacific Software Research Center. Oregon Graduate Institute of Science & Technology. TRI-Ada '94 Proceedings, ACM Press, November, 1994, pages 396-404.

Bererd, Edward V., (no date). *Motivation for an Object-Oriented Approach to Software Engineering*. http://www.toa.com/pub/net_articals/motivation_article.txt

Biondo, S. J. (1990). *Fundamentals of Expert Systems Technology*: Principles and Concepts. Norwood, NJ: Ablex Publishers.

Biggerstaff, T. J., and Perlis, A. J. (1989). *Software Reusability.* New York, NY: ACM Press.

Biggerstaff, T. J.; and Richter, C. (1987). *Reusability Framework, Assessment, and Directions*, IEEE Software 41-49.

Blissmer, R. H. (1991). *Introducing computers: Concepts, systems and applications*. New York: Wiley and Sons.

Boehm, B.W (1981). *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

Brown, Kyle (no date) *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk.* www2.ncsu.edu/eos/info/tasug/krown/thesis2.htm

Buchanan, B. G., & E. H. Shortliffe (1985). *Rule-Based Expert Systems.* Reading, MA: Addison-Wesley.

Budinsky, F.J; Finnie, M.A.; Vlissdes, J.M and Yu, P.S (1996). *Automatic code generation from design patterns*. IBM Systems Journal. V35, n2 (1996). http://www.almaden.ibm.com/journal/sj/budin

Busch, E. (1988). *A CASE for Existing Systems*. Salem, MA: Language Technologies, Incorporated.

Caldwell, B. (1994, Nov. 14). *Software reuse comes of age*. Information Week, 501:122-124.

Carmichael, A. (1994). *Object Development Methods*. New York: SIGS Publications.

Cawsey, Alison (1994*). Databases and Artificial Intelligence 3 Artificial Intelligence Segment*. http://www.cee.hw.ac.uk/~alison/ai3notes/subsection2_4_2_2.html

Chen, D.J.; Chen, David T.K. (1994). *An experimental study ofusing reussable software design framworks to achieve software reuse*. Journal of Object-Oriented Programming v7, n2 (May, 1994):56.

Chen, H. et al . (1994). Explaining and Alleviating Information Management Indeterminism: A Knowledge-Based Framework. http://ai.bpa.arizona.edu/papers/ipm91/listoffigures3_2.html#SECTION0002000000000000000

Chen, R. W., & Prinz, F. B. *A Cost-Benefit Model of Product Design for Recyclability and its Application*. IEEE Transactions on Computers, 1994, pp. 502-512.

Cho, Youngsuck , (1994). *A Multi-Faceted Software Reusability Model: The Quintet Web*, The Department of Computer Science, Laouisiana State University and Agricultural and Mechanical College.

Citrin, Wayne; McWhirter, Jeffrey D. (1995*). Diagram Entry Mechanisms in Graphical Environments*. CHI '95 Proceedings. http://www.acm.org/sigchi/chi95/proceedings/shortppr/wc_bdy.htm

CMU Artificial Intelligence Repository Home Page (1996). Internet Publishers: Author.

Coad, P., et al. *Object Models: Strategies, Patterns and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

Codd, E. F. (1970). *A Relation Model of Data for Large Shared Data Banks*. Communications of the ACM, v13, n6 (June, 1970):p377.

Cohen, Daniel I. A. (1991). *Introduction to computer Theory*. John Wiley & Sons, Inc.

Cohen, P. R and Kjeldsen, R (1987). *Information retrieval by constrained spreading activation in semantic networks.* Information Processing and Management 1987, v23, n4, Page 255-268

Cohen, Sholom G. et al (1992). *Application of Feature-Oriented Domain* Analysis to the Army Movement Control Domain(CMU/SEI-91-TR-28, ADA 256590). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

Compton, P. et al (1989). Maintaining an Expert System. In I. Quinlan & J. Ross (eds.), *Applications of expert systems, Vol. II,* Addison-Wesley, 366-384.

Constantine, L. L. & Yourdon E. (1989). *Structured Design.* Englewood Cliffs, NJ: Prentice-Hall.

Day, M; Gruber, R; Liskov, S; A.C. Myers. (1995). Subtypes vs. Where Clauses: *Constraining Parametric Polymorphism.* In the proceedings of OOPSLA '95. Austin TX, October 1995. www.pmg.lcs.mit.edu

Davis, M; Sigal, R.; Weyuker, E. (1994). *Computability, Complexity and Languages.* Fundamentals of Theoretical Computer Science, Second Edition, Academic Press, New York, NY, 1994.

DES (no date). *The Data Encryption Standard.* raphael.math.uic.edu/~jeremy/crypt/ des.html

Diaz-Herera, Jorge L.; Cohen, Sholom and Withey, James (1996). *Institutionalizing Systematic Reuse: A Model-based Approach.* Software Engineering Institute, Carnegie Mellon University. (http://www.sei.cmu.edu)

Dixon, John D. (1967). *Problems in Group Theory.* Blaisdell Pub. 1967. QA 171 D59.

Downie, N., & Heath, R. W. *Basic Statistical Methods* (5th ed). New York: Wiley, 1984.

Endoso, Joyce (1992). *Army center spearheads domain analysis. (software reuse in US Army).* Government Computer News v11, n23 (Nov 9, 1992):71.

Esteva, Juan Carlos (1995). *Automatic Identification of Reusable Components.* IEEE 1995

Ettel, R., Fingar, P. , & D. Read (1995). *Shared Objectives*: Object-Oriented Information Systems. *CIO,* 8, 68-72.

Faris, C. (1995). *Reuse Initiative.* Andersen Consulting Home Page. http://www.ac.com

FDR Manual. (1997). Formal Systems (Europe) Ltd. Failures-Divergence Refinement FDR 2. *User Manual*, 1997. ftp.comlab.ox.ac.uk/pub/Packages/FDR.

Finch, Steven (1996). *Favorite Mathematical Constants.* Research and Development Team, MathSoft, Inc. (www.mathsoft.com)

Fischer, G. (1992). *Domain-Oriented Design Environments* in KBSE '92 Knowledge-based Software Engineering Conference, IEEE Computer Society Press, page 204

Forslund, G. (1995, August). *Toward cooperative advice-giving systems: A case study in knowledge-based decision support.* IEEE Expert, 10(4), 27-37.

Fox, Edward A.; Chen, QI Fan; Daoud, Amjad M. and Heath, Lenwood S. (1991). *Order-Preserving Minimal Perfect Hash Functions and Information Retrieval.* ACM Transactions on Information Systems, Vol. 9, No. 3, July 1991, 281-308.

Frakes, W. B. and Gandel P. (1990). Representing Reusable Software, Information and Software Technology, Vol. 32, No. 10, pp. 653-664.

Frakes, W. B., and Pole, T. B. (1994). *An empirical study of representation methods for reusable software components.* IEEE Transactions on Software Engineering, 20(8): 617-630.

Frakes, W. B.; Gandel, P. B. (1990). *Representing reusable software.* Information and Software technology, v32 (Dec, 1990)

Freitag, Burkhard (1994). *A Hypertext-Based Tool for Large Scale Software Reuse. Advanced Information Systems Engineering*, 6th International Conference, CAiSE'94, Utrecht, The Netherlands, June 1994. Springer-Verlag.

Frenzel, L. E. *Crash Course in Artificial Intelligence and Expert Systems.* Indianapolis, IN: Sames Publishing Co, 1989.

Fritzson, Peter; Viklund, Lars; Herber, Johan (1995). High-Level Mathematical Modeling and Programming. IEEE Software. V12, n4 (July, 1995). p77

Frohn, Jürgen; Lausen, Georg; Uphoff, Heinz. (1994).Access to Objects by Path Expressions and Rules. VLDB 1994: 273-284. http://researchsmp2.cc.vt.edu/DB/db/conf/vldb/vldb94-273.html

Gales, L., & D. Mansour-Cole (July, 1995). User Involvement in Innovation Projects: Toward an Information Processing Model. *Journal of Engineering and Technology Management*, 12(1-2), 77-109.

Gaska, Marilyn T. (1996). Reuse Lessons Learned from Architecture and Building Systems Integration. Loral federal Systems-Owego. (www.lfs.loral.com).

Gehani, N. H.; Jagadish, H. V.; Roome, W. D. (1994). OdeFS*: A File System Interface to an Object-Oriented Database*. VLDB 1994. 249-260. http://researchsmp2.cc.vt.edu/DB/db/conf/vldb/vldb94-249.html

Gentleman, Robert (no date). *The R Language*. http://www.stat.math.ethz.ch/R-manual/funs/data.matrix.html

Giarratano, J., & G. Riley (1993). *Expert Systems Principles and Practice.* Boston, MA: PWS Publishing.

Glen, Ron (1993). Common traits and reuse rates. I.T. Magazine v25, n9 (Sept, 1993):32.

Goble, T. (1989). *Structured Systems Analysis through PROLOG.* Englewood Cliffs, NJ: Prentice-Hall.

Gold, D., and R. T. Plant (1990). *Towards the Formal Specification of an Expert System.* Working Paper CIS/RTP/90/2, CIS Dept, University of Miami, Coral Gables, Florida.

Girardi, M. R.;Ibrahim, B. (no date*). An approach to improve the effectiveness of software retrieval.* www.unige.ch/eao/www/ROSA.papers/ISS93.

Gutowitz, Howard (1994*). A Massively parallel Cryptosystem Based on Cellular Automata*. http://www.santafe.edu/~hag/ca11

Hanink, Joe (1997). *Function-Based Encryption*. home.earthlink.net/~ortech/fbeinfo.html

Harris, D. G. et al. (1990). *Report on Computer Board Initiative on Knowledge Based Systems*. Sterling, UK: University of Sterling, Institute for Retail Studies.

Hinchey, Michael G. (1995). *Formal Development in CSP*. McGraw-Hill, 12/95

Hislop, Gregory W. (1995). Reuse versus reusable. Information Week, n510 (Jan 16, 1995):67.

History of Mathematics (1995). *Re-use and Abstraction.* http://kalypso.cybercom.net/~rbjones/rbjpub/matchs

Hoare, C.. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall, 1985.

Holmes, Steve (1995). *Finding Information about Library Functions. C Programming*. University of Strathclyde Computer Centre. (http://www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html)

Humphrey, W. S. (1990). *Managing the Software Process*. New York: Addison-Wesley.

Hutt, A. T. (1994). *Object Analysis and Design: Description of Methods*. Burr Ridge,IL: Irwin.

IMSL, Inc. (1987). *Library Reference Manual*. Houston, TX.

IMSL, Inc. (August, 1989). *Math/Library Quick Reference*. Document Number MALB-QRF-EN8909, Houston, TX .

Jackson, P. *Introduction to Expert S* Klein, M. *Knowledge Based Decision Support Systems: With Applications in Business*. New York: Wiley, 1995.

Jagadeesan, Radha & Pingali, Keshav. (1991). *A Fully Abstarct Semantics for a First-Order Functional Language with Logic variables*. ACM Trans. Lang. And Systems, v13, n4 (Oct, 1991):577-625

Johnson, Jeff A; Nardi, Bonnie A.; Zarmer, Craig L.; Miller, James R. (1993). ACE: *building interactive graphical applications*. Communications ACM, v36, n4 (April 1993), Pages 40-55

Johnson, Ralph; Foote, Brian. (1988). Designing Reusable Classes. Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22-35.

Jones, Capers (1994). *Economics of software reuse*. Computer v27, n7 (July, 1994):106.

Jones, Roger Bishop (1995). *Propositional Logic*. http://www.rbjones.com/rbjpub/logic/log003.htm

Kale L. V. (1996). *The Chare Kernel parallel programming language and system.* Proceedings of the International Conference on Parallel processing. v.2 (Aug, 1990) p.17

Kang, Kyo C. And et al (1990). *Spencer Feature-Oriented Domain Analysis* (FODA) Feasibility Study(CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, Pa.:Software Engineering Institute, Carnegie Mellon University, 1990.

Keenan, B. T. (no date). www.cs.ac.uk/User/B.T.Keenan/project/know.html

King, B. (October 27, 1993). *Expert Systems in Manufacturing.* School of Computing and Information Systems, University of Sunderland. Occasional Paper 93-1.

Kirk, R. E. (1978). *Experimental design: Procedures for the Behavioral Sciences.* Belmont, CA: Wadsworth.

Klein, M. *Knowledge Based Decision Support Systems: With Applications in Business.* New York: Wiley, 1995.

Kochen, Manfred (1984). *Coding For Recording and Recal of Information.* Information Processing & Management, Vol. 20, No. 3, 343-354.

Krueger, Charles W. (1992). *Software reuse. (creating applications from existing elements).* ACM Computing Surveys v24, n2 (June, 1992):131.

KSL (Stanford, no date) http://www-ksl.stanford.edu/testfiles/htw/experimental-ontologies/frame-ontology/REFLEXIVE-RELATION.html

Kumar, Vive S. (1996). *Personal Software Process in Meta-CASE CMPT 856 - Project.* http://www.cs.usask.ca/grads/vsk719/academic/856/project/node27.html

Kuokka, D. (No Date) MAX: *Meta-reasoning Architecture for "X"* krusty.eecs.umich.edu/cogarch2/specific/max.html

Landau, Susan et al (no date). Chapter 1: Information Protection in the Information Age. http://info.acm.org/REPORTS/ACM_CRYPTO_STUDY/_WEB//chap1.html

Lenz, Manfred; Schmid, Hans Albrecht; Wolf, Peter F. (1987*). Software Reuse through Building Blocks.* IEEE Software. v4 (1987)

Leong, L. H. S. (1992). Applications of Expert Systems in Manufacturing: A Case Study. *Computers in Industry*, 18, 193-198.

Li, Shu-Xiang & Loew, Murray H. (1987). The quadcode and its arithmetic. ACM v.30 n.7 (July 1987), p.621-626

Maiden, Neil A.; Sutcliffe, Alistair G. *Exploiting reusable specifications through analogy. (Computer-aided software engineering tool support for software specification reuse).* Communications of the ACM v35, n4 (April, 1992):55

Manley, Kirk C. (1997*). Composite Grid/Frame Reinforcements for Concrete Structures*. http://www.cecer.army.mil/facts/sheets/PD13.html

Mannino, Michael V. (1990). *The Object-Oriented Functional Data Language.* IEEE Trans. On Software Engr. V16, n11. (Nov,1990):1258

Marais, Johannes L (1994*). Oberon System 3: designed with software reuse in mind.* Dr. Dobb's Journal v19, n11 (Oct, 1994):42.

Marcus, S., & J. McDermott (1989). SALT: A Knowledge Acquisition Language for Purpose-and-Revise Systems. *Artificial Intelligence*, 39, 1-37.

Marlin, C. D. (April, 1995). *Exploring the Role of the ProgrammingLlanguage in an Integrated Software Development Environment.* Presented at Workshop on Research Issues in Software Engineering and Programming Languages. Seattle, Washington.

Martin, J. (1990). *Structured techniques: The Basis for CASE.* Englewood Cliffs, NJ: Prentice-Hall.

Martin, J. (1983). *Managing the Data-Base Environment.* Englewood Cliffs, NJ: Prentice-Hall.

Martin, J., & J. Odell (1993). *Principles of Object-Oriented Analysis and Design.* Englewood Cliffs, NJ: Prentice-Hall.

Math77, *Mathematical Subprograms for Fortran77.* Fortner Reserach LLC, 2rd Printing November 1995.

Mattison, R., & M. L. Sipolt (1995). An Object Lesson in Management. *Datamation*, 41, 51-55.

Matuszek, David (1996). *P and NP.* http://www.netaxs.com/people/nerp/automata/p-and-np1.html

McCarthy, John (1996). *Mathematical Theory of computation.* www.formal.stanford.-edu/jmc

McClure, C. (1995, June). *Reuse finds common ground.* Software Magazine, 15(6): 5-6.

McNaughton, R., & D. Patel (1992). A Framework for a Quality Management Cycle for Expert System Development in a Prototyping Environment. In *Sunderland Advanced Manufacturing Technology*. International Conference on Manufacturing Technology, Proceedings 27-30.

Mili, Hafedh; Radai, Roy; Weigang, Wang; Strickland, Karl; and others. *Practitioner and Softclass: a comparative study of two software reuse research projects*. Journal of Systems and Software v25, n2 (May, 1994):147

Minsky, M. (1975). *A framework for representing knowledge*. In P. H. Winston, editor, The Psychology of Computer Vision, pages 211 - 277. McGraw-Hill, New York

Mizoguchi, R., & H. Motoda (October, 1995). AI in Japan: Expert Systems Research in Japan. *IEEE Expert*, 10(5).

Mitchell, Gail Anne (1993). Extensible Query Processing In An Object-Oriented Database. Department of Computer Science, Brown University.

Moldovan, Dan I.; Wu, Chung-I (1988). *Concise Papers. A Hierarchical Knowledge Based System for Airplane Classification*. IEEE Transactions on Software Engineering, v14, n12, December 1988. (1829).

Montlick, Terry, (1997). *What is Object-Oriented Software?* www.softdesign.com/software/objects.html

NAG, Ltd. (1986). *Library Reference Manual*. Oxford, England.

Navak, Gordon S., Jr.; Member, IEEE, Hill, Fredrick N.; Wan, Man-Lee and Sayrs, Brian G. (1992). *Negotiated Interfaces for Software Reuse*. IEE Transactions on Software Engineering, Vol. 18, no. 7, July 1992.

Novak, Gordon S. Jr.; Hill, Fredrick N.; Wan, Man-Lee; Sayrs, Brian G. *Negotiated interfaces for software reuse*. IEEE Transactions on Software Engineering v18, n7 (July, 1992):646.

Novak, Gordon S. Jr. (1983). GLISP: *A Lisp-based language with Data Abstraction*. A.I Magazine,v4,n3 (Fall 1983). http:://www.cs.utexas.edu/users/novak/aimag83.html

O'Brien, L. (1996). From Use Case to Database: Implementing a Requirements Tracking System. *Software Development*, 4(2), 43-47.

O'Keefe, R. M., & D. Rune (1993). *Understanding the Applicability of Expert Systems*. International Journal of Applied Expert Systems, 1(1), 17-27.

Olson, J. R., & H. Reuter (1987). Extracting Expertise from Experts; Methods for Knowledge Acquisition. *Expert Systems*, Vol. 4(3), 152-168.

Orfali, R., & D. Harkey (1995). Object Component Suites: The Whole is Greater than the Parts. *Datamation*, 41, 44-46.

Pearce, C. (1994). *A Dynamic Hypertext Environment through N-Gram Analysis*. Ph.D. Dissertation, UMBC.

Peter (no date). *Data Abstraction*. http://www.cs.kun.nl/~peter88/PeterThesisCh5.html

Plant, R. T. (1992). *Expert system development and testing: A knowledge engineer's perspective.* Journal of Systems and Software, 19, 141-146.

Poulin, J.S.; Caruso, J.M.; Hancock, D.R. *The business case for software reuse.* IBM Systems Journal v32, n4 (Dec, 1993):567.

Practical Application of Small Expert Systems (1996). TOPS Exhibit #S-405. NASA Exhibits. Internet page.

Price, R. T.; Girardi, M. R. (1990). *A class retrieval tool for an object-oriented environment*, in Procs. 3rd Int. Conf. Technology on Object-Oriented Languages and Systems, Sydney, 28-30 Nov. 1990.

Prieto-Diaz, R. (May, 1993). Status Report: Software Reusability. *IEEE Software*, 10(3).

Prieto-Diaz, R. (1985). *A Software Classification Scheme.* Department of Information and Computer Science, University of California at Irvine.

Prieto-Diaz. R. (1987). *Classification of reusable modules*, IEEE Software, 6(1).

Prieto-Diaz, R. & P. Freeman. (January, 1987). *Classifying Software for Reusability. IEEE Software*, 4(1), Jan 1987.

Prieto-Diaz, R. (1991). *Implementing Faceted Classification for Software Reuse,* CACM, vol. 34, no. 5, May 1991, pp. 88-97.

Purtillo, James. (1990). *The Polylith software toolbus.* Technical report CS-TTR-2469. University of Maryland, Department of Computer Science and UMIACS, 3/90

Quantitative Data Systems (1996). *Software Reuse: Establishing a Framework of Reuseable Software.* Internet Home Page (www.qds.com/qds.hom.page).

Quinlan, I., and J. Ross. (1989). *Applications of expert systems.* Volume II. Reading, MA: Addison-Wesley.

Radding, Alan (1993). Not quite ready for prime time. (Object-oriented programming) (includes related article on object-oriented databases, definitions of objects, important trends and software reuse). Computerworld v27, n24 (June 14, 1993):107

Ransom, K. J. & C. Marlin (April, 1995). *Supporting Software Reuse within and Integrated Software Development Environment.* Proc. ACM SIGSOFT Symposium on Software Reusability. Seattle, Washington, 233-237.

Ray, Garry (1992). *Software reuse not a panacea; some firms pursue it as a development goal; others question its viability.* Computerworld v26, n51 (Dec 21, 1992):47.

Reid, Glenn (1994). *Today's reuse recipe comes in three flavors.* Computing Canada v20, n4 (Feb 16, 1994):25.

Reuse Issues. (1993). ICSE 15 Parallel Session 2F. Baltimore, MD: ICSE.

Rhodes (no date). www/Juniata.edu/~Rhodes

Roth, F. H. (1984). The Knowledge Based Expert System: A Tutorial. *IEEE Computer*, 17(9), 11-28.

Rowson, James; Sangiovanni-Vincentelli, Alberto (1997). *Interface-based code design weighs reuse.* Electronic Engineering Times, n935 (Jan 6, 1997)

Rubinovitz, Harvey and Thuraisingham, Bhavani (1993). *Design and Implementation of a Query Processor for a Trusted Distributed Data Base Management System.* J. Systems Software, 1993; 21:49-69.

Salamon, W.J; Wallace D.R. (no date). *Quality characteristics and Metrics for Reuseable software.* hissa.ncsi.nist.gov/HHRFdata/artifacts/Itdoc/5459/metrics.html

Salton, G. (1989). Automatic Text Processing: *The transformation, Analysis and retrieval of Information by Computer.* Addison-Wesley, 1989.

Sander, Evelyn (1994). *Groups Symmetry.* http://www.cee.hw.ac.uk/~alison/ai3notes/subsection2_4_2_2.html

Schlukbier, A. (1995, May 8). *The future is reuse.* Computerworld, 29(19): 77-78.

Schmidt, Coplien. D. (1995). *Pattern Languages of Program Design.* Addison Wesley

Schrage, Michael. (1995, August 21). *Software: reuse it or lose it.* Computerworld, 29(34): 31-33.

Schwartz, Karen D. (1993). *DOD resue libraries get linked today. (Department of Defense project boosts software engineering efficiency).* Government Computer News v12, n8 (April 12, 1993):1.

SER (no date). *Solutions for Software Evolution and Reuse.* www.sema.es/projects/SER

Shai, Ben-Yehuda (no date). *The Canonical OO Framework Pattern.* http://www.sela.co.il:8080/~shai/canon_fw.htm

Skipjack. (no date). Www.austinlinks.com/Crypto/skipjack-review.html

Slofstra, Martin (1995). Netron takes to the road to talk of frames and reuse. Computing Canada v21, n9 (April 26, 1995):20.

Smith, A. (1992). A. Smith. *Cache memories.* ACM Computing Surveys. V14, n3. Sep, 1992. p.473

Smith, D. (1992) KIDS: A knowledge-based software development in KBSE '92 Knowledge-based Software Engineering Conference, IEEE Computer Society Press, page 204.

Smith, P. et al (1992). The Use of Expert Systems for Decision Support in Manufacturing. *Expert Systems with Applications*, 4, 11-17.

Special Issue on Systematic Reuse (Sept., 1994). *IEEE Software*, 11(5).

Solderitsch, James (no date). *Creating an Organon Intelligent Reuse of Software Assets and Domain Knowledge.* http://source.asset.com/stars/lm-tds/Papers/cr-org/ancoat-final.html

Srinivasan, P., & and Rama, D.V. (1989, January). *An Investigation of content representation using text grammars.* ACM Transactions on Information systems, 11(1): 48-53.

Stanford University (no date). www-ksl.stanford.edu/KSL_Abstracts/KSL-93-68.html

Stroustrup, Bjarne (1996). *The C++ Programming language.* Addision Wesly, 1996.

Sun Microsystems, Inc. (no date). *Special Java Issue.* http://www.tdmi.com/hottest0108/

Taylor, D. A. (1990). *Object-Oriented Technology: A Manager's Guide.* Reading, MA: Addison-Wesley.

Teer, F. (1994). The Trend in Expert Systems Coverage in Business Schools and Implications for People in Industry. *International Journal of Applied Expert Systems* 2(3), 127-133.

Terlouw, J.P. (1997). *Groningen Image Processing System. System for Astronomical Image Processing.* http://www.astro.rug.nl/~gipsy/gds/input_p.html

The Full Computing Reviews *Classification Scheme* (1990, January). ACM Computing Reviews, 31(1).

Tibbetts, J., & Bernstein, B. (1995, March 27). *Build on what's already there: With patterns, developers don't always have to start from scratch.* Information Week, 520:126-128.

Todd, Daniel (1990*). Code recycling: reuse of software can save on development.* Information Week, n270 (May 14, 1990):50.

Tracz, W. (1988). *Software Reuse: Emerging Technology*, Washington, DC: IEEE Computer Society.

Turban, E. (1995). *Decision support and expert systems:* Management support systems. Englewood Cliffs, NJ: Prentice-Hall.

University of Liverpool, (1997). *Object Oriented programming.* University of Liverpool, 1996. BST 1997. www.liv.ac.uk

Urban, M.L.; Chang, I. N. (1995). *Information Architecture as a Framework for reuse.* sw-eng.falls-church.va.us/reuseic/pubs/proccedings

Van Horn, M. (1986). *An understanding of expert systems.* New York: Bantam Books.

Verena (no date). *Comparision of Fortran77, C, C++, and Fortran 90.* http://csep1.phy.orm.gov/pl/node2.html

Verstraete, (1997). *System Decomposition. Revised*: September, 1997. www.smeal.psu.edu/misweb/systems/sycodeco.html#HIER

Virginia Center of Excellence for Software Reuse and Technology Transfer. *Reuse Adoption Guidebook*, Version 2.0. (www.software.org).

Wasmund, M. (1993). *Implementing critical success factors in software reuse.* IBM Systems Journal v32, n4 (Dec, 1993):595.

Weigret, T., & and Jang, H. C. (1992, December). *A hybrid knowledge representation as a basic of requirement of specification and specification analysis.* IEEE Transactions on Software Engineering, 18(2): 1076-80.

Welbank, M. (1983). *A Review of Knowledge Acquisition Techniques for Expert Systems.* Martlesham Health: UK, British Telecom Research Laboratories.

Werth, J. S., & L. H. Werth (May, 1991). *Directions in Software Engineering Education.* In Proceedings of Thirteenth International Conference on Software Engineering.

Wexelblat, R. L., (1981). *History of Programming Languages.* Academic Press, The history of Fortran. (www.csc.liv.ac.uk/~u4sdg)

White, A. P. (1995). An Expert System for Choosing Statistical Tests. *New Review of Applied Expert Systems*, 1, 48-53.

Whitten, J. L. & Bentley, L. D. (1989). *Systems Analysis and Design.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

Wilkes, Maurice (1993). Communication of the ACM. V36, n7 (July, 1993). p21

Willis, P. J. (1994). ANSI *C Programming.* www.bath.ac.uk/~maspjw/NOTES/ansi_c

Woolnough, Roger (1990). *Esprit: let's reuse software.* Electronic Engineering Times, n620 (Dec 10, 1990):20.

Wyder, T. (1996). Capturing Requirements with Use Cases. *Software Development*, 4(2), 37-40.

Xmath (no date). www.isi.com/Products/MATRIXx/Techspec/MATRIXxXmath/

Zimmerman, H. J. (1987). *Fuzzy Sets, Decision Making, and Expert Systems.* Boston, MA: Kluwer Academic Publishers.

Annotated Bibliography.

Ang, T. S. L., Arnott, D. R., & P. O'Donnell (1995). *Formal Problem Formulation in the Development of Decision Support Systems.* In the Proceedings of the Pan Pacific Conference on Information Systems, Singapore, 179-182.

In the development of decision support systems, formal problem formulation is an important concern, according to the authors in their report at the Pan Pacific Conference. They point out the pitfalls of inaccurate problem formulation and provide guidelines for effectiveness. They conclude that problem formulation invariably relies on logical, detailed thinking processes, as based on tested approaches such as case-based reasoning.

Arnott, D. R., & P. A. O'Donnell (1991). *What are Decision Support Systems?* In D. R. Arnett and P. A. O'Donnell (eds). *Readings in decision support systems.* Melbourne: CISR, 309-322.

This article provides an in-depth definition with accompanying examples of decision support systems in a large collection of readings on decision support problems, benefits, and areas of concern. The authors note that these systems are used mainly where similar decision processes are repeated, but where the information to decide upon differ. Typical application areas include medical diagnosis, credit evaluation, and process control systems.

Baker, Rajiv D.; Kauffman, Robert J. and Zweig, Dani *(1994). Automating output size and reuse metrics in a repository-based computer-aided software engineering (CASE) environment.* IEEE Transactions on Software Engineering, Vol. 20, No. 3, March 1994:169.

Researchers describe three reuse metrics: leverage, value, and classification, but continue to erroneously define reuse as the "use" of a component and include repeatedly "using" a component within a team (external reuse is defined as software from other applications). The classification metric differentiates between internal and external reuse. mainstream effort in software reuse are ignored. The best part of the article comes in the appendix in which the authors provide an excellent summary of how to calculate function points.

Baker, Rajiv D.; Kauffman, Robert J. (1991). *Reuse and productivity in integrated computer-aided software engineering: an empirical study.* MIS Quarterly v15, n3 (Sept, 1991):375.

In this assessment of reuse on productivity, the focus is on reuse in integrated computer-aided software engineering. A number of companies are included in the analysis. A later paper by the same team raises the issues to a higher level. They concluded that few companies know what programs are in their current inventory. Even less have solid productivity measurement systems in place to monitor systems development efforts.

Baxter, Ira D. (1992). *Design maintenance systems*. Communications of the ACM v35, n4 (April, 1992):73.

In Baxter's view, efficient design maintenance systems must be in place for system success. There are not as many design issues in expert systems as there are in neural networks. Fewer choices are available to represent knowledge. In neural networks a series of decisions must be made on system structure. Designers must determine the number of layers, the type of connectivity, the method of learning, and learning parameters. A number of other design and maintenance issues are reviewed.

Benoit, L. Lemaire & J. Moore, (1994). *Human Factors in Computing Systems*. Boston: Allyson & Bacon.

A major point in the book is that human factors have not been totally considered in current computing programs. Experts often have the knowledge but do not know how to put it into a workable form. Using shells taking into account human factors has been less rewarding, but can overcome problems because they contain the explanation generator and inferencing code. Experts and the developer need only enter the knowledge base rules and customize the user interface. A number of other issues are discussed in the book.

Berry, C., & D. E. Broadbent (1986). *Expert Systems and the Man-Machine Interface. Expert Systems*, 3(4), 228-231.

The authors define expert systems as computer programs that have been constructed in such a way that they are capable of functioning at the standard of human experts in given fields. They briefly discuss the need for improved user interfaces to effectively use expert systems, noting that man-machine interfaces have improved over time. They also point out that the process of collecting knowledge remains a time-consuming task..

Berry, C., & D. E. Broadbent (1987). *Expert Systems and the Man-Machine Interface. Part Two: The User Interface. Expert Systems*, 4(1), 18-27.

This is a continuation of the article cited above. In this section, the authors thoroughly review problems associated with user interfaces. They explain that interfaces are the user's means of communicating with the system and tend to use a pseudonatural dialogue. Full natural-language interfaces using a syntax close to the user's native language are largely a future possibility.

Biggerstaff, Ted J. (1994). *The Library Scaling Problem and the Limits of Concrete Component Reuse*. Proceedings of the 3rd International Conference on Software Reuse, Rio de Janeiro, Brazil, 1-4 November 1994, 102-109.

This is a well-written paper describing issues surrounding growing the stereo-typical reuse library of a limited number of small components to one with larger and more components. The researcher describes techniques to abstract variations of components such as variable macros, parameterized types, and module interconnection languages, among others. He concludes from his study that Draco and P++ seem to have taken the best approach.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Reading, MA: Addison-Wesley.

Grady Booch suggests that the discipline of object-oriented technology will soon give rise to a marketplace of reusable software components that can be assembled into robust and scaleable software solutions. He provides descriptions of o-o design and general applications. This book builds on his previous works discussing software engineering with Ada. Unfortunately, no reusable software component marketplace has yet to arise.

Boose, J. H., & J. M. Bradshaw (1987). Expert Transfer and Complex Problems: Using AQUINAS as a Knowledge Acquisition Workbench for Knowledge-Based Systems. *Journal of Man-Machine Studies*, 27, 167-179.

Complex problems associated with transfer of expert knowledge is the focus of this article. Bottlenecks are discussed. Conceptualization tools are noted to support domain elicitation and owe their origin to psychological theories or repertory grids and card sorting techniques. Examples cited include AQUINAS, a successor to the Expertise Transfer System which has a broader scope of functionality. Use of AQUINAS is recommended.

Borgatta, E. F., & G. W. Bohrnstedt (1980). *Sociological Methodology*. 2nd ED. San Francisco, CA: Jossey-Bass.

This represents a well-organized, clearly written, and practical basic primer for research design and methodology. The work seeks to introduce the basic tools of research by explaining the various research techniques and methodologies. Numerous examples are provided. Included are various types of study designs and methods of statistical analysis. The authors also provide guidelines for collecting study and research data.

Bose, Ranjit. (1995). *Organizational Computing, Coordination, and Collaboration : An Expert Systems Framework.* New Review of Applied Expert Systems, 1 (1), 27-32.

This paper illustrates design and construction of an automated group support system using expert systems technology. Software framework automates collaborative organizational processes, based on augmenting human members with computerized assistants, called intelligent agents. Framework organizes intelligent agents to match the human group. Designed using o-o technology; implemented using logic programming language, Prolog2.

Boyd-Williams, Michael. (1993). *Expert System Support for Object-Oriented Database Design.* International Journal of Applied Expert Systems, 1(3): 91-96.

Reports on the development of the Object Design Assistant (ODA), an expert system for providing intelligent assistance in design of structural aspects of object-oriented databases. The purpose of ODA was to allow a user to design the database without having extensive knowledge. Tool illustrated that it was effective. Demonstrates practical application of expert system techniques. Concludes with discussion of areas for future development.

Bradley, John, & Gupta, Uma G. (1995). *A Classification Framework for Case-based Reasoning Systems.* New Review of Applied Expert Systems, 1 (1).

Case-based reasoning, a powerful approach to solving problems that require experience, intuition, and judgment, has been discussed as if all case-based systems are alike. This paper presents a framework, useful for differentiating case-based reasoning systems into one of four categories. Authors believe that this categorization will assist the developers in planning, since there are common characteristics to the systems in each category,

Brown, Carol V., Ph.D., and Bostrom Robert P., Ph.D., (1990). *Choosing the Right approach for End-User Computing Management.* Information Executive, Fall 1990, 30-33.

The right approach for management of end-user computing is essential if a both larger systems and subsystems are to function adequately, according to the report. Selecting the right approach is essential to responding effectively and meeting the needs of company executives. They describe a number of approaches that have been used and provide several techniques as well as their view of the better method.

Chandrasekaran, B. (1986). *Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design. IEEE Expert*, 1, 23-30.

This report presents an excellent review of generic tasks that should be associated with knowledge-based reasoning. In his view, reasoning is often based on common sense and logic. As noted in other studies, good mental models are necessary for knowledge-based reasoning. The author notes that certain repertory grids and card sorting tools have been developed to handle specific categories of problems known as generic tasks. He provides an example of a trouble-shooting tool called TDE.

Chapman, A. J. (1994). Stock Market Trading Systems through Neural Networks: Developing a Model. *International Journal of Applied Systems*, 2(2), 1-8.

This article explores the use of neural networks for stock market trading application, noting that neural networks serve as strong predictive models to uncover complex relationships. The author provides definitions, reviews the impact of neural networks, and presents an example of how neural networks can be used as a tool for analyzing market share. Trading systems are defined in the context of stock market trading and forecasting.

Chauvet, Jean-Marie (1995). *Maintain your software: efficient use and reuse of object-oriented technology*. Software Development v3, n8 (August, 1995):34(5).

The focus of this paper is on the efficient use and reuse of o-o technology in maintaining software. The author believes that o-o programming and languages give users a better chance to reach their goal. Object-oriented design focuses on abstractions (classes), which seem to be practical units for reusable components. Also, the improved techniques on data-hiding, genericity and the use of inheritance available in current object-oriented languages are better suited for the needs of reusable software components.

CMU Artificial Intelligence Repository Home Page (1996). Internet Publishers: Author.

The Internet posting discusses MIKE (Micro Interpreter for Knowledge Engineering), a full-featured, free, and portable software environment designed for teaching purposes. MIKE forms the core of an Open University course on Knowledge Engineering, written in Prolog. Features include: forward/backward chaining rules with user-definable conflict resolution strategies, a frame representation language with inheritance, and automation of "how" explanations.

Collantes, Lourdes Y. (1992). *Agreement In Naming Objects and Concepts For Information Retrieval.* Graduate School-New Brunswick Rutgers, The State University of New Jersey.

This represents a very readable layman's book detailing current concepts and approaches for the standardization of naming objects across projects. In the author's view, this is critical for accurate information retrieval. The move from reuse to object-oriented technology is reviewed. The need for a standardized library of objects, one which can be shared, is discussed. Author cites several groups addressing the problem.

Comaford, Christine, (1995). *Stop obsessing over reuse - just do it already.* PC Week v12, n10 (March 11, 1995):20.

Comaford, a major contributor to PC Week, believes that developers spend so much time obsessing about the negative and positive aspects of reuse that they stop adding business value. The balance between meeting business needs and disciplined development is always tricky, she warns. Her advice is that IT management should not get obsessed. In order to realize the potential and cost effectiveness of reuse, just move forward and use it.

Constantine, Larry (1992*). Rewards and reuse. (the benefits of reusing software designs and models).* Computer Language v9, n7 (July, 1992):104.

The potential benefits of reuse of software components is the focus of this article. Author attempts to consolidate views and provide a clear and non-subjective assessment in a comprehensive report. Rewards associated with reuse specifically of modules of software design as well as models are detailed. Realization of the such benefits depends upon the degree to which new modes of reuse may be successfully substituted for selection.

Copley, M. G. (1994). An Expert System to Aid Probation Officers in Sentence Recommendation. *International Journal of Applied Expert Systems*, 2(1), 42-55.

Probation Officers are required to make sentence recommendations for some offenders. The decision-making necessary to make recommendations matches the criteria for expert system development. This paper describes an expert system that performs this task, points out the improvements that using the system makes possible, and suggests research and evaluation of the system's effects on its environment.

Crestani, F. (1994). Domain Knowledge Acquisition for Information Retrieval using Neural Networks. *International Journal of Applied Expert Systems*,2(2).

The results of experiments investigating neural network use in the learning engine of an CIRS information retrieval system are presented in this paper. CIRS uses the learning/ generalization capabilities of the Black Propagation algorithm to acquire and use application domain knowledge in the form of sub-symbolic knowledge representation. CIRS architecture is described. Experiments on three learning strategies are reported.

Cullen, J., & A. Bryman (August, 1988). The Knowledge Acquisition Bottleneck: Time for Reassessment? *Expert Systems*, 5(3), 216-225.

Examines resistance to expert systems technology which derives from the idea that expert systems would replace personnel. Focuses on knowledge acquisition problems, perceived as a major bottleneck to development. Survey concludes that knowledge acquisition viewed as difficult and time-consuming is unjustified. Attitudes based on inappropriate techniques. Acquisition accounted for only a small portion of development time.

Deng-Jyi, Chen; Lee, P.J. (1993). On the study of software reuse using reusable C++ components. Journal of Systems and Software v20, n1 (Jan, 1993):19.

Although the study of software reuse using reusable C++ components is the focus of this report, other problems are reviewed. It is noted that current object-orientation has largely been focused on o-o programming; more code is still created instead of effectively reusing existing objects and classes. Much C++ code takes no account of the concepts of object reuse. Management still tries to measure productivity by lines of code written.

Depompa, Barbara, & John Foley. (1996). *IBM to HELP Data Miners*. Information Week, 5743, 32.

This short article reports on the efforts of IBM to provide users with better tools to analyze information contained in company large data warehouses. IBM offers an object-oriented programming environment to simplify working with databases running on parallel-processing computers. Many believe o-o technology is the answer to reuse.

Dologite, D. G., & R. J. Mockler (1994). Designing the User Interface of a Strategy Planning Advisory System: Lessons Learned. *International Journal of Applied Expert Systems*, 2(1), 23-30.

Uses a framework of three central user interface issues as basis for examining knowledge- based expert system, the Strategy Planning Advisor (SPA), an experimental system intended to support managers with strategic business unit planning. User interface design, anchored in human factors research, doubled development time and cost. The authors report on the design process and lessons learned.

Due, Richard. (1995). The Economics of Reuse. *Information Systems Management*, 12, 70-78.

Reports on economics of reuse. Code reuse has been a failure at application level. Data reuse is limited at the enterprise/industry level. Techniques of software engineering designed to promote code reuse are sound; the problem has been with implementation and support. Concludes that designers should consider the reuse of existing designs and requirements instead of trying to promote code reuse. This shift in emphasis involves thinking about systems in terms of frameworks and patterns.

Edwards, Stephen H. (1996). Good Mental Models are Necessary for Understandable Software. Department of Computer and Information Science, The Ohio State University. (www.cis.ohio-state.edu)

Edwards believes conventional programming languages still do little to help programmers develop good mental models of software subsystems. Psychological insight has only been informally applied. He proposes that modules should not be merely syntactic units, but must ``mean" in the sense that they have denotations in the semantic framework that are not hierarchically constructed from meanings of implementations. He suggests a model for those who reason about interacting software parts collections during design.

Eichmann, David and Irving, Carl (1996). *Life Cycle Interaction in Domain/ Application Engineering*. Repository Based Software Engineering Program, Research Institute for Computing and Information Systems, University of Houston-Clear Lake. (rbse.jsc.nasa.gov).

The importance of life cycle interaction in domain and application engineering is reviewed. Designers are only now beginning to understand that by bringing software reusability issues to the first phases of the software life-cycle they can greatly enhance the impact of software reusability. In their view, by studying the designs of many systems within a particular application domain, programmers are finding out that designs are very reusable.

Endoso, Joyce (1994). *Look for a new version of the Defense Software Repository System. The Army Reus Center is converting its DSRS version 3.5.* Government Computer News v13, n6 (March 21, 1994):77.

Describes a new version of the Defense Software Repository Systems, an automated repository for storing and retrieving reusable assets. DSRS serves as a central collection point for quality assets and facilitates software reuse. It will now will support storage and retrieval of other than Ada-related products. It uses the ORACLE database management system and operates on the UNIX platform. DSRS will provide an on-line help facility, dependency information, session maintenance, and user suggestion facility.

Endoso, Joyce (1992). *Business issues impede software reuse. (includes related articles on standard Army Management Information Systems, software reuse terminology).* Government Computer News v11, n23 (Nov 9, 1992):1.

In Endoso's view, getting programmers to write good reusable code is an educational process. But besides technical problems to address, other connected issues must be considered: business issues of a military, political, legal, financial, and managerial nature. She outlines a number of general issues that continue to impede software reuse for Army MIS development. She includes related articles and suggests approaches to resolution.

Faris, C. (1995). *Reuse Initiative.* Andersen Consulting Home Page. (http://www.ac.com).

Farris discusses the Andersen Consulting Center, which focuses on developing technology solutions that enable reuse. Anderson now leads an effort to raise the level of reuse within the firm, using existing technology. Their goal is to provide as many leverage points as possible, to allow firms to step up to reuse in a planned manner, and to leverage existing knowledge on how reuse should best be accomplished. Four activities are described.

Fowler, F. J. (1984). *Survey Research Methods.* Newbury, CA: Sage Publications.

The author presents a thorough review of basic survey research methods. Included are descriptions of research methodology, testing procedures, sample population selection and basic survey techniques. Theory building, the research process, and measurement concepts are also discussed. This work provides support for various types of methodology such as that selected by the present study.

Gentle, C. R., O'Neil, M., & J. V. Sealey (1995). Nominal Group Technique as a Method of Knowledge Elicitation for Expert Systems: A Case Study Involving Assessment of Undergraduate Projects. *New Review of Applied Expert Systems*, 1, 54-67.

This was a case study to discover how Nominal Group Technique (NGT) can be used to gather data as part of knowledge elicitation involved in expert system development. Adopts methodology for interviewing large numbers of students and analyzes responses to project performance. Data were incorporated into an expert system for assessing progress. Concludes that NGT provides a simple method of gathering knowledge from large numbers of low grade experts and putting into a form directly suitable for expert systems.

Griss, M. L. (1993). *Software Reuse: From Library to Factory*. IBM Systems Journal, 32(4), 548-551.

The report provides a good, all-around discussion of software reuse issues and problems, as well as the problems associated with moves of standardized library components to the industrial environment, in terms of real-world application. Identifies the potential benefits to be realized from implementing reuse.

Grudin, J. (1990). Groupware and Cooperative Work: Problems and Prospects. In B. Laurel, *The Art of Human-Computer Interface Design*. Reading, MA: Addison-Wesley, 171-185.

The first half of the chapter describes problems that have led to expensive and repeated failures of GroupWare development efforts, after providing a thorough explanation of uses and reuses. In the second half, the author describes a groupware success story, which demonstrates the importance of focusing the analysis on the work setting. The chapter concludes that this focus provides a basis for speculating about the future.

Hajsadr, S. M., & A. P. Steward (1990). *An Approach to Knowledge Elicitation of Manufacturing Skills and Production Behavior in Industrial Environment.* Proceedings of UKIT90 Conference, IEE, London.

Reviews approaches to knowledge elicitation of manufacturing skills. Believes another use for expert systems arises from lack of communication between worker teams operating on different shifts. Approach advocates storing knowledge in the system from one shift and passing it to the next. Concludes prototype system using HyperCard with a hypertext interface has increased production behavior and efficiency in the industrial environment. Discusses other ways to elicit knowledge from experts.

Hodges, Julia E. and Cordova, Jose L. (1993). *Automatically Building a Knowledge Base through Natural Language Text Analysis.* International Journal of Intelligent Systems, Vol. 8, 921-938.

An excellent review of the need to build a knowledge base through natural language text. Pressure to provide this attribute gave rise to this report. States that current systems are expected to acquire natural human language; research has focused on systems that permit access to databases by queries posed in natural language. Restricting systems to a limited domain allows developers to simplify the linguistic processing problem.

Holden, P. (1992). Expert Systems in Manufacturing. Part I. *A User's Perspective on Expert Systems Innovation. Knowledge-Based Systems*, 5(2).

Surveys resistance to expert system technology in the manufacturing industry. Found that, although companies had an understanding of the benefits, many (60%) had not ventured beyond the provision of a prototype demonstration model. Of those not interested, many confessed a lack of awareness, believed expert system technology was of no use to them, felt it was too costly, or believed the problem of domain was too complex. Resistance from both experts and potential users was shown as another barrier.

Huff, Sid. (1993). *Object-Oriented Programming.* Business Quarterly, 58, 85-90.

Article discusses new method of building/maintaining computer software, object-oriented programming. Traditional view first explained: software development emulates traditional engineering work, emphasis is placed on standardized approaches, and software is reused wherever possible. But software is hard to reuse. Once a full library of objects has been defined, building software systems can be done very rapidly, but developing the library in the real world is a major undertaking.

Ignizio, James (1991). *Introduction to Expert Systems: Development and Implementation of Rule-Based Expert Systems.* New York: McGraw-Hill.

This book is concerned with knowledge-based models and their proper implementation from a decision-making perspective. Also covered are knowledge acquisition, inference, and validation. The work is especially good for students in fields other than computer science, such as business and engineering. There are exercises at the end of every chapter, clear and concise explanations, and good examples are also provided.

Johansen, R. (1989). User Approaches to Computer-Supported Teams . In M. Olson, *Technological Support for Work Group Collaborations* . Hillsdale, NJ: Lawrence Erlbaum Associates, 1-32.

Johansen provides a tour of seventeen different approaches to using computers and component program parts to support work teams, showing that the field of technological support for collaboration is still emerging. He shows how many seemingly unrelated tools can be labeled as team support. Predictions are made of how the field will develop, what its new and reused products will be, and who will be users and vendors.

Jones, Capers (1994). *Economics of software reuse.* Computer, v27, n7 July, 1994:106.

The author asks how much can be saved by using pre-existing or modified software components when developing new software systems. With the increasing adoption of reuse methods and technologies, this question becomes critical. Directly tracking actual cost savings is difficult. States that a worthy goal would be to develop a method of measuring savings indirectly by analyzing the code for reuse of components.

Klinker, G., Linster, M., & Yost, G. *Cooperative systems for workgroups.* IEEE Expert, 1995, 10, 37-44.

Central problems that arise when building cooperative expert systems are the focus of this report. Authors note recent shift from traditional expert systems to cooperative systems for workgroups. They discuss the need for consideration of the workplace's contextual information to create successful applications, rather than the development of applications to perform tasks in isolation. They conclude that traditional expert system development methods are insufficient to create effective cooperative systems.

King, James A. (1995). *Software reuse and knowledge reuse.* AI Expert v10, n4 (April, 1995):13.

A detailed comparison and contrast of software and knowledge reuse is provided by this author. Differences associated with software and knowledge reuse are explored in depth. The article provides a well-written assessment of both software and knowledge reuse, noting the benefits and drawbacks of reuse associated with each.

Kirk, R. E. (1978). *Experimental design: Procedures for the Behavioral Sciences.* Belmont, CA: Wadsworth.

Major components of experimental design are discussed in detail in this book. Represents a comprehensive examination of survey and experimentation. Provides numerous examples of all phases of design, including: types, research steps, population samples, and other aspects of method. Included are explanations of various statistical and quantitative approaches and techniques. Statistical formulae and application reasons are reviewed.

Maiden, Neil A.; Sutcliffe, Alistair G. *Exploiting reusable specifications through analogy. (Computer-aided software engineering tool support for software specification reuse).* Communications of the ACM v35, n4 (April, 1992):55

Problem-scooping is an important concern. The development of computer-aided software engineering tool support for reuse has focused on knowledge-based CASE tools. Little thought has been given to the practical problem of initially eliciting such information. Analogical specifications are discussed in both technical and methodological terms. The authors believed it can provide relevant domain models with similar boundaries to assist problem-scooping. A report was provided with an in-depth definition of an intelligent reuse advisor (Ira). Accompanying examples also included in the report was the problem identifier, the analogy engine, and the specification advisor.

Mili, Hafedh; Radai, Roy; Weigang, Wang; Strickland, Karl; and others. *Practitioner and Softclass: a comparative study of two software reuse research projects.* Journal of Systems and Software v25, n2 (May, 1994):147

The article provides an excellent discussion for software reuse issues as well as problems associated with the technical aspects. It includes a detailed examination of reuse methods and technologies. Some of the topics are building reusable software, repackaging existing software (to make it more readily reusable), and providing support for software development with reusable components. The technical aspects among all other surrounding factors in the development of software reuse is equally important to its success.

Montgomery, George (1992). *Matermind: Improving The Search.* AI Expert, April 1992, 41-47.

This paper illustrates an in-depth technical search into the mastermind game. It focuses on the searching problems the game represents and shows how to minimize number of plays needed to finish. An assessment of the impact of a state-space search and examples as well as definitions will be provided. Some of the major points will be organizing the state-space into a tree hierarchy, exploring state-space size, transforming a parent node into its child nodes, and defining suitable predicates for testing branch nodes.

NASS-38035 (1994). *Automated Information Mining of Large Software Collections for the Extraction of Reusable Code.* NAS5-38035, July 21, 1994.

The technique of information mining - the search for relationships and global patterns that exist in large databases but are hidden among the vast amounts of data - has enjoyed a recent resurgence of interest as databases grow increasingly larger in today's companies. The report discussed the techniques of automation pertinent to large software collections for the purpose of reusable code extraction. It explains major problems and the need for intelligent search strategies.

Perry, William E. (1992). *For DOD software reuse to succeed, it must be easy.* Government Computer News v11, n22 (Oct 26, 1992):22.

The Department of Defense (DoD) Center for Software Reuse Operations is pursuing a comprehensive reuse initiative. This is a direct result of rapid growth in software programs and increasing developmental costs. According to the Perry, the DoD center has a collection of 1,531 reusable software modules that contain 2.2 million lines of Ada and Cobol code. The reuse center provides 50% of programmer needs from reusable elements. The best part of the article is when the author suggests that the search system should allow key words and phrases search. This will enable users to identify a small group of modules to satisfy specific needs.

Seybold, Patricia (1993). *The road to reuse. (advantages to reusing software).* I.T. Magazine v25, n6 (June, 1993):12.

Reuse methods can be very appealing due to the potential for significant cost savings although not all businesses are adopting this technology. An author suggests in promoting reuse software, one must show how it will reduce corporate computing costs. A technique which must considered in development is building software systems from common reusable components. The best part of the article comes in with a suggestion the modules should be able to link to other modules to create new applications. An intelligent system must exist in these modules to assist in interface.

Appendix A

Numbers Of Modules From GAMS Are Provided By Boisvert Who Was Personally Contacted On 10/14/97.

From: Dr. Ronald F. Boisvert
Leader, Mathematical Software Group
Editor-in-Chief, ACM Transactions on Mathematical Software

Date: Tue, 14 Oct 97 08:07:43 EDT
From: boisvert@cam.nist.gov (Ronald F Boisvert)
Message-Id: <9710141207.AA04779@fs3.cam.nist.gov>

Package	Number of Modules
A	16
AMD	6
AMOS	16
BESPAK	1
BIHAR	12
BLACS	4
BLAS	24
BLAS1	42
BLAS2	66
BLAS3	30
BMP	1
C	8
CBLAS	136
CLAPACK	598
CMLIB	739
CONFORMAL	5
CONTIN	2
COULOMB	1
CRAYFISHPAK	23
DATAPAC	169
DATAPLOT	87

DERIV	1
DIERCKX	29
DIFFPACK	3
DISSPLA	1
EISPACK	70
ELEFUNT	20
ELLPACK	7
ENVELOPE	1
F90GL	1
FFTPACK	19
FISHPACK	19
FITPACK	1
FN	187
FORMAT	1
FORTRAN	2
FP	3
GO	12
GRAPHICS	5
HBIO	1
HOMPACK	7
IML++	1
IMSLM	1049
IMSLS	752
ITPACK	4
JAKEF	1
JCAM	4
LANZ	1
LAPACK	598
LASO	4
LINALG	23
LINPACK	176
MA28	7
MACSYMA	1
MANPAK	2
MAPLE	1
MATHEMATICA	1
MATLAB	1
MINPACK	11
MISC	16
MPFUN	4
MV++	1
NAG	2148
NAPACK	140
NASHLIB	19
NCAR	1

NLR 4
NMS 52
NSPCG 1
ODE 30
ODEPACK 6
ODRPACK 2
OPT 13
PARANOIA 6
PDELIB 3
PDES 2
PLTMG 2
PORT 659
QUADPACK 58
RANDOM 5
SAS 40
SCALAPACK 4
SCILIB 169
SCRUNCH 9
SEISPACK 70
SLATEC 899
SMINPACK 11
SODEPACK 6
SPARSE 1
SPARSE-BLAS 4
SPARSELIB++ 1
SPBLASC 1
SPECFN 3
SPECFUN 16
SPM_MORPH 1
STARPAC 145
STOPWATCH 1
TEMPLATE 1
TEMPLATES 6
TOMS 268
TRANSFORM 1
VANHUFFEL 3
VECLIB 118
VFFTPK 13
VFNLIB 16
VOLKSGRAPHER 1
VORONOI 2
Y12M 3

Appendix B

Reusability Tally Sheet:

Components Classified by the Researcher as Reusable

System Number: _____

Size and Number of Components Selected: _____

Specific System Component Selected:_____

1. There are not too many answers to a question. ___Yes ___No

2. There are not too many questions. ___Yes ___No

3. A shortcut has been provided. ___Yes ___No

Amount of time it took to retrieve the selected component for each system identified above.

Time:_____

Appendix C

Recording Sheet:

Components Classified by the Guides-Search as Reusable

System Number: _____ (identify the system number and name here)

Size and Number of Components Selected: _____ (list size and number)

System Component Selected:_____ (name the component)

Time taken by Guides-Search to retrieve the selected component for each system identified above.

Time:_____