UC Irvine
ICS Technical Reports

Title
A software classification scheme

Permalink
https://escholarship.org/uc/item/78j2r3c4

Author
Díaz, Rubén Prieto

Publication Date
1985

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

A Software Classification Scheme

Rubén Prieto–Díaz

Technical Report 85–19

Information and Computer Science
University of California
Irvine, California 92717

1985

Contents

List of Tables v
List of Figures vi
Acknowledgments vii
Abstract viii
Chapter 1: Introduction 1
General Problem 1
Why Classify Code Fragments? 4
Plan of Presentation 6
Chapter 2: Code Reuse 8
Motivation 8
Use of Existing Software Inventories 8
Improve Software Productivity 9
Reusability 12
Reusability in Established Disciplines 12
Reusability in Software 16
Characterization of the Reusability Problem 25
Models of Code Reuse 26
Functional Collections 26
Module Interconnection 28
Code Fragments 29
Proposed Model 31
Problems in Code Reuse 35
Large Collections 35
Component Descriptors 36
Retrieval 37
Program Understanding 38
Selection 39
Adaptation 40
Conclusions 43
Chapter 3: Classification Related Work 44
What is Classification 44
Syntactical and Hierarchical Relationships 46

Enumerative and Faceted Schemes 48
Citation Order 49
Notation 52
Use of a Classification Scheme 53
Making a Faceted Scheme 54
Summary 54
Library Classification 55
Dewey Decimal Classification 57
Library of Congress 60
Universal Decimal Classification 62
Bibliographic Classification 64
Colon Classification 66
Summary 68
Software Classification 70
Computer Program Libraries 71
Software Catalogs 74
Summary 76
Conclusion 77
Chapter 4: A Classification Scheme 78
Flexible Facets in Software Classification 78
A Formal Classification Model 80
Component Descriptors 82
Measuring Relevance Using Facets 84
Measuring Closeness Between Attributes 86
Example 89
Discussion 93
Chapter 5: Classification of Software Components 95
Descriptor Synthesis 95
Functionality 98
Environment 100
Schedule Construction 102
Vocabulary Control 107
Implementation 110
Computing Conceptual Distances 115
Conclusions 116
Chapter 6: Evaluation of Selected Components 117

Reusability Related Attributes . . . 119
Program Size . . . 120
Program Structure . . . 123
Documentation . . . 126
Programming Language . . . 130
User Experience . . . 135
Multi-dimensional Evaluation . . . 137
Dimensional Normalization with Fuzzy Sets . . . 140
Fuzzy Logic Concepts . . . 141
Fuzzy Modifiers . . . 143
Conclusion . . . 148
Chapter 7: Library System Evaluation . . . 150
Retrieval Effectiveness . . . 151
Making the Key . . . 152
Retrieval Evaluation . . . 154
The Effect of Citation Order . . . 156
The Effect of Conceptual Ordering . . . 158
Experiment Conclusion . . . 159
User Classification . . . 160
Classification Difficulty . . . 162
Classification Consistency and Accuracy . . . 166
Test Results . . . 167
Reuse Effort Estimation . . . 167
Experiment Results . . . 172
Conclusion . . . 174
Chapter 8: Summary, Future Work, and Conclusions . . . 176
Summary . . . 176
Recommendations for Future Work . . . 180
Suggested Problems . . . 181
Testing the Prototype in a Real Environment . . . 182
Conclusions . . . 184
References . . . 185
Appendix A: A Preliminary Schedule for Application Programs in Communication and Media . . . 195
Appendix B: Partial Classification Schedules for Software Components . . . 197
Appendix C: A Classification Example . . . 204
Appendix D: Sample Library System Entry . . . 206

List of Tables

4.1 An Ordered List of Descriptors 92

6.1 A Programming Language Closeness Matrix 134

7.1 Summary Programs for Classification Test 161
7.2 Reuse Metrics of the First Experimental Sample 170
7.3 Ranked Components for First Sample 170
7.4 Reuse Metrics of the Second Experimental Sample 172
7.5 Ranked Components for Second Sample 172

List of Figures

2.1 An RC Circuit 14
2.2 Tension in Software Reusability 18
2.3 A Simplified Physical Model of Reusability 23

3.1 A Demonstration Faceted Scheme [BUCH79] 49
3.2 A Demonstration Enumerative Scheme [BUCH79] 50

4.1 Measuring Conceptual Distance in a Tree Hierarchy 86
4.2 Measuring Conceptual Distance in a DAG 87
4.3 Labeling the Nodes in a DAG Hierarchy 88
4.4 A Partial Conceptual Weighted Graph for the Function Facet 90
4.5 A Partial Conceptual Weighted Graph for the Objects Facet 91

5.1 A Preliminary Faceted Schedule for Compiler Components 104
5.2 SADT Context Actigram for The Library System 111
5.3 SADT Level 0 Actigram for The Library System 113

6.1 Relative Closeness to Target Language Pascal 135
6.2 Two Hypothetical Utility Functions 138
6.3 Two Reuse Effort Utility Functions 139
6.4 Fuzzy Functions for Young and Old with Some Modifiers [ZADE84] 143
6.5 Reuser Experience as Modifier for Small Component 144
6.6 Programming Language as Modifier for Small Component 145
6.7 User Experience as Modifier for Attribute Functions 145
6.8 Membership as Function of Reuser Experience 147

7.1 Recall and Precision in Retrieval 156
7.2 Use of Recall and Precision to Compare Retrieval Systems 157
7.3 Specification for a Simple Payroll Program 171

Acknowledgments

Going through graduate school is analogous to a climbing expedition. The doctoral candidate is the climber selected to reach the summit. However, the feat cannot be accomplished without the support of the other members of the expedition. This support may take the form of route planning, infrastructure logistics, or stimulating friendships. All this is also true of a Ph.D. program, and this thesis could not have been possible without the support and friendship of many people. All of them deserve more than just an acknowledgment.

I am indebted to my advisor Peter Freeman for his guidance and support during my years of study. His patience has been invaluable to the success of this work. I would like to thank my committee, Rob Kling and Dennis Kibler, for their support and reading of my work and for their valuable comments and suggestions.

My special thanks to my colleagues of the REUSE Project, Guillermo Arango, Ira Baxter, and Chris Pidgeon for their persevering encouragement, intellectual nurturing, and elightening discussions. Their comments and ideas through the slow refinement of the several drafts of this thesis were most worthy.

The enduring moral and personal support of my wife Haydée was of utmost importance. She was always behind me and ready to help in the most difficult times and happy to celebrate any successes. Thank you.

Finally, I would like to thank Joan Isenbarger for her help in proof reading the thesis, Frank Murgolo for his comments on formal notations, and the graduate students of the ICS Department that participated in the classification experiments.

This work was supported by the National Science Foundation under grant MCS-83-04439 and by the Consejo Nacional de Ciencia y Tecnología, México.

ABSTRACT

Reusing code is one approach to software reusability. Code is the end product of the software lifecycle. It is delivered in a low level representation that is difficult to reuse unless an almost perfect match exists between available features and required specifications. There is a need to organize large inventories of software such that reusable code is easy to locate and exchange. The relative success in the reuse of code fragments reported by some software factories is due in part to their capacity to encapsulate domain specific functions and create specialized libraries of components classified by these locally standardized functions.

A general software classification scheme that organizes reusability related attributes and common functions from different domains is proposed as a partial solution to the software reusability problem. For the problem of selecting from similar, potentially reusable components, a partial solution based on evaluation of common characteristics is also proposed. A library system is presented that integrates the proposed classification scheme with an evaluation mechanism based on inherent component attributes, programming languages characteristics and reuser experience.

The fundamental contribution of this dissertation is a formal treatment of a faceted scheme for software classification leading to better understanding of reusability at the code level. This approach has been prototyped in a library system for the semi–automatic classification of software components. Analysis were performed to evaluate the classification scheme. The results show the potential of the scheme in organizing collections of code fragments, in improving retrieval, and in simplifying the classification process. Tests of the evaluation mechanism showed positive correlation with evaluations conducted by potential reusers.

Abstract of the Dissertation

A Software Classification Scheme

by

Rubén Prieto Díaz

Doctor of Philosophy in Computer and Information Science

University of California, Irvine, 1985

Professor Peter Freeman, Chair

Reusing code is one approach to software reusability. Code is the end product of the software lifecycle. It is delivered in a low level representation that is difficult to reuse unless an almost perfect match exists between available features and required specifications. There is a need to organize large inventories of software such that reusable code is easy to locate and exchange. The relative success in the reuse of code fragments reported by some software factories is due in part to their capacity to encapsulate domain specific functions and create specialized libraries of components classified by these locally standardized functions.

A general software classification scheme that organizes reusability related attributes and common functions from different domains is proposed as a partial solution to the software reusability problem. For the problem of selecting from similar, potentially reusable components, a partial solution based on evaluation of common characteristics is also proposed. A library system is presented that integrates the proposed classification scheme with an evaluation mechanism based on inherent component attributes, programming languages characteristics and reuser experience.

The fundamental contribution of this dissertation is a formal treatment of a faceted scheme for software classification leading to better understanding of

reusability at the code level. This approach has been prototyped in a library system for the semi-automatic classification of software components. Analysis were performed to evaluate the classification scheme. The results show the potential of the scheme in organizing collections of code fragments, in improving retrieval, and in simplifying the classification process. Tests of the evaluation mechanism showed positive correlation with evaluations conducted by potential reusers.

CHAPTER 1

Introduction

General Problem

The goal of this dissertation is to propose a classification scheme for code fragments that permits easy location and exchange of reusable code. One of the essential problems in reusing code fragments is to be able to locate an appropriate piece of code that is stored in a large collection of fragments. Code reuse in this context is the use of an existing code fragment as a component for a new software system. Reusability implies the process of modifying or adapting an existing code fragment into a new system.

A classification scheme is a domain knowledge structure that organizes collections of items to satisfy the needs of the users of the collections. In the domain of code fragments, a classification scheme should organize the collection to satisfy the needs of the reusers. It is necessary to identify the factors involved in the code reuse process and use them in designing a scheme that will maximize the utility of the collection.

In order for code reuse to be attractive the overall effort of reusing code must be less than the effort of creating new code. The code reuse process involves three steps: access the code, understand the code, and modify the code. A classification scheme is central to code accessibility, code understanding depends on both reuser experience and program characteristics, and modification effort is related to the difference between requirements and features offered by existing components.

Code understanding effort and modification effort can be reduced by proper classification of the collection. Understanding a group of very similar code fragments

implies relatively less effort than understanding a group of different code fragments. Once the effort is invested to understand one of the group, the effort required to understand the remaining members belonging to the same class is minimal assuming a certain degree of similarity between members of the same class. If the collection is organized by attributes used to define software requirements, then the probability of retrieving components that are different from the given requirements is very small. This reduces modification effort.

Classification and organization of the collection is a necessary condition to make code reusability an attractive approach to the software development process. However, an organized collection is of no use if it does not provide the mechanisms to use it. A search and retrieval mechanism is necessary to access the collection and a well defined classification structure is essential to the design of an effective retrieval system. If the retrieval system has knowledge about the collection, its retrieval performance is improved. Another consideration is that the classification structure–based retrieval system just proposed (call it library system) needs to help its users discriminate among very similar items in the collection.

A typical retrieved sample may consist of several very similar components differing only in minor implementational details. A reuser is faced with the problem of selecting the components that would require the least adaptation effort. An inspection of each element in the sample is needed to select the best. This process alone may discourage a potential reuser from using the collection, especially if the number of elements in the sample is large. A system to conduct an evaluation of very similar components is therefore needed to assist reusers in selecting the components that would require the least conversion effort.

A proper classification scheme requires, therefore, an integral (holistic) view of the problem of code reusability. All factors involved during the code reuse process must be considered.

The contribution of this dissertation is the derivation of a software classification scheme as the result of a comprehensive analysis of the code reuse problem. This approach calls for an integrated solution—a classification scheme embedded in a retrieval system and supported by an evaluation mechanism.

The main features of the proposed classification scheme are expandability, adaptability, and consistency. Expandability means that new classes can be added with minimal disturbances to the present collection, that is, with a minimum of reclassification problems. Adaptability means that the scheme can be customized to a particular environment. Consistency means that components from different collections in the same class share the same attributes. This feature will allow different organizations to share their collections.

This holistic approach resulted in a prototype library system that was used to evaluate the proposed classification scheme in particular and the code reuse problem in general.

The approach taken by the author was to study the problem of code reuse and survey the domain of classification in order to identify feasible classification schemes that could be applied to software. A special scheme for software code fragments resulted. Next, the domain of software metrics was researched to identify reuse related metrics that could be used to estimate reuse effort. An evaluation technique to compare similar components based on reuse related attributes was selected. The classification scheme was integrated with the evaluation technique into a prototype system built around a data base management system and including a retrieval front end. The resulting prototype library system was then tested and evaluated.

Why Classify Code Fragments?

In 1968, at the NATO Software Engineering meeting in Germany, McIlroy [McIL69] proposed the idea of a software components catalog from which one could order software parts which could be assembled, much as we do mechanical or electronic components.

> Software components, to be widely applicable to different machines and users should be available in families arranged according to precision, robustness, generality and time space performance. . . . software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical designs are abetted by the existence of families of structural shapes, screws or resistors.

It is believed that this idea and similar approaches taken by the software publication industries have not flourished because individual concrete programs are too specialized to be reused in most cases—they contain too many detailed representational choices to be adaptable to new circumstances of potential use.

Other researchers [PRYW79, NEIG80, SWAR82] proposed effective methods for expressing software abstractions and for generating particular instances by transformation or refinement. This most recent approach, however, is considered a long term alternative to the software reuse problem in particular [FREE83], and to the software development process in general [BALZ83].

As a short–term alternative to improving the software development process, reuse of software components from available collections has shown some success. Lanergan and Poynton [LANE79] at Raytheon report significant gain in productivity by reusing available components. They identified and classified a large number of pieces of code and a few standard control schemes that could be used in many of their applications. They established libraries and tools to facilitate the reuse of these code fragments, management procedures to encourage and control their reuse,

and training procedures to teach new programmers how to use this approach. They report reusability figures on the order of 60%.

Following a similar approach, Japanese software factories report high improvement in programmer productivity through an integration of known techniques from different disciplines like resource management, production engineering, quality control, software engineering, and industrial psychology [MATS80, KIM84, TAJI84]. A significant part of this productivity improvement is due to code reuse. Frequently used programming structures, including parametric parts, are standardized and cataloged in libraries as "reusable pattern modules." These structures can be recalled and combined with parametric variables and other pattern modules to generate new programs.

These cases, although very successful, are limited in number. They deal with very specialized domains of applications where certain programming structures are readily identified and easily standardized. Relatively few categories of programs are needed to fill the small number of programming structures resulting in easy–to–access collections.

One of the factors preventing a wider use of software component collections in software development is the problem of locating the components within the collection. When the number of program categories increases along with the size of the collection, collection access becomes difficult. A specialized classification scheme is necessary to organize a collection for the needs of the users thus facilitating the location of the appropriate components. The resulting classification scheme must be easily expandable to support growing collections, it must be adaptable to be used by different organizations working on different domains of applications, and it must be consistent to serve the needs of a broad variety of users.

Another important aspect of classifying software components is its potential in identifying what kinds of programs recur sufficiently often that it would be advisable

to develop reusable versions of them. Work by Goodell [GOOD83] from Burroughs Corp. is the first reported in classifying components for this purpose. The need to classify existing collections has recently been recognized by some industries[1]. Their interest includes both direct reuse and component standardization.

Alan Perlis expressed clearly to the author[2] the need for a software classification scheme. He indicated that a classification of software is necessary to properly address the problem of software reusability. It should identify the software attributes that are considered during reusability.

Plan of Presentation

Chapter 2 motivates the reuse of code as an approach to improve software productivity. It contrasts reusability as conducted in other disciplines with reusability in software. A detailed analysis of the software reuse process is conducted and a model is presented that provides a more flexible approach to code reuse.

Chapter 3 surveys classification related work, in particular, classification in library science and the existing schemes used in software classification. An outline on how to construct a faceted scheme is presented indicating the advantages of this approach to classifying code fragments.

Chapter 4 presents the development of a coherent scheme for organizing descriptive terms for classifying and retrieving of software components. The advantages of using a flexible–facet classification scheme are discussed, and some examples are presented to illustrate the use of the scheme.

Chapter 5 describes how the classification scheme is implemented to classify software components, how to start a collection of software component descriptors,

---

[1] In particular, Intermetrics Corp. of Huntington Beach, Ca. and GTE Laboratories Inc. from Boston, Ma.

[2] Personal communication, Irvine, 1983

and how the scheme and collection are implemented in a library system.

Chapter 6 presents a detailed analysis of selecting the appropriate component after retrieval. The inherent attributes used to discriminate among similar components are defined. These same attributes are used to estimate reusability effort. An approach to evaluate similar components is introduced. The approach is based on attribute weighting and dimension normalization using concepts from fuzzy logic.

Chapter 7 reports on an evaluation of the proposed classification scheme and the prototype library system. Tests on retrieval effectiveness, on classification effectiveness, and on reuse effort estimation were conducted to demonstrate the potential of the approach. The results of these tests confirm some of the arguments that were used as a basis for the construction of the library system.

Chapter 8 concludes the dissertation with a summary of the contributions of this research. Limitations of the proposed approach are discussed with some suggestions for future research.

CHAPTER 2

Code Reuse

This chapter offers reasons for the reuse of code. It contrasts reusability as conducted in other disciplines with reusability in software. Reusing software components involves certain variables not considered when reusing physical artifacts. Besides internal characteristics, reuser skills and environment may determine the degree of reusability of a software component. Deficiencies in some successful models of code reuse are discussed, and a model is presented that provides a more flexible approach to code reuse. Some of the problems in implementing this model are discussed in the final section.

Motivation

There are at least two major reasons of a strong economic nature. One is the need to take advantage of existing software inventories in the construction and development of new software systems to improve software productivity. The second is the need, specially in the software industry, to improve quality. In both cases code reuse is a key factor. This section discusses both reasons.

Use of Existing Software Inventories

Software has been recognized as a very valuable asset. It is an owned resource that contributes to the means of production. It is costly to acquire and even more costly to replace. The typical size of a software inventory for Fortune 500 corporations is on the order of tens of millions of lines of code [LYON81]. At an average cost of $10 per line of code, that brings the investment bill close to the billion dollar mark.

According to Morrisey and Wo [MORR79], six languages (Cobol, RPG, Fortran, PL/1, Basic, and APL) account for 90% of these inventories, and Cobol alone accounts for over 1/2 of this figure. Most of that available code was produced during the early stages of software development technology and has been considered untouchable because of its obscure structure and unreliable documentation.

There is a trend, among large corporations in particular, to organize these inventories of programs for reuse in the construction of new systems. The use of techniques such as *Structured Retrofit* [BALB75, GOME79, MILL80, LYON80] to automatically improve the structure of existing programs is stimulating the conversion of old inventories into better structured and documented programs that are easy to understand, maintain, and, above all, modify. This trend, although motivated mainly by maintenance needs, shows an enormous potential of reusability.

As more potentially reusable code becomes available, either by retrofit of old inventories or by generation of new software systems, new approaches to organize large collections of components for effective classification, cataloging, and retrieval are needed. This need creates a fertile ground for research in code reusability and software classification.

Improve Software Productivity

Another source of motivation for the reuse of code is the strong economic need to reduce the cost of software development. Some of the major reasons for the extraordinary growth in the cost of software are the growing complexity of software systems, the increased demand for qualified software professionals, and the lack of appropriate software development tools and methodologies.

A bottleneck exists in the software development process caused mainly by the slow rate of improvement in the software creation process. Morrisey and Wo [MORR79] report an increment in the software production process of only 3% to

8% per year during the last 20 years while the total installed processing capacity has increased at a rate greater than 40% per year.

The future does not look any better. Software demand is increasing by at least 12% per year while the supply of software professionals is increasing at about 4% per year [BOHE83]. The current shortfall between the supply and demand of programmers is estimated at 50–100,000 and it may rise to 1.2 million by 1990 [STAN83]. There is a definite need to improve software productivity. Reusability is one of the alternatives being explored. The potential of reusability in improving the software production process is significant.

> . . . we believe [the reuse concept] has the potential for increasing software productivity by an order of magnitude or more. [HORO83A]

Significant improvement in software productivity by enforcing code reuse has been reported by software factories. Software Factories [LANE79, MATS80, MACN83, KIM84, TAJI84] show reusability figures on the order of 50% to 85% and report substantial improvement in development time, reduced cost, increased product reliability, and more efficient use of manpower. Management strategies, quality control, and special training are used as supporting factors for achieving these figures.

Functional collections have proved successful in improving programming productivity of non–programmers in certain domains of application, in particular, in the field of scientific programming where problems are usually well–defined and encapsulated. Functional collections are large groups of functionally cohesive components designed for the solution of very specific problems. Although functional collections are not usually utilized by programmers or software designers to construct software systems, they are used by engineers, scientists, and technicians as tools to solve domain specific problems. As problem solving tools, functional collections have significantly improved the productivity of non–programmers. SPSS,

for example, is considered an essential tool in statistical problem solving. Courses on how to use SPSS are taught in most American universities.

Software development systems that rely on the Module Interconnection Language (MIL) paradigm have shown limited success in the reuse of code. MILs have been successfully integrated in some research and industrial software development systems [PRIE82]. Some MIL–based environments such as the GANDALF System [HABE81] are capable of generating complete systems out of available components by specifying, through a MIL, how to "knit" them together. The GANDALF system demonstrates the potential of the MIL paradigm in the generation of new systems from available modules.

Recent proliferation of commercially available software packages is another indication of the trend for using available code instead of developing in–house systems. The software publishing industry has experienced a remarkable growth in the recent years. Software directories for application packages have been the main contributors.

As indicated in a recent study on Reusable Software Implementation Technology conducted at Hughes Aircraft Co. [GRAB84], current software development methodologies, unfortunately, do not emphasize the reusability paradigm. In this study 19 current software development methodologies were evaluated. The methodologies evaluated covered a cross section of three categories: research, commercial, and industrial. The first three conclusions of this study are:

1. No credible methodology was examined which purported to provide the reuse of source code between dissimilar application areas. In fact, where source–code reuse occurs at all, it happens within narrow application areas.
2. None of the methodologies used in large–scale development efforts provide a reliable way of storing and retrieving items from a code–level library. Some methodologies were able to implement libraries, but the retrieval of

the correct item from the library was a manual process which was often so difficult that it was easier to code a new item than to look for one to reuse.

3. Within industry, the prime means of reusing software products is via the reuse of the personnel who created the products. Much of the knowledge which advanced methodologies attempt to capture is already resident within these knowledgeable personnel.

Motivation for code reuse can be summarized in the following points:

- Know–how on code reuse is needed to reuse current software inventories.
- Code reuse has proven to be, in some areas, an effective technique to improve software production.
- There is a need to improve current code reuse techniques.
- There is evidence that, even in 'state of the art' software industries, code reuse is conducted informally.

Reusability

In this section we address the following questions: What is reusability? How is reusability conducted in other disciplines? What is reusability in software? Two trends in software reusability research are described, and a characterization of the software reusability problem is presented.

Reusability in Established Disciplines

Reusability is usually construed to be the use of previously acquired concepts or objects in a new situation that is similar to the one that triggered the original use of those concepts or objects. Two problems arise when a situation of potential reusability appears: 1) how to recognize when the new situation is similar to a previous situation and 2) how to modify the previously acquired concept to fit the new situation. Answering these two questions is beyond the scope of this thesis. They are mainly related to problems of learning, knowledge representation, and

psychology. There are, however, two concepts that attempt to simulate the answering of these questions. The concept of storage of past situations or facts, known as knowledge base, and the concept of search and retrieval of stored situations to match and explain a new situation. Reusability could be considered as the process of matching new situations to old situations, and, if the matching succeeds, duplicating the same actions in the new situation.

This is a very general view of reusability. Terms like experience, practice, or skills could be identified with the term reusability. Even if we restrict reusability to objects or concepts that are the result of an activity, the term remains very general.

> The term *reuse* has the connotation that one may use a workproduct in a situation other than the original one for which it was created with less effort than would be required to create a new workproduct. [FREE83]

Reuse then, may have several interpretations as Wegner [WEGN83] illustrates with the following analogy:

> We have as many words for different kinds of reusability as there are words for different kinds of snow in the language of the Eskimo.

Regardless of its generality, reusability implies that knowledge has been coded at different levels of abstraction and stored for future reuse.

In well established disciplines like civil or electrical engineering, reusability is based on the existence of previously coded knowledge. There are two different levels of reusability to consider: the reusability of ideas or knowledge and the reusability of particular artifacts or workproducts either as new components or as complete modified assemblies. In civil engineering, for example, reuse of ideas consists of applying general engineering concepts such as stress analysis, truss analysis, or mechanics and applying standard design equations to particular problems like determining the dimensions and materials of a beam, a column, or a wall. An example of the reuse of particular workproducts, on the other hand, would be determining from a set

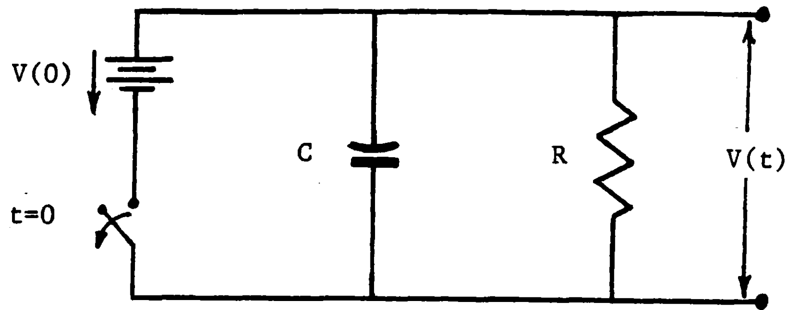


Figure 2.1: An RC Circuit

of standard beam shapes, cross sections, and materials what would best meet the established design criteria.

A design is typically based on availability of standard components. Electrical engineers, for example, consult component catalogs continuously during the design process. Before making a design decision, they check what available part best fits the design constraint. In most cases, the original design requirements are modified to take advantage of existing components.

The example below illustrates the typical design process. Assume the design specification of an RC circuit, like the one shown in figure 2.1, is given by:

$$V(t) = 5\ e^{-t/3.25\times10^{-6}}$$

The general equation for that RC circuit is given by:

$$V(t) = V(0)\ e^{-t/RC} \tag{2.1}$$

$V(0)$ is the initial condition and the product RC is called the time constant τ. The design asks for a circuit that damps an initial 5 volts load to its RMS value (36.8%) in 3.25 microseconds. Assume that the catalog only lists the following components

for available resistors (R) and capacitors (C):

	R		C
a)	$1.5\,\Omega$	A)	$2 \times 10^{-6}\, f$
b)	$1 \times 10^{3}\,\Omega$	B)	$17.5 \times 10^{-6}\, f$
c)	$.2\,\Omega$	C)	$.5 \times 10^{-3}\, f$

The designer is now faced with a selection problem. There are only two feasible combinations of R and C that come close to the design requirement of $\tau = 3.25 \times 10^{-6}$ sec.: a–A and c–B. Combination a–A results in $\tau = 3 \times 10^{-6}$ sec. Combination c–B results in $\tau = 3.5 \times 10^{-6}$ sec. Now the designer has to go back to the drawing board and 'make a concession' on the design requirements. The 'concession' is based on which of the two alternatives has the best trade–off on the requirements of the problem: a longer τ or a shorter τ. In real design problems, the number of alternatives is usually large. Several combinations of components may give feasible solutions, and the designer normally increases the number of design requirements to reduce the number of alternatives.

From this example we can make the following observations:

- Design constraints are usually modified.
- Design implementation becomes a selection problem.
- Components are usually acquired rather than created.
- Components are described by standard units that capture their functional characteristics.

In the next section, these observations are considered when reusing code.

Designing with standard components has been practiced more in physical assemblies where the creation of customized parts is expensive. In software systems construction, standard components have not been widely used mainly because of the generally accepted belief that customizing software is not as difficult as customizing material objects. This may be true as long as the components remain relatively

simple. When complexity of the component is considered, the software designer can create or reuse. In the case of physical artifacts, the reuse option is usually assumed.

Returning to the reuse of ideas, the circuit example also illustrates the reusability of ideas or concepts. Equation (2.1) is the solution to the differential equation

$$C\frac{dV}{dt} + \frac{V}{R} = 0$$

with initial condition $V(0)$ at $t = 0$. This equation is a mathematical model of the physical circuit and is considered the standard representation of RC circuits. Engineers use the solution for this equation and plug in the required values rather than solve the model equation each time. To reuse this concept, the designer must search in catalogs, books, or the designer's own knowledge base (memory) for the RC circuit solution formula given by equation (2.1).

Reusability in Software

In software engineering, as in civil or electrical engineering, there are two views of reusability: the reuse of software components, analogous to the reuse of artifacts in the other disciplines, and the reuse of software workproducts, equivalent to the reuse of established models or procedures discussed above. A software component usually refers to an executable image of a module or body of code or to a module or body of code written in a high level–language which can be compiled into an executable image. In this thesis, we use code fragment and software component interchangeably. A software workproduct, on the other hand, is considered to be any piece of software, code, or documentation.

The reuse of components in software is considered the "typical" [FREE80] view of reuse, and the reuse of workproducts is considered the "new" view of reuse. The typical view of software reuse is that of reusing executable programs from

available collections. Back in 1968, McIlroy [MCIL68] proposed the creation of a "software components manufacturing facility" to generate standard interchangeable software components. This idea is still very attractive and continuously mentioned as feasible [BOWL83, ICHB83, STAN83] but very little has been accomplished to date. The high potential for variability in implementation and performance of the usually complex software components has precluded their use as standard components in the construction of software systems. Another contributing factor to this impediment is the nature of software. It is relatively easy to modify, in contrast with physical objects, thus tempting the implementor to make rather than reuse.

The new view of software reusability may be defined as:

> . . . use of non–executable workproducts (e.g. a requirements definition, a design, a test plan) in the lifecycle of a piece of software other than the one for which it was originally produced. [FREE80]

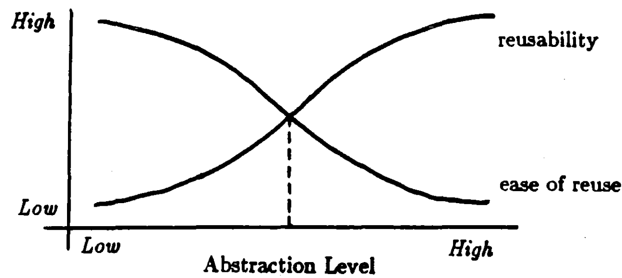
According to this view, the objects of reuse are not only the executable code which is the end product of the software lifecycle but the reuse of the intermediate workproducts as well. The term workproduct is defined as:

> . . . the result of performing an activity like analysis, design, or construction; we will always mean a tangible result that can be read and about which we can ask questions (e.g., a requirements definition containing certain information or a design specifying the modules of a system and their interconnections). [FREE80]

This view of reusability has been recognized by researchers as a very important area of research with short and long range objectives [ITT83]. Wegner [WEGN83] agrees with the need to broaden the research in software reusability.

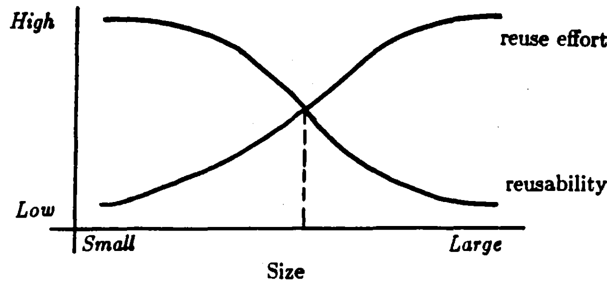
> The term *reusable software* was initially introduced to describe off–the–shelf software components reusable as building blocks of larger systems. . . . But it excludes many important kinds of reusability, such as the reusability of concepts and tools.

MORE *vs.* LESS Reusable Components:



Optimally reusable components are at medium level of abstraction

LARGE *vs.* SMALL Components:



Optimally reusable code is of medium size

Figure 2.2: Tension in Software Reusability

Another problem in software reusability is the tension that exists in the levels of abstraction and in the granularity of potentially reusable components. This tension problem is illustrated in figure 2.2 and discussed below.

More vs. Less Reusable Components

There exists certain controversy between the degree of reusability of software workproducts and the degree of reusability of software components. It is often argued that software representations at high levels of abstraction (e.g., analysis and design) are *more reusable* than software representations at lower levels of abstraction (e.g., code). Analysis and designs are, however, *more difficult to reuse* [WEGN83]. The specialization process of going from a general (reusable) design to a particular

implementation is no different than implementing a design from scratch. Specializations (e.g., code), on the other hand, may be *less reusable*, because of the large number of implementation constraints, but they are *easier to reuse* because there is no specialization process involved. Optimally reusable components are at a medium level of abstraction.

Some researchers, [NEIG80, PRYW79, SWAR82], see reusing analysis and design as more effective, even though more difficult than reusing code. They argue that the production of code is only a small portion of the total software development cycle and that, eventually, code generation will be automated. Other researchers, [ICHB83, STAN83, BOWL83], see the reuse of code as a feasible alternative and as a significant factor in reducing software development costs using current technology.

Both approaches to software reuse are important and are not exclusive of each other. Research in both areas is necessary. These two areas of research are placed into perspective by the concept of *Reusable Software Engineering* [FREE83], a discipline concerned with the study of software engineering activities that:

- "*use* reusable information" and
- "*produce* reusable information."

Freeman presents two distinct research perspectives in this new discipline: "*Reusable–Software* Engineering" and "Reusable *Software–Engineering*".

Reusable–Software Engineering can be identified with the "typical" view of reuse. Its focus is in the development of tools, techniques, and methodologies to improve the reuse of components. It is concerned as well with management procedures to implement them.

Reusable Software–Engineering relates to the "new" view of reuse. The objective here is in the different types of information that can be reused during the development of software systems.

In this frame of reference, the focus of the thesis is in an aspect of Reusable–Software Engineering concerned with reusing medium size components at a medium level of abstraction. These components are usually source code fragments written in high level languages.

Large vs. Small Components

One major problem in Reusable–Software Engineering is the definition of granularity of the components to be reused. At one extreme of the granularity spectrum, small granules are characterized by single programming language instructions while at the other extreme large granules consist of complete systems. Neither extreme is an issue in Reusable–Software Engineering.

Reusing very small components is no different than programming, and very large components are not reusable unless they can be combined with other components in some interesting composition paradigm to make yet bigger components. Use 'as–is' is the common approach for large components. The area of interest in code reuse is thus in the middle of the spectrum (not–too–small and not–too–large components) and in components that can be used to create other components.

In this middle range of component size, we see that, by observing the behavior at the extremes, ease of reuse is monotonically related to component size. Smaller components are usually easier to reuse than larger components. On the other hand, effectiveness of reuse is inversely related to size. The larger the component successfully reused, the more time and effort is saved. Optimally, reusable code is of medium size.

Supporting Environment and Reuser Qualifications

Size is only one of the factors. The environment provided to the reuser and the qualifications of the reuser are two additional factors that contribute to the reusability effort. The following analogy would help in visualizing the interaction of these and other factors when reusing code:

When editing texts, for example, previously written words are not reused; they are rewritten. The effort (even with a very powerful editor) to pick an existing word in our manuscript and use it in a different place overcomes the effort of just writing it from scratch. Notice that words are very much context free and the trade–off is merely one of mechanical effort: search for the word using the first few letters, pick, return to our place in the text, and deposit.

In contrast, if someone is composing a manuscript and wants to reuse a major section from another text, he or she either modifies the context of the paper so that the section will fit without major modifications and become part of the text, or keeps the context rigid and tries to modify the existing section. Here the major component in effort is not mechanical but cognitive. The same mechanical process is conducted as in selecting a word, but it is considerably less significant than the effort to understand the section, modify it to match the new context, and change the style.

Midway between these two examples would be moving paragraphs, with a few context dependencies, around in the text. Here the components in mechanical and cognitive effort are more even, making reusability more attractive. There are so many ways to express an idea in writing that defining a standard paragraph for each idea is unrealistic. Instead, the idea is captured from reading a few keywords from the paragraph, then the text is modified to express that idea, perhaps in a different form. Searching for 'reusable' paragraphs is usually conducted informally by looking

for keywords that provide a clue on the required idea. A more formal approach would be to establish a library of paragraphs and classify them by clue–providing key words. A problem would arise if the library is used by a different person, and for a different topic. The classification scheme may not be as cost effective. A key word defined by the original user may not be as relevant or meaningful for the new user. A better approach would be to classify them by the ideas they represent. Extract the attributes that represent the idea and use them for classification.

In this example, the circumstances of reuse play an important role as well. If the language used requires long complicated words and the user is unskilled at typing but an excellent editor–hacker, then reusing words would make sense. In the case of reusing sections, if we are composing a paper that is only a new version of an existing one, then context would not be much of a problem.

Reusability at the ends of the spectrum may therefore be effective if the setting and conditions of the reuse environment are considered. What remains to be discussed are the trade–offs made in reusing middle size components (e.g., paragraphs.)

Reusing paragraphs also depends on the setting and conditions of the reuse environment. The size dimension of the objects being reused form a continuum between large and small, and the reuser moves along this continuum as the conditions demand. The reuser may reuse words, paragraphs, or complete sections and the trade–offs would be determined at the point of reuse. If the reuser is very knowledgeable about the area of discourse, capturing the idea from a given paragraph or section may not be a problem. If the reuser is skilled in composition, then modifying the paragraphs may not be a problem either. In both cases, the skilled reuser may be more capable than others at reusing larger size paragraphs.

From this analogy it becomes clear that the effective reuse of code is dependent upon the particular characteristics of the components, the environment provided

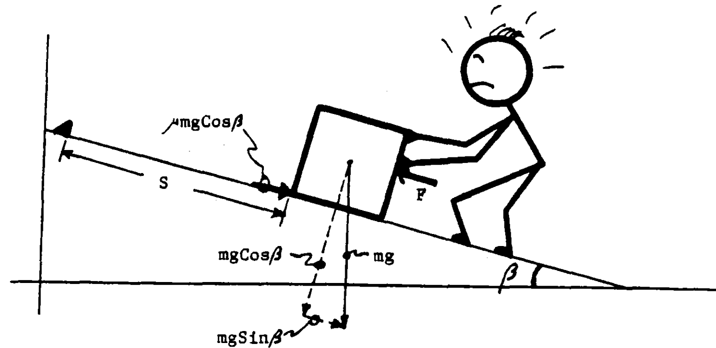


Figure 2.3: A Simplified Physical Model of Reusability

for the reuser, and the qualifications of the reuser.

Code Reuse Effort

The model below illustrates some of the factors involved in the code reuse process and how these factors may influence the code reuse effort. Lack of standard software metrics is a major constraint in measuring reuse effort. The effort in reusing code can be seen as analogous to the work required to move a heavy object a given distance up an inclined plane. A physical representation of this model is shown in figure (2.3).

In this model:

$$W = S \cdot F \qquad \text{and} \qquad F > \mu\, mgCos\beta + mgSin\beta$$

Where:

W = Work done
F = force applied to block
S = distance block is moved
μ = coefficient of friction
m = mass of the object
g = gravity constant
β = angle of inclination of the plane

The equation is an exact representation of the *work* performed on the block by force F to move it up the inclined plane a distance S. In reusing code, the following analogy is observed:

$F \sim$ reuser proficiency (experience and determination)
$S \sim$ extent of conversion
$\mu \sim$ effectiveness of reuse tools and infrastructure
$mg \sim$ size and complexity of program
$\beta \sim$ documentation, code expressiveness, and refinement process

The equation is valid after replacing the analogies, only if it can express each side in the same units. Meaningful units of work (e.g., *ft–lb*) result from this equation only if compatible standard units of measurement are used. (e.g., S in ft, F in lbs, m in $slugs$, g in ft/sec^2, and β in degrees or radians).

In reusing code however, we do not have such standard units. This model could be a truly representative model of conversion effort (work) but we do not have standard units to verify it. Halstead did a pioneer work in describing software through standard units of measurement [HALS77]. Boehm [BOEH81] has shown that some units of measurement (lines of code) are good predictors in estimating software development effort. Their contribution although significant, is far from being standard or universally accepted.

The objective of presenting this model is mainly to illustrate the major obstacles in reusing code. It helps to visualize the relationships among the various parameters involved. Reuse effort can significantly be reduced by decreasing β and improving μ. Even with ideal values of β and μ (~ 0 in both) some $F > 0$ is still required to move the block. To make β and μ as close as possible to their ideal values implies a need to improve the effectiveness of reuse tools and the infrastructure and to provide guidelines for better organization and expressiveness of code and documentation.

Characterization of the Reusability Problem

Freeman [FREE76, FREE80, FREE83] characterizes the reusability problem as the problem of "successfully answering the five reusability questions." A model of software development must be assumed first to include the following aspects:

- A need to create software and the availability of resources to develop it including human and economic as well as existing *workproducts*.
- A well–defined methodology to carry on the development process that would allow for the effective reusability of the existing workproducts as well as for creating new reusable workproducts.

The five reusability questions are:

1. Does there exist a workproduct that may be considered for reuse at this point in the development process?
2. In what ways does an existing workproduct not meet our specific needs?
3. In what ways can the workproduct be changed so that it does meet our needs?
4. What side–effects (unintended changes in the workproduct characteristics) will these changes induce?
5. What effects will these changes have on the work products that are derived later in the development chain from this workproduct? [FREE83]

These reusability questions "capture the essence of the reuse problem." In this thesis a library system is proposed that concentrates on the solution of the first three questions when the workproducts are code fragments. These questions help us formalize the *code reuse process* implemented in the library system.

The following conclusions are derived from this discussion on reusability:

- Reusability of software components is a special case of the reusability of artifacts, as conducted in other disciplines.
- Code reuse takes into consideration three important factors:
    1. Complexity of the component.
    2. User ability to modify a component.
    3. Environmental support to carry on the modification.

Models of Code Reuse

This section discusses the difference between use and reuse of software components. Three models on reusing software components are presented and analyzed. The idea of reuse explored in this thesis is introduced.

Instrumental use of software components is of no interest here. Such is the case of the repetitive use of application programs, the repetitive use of software development tools, and the portability of complete systems to other environments. The type of code reuse pertinent here concerns integrating existing components into a new system. Components do not have to fit precisely and some adaptation may be necessary.

Repetitive execution of software packages, such as payroll or inventory control programs, are of no interest here because there is no software development process involved. Repetitive use of software tools such as compilers and loaders, although used for software development, are not used as components of new systems. Software portability such as moving a compiler to a new hardware may involve a myriad of adaptation problems, but it is not used as a component of a new system.

There are three successful models of reuse that meet our criteria: functional collections, Module Interconnection Language–based systems, and the reuse of code fragments as conducted in some software factories.

Functional Collections

Functional collections consist of large groups of functionally cohesive components such as subroutines, functions, or subprograms designed for the solution of very specific problems. Functional collections have been very successful in the field of scientific programming where problems are usually well–defined and encapsulated. By and large, the most popular and widely used are the International

Mathematics Scientific Library [IMSL83] and the Statistical Package for the Social Sciences [SPSS84]. IMSL consists of a collection of FORTRAN subroutines for numerical analysis mainly used by physicists and engineers. SPSS is a collection of statistical analysis programs with a high–level job control language to facilitate integration. There are several other collections aimed at different specialized scientific and technical areas.

What characterizes the success of functional collections is their functional cohesion, excellent information hiding, and good documentation with illustrative examples and well–defined boundary conditions. Components of these collections have well–defined parameterized interfaces. For an effective application, the user needs to know the performance characteristics with different values of the parameters, the limitations, and, in some cases, the algorithm used. The implementation details are of no concern since the typical user is not interested in modifying its internals for adaptation.

It is important to note that components in functional collections are not intended to be modified. Functional collections are organized for this objective, and usually, no information on component internals is provided. Reusability is thus restricted to a function or subroutine call.

A major limitation of functional collections is the small number of domains where they have been used. There is a need to extend functional collections to other areas of application. In domains that are not as formal and well–defined as mathematics, statistics, or numerical methods, it is difficult to collect some significant number of standard procedures. Such is the case of business applications and several other domains.

Research on domain analysis is needed to identify common features among components to help organize code fragments in domains with less standard procedures. Common features could then be used to define standards.

Module Interconnection

Most of the current software development methodologies follow the 'successive decomposition' paradigm for reducing problem complexity. Programming activity is usually reduced to coding small well–defined modules that are integrated later into the originally planned system. Programming–in–the–large is the term used to describe the activity of 'knitting' those modules together. Programming–in–the–small deals with individual module implementation, for which typical programming languages are used.

Module Interconnection Languages (MILs) are languages for programming–in–the–large. Their development can be traced to four primary sources: [DERE76, THOM76, COOP79, and TICH79]. MILs have a formal syntax for the description of module interfaces, interconnection operations, and resource flow. MILs are used to describe complete system structure and to specify and verify interconnection requirements between modules. MILs are also used as management tools to encourage structuring before coding and as design tools to verify designs.

MILs have been successfully integrated in some research and industrial software development systems [PRIE82]. The objectives of MILs in these systems are, typically, version control and system description. In the GANDALF [HABE81] environment in particular, MIL capabilities are extended to perform other functions. A system generation sub–system is used to generate new system versions from MIL descriptions. A system description "coded" in a MIL is given to the system. Source code is "compiled" to verify interface compatibility, and, if correct, it is "executed". Execution consists of retrieving required modules from a data base and integrating them in the format specified by the source code. A complete, fully integrated system is returned.

The GANDALF environment, however, is limited to generation of versions

of the same system. It can not integrate modules from different systems. Research in the area of standard interfaces is needed to extend the capabilities of the MIL paradigm. Research in component interfaces is currently conducted to free interfaces from syntactic dependencies and concentrate on semantic interfaces, but preliminary results indicate that semantic interfaces may have as many dependencies as their syntactic counterparts [RICE83]. An alternative approach such as flexible or modifiable interfaces may be the answer. Basic research on reusability effort is needed to determine how flexible interfaces can be. How much effort is required to modify an interface? If an exact match is not found, how much deviation is allowed before modification effort overcomes coding effort? Extension of the MIL paradigm also requires research on module descriptions and on techniques for module selection and evaluation.

Code Fragments

The best example of reuse of code fragments is the approach taken by several software factories, especially those in Japan. The core of their success lies in the creation of a library and a catalog of "reusable" components and the selection of certain common "logic structures" for standardization [LANE79]. Programmers are then trained to reuse what is in the library rather than code from scratch. Programs with common structures are generated by filling the skeletons of the established logic structures with library components. Management strategies focus mainly on providing the right tools for library use and the right environment to promote reuse. Approaches such as changing productivity indicators from lines of code to numbers of modules reused are typical.

One reason for their success is that they usually concentrate on a single domain of application. An advantage of concentrating on a single domain is that, once logic structures are standardized for a particular domain, reusability of components

within that domain is carried on very effectively. This, however, is a limitation on their scope of reusability. Reusability across domains is not practiced in software factories.

Reusing software components across domain boundaries presents difficult problems. Context dependencies, implementation constraints, and application–specific functions are some of the problems that have discouraged software designers from reusing components in new application domains. Reusing code under this mode requires some extra information. There is a need, therefore, to conduct research in the problem of reusing code across domains in order to answer questions about describing components, measuring the reusability effort, and determining what information is required to capture domain information.

Another limitation of reusability as practiced in software factories is the adoption of a single programming language. A single programming language improves reusability within the organization but limits its scope. Available reusable components in other programming languages outside the organization may not be worth using because of the effort needed to solve their incompatibilities. There is a need, therefore, to research the problems of reusing components written in different languages, such as, how to measure or estimate the conversion effort from one language to another.

A third limitation of the software factories is that programmers are trained to be library users, not library contributors. From the software factory perspective, this measure is designed to improve productivity within the organization. From a broader perspective, however, if software factories were to share their libraries to increase overall productivity, a large staff of librarians would be required to provide services to a wide range of contributors. Research on how to reduce software library overhead is therefore needed. Problems on software classification and cataloging must be addressed.

Proposed Model

The model of reuse proposed in this thesis is very similar to code reuse in software factories. Emphasis is placed on the assumption that available code fragments usually do not match the requirements perfectly and adaptation is the rule rather than the exception.

Code fragments do not have to exist isolated in a library. All that is needed, in the proposed model, is a pointer to indicate their location inside some working system, although isolated standard components, like the ones used in software factories, may be available. Also, components are not restricted to a particular language. The objective in this approach, is to provide an environment that helps locate components and offers an estimation of the adaptation and conversion effort by proper evaluation of attributes related with reusability.

The idea of creating new code by searching in a library of code fragments is supported by this model. In this approach, workproducts such as requirement analysis, specifications, and designs are used as intermediate means to reuse code where code reuse is the final objective. The role of intermediate workproducts here is merely instrumental, as documentation and code understanding aids.

The reusability scenario advocated is, thus, one in which a new software system is to be built from available components which may be part of systems intended for applications or functions other than those desired in the new-system.

Collection organization is a key issue in this model. Code fragments must be classified so that similar components are grouped together and the component descriptions must serve as discriminating aids in selecting the best among similar components. The importance of these factors is evident when going through the process of reusing code.

Code Reuse Process

Reusing code where modification is one of the activities, can be described by the following process: A set of functional specifications is given. The reuser then searches a library of available components to find a group of candidates that satisfy the specifications. If a program that satisfies all the specifications is available, then reusing it becomes trivial. More typically, several candidates exist, each satisfying some specifications. In this case, the problem becomes one of selecting and ranking the available candidates based on the effort required to modify the non–matching specifications to the desired requirements after considering degree of relevance. This implies that certain specifications are more relevant than others, and the candidates must be ranked accordingly.

The following algorithm illustrates the code reuse process proposed in this model. The paragraphs below explain in detail what is involved in each step.

```
given a set of specs
begin
    search library
    if identical match then terminate
    else
        collect similar components
        for each component {evaluate}
            compare features to requirements
            [modify specs]
        end
        rank and select best
        modify component
    fi
end
```

The order of relevance for specifications is a function of several factors. Specifications of essential requirements come first and are usually few. Essential requirements are those that 'must' be met. Optional or weak requirements are next

and usually numerous. Optional requirements come in the form of *adjustable* specifications. The order of relevance of these adjustable specifications is determined by the reuser upon examination of each available candidate through weighting and matching available features, defined requirements, and personal factors like experience and bias or preference. The function performed by a component is the most relevant factor. Thus, a functional description is defined first and all matching components are selected. The *original sample* of functionally equivalent components is kept all through the selection process and is ordered and reordered iteratively through each stage of the process. In going through each reordering step, the reuser needs more information about each component until a point is reached where any new information about a component does not alter its position in the ranked list. At this point the selection process stops and the component at the top is inspected first.

The initial ordering of the original sample is considered a 'coarse' ordering that is 'refined' as more features are considered for comparison. The order in which these features are incorporated into the selection process is by order of relevance; more relevant features enter the process first.

The problem of matching available features with requirements has been reported in program maintenance [CURT79A, GORD79, BASI84, SELB85], in code understanding [WEIS74, LOVE77, JEFF81, SOLO83B], and in software reusability [CURT83, SOLO83A]. The software component selection problem has been recognized by Neighbors [NEIG80] and the approach to ordering alternatives by relevance is used in decision making theory [HOGA80, KAHN82] and in systems theory [ATHE82].

It was observed, as reported in Chapter 5, that *function* and *environment* are usually the most relevant attributes considered when attempting to reuse a code fragment. Adjustable requirements consist of those attributes related to per-

formance, economics, complexity, size, development effort, and others. They are compared against the reuse effort of the available components by the question: Can I neglect these requirements if by reusing this component I save or improve elsewhere? Thus, relevance of adjustable requirements may change for each reusable component considered.

Once a small sample of properly ranked candidates is available, the reuser is faced with the problem of understanding the contents of each item in the sample in order to determine which is better suited for conversion. This may be a time consuming process unless relevant attributes are properly abstracted and structured. Here proper documentation on intermediate workproducts plays a catalytic role in the process. Then comes the actual conversion effort where the selected component must be adapted to the new environment. A measure of estimation for the conversion effort would significantly help in the decision process and, consequently, in the effectiveness of the reuse paradigm.

The estimation problem has been handled in an oversimplified fashion in the *MetaCAD* System [COOP82] by defining a single ***reusability index*** called "Cost". This cost reduces to a single number all code attributes needed to determine reusability. Under different circumstances, however, some reusable attributes may be more relevant than others thus forcing the same component to have different costs for different reusability instances.

The model presented here provides a more flexible and complete approach to evaluate all relevant attributes each time to determine reusability. Several problems must be addressed before the described code reuse model can become feasible. Some of these problems are discussed below.

Problems in Code Reuse

In this section we address the problems of creating and managing large collections of components, defining component descriptors, component retrieval, program understanding, selection of the 'most' reusable component, and conversion or adaptation for integration.

Large Collections

Creating a pool of available components requires well–defined guidelines to determine which parts of available systems should be considered for reusability. Poorly structured systems are difficult to separate into functionally definable modules. Structured retrofit is sometimes used to restructure old systems. Systems developed with structured methodologies are usually easier to separate into reusable components. However, defining the boundary between any two components may not be a simple task.

Functionally characterized modules are not always easy to isolate from complex systems, and, it is even more difficult, to save them as separate components in a library. An approach to this problem is to leave the potentially reusable component attached to its system and use a pointer for access and a descriptor for relevant reusable information. The collection becomes a collection of descriptors and pointers. The problem is shifted to the definition of effective descriptors. With this approach, precise boundary definition is not required.

Not all components of any system should be expected to be 'reusable'. Some may have too many dependencies, or may be too large and monolithic, or too small to be worth marking for reuse. On the other hand, some may be reused either alone or together with others as members of a subassembly, and subassemblies may be reused alone or as parts of an integrated system. This is one advantage

of this approach over the *standard components* approach where components are isolated from their assembly and generalized for multiple reuse. Another problem of the standard components approach is that true standard components, unless very small, require a myriad of configuration parameters and suffer from performance inefficiencies [RICH83].

Managing large collections of documents is a recognized problem in library science. It is directly related to how the collection is organized and, consequently, the classification scheme used is essential to the organization of the collection. In library science, different approaches to classification are used for different types of documents. While there are some general schemes (e.g., Library of Congress) that are used to classify almost any kind of document, specialized libraries, generally depend on their own specialized classification schemes.

Component Descriptors

Component descriptors deal with the problem of how to represent the *minimal information* required to determine the reusability of an available component. Code descriptions may be many times more extensive than code itself [NEIG83]. A complete description must include information at all levels of abstraction available, from requirements and specification at the highest level to code and operation at the lowest. Development information is also needed to understand the final state of a given software component.

For reusability to be attractive, however, the reuser should spend as little time as possible inspecting a candidate component. Size and descriptive capacity are conflicting goals. Short descriptors are desirable, but, at the same time, a complete description of the components is critical. Both requirements are met only in the extreme situation where descriptors are pointers only, making source code the detailed description of the component. Reuse in this extreme situation is

impractical.

A practical approach could be to define all the 'reusability factors' that can be assigned to components and use them as a standard vocabulary for describing each component. The assigned terms for reusability factors would become the reusability descriptor of a particular component. Other types of descriptors could be defined in the same way. A software catalog would list these factors for each component in a proper format.

Retrieval

A library system is central to code reusability. It is based on a knowledge structure called a classification scheme. A classification scheme consists of an index language and a vocabulary of terms that can be used for making queries. Component retrieval is only as effective as the queries. A classification scheme is a key factor in the effectiveness of a library system.

Although a reusable software library can be implemented as a data base system, with all its benefits regarding data retrieval, it should be noted that software retrieval is different from both data retrieval and document retrieval. According to Blair [BLAI84], data retrieval systems are essentially deterministic whereas document retrieval systems usually have only a probabilistic relation between the formal request and the likelihood that the user will be satisfied with the response. Software components are neither as specific as data nor as general as documents. If software components are described at their lowest level of detail (i.e., code) they can be treated as deterministic pieces of data. For example: list all components that have the instruction $X = X + 1$. On the other hand, if software components are described using only a few abstract terms, they resemble document abstracts and are treated like documents. For example: retrieve all programs that compress files.

An increased hit ratio may result from probabilistic queries, even with a limited vocabulary, if the terms used are semantically rich, unambiguous, and describe attributes relevant to the objects in the collection and to the intention of the user.

Program Understanding

Program understanding is a key problem in code reuse. Reusability effort is usually related to the reuser's capacity to understand code. If the time required to understand a program is reduced, the overall reusability effort is reduced. However, a program that is easy to understand is not necessarily simple. Some complex programs may be easy to understand but difficult to modify.

The problem of understanding programs is a field of research in itself. Despite the effort of researchers [WEIS74, LOVE77, BROO76, SHNE77], very little has been formalized to determine the degree to which a person understands a program. In works by Curtis [CURT79a] and Evangelist [EVAN83], a strong correlation between software complexity measures (e.g., Halstead's and McCabe's) and program comprehension is shown while Shen [SHEN83] cautions their use for that purpose. Coulter [COUL83], on the other hand, presents evidence that Software Science Metrics do not reflect and can not predict program understanding behavior. Several variables are involved, and work is still being done to isolate (and hopefully control) all pertinent variables [MOHE81].

A recent study [SOLO83] shows evidence that reusers use "schemes" to recognize (by abstraction) familiar structures in programs. More experienced programmers use more complex schemes in their understanding processes. This evidence is also supported (at the design level) by Jeffries et.al. [JEFF81]. Curtis [CURT83] advocates the idea that "reusers" understand code by extracting conceptually bounded "chunks" of code which they can abstract and use as conceptual units in their understanding process.

A software library may provide the reuser with a means to speed up the process of incremental abstraction by complementing code with different levels of documentation. If documentation describes conceptually bounded chunks of code in a hierarchical structure, then, the incremental abstraction process could be simplified by searching through the hierarchy.

Selection

Selecting programs from a set of candidates that meet some of the required specifications may be the most time consuming task of a reuser. Specifications must be ranked by the reuser (maybe just mentally) before embarking on the selection process. For example, a particular programming language may be more important than documentation quality.

Features available on each of the listed candidates are checked and compared against required specifications. An evaluation process follows. This process is not unique to software selection. It is a problem of decision making with complex alternatives. Several decision strategies to solve this problem have been proposed and have been widely reported [COOM70, LIND72, KAHN82].

The software selection process is unique in that the ranked list of specifications initially defined by the reuser does not remain constant throughout the process. Two situations may occur:

1. The reuser can find better suited features than the original specifications in the available components. This causes a reordering in the specifications list.
2. The reuser may be forced to make 'local concessions' to a particular candidate. For example, a program may not meet some of the most relevant specifications but may have an excellent documentation (assuming documentation was considered of lower relevance). For this particular case, the reuser disregards the original list or changes its order by giving more relevance to documentation. On the other hand, documentation may be irrelevant if the program is small.

Another problem is that of *cognitive strain*. It is the amount of strain that decision operations place on the cognitive capacities of an individual. A normal individual attempts to minimize or eliminate cognitive conflicts [LIND72]. If we apply this general principle to the software selection process described, the reuser is subjected to a high level of cognitive strain. The natural approach is to disregard as many specifications as possible and as many candidates as possible and make a decision with a reduced set of alternatives.

Presumably, an experienced, objective, unbiased, stress-free reuser would drop the poorest candidates and the less relevant specifications, invest very little effort, and come out with an optimum selection. However, when the sample size becomes larger and the differences between candidates smaller, the probability of missing the best candidate increases.

A partial solution to these problems is a system capable of helping the reuser to make appropriate selections (if not optimum). Starting from an initial set of candidates, an indicator of reuser experience, and a set of specifications, a ranked list of candidates can be defined. A matching algorithm can be used with background rules to modify the original specifications list according to reuser attributes and according to features offered. This approach is objective thus helping the reuser handle the cognitive strain better by presenting a preordered sample where partial alterations could easily be determined.

Adaptation

Once a program is selected, the reuser is faced with code adaptation. This problem is tightly coupled with program understanding. Code adaptation can be carried on in any of the following ways:

1. Specialization—the reuser relies mainly on high level information to implement new code or to adapt existing code.

2. Conversion—the reuser adapts code piece wise to new requirements.
3. Abstraction–Specialization—the reuser abstracts information from code, generalizes it, and implements new code.

In **Specialization**, the available code is used as documentation while the details of the new implementation are worked out. In this case, new code is created from scratch, not reused. Experienced reusers and highly qualified analysts/programmers would take full advantage of this approach.

Conversion may be very attractive in cases where most functional specifications are met and environmental specifications are different. Automatic language conversion tools are available for a very small set of languages [WOLB83]. For the scenario advocated here, however, the conversion process may require the conversion of different code fragments coming from different (functional and environmental) sources and coded in different programming languages. The recommended process is to convert them manually.

Reusers tend to project their own programming patterns into the new code. For example, if a COBOL programmer is asked to do a program in FORTRAN, the FORTRAN program would probably reflect COBOL–like structures and logic. A measure to determine how difficult it is to convert code from one programming language to another, based only on the characteristics of both languages, may help the conversion process.

Abstraction–Specialization may be the most typical case of code conversion. Starting from a piece of code, the reuser abstracts (through understanding and documentation) functional concepts and then implements them in the new environment. Enough abstraction is carried on to grasp the functional concept in question and apply it to the new environment. During this process, only the features that do not match with selected specifications are modified. The reuser is only concerned

with local transformations to code and does not usually worry about structural modifications.

Code conversion is carried–on more effectively by experienced programmers. They seem to have a large repertoire of "programming plans" [SOLO83] that they can retrieve at will. To make code conversion practical for 'non–masters', an environment is needed where programmers can easily discover programming plans in the available code. A first cut approach is to provide information (documentation) at different levels of abstraction to stimulate local abstraction–specialization. A long range solution would be to identify, classify, and describe programs by means of standard programming plans.

Required Infrastructure

From this analysis of the problems in reusing code, the following infrastructure is proposed as a partial solution to the software reuse problem:

- A specialized library of program descriptors and pointers to their respective source code and different types of documentation, perhaps in the form of a data base system.

- A classification scheme that can be used not only as a classification guide but as a frame for building queries and conducting selective searches. The classification scheme should serve as an organizing structure for the data base and as a knowledge representation structure. Reuse related attributes should be terms defined in the indexing language and have a definite position in the classification structure.

- A Support System to reduce reuser *cognitive stress* and speed up the selection process. This system would guide the reuser through the classification scheme

to select the appropriate index terms required for the initial query. The system would use reusable related metrics and a reuser experience metric to rank the initial selection of candidates in a *more* to *less* reusable order.

Conclusions

An overall perspective on code reuse has been presented. Strong motivational arguments were given to support this approach to code reuse based on collections of reusable components. Three successful models of code reuse were analyzed. Selected features from each were used to propose a more comprehensive model, one based on retrieval of components that are 'similar' to the proposed query. A mechanism was suggested to provide support for the evaluation of the selected candidates. This proposed model relies very strongly on the classification scheme used.

Problems in reusing code were analyzed. For each problem presented, some ideas to handle them were proposed. An experimental library system with a specific infrastructure was proposed as essential to study the code reuse problem and, eventually, understand it better.

The following chapters present a classification scheme for software components as a partial solution to the code reuse problem. A proper organization of the collection aimed at serving the needs of the reuser is a stepping stone towards solving some of the reusability questions. A prototype library system is implemented to test the proposed scheme and some encouraging results show the potential assistance of classification in reusing code.

CHAPTER 3

Classification Related Work

This chapter introduces the basic concepts of classification theory as used in library science. A survey on classification schemes, in particular those used in library classification is presented. Their major features are identified and their potential in software classification indicated. The area of software classification is surveyed indicating the major drawbacks of current software classification schemes. It is concluded that a faceted scheme is better suited for classifying a collection of code fragments.

What is Classification[1]

Classification is the act of grouping like things together. All members of a group—or class—produced by classification share at least one characteristic which members of other classes do not possess. What gets classified may be concrete entities, the ideas of such entities, or abstractions. For example, eagles, hawks, and owls would be housed close to each other in a zoo because we observe that they share the characteristic of preying on their victims which doves, chickens, and swallows do not possess. In the supposed case that zoos did not exist and the actual birds could not be grouped, we could still appreciate the relationship between the idea of an eagle, the idea of a hawk, and the idea of an owl, and group them in our minds. Similar groups could be formed with properties of things and with operations and activities performed by things or on things.

Classification displays the relationships between things, and between classes of things. From the previous example, it is observed that members of the class

[1] Most concepts and definitions in this section were taken from [BUCH79].

EAGLES differ from members of the class HAWKS; but also that these two classes have a closer relationship with each other than either has with the class OWLS, because they share the quality of being day hunters. All three however are related because they are all BIRDS OF PREY. The result of classification is the display of a network or structure of relationships which is used for many purposes, unconsciously or consciously.

Human recognition of likeness or of shared characteristics may be an intuitive process or it may be the result of conscious thought. Recognition of likeness, in some cases, can also be carried on by machines through cluster analysis either by some numerical proximity measure in a multidimensional space [KRUS78, ANDE73] or by some conceptual criterion [MICH83].

Through classification we cope with the multitude of unorganized impressions we receive through our senses. We can use this preconceived classification structure to place what we see, hear, feel, smell and taste within it. Classification simplifies the process of thought, because there are far fewer classes than there are members of classes. *Definition* of a thing is a classification process that separates its *genus* and the specific *differentia* of the concept defined. That is, in defining, a reference is made to the class which contains the thing being defined, and then the characteristic which differentiates the thing from the other members of the same class—as in 'an owl is a bird of prey that hunts at night'.

Classification is a fundamental tool for the organization of knowledge and pervades everyday life from supermarkets to warehouses to schools. A particular application of classification is in libraries to classify documents conveying information. A central task in a library is the organization of the collection for easy access and document retrieval. A collection owes its organization to a **classification scheme** which may be defined as a tool for the production of systematic order based on a controlled and structured index vocabulary.

An *index vocabulary* is the set of names or symbols which represent concepts. A concept may have many names (synonyms); a name may stand for many different concepts (homonyms); and some concepts may not have their own names.

In a library of documents the concept index provides the means to find all works on a concept no matter what name is used for the concept by the user or what name is used in the document. An index vocabulary is *controlled* because the indexer has controlled synonyms and homonyms in order to find the desired works on a given concept. Controlled vocabularies are usually also *structured*; that is, they display the relationships between different concepts either by cross–reference or by keeping works on related subjects close to each other through systematic order. **Systematic order** between documents is an order which, in itself, displays the relationships between the subjects of documents. For example, in the sequence VERTEBRATES-AMPHIBIANS-FROGS-TOADS-FISH-TROUT it can be observed that the class FROGS is kept with its closely related class TOADS; these are preceded by the class which contains them, AMPHIBIANS and followed by FISH which is related to the class AMPHIBIANS; FISH is in turn followed by the class it contains: TROUT.

A classification scheme, thus, is a structured index vocabulary and the list of all classes in a prescribed systematic order is called the **classification schedule.** Classification, then, is about the discovery and the display of relationships in some systematic order.

Syntactical and Hierarchical Relationships

There are two kinds of relationships a classification scheme must be able to express: syntactical and hierarchical. Syntactical relationships are those between classes which occur together in statements which represent the subjects of documents. Such as between *learning, programming* and *home computers* in the title "The Role of Home Computers in Learning How to Program". A classification

scheme should enable its users to differentiate the above title from, say, "Learning to Program with Home Computers". The concepts used in both titles are the same but they are displayed in a different syntactical relationship. There are four kinds of syntactical relationships: elemental (e.g., Birds), superimposed (e.g., Migratory Birds), compound (e.g., The Migration of Birds), and complex (e.g., Comparison Between Migratory and Hibernating Animals).

Hierarchical relationships are based upon the principle of subordination or inclusion. A classification scheme must display when a class wholly includes another or if two classes are included by a third like in MAMMALS-RODENTS-RATS-MICE. This kind of hierarchical relationship is called *generic* because it is an absolute relationship which does not depend on the existence of documents on the related subjects.

A second kind of hierarchical relationship is the one that is not generic and only exists because an author has produced a work involving it. For example, the relationship between "Software Metrics" and "Validating Software Metrics" or the relationship between "Rats" and "The Intelligence of Rats". This type of hierarchical relationship displays a broader–narrower relationship between a thing and its activities (Software Metrics—Validation) or between a thing and its properties (Rats—Intelligence). Broader–narrower relationships between more complicated subject statements may be difficult to perceive. For example, "Validation of Software Metrics" and "Validating Program Length as an Objective Software Metric".

In hierarchical relationships, a class that contains another is said to be superordinate to that class and the contained class is said to be subordinate to the containing class. This is equivalent to a parent–child relationship in an ordered tree. Classes that are neither broader nor narrower than each other but share the same immediate superordinate class, are said to be coordinate (e.g., siblings in an ordered tree); those not sharing the same immediate superordinate class, although

in the same hierarchy, are said to be collateral (e.g., descendants of a common ancestor).

Enumerative and Faceted Schemes

A classification scheme can be arranged to express syntactical and hierarchical relationships in two ways: enumerative and faceted. The enumerative or traditional method is to postulate a ***universe of knowledge*** and to divide it into successively narrower classes which will include all the elemental, superimposed, and compound classes which the scheme may have to accommodate, arranged in an order displaying their hierarchical relationships. All the required classes are listed or 'enumerated' in this kind of schemes, except complex classes which are 'author attributed relationships' and impossible to predict. Dewey Decimal classification is an obvious example of an enumerative scheme. It divides the whole knowledge into ten main classes, and each of these into ten principal subclasses, and each of these subclasses into ten, and so on as far as is necessary.

The faceted method relies not on the breakdown of a universe, but on building up or 'synthesizing' from the subject statements of particular documents. By this method, subject statements are analysed into their component elemental classes, and it is these classes only which are listed in the scheme; and their generic relationships are the only relationships displayed on its pages. When the classifier using such a scheme has to express a superimposed, complex or compound class, he does so by assembling its elemental classes. This process is called ***synthesis***.

Facets are the arranged groups of elemental classes that make the scheme. Figures 3.1 and 3.2 illustrate the difference between a faceted scheme and an enumerative scheme. Figure 3.2 shows the enumerative version of the faceted scheme shown in figure 3.1. Both schemes can be used to express precisely the same number of classes. The difference is that, in the enumerative scheme, classes with more than

```
(process facet)
  Physiology
    Respiration
    Reproduction

(animals facet)
  (by habitat subfacet)
    Water Animals
    Land Animals
  (by zoologists' taxonomy subfacet)
    Invertebrates
      Insects
    Vertebrates
      Reptiles
```

Figure 3.1: A Demonstration Faceted Scheme [BUCH79]

one elemental component are listed ready–made, while, with the faceted scheme the classifier will have to make multi–element classes by synthesis. The construction of an enumerative scheme is obviously, much more involved and time–consuming.

Enumerative schemes are usually inefficient. They are not normally compiled to the level of detail shown in this demonstration example, but, because of the size and complexity of the operation, compilers usually omit classes which they think are not needed. If some classes are omitted, it will not be possible for the users of the scheme to express them when necessary.

Citation Order

The purpose of a classification scheme is to show relationships by collocation—that is, to keep related classes more or less together according to the closeness of the relationship; for example, a zoologist would place the classes WATER INVERTEBRATES and LAND INVERTEBRATES close together because both classes are invertebrates. A marine biologist would rather have WATER VERTEBRATES and WATER INVERTEBRATES close together and separated from LAND INVERTEBRATES. The

```
Physiology
  Respiration
  Reproduction

Water animals
  Physiology of water animals
    Respiration of water animals
    Reproduction of water animals
Land animals
  Physiology of land animals
    Respiration of land animals
    Reproduction of land animals

Invertebrates
  Physiology of invertebrates
    Respiration of invertebrates
    Reproduction of invertebrates
  Water invertebrates
    Physiology of water invertebrates
      Respiration of water invertebrates
      Reproduction of water invertebrates
  Land invertebrates
    Physiology of land invertebrates
      Respiration of land invertebrates
      Reproduction of land invertebrates

  Insects
    Physiology of insects
      Respiration of insects
      Reproduction of insects
    Water insects
      Physiology of water insects
        Respiration of water insects
        Reproduction of water insects
    Land insects
      Physiology of land insects
        Respiration of land insects
        Reproduction of land insects

Vertebrates
  Physiology of vertebrates
    Respiration of vertebrates
    Reproduction of vertebrates
  Water vertebrates
    Physiology of water vertebrates
      Respiration of water vertebrates
      Reproduction of water vertebrates
  Land vertebrates
    Physiology of land vertebrates
      Respiration of land vertebrates
      Reproduction of land vertebrates
  Reptiles
    Physiology of reptiles
      Respiration of reptiles
      Reproduction of reptiles
    Water reptiles
      Physiology of water reptiles
        Respiration of water reptiles
        Reproduction of water reptiles
    Land reptiles
      Physiology of land reptiles
        Respiration of land reptiles
        Reproduction of land reptiles
```

Figure 3.2: A Demonstration Enumerative Scheme [BUCH79]

habitat facet for the marine biologist may be more relevant than the *zoologist's taxonomy* facet. *Habitat* determines the primary division and *zoologist's taxonomy* may be subordinated to *habitat.* The order in which facets are considered is called **citation order.** The choice of citation order is arbitrary and determines which classes are close together and which are far apart.

The divisions within a facet are defined by the list of subfacets and classes. Closeness among classes is determined by their relationship. Classes listed in facets are called **terms**, and their relationship is shown by their collocation in a linear list. If the habitat facet were extended as illustrated in the example below, RIVER, LAKE, and SEA are close together in the list and far from DESERT or TROPIC or MOUNTAIN. Subfacets in this example are land, aquatic, salt water, and fresh water.

(habitat)

Land
    DESERT
    TROPIC
    MOUNTAIN
Aquatic
  Salt Water
    SEA
  Fresh Water
    RIVER
    LAKE

Different criteria can be used for term ordering. One is *developmental* like INVERTEBRATES listed before VERTEBRATES since this is the order followed in the evolutionary process; *chronological* like the order ROMANESQUE, GOTHIC, RENAISSANCE in a scheme of arquitecture styles; *spatial* like SUN, MERCURY, VENUS, EARTH, MARS in an astronomy scheme; or *increasing complexity* like SOLITARY ANIMALS, HERD ANIMALS, SOCIAL ANIMALS in an animal behavior scheme. Term

ordering is selected to satisfy the needs of the users.

Notation

Notation is needed for identification of classes during classification, for shelving documents, and for arranging entries in a catalog. A given code is assigned to each class in the schedule and documents are marked with the assigned code when classified. Codes tend to make it easier for users of the library to locate documents on the shelves. One advantage of notation is that neither the user nor the classifier need to know the terms and term ordering needed to describe a particular document. In the Library of Congress scheme, for example, the title "Structured Systems Programming" has been assigned the code QA76.6 (i.e., *call number*) instead of the list of terms:

General Science
  Mathematics
    Computer Science
      Software

For most people the notation is the classification scheme, but it must be pointed out that notation is only a relatively unimportant part of the scheme. The acceptability of a scheme to its users, in particular where physical collocation is essential, depends largely on the qualities of its notation. In library science, classification scheme notation is very important. Much effort has been invested in making notation as compact as possible to accommodate the classification codes in the narrow spines of books. In a scheme where physical collocation is not important as is the case in a collection of software components, notation may not be important.

A drawback of notational schemes is that once a list of symbols is assigned to a particular schedule of terms, further additions of new terms may require either a remapping of the term sequence into a new list of symbols which means complete reclassification or, patching the sequence (as is usually done) thus losing its logical order. The burden of a notational scheme is carried by the classifier who must learn

the notational rules of translation from classification terms to code, even though the library user must learn, in many cases, how to decode them. The notational language becomes the common reference language in a library instead of the more natural index language.

Use of a Classification Scheme

The process of using a classification scheme to produce some systematic ordering is called *classification*. Classification in general is carried on by choosing certain characteristics of the object to be classified and selecting index terms from the classification schedule that properly describe those characteristics. The resulting ordered list of index terms is then used to generate a classification code. Items sharing the same codes belong to the same classes. In the case of classifying library documents, the task of identifying object characteristics is substantially reduced when a title is available.

The following example illustrates a typical classification procedure. The title is "The Nutritional Requirements of the Panda". In an enumerative scheme the title would be entered either in the category BEARS from where pandas is a term or in the category NUTRITION from where nutrition requirements is a term. Unless a specific category NUTRITION OF BEARS is available (very unlikely), the librarian has to determine which category to select based on a thorough inspection of the book. (Librarians decisions are usually non–optimal because of monocular perspective.)

If a faceted scheme was used, the term nutritional requirements would be listed in facet GENERAL PROCESSES from where NUTRITION is a term. Pandas would be listed in class BEARS and bears in MAMMALS and mammals in VERTEBRATES all of them generic terms of facet ZOOLOGIST'S TAXONOMY. The librarian in this case would *synthesize* a classification 'code' from both facets resulting in a precise and unique classification NUTRITION OF PANDAS.

Making a Faceted Scheme

The construction of a faceted scheme is much easier than that of an enumerative scheme. A faceted scheme is constructed by selecting a representative sample from the collection to be classified. In the case of books, the sample is a list of book titles, in the case of programs, the sample is a list of program descriptions.

The first step is to separate terms (e.g., keywords) from the sample list and group them into facets. All related terms are grouped together. After all terms have been grouped, an iterative process of regrouping is conducted to form subordinated groups within each facet (i.e., subfacets) or to break groups into separate facets. After terms in all facets and subfacets have been defined, the next step is to order terms in each facet and subfacet according to their relationship and to the needs of the users (e.g., spatial, chronological). Next, subfacets are placed in order in their respective facets reflecting the needs of the users. Next a citation order is assigned to the facets as indicated before. At this stage, the resulting scheme should provide a preferred collocation and systematic order of the items of the collection. This resulting scheme is essentially the desired faceted scheme.

To make the use of a faceted scheme easier, library science recommends two further steps: Adding notation to each class (i.e., term and facet) and producing an alphabetical index of the classes. Buchanan [BUCH79] and Vickery [VICK60] are two excellent references on the construction of faceted schemes.

Summary

Classification is about the discovery and display of relationships; it simplifies the thought process and provides the basis for a definition of things. Classification is a fundamental tool for the organization of knowledge, and a classification scheme establishes a systematic order to a particular set of concepts listed in a controlled

and structured index vocabulary. A classification schedule is the resulting ordered list of concepts.

A classification scheme must be able to display syntactical as well as hierarchical relationships. The two kinds of schemes that display these relationships are enumerative and faceted. An enumerative scheme is a list of all possible classes and their relationships while a faceted scheme lists only the essential classes, and the classification process is conducted by synthesizing essential classes to express specific relationships.

Citation order determines which facets in a classification scheme are listed together and which are kept apart based on the needs of the users. Term ordering is the order in which classes are arranged within a facet. This ordering defines some conceptual closeness based also on user needs.

Notation is used to codify, in a compact set of symbols the terms and term ordering that describe a class. Classification codes are useful for shelving documents and arranging entries in a catalog but are not a necessary component in a classification scheme. Classification consists in identifying terms in the schedules that describe the characteristics of an object or the title of a document and assigning the corresponding code.

Library Classification

There are five major general classification schemes: Dewey Decimal Classification (DDC), Universal Decimal Classification (UDC), Library of Congress (LC), Colon Classification (CC), and Bibliographic Classification (BC). This section provides an overview of these schemes consisting of a brief background, some basic concepts, major features, and their relationship to the other schemes. These schemes can be grouped into two major classes: enumerative and faceted. None of them is purely

faceted or purely enumerative but all have some of each class. DDC and LC are considered almost enumerative, CC and BC are considered almost faceted, and UDC is a combination of both. Their relative position in a scale from enumerative to faceted is illustrated below:

Library classification schemes are based on philosophical schemes of knowledge classification. Philosophers were interested in studying the mutual relations between ideas and their logical sequence. This led to a number of schemes of knowledge classification dating as far back as the ancient Greek classification proposed by Aristotle. The Greek scheme divided knowledge into three main groups, theoretical philosophy, practical philosophy and productive arts. This scheme follows an order from the abstract (pure knowledge) to the specific (use of knowledge to create things) and is "utility–centered".

Francis Bacon (1561–1626) proposed a new scheme that strongly influenced the later schemes such as Kant's, Hegel's, and Comte's and set the basics for modern library classification. The Baconian system is "psychology–centered" and divides the universe of ideas into three main groups in the following order: history—"emanation from memory", poesy—"emanation from imagination", and philosophy—"emanation from reason". DDC and UDC are based on an inverted Baconian order; philosophy, poesy, and history.

Dewey Decimal Classification

DDC was developed by Melvil Dewey in 1876 as an alternative to the "fixed location" document arrangement practiced by contemporary libraries. It was the practice to assign shelving areas to various subjects and within each subject books were arranged by accession number. Each book was assigned a shelf mark denoting its exact position on the shelves. Melvil Dewey introduced the idea of "relative location" as opposed to "fixed location." He assigned decimal numbers to books and not to shelves. As a result, a new book on a given subject could be interpolated in the existing sequence in a position prescribed by a decimal number.

DDC is a hierarchical scheme of classification, which proceeds from the general to the specific. The universe of subjects has been divided into ten main *classes*; each class into ten *divisions* and each division into ten *sections* [DEWE65]. The ten main classes are:

000 Generalities
100 Philosophy and related disciplines
200 Religion
300 The social sciences
400 Language
500 Pure sciences
600 Technology (applied sciences)
700 The arts
800 Literature
900 General geography and history and their auxiliaries

These main classes follow the inverted Baconian order. Each main class represents either a broad discipline or a group of related disciplines, except for the 000 class which includes varied subjects such as bibliographies and catalogs, general encyclopedias, and serial publications.

The ten complete divisions of the main class 700 are given below:

700	The arts
710	Civic and landscape art
720	Architecture
730	Plastic arts and sculpture
740	Drawing, decorative and minor arts
750	Painting and paintings
760	Graphic arts and prints
770	Photography and photographs
780	Music
790	Recreational and performing arts

At each subdivision, the scheme preserves the same inverted Baconian order.

The ten sections of the 720 division are:

720	Architecture
721	Architectural construction
722	Ancient and oriental architecture
723	Medieval architecture
724	Modern architecture
725	Public structures
726	Buildings for religious purposes
727	Buildings for educational purposes
728	Residential buildings
729	Design and decoration

Sections labelled 0 (e.g., 700 and 720), as in class 000, are allocated for general works on the entire division and 1–9 are used for subclasses.

Further subdivisions in DDC follow the same decimal structure and are represented by digits after a decimal point in the notation, each digit representing a subordinate level. The idea of successive division is illustrated below:

700	The arts—Fine and decorative arts
720	Architecture
725	Public structures
725.8	Recreation buildings
725.82	Buildings for shows and spectacles
725.822	Theater and opera buildings

DDC is essentially an enumerative scheme largely providing ready–made com-

pound classes. The primary arrangement of subjects in DDC is by discipline and specific subjects like computers, Mexico and underdevelopment may appear in any of the disciplines. The result being that DDC scatters subjects by discipline. Being an enumerative scheme, the DDC index listing is very extensive since it includes all compound classes (e.g., Elizabethan England—942.055, Elizabethan drama—822.3).

DDC has been criticized for its restricted decachotomy base (limited to a branching factor of ten). The universe of knowledge has been fixed to 1000 main sections and there is no provision to add more. New sections are added at lower hierarchical levels. In some new dynamic disciplines like in electrical engineering, it has created very deep hierarchies. For example, 'transistorized circuits' has the code 621.381 530 422. Disciplines that have not seen very much development since the last century like logic keep very shallow hierarchies (e.g., 'sylogisms' is coded 166).

DDC is a general classification scheme that is also being used in bibliographical listings, book catalogs, book guides and reading lists. DDC is the oldest and most widely used scheme of classification. It has been adopted in libraries all over the world mainly for its simple notation, its ease of application, the adaptability of its notation to the requirements of libraries of different sizes, and its availability in a variety of editions (i.e., 18 editions since 1876).

Its major drawbacks are being enumerative and having a rigid framework. As Arthur Maltby [Malt75] puts it:

> . . . the system still inevitably lags behind modern theories; the whole-hearted supporter of the clear analysis of the elements which make up a subject into definite categories, or facets, is likely to be tempted to regard such an enumerative system as almost antediluvian in many respects. Yet no classification devised in the nineteenth century and at all concerned with number integrity could hope to be completely abreast with modern theories of classification or with modern knowledge. The fact remains that [DDC] works well in a very large number of libraries.

Library of Congress

The Library of Congress was founded in 1800, and the arrangement of the collection was according to size until 1815 when Thomas Jefferson's collection came into its possession. The library adopted the classification used by Jefferson which was based on Bacon's Classification of Knowledge. When the library moved to a new building in 1897, a decision was made to abandon the Jeffersonian system and to reclassify according to a more practical and flexible arrangement. Thus a new classification scheme was specially designed for the collections of the library and for the library building itself. The theoretical approach, whereby knowledge is first mapped out and then adapted to meet the needs of books, was rejected in favor of a gradual build–up in a classification based on the needs of the collection of a particular library. This approach of gradual build–up based on the needs of an existing collection is generally called **literary warrant.**

LC consists of a series of special classifications each covering a major class. Each class has been devised by subject specialists. LC is considered as a coordinated series of special classifications where each major class is virtually independent of the others. There are 20 major classes in LC with one additional class for general works.

A	General works and polygraphy
B–BJ	Philosophy
BL–BX	Religion
C	Auxiliary sciences of history
D	History: General and old world
E–F	History: America
G	Geography, anthropology, folklore, etc.
H	Social sciences
J	Political science
K	Law
L	Education

M Music
N Fine arts
P Language and literature
Q Science
R Medicine
S Agriculture
T Technology
U Military science
V Naval science
Z Bibliography and library science

Main classes are denoted by single Roman capitals and two Roman capitals are used for subdivision of main classes. The class Mathematics, for example, is a subdivision of the major class Science and represented by the two Roman capitals QA. Further subdivisions are represented by decimal numbers, but, in some areas, the subdivision is not sufficiently detailed. For example, the subject Software—QA 76.6 (i.e., Q—General science, A—Mathematics, 76—Computer science, .6—Software) which is considered a very general subject is the most detailed subdivision in this hierarchy. All books related to software are grouped in class QA 76.6 and ordered alphabetically by author name. Further subdivision of software is not yet provided in LC.

Sequence within classes in LC is achieved by using appropriate characteristics for the division of subjects. It attempts to achieve a helpful sequence of various groups of books, rather than groups of subjects. In other words, it avoids a purely theoretical point of view. That is why the scheme does not strictly follow the scientific order of subjects. The sequence and coordination of main classes in LC is based on a late nineteenth century educational and scientific consensus.

LC is considered by far the most enumerative of the general classification schemes. Sequences of terms repeat over and over in each class as illustrated in the example below [MALT75]:

HA Statistics	HB Economic theory	HD Economic history (land and agriculture)
1 Periodicals	1–9 Periodicals	101 Periodicals
9–11 Congresses	21–29 Congresses	103 Associations
13–15 Collections	31–35 Collections	105 Congresses
16 Comprehensive works	61 Encyclopedias	113–1565 History: General
17 Essays	71–74 Method: Utility	166–279 United States
19 History	75–125 History	301–1130 Other countries
23 Biography	151–195 Theory: General works	
29–39 Theory: Method		

LC is criticized for having a structure that avoids logical hierarchies and clear subject analysis and is considered a "vast pragmatic pigeon–holding devise" as Kumar [KUMA79] puts it:

> . . . thus, we may regard LC as basically a book classification scheme not rooted in philosophical systems which are based on analysis of ideas. . . . Most of the libraries using LC have simply adopted it as a method of shelf classification for subject location and a marking and parking device.

The lack of a clear and predictable theoretical basis for subject analysis prevents the wide use of LC in retrieval systems.

Despite these deficiencies, LC has been very successful especially in the academic libraries of the United States. One major reason for the success is the extensive support offered by the Library of Congress in the form of catalog card service, continuous schedule revision, and cataloging standardization procedures.

Universal Decimal Classification

UDC was originally derived from the fifth edition of DDC in 1895. The scheme was initiated by the Institut International de Bibliographie in Belgium and was prepared for the classification of all published literature, so that a comprehensive classified index to all literature could be produced, a rather colossal task.

UDC is aimed specifically at retrieval rather than at shelf arrangement unlike the other general schemes [FREE68, RIGB72]. For this purpose, in order to accomplish arrangement by subject in a vast and growing catalog, an extremely detailed and expandable classification scheme was required. The requirements asked for a scheme capable of showing the relation of books to subjects, places, languages, eras, etc., and for a series of common subdivisions much more comprehensive than those in any of the then existing systems.

UDC is a practical classification scheme based on the demands of abstracts, reports and periodical literature rather than on the framework of theory. It is a system for numerically coding information, so designed that any item, once coded and filed correctly, can be readily found from any perspective it is sought.

The major contribution of UDC is the introduction of auxiliary schedules and special notation to connect and relate terms listed in separate schedules; a *synthesis mechanism.* UDC is considered the first scheme to introduce a system of common facets.

UDC divides the entire field of knowledge into the same ten main classes DDC does and follows the same hierarchical structure for subdivisions. Centesimal divisions however, are used if more than ten divisions are needed. The auxiliary schedules or facets are of two kinds—*common* and *special.* Common facets include language, form, place, time, point of view and race and nationality. Special facets have different meanings depending upon the context. For example, from the schedule of Political Science, special facets are *by political ideology* and *by duties of state.*

A major drawback in UDC is notational complexity. An elaborate notational scheme has been devised to properly indicate how terms from different facets are combined to synthesize different types of relationships. For example, 666.113'41'28 denotes 'lime–silica glass'. The synthesis comes from 666.113 glass, 546.41 calcium, and 546.28 silicon. Single quotes (') mean combination and simplification. A sec-

ond example, 61=03.82=40 denotes 'medical documents translated from Russian to French' where 61 denotes medical documents, 03 denotes translation, period (.) denotes the source language (in this case Russian—82), 40 denotes French and the '=' sign means that 82 and 40 are from the language facet. UDC is a complex encoding scheme where each term in each facet is assigned a decimal number and a set of special characters are used, through complex encoding rules, to combine terms for denoting very specific subjects.

Despite this drawback, UDC is used by a large number of libraries around the world. It is particularly popular in Europe, Latin America, and Japan and has become the official classification scheme in scientific and technical libraries in the USSR and Eastern Europe. UDC is considered an almost faceted scheme that has proved most useful for the exact classification of highly specific subjects.

Bibliographic Classification

BC was developed by Henry E. Bliss in 1908. BC is based on theoretical principles explained by Bliss in his *Organization of Knowledge* [MILL77]. The leading principle used in BC is scientific and educational consensus. That is, consensus should be sought from expert opinion in all areas. According to Bliss, it is through the process of science and education that knowledge is ultimately affected and the more closely a scheme reflects the consensus the more stable it is.

BC is an almost faceted scheme. It provides 22 principal systematic schedules with several subschedules each and several combination rules to obtain precise synthesis of very specific subjects as in UDC but with a simpler notation. The reason BC is not fully faceted is because most schedules are not general; they are specific and context dependent, thus limiting synthesis, in most cases, to subjects within the same general class.

There are 21 major classes in BC:

1–9	Anterior numeral classes
A	Philosophy and general science
B	Physics
C	Chemistry
D	Astronomy, geology, geography
E	Biology
F	Botany
G	Zoology
H	Anthropology
I	Psychology
J	Education
K	Social sciences
L–O	Social–political history
P	Religion, theology, ethics
R	Political science
S	Law
T	Economics
U	Arts in general, useful arts
V	Aesthetic arts, recreative arts and pastimes
W–Y	Philology: language and literature
Z	Bibliography, bibliology, libraries

A feature of BC is the collocation of related subjects and the proper subordination of each special topic to the appropriate general one. This feature of serial dependence is called *gradation by specialty.* Certain subjects are derived from findings in others and are, in this sense, more specialized than the disciplines from which they were derived. This gradation by specialty can be appreciated in the following extract from the main schedule:

AW	Statistics and probability
AX	Systemology and organization theory
AY	Science and technology
AZ	Science and empirical science
AZB	Physical Science
B	Physics
BA	Mathematical and theoretical
BB	Experimental
BC	Thermodynamics

Bliss recognized that, for certain subjects, there were two or more possible locations in the sequence of classes that were almost equally acceptable. For example, photography can be with technology and with the arts; economic history can be subordinated to general history but can also go under economics. This provision in the scheme led to strong criticism from the library classificationists arguing that books in a given subject area should be grouped together and not scattered. From the point of view of general classification, however, this feature is very important for document retrieval in finding related subjects.

BC notation is very compact and follows an *ordinal* form rather than the hierarchical form of DDC and UDC. The example below illustrates the difference:

	DDC		BC
796.3	Ball games	HKE	Ball games
796.33	Football	HKF	Football
796.333	Rugby	HKG	Soccer
796.334	Soccer	HKH	Hockey
796.34	Racket games	HKI	Polo
796.342	Tennis	HKJ	Lacrosse

Despite its qualities of helpful order, adaptability, a clear and predictable facet structure and a generally concise notation, BC has been adopted by very few libraries. The reason is an incomplete schedule. A partial, substantially modified second edition of BC was published in 1977 [MILL77], and an edition of the complete scheme is expected soon. Adoption of the new edition by libraries already using it is unlikely due to the enormous reclassification effort required. It is expected that some new academic libraries may adopt it.

Colon Classification

CC was developed by Shiyali R. Ranganathan in 1924 as the first truly faceted library classification scheme [RANG67]. The name is due to the role the colon

symbol (:) plays in the rules to synthesize subjects from different facets.

CC's main contribution is the set of rules and "canons" prescribed to synthesize terms from different facets to compose proper subject headings. CC is not an enumerative scheme and there is not a continuous listing of all subjects describing the whole universe of knowledge.

Ranganathan has produced an entirely synthetic system in which most topics are compounds specified by linking terms from different facets.

The fundamental idea in CC is that all facets are related to one another in a fixed citation order given by five fundamental concepts—Personality, Matter, Energy, Space and Time (PMEST). Time and Space are general and can be applied to any division while Personality, Matter and Energy are specific of each class. PME are considered more relevant and are first in citation order while S and T are considered less relevant. There is a schedule of basic subjects that usually defines the main subject area. Facets in all the schedules are ordered in the citation order—PMEST. This citation order provides the 'general' formula used for synthesizing terms from different facets as shown in the following examples:

1. X,61.73'N6 "Monetary economics in USA during the 1960's". Here the main class is X Economics, and the facets represented are Personality (money), Space (USA), and Time (1960) in the order PST. "Monetary economics in the 1960's" would be X,61'N6 and "Economics in the USA" X.73.

2. L,185:5 "Hygiene of the eye". Here L Medicine is the main class, and the facets are Personality (eye) and Energy or activity (hygiene) in the order PE.

Another contribution of CC is the introduction of the *empty digit* concept which is a notational devise to allow hierarchies to expand at any level without a need for reclassification. If there is a need to include a new class, say, between classes L and M, then X can be used as an emptying digit so that the new class

between L and M would be denoted by the symbol LX. The resulting arrangement would be L, LX, M.

The major drawbacks in CC are its complex notation, as in UDC, and the complexity of the synthesis rules. These require extensive intellectual and practical effort to master. CC is mainly used in India and, because of its late appearance, has not been widely accepted.

CC is considered the most advanced classification scheme and has contributed significantly to modern theory of classification. The concepts of facet analysis and concept synthesis were developed by Ranganathan.

Summary

From this brief survey of the major library classification schemes the following observations are drawn:

1. The adoption by a classification scheme of a particular order of universal knowledge does not seem to have an effect on its success or failure as a tool for the production of systematic order.

2. An enumerative scheme is not practical beyond small collections and for uses other than physical collocation. Enumerative schemes are usually avoided for document retrieval, index construction or classification of document abstracts.

3. Schemes based on rigid, centralized hierarchies become malformed structures as knowledge expands. Hierarchies usually grow deeper in certain disciplines failing to accommodate new disciplines at the appropriate level.

4. The success of a classification scheme does not depend on its sound theoretical foundation but rather on how much effort is invested by its sponsors to promote its adoption.

5. Schemes based on literary warrant seem to be adaptable to the needs of users and naturally expandable.

6. Faceted schemes are well suited for very detailed subject descriptions and for document retrieval. Faceted schemes are also naturally expandable.

7. Notation in faceted classification schemes seems to be a self–imposed barrier and an unnecessary hindrance. The same terms in the schedule could be used in the synthesis rather than some artificially assigned symbol.

8. A generic citation order applied to all facets and a set of generic synthesis rules are essential requirements for a scheme aiming at a certain degree of standardization.

9. Faceted schemes seem to be amenable to automation. Their organization as separate lists of terms and the prescribed order in which terms from different facets can be combined make faceted schemes attractive for data base implementation.

This survey has shown how library classification has evolved from highly enumerative and monolithic hierarchical structures into faceted and hierarchically distributed structures based on synthesis for construction of classes. Historical evidence has been presented that faceted schemes are more advanced and better suited for large, continuously expanding collections requiring exact classification of highly specific subjects. New faceted schemes have not been widely accepted mainly because of the enormous inertia current schemes have accumulated along several decades of intensive promotion.

Software Classification

Software classification schemes are very recent compared with general library classification schemes. The first general classification scheme for 'computing' was introduced by the Computing Reviews (CR) of the ACM in 1960 [SAMM85] listing 33 computer related topics in alphabetical order with no subdivisions. Computer Science classification schemes have been developed ad–hoc as the field has evolved in recent years. The CR scheme has been revised several times and has changed from the original informal subdivisions of the 1960 edition to a three level hierarchy with eleven main subject descriptions for the 1982 edition [SAMM82].

The most developed schemes in the field are the ones for ***general computer science*** (e.g., CR), for ***computer program libraries*** mainly of functional collections for scientific applications (e.g., GAMS, SHARE, SSP, IMSL, SPSS), and for ***software directories*** listing descriptions of integrated application programs. These schemes share the following general characteristics:

- They are enumerative.
- Each has its own notation.
- Their notations are borrowed from the major library classification schemes.
- They are not based on a logical arrangement of knowledge.
- Their classes are not arranged in any particular logical order.

Each major library classification scheme has developed its own schedule for Computer Science. These schedules are very general as illustrated by the example in the previous section where Software is the lowest level class in the LC scheme. The CR scheme (1982 edition) which is essentially derived from the AFIPS' Taxonomy of Computer Science and Engineering [AFIP80] is a much more specific scheme

than the ones in the major classification schemes. Its structure is similar to the DDC scheme but notation is carried to the third level only; lower level classes are not assigned any notational code. Lower level classes are kept as lists of terms thus facilitating expansion of the scheme. The basic high level hierarchical structure (first three levels) has been kept constant since 1982 in an effort to preserve a standard high level structure as was done in the DDC system.

The CR scheme is a very comprehensive listing of 'computing' related subject areas aimed at organizing collections of computer science related reviews, but, unfortunately, it is not appropriate for classifying computer programs. The schemes developed for program libraries and for software directories, on the other hand, are aimed at classifying programs. Computer program libraries are essentially functional collections of small, problem–oriented single–function programs classified by the type of problem they solve. Software directories tend to be collections of descriptions of large, integrated, multi–function systems classified by application area. The subsections below discuss the classification schemes used in program libraries and in software catalogs.

Computer Program Libraries

Computer program libraries are mainly collections of functions aimed at solving specific problems. The classification is by the type of problem they solve. The first classification scheme for computer programs was introduced by the IBM Users Group (SHARE) in 1963 [SHAR63] and a 'new' revised version appeared in 1973 [SHAR73]. The SHARE system is a three level hierarchical enumerative scheme with a decimal notation similar to DDC. John Bolstad [BOLS75] proposed a modification to the SHARE scheme by adding new subdivisions and by rearranging some classes and some subdivisions. Bolstad's scheme has 22 main classes:

A	Arithmetic, Elementary Operations on Polynomials
B	Evaluation of Elementary and Special Functions
C	Roots/Zeroes of Functions, Simultaneous Nonlinear Equations
D	Operations Involving Derivatives and Integrals
E	Interpolation and Approximation
F	Operations on Matrices and Vectors
G	Statistical Analysis and Probability
H	Operations Research Techniques, Simulation and Management Science
I	Optimization: Minimizing or Maximizing a Function
J	Input/Output
K	Internal File Manipulations
L	Language Processors
M	Data Handling
N	Debugging
O	Simulation of Computers and Data
Q	Service Routines: Programming Aids
R	Logical and Symbolic
S	Information Retrieval
T	Applications and Application–Oriented Programs
Z	All Others

This scheme's emphasis is on mathematical software. Classes A through I are basically classes of mathematical problems, each with an average of 25 subclasses. Classes J through T, on the other hand, are not mathematical problems. They average only eight subclasses. Bolstad's scheme is very specific in describing classes of mathematical problems. Some classes have up to six hierarchical levels as illustrated below:

I	Optimization
I2	Constrained Optimization
I2b	Nonlinear Programming
I2b2	Nonlinear Constraints
I2b2a	Equality Constraints Only
I2b2a1	Derivatives Not Required

The scheme attempts to capture all constraints for each class of problem in an enumerative listing. The notation consists of different types of symbols for each

hierarchical level with capital roman letters for the first level and alternate numerals and lower case letters for subordinated hierarchies.

Bolstad's scheme has served as a basis for newer schemes. The classification scheme of the Guide to Available Mathematical Software (GAMS) of the National Bureau of Standards [GAMS80, BOIS83], for example, is a modification of Bolstad's scheme specialized for classes of mathematical and statistical programs. The scheme used by the IMSL Library [IMSL84] is another example of a modified Bolstad's scheme. The scheme used by the SPSS [SPSS84] is basically an expansion of Bolstad's class G—Statistical Analysis and Probability including some terms from other classes.

These schemes for program libraries do not have their classes arranged in a logical order as do library classification schemes. Some related classes are next to each other while others are separated. In the GAMS scheme, for example, related classes H—Differentiation and Integration and I—Differential and Integral Equations are next to each other while related classes E—Interpolation and K—Approximation are kept separated.

Despite these drawbacks, these schemes seem to meet the needs of the users by imposing some particular order to the collections. The users of the collections are interested in using the programs to solve particular problems, that is, a program in the collection is 'used as is' rather than modified for a new environment. With a 'use as is' objective in mind, an enumerative scheme, although extensive, seems to work effectively for functional collections because each listed problem is unique and usually has few methods to solve it. For a collection of code fragments intended for modification, a more specific classification, including implementation details, would be required. This makes an enumerative scheme impractical.

Software Catalogs

The recent growth of commercially available software packages has created an explosion of software directories, each with its own classification scheme. (e.g., [ICP83, IDS83, IBM83, APPL83]) All these schemes, however, share the following two characteristics:

1 All classify by application area
2 The schemes are changed continuously, usually for each edition.

The most refined scheme is the one used by the International Computer Programs (ICP). It is a hierarchical scheme with decimal notation similar to DDC with classes listed in alphabetical order. The ICP classification is substantially revised with each new edition, usually twice a year. The scheme of one given edition can not be used to locate programs in another edition. The 1982 edition of ICP consisted of three main divisions:

1 Systems Software
2 General Applications
3 Industry Specific Applications

There were for example, 13 main application areas for the Industry Specific Applications division ranging from class 31—Communication and Media to class 43—Utilities with up to four hierarchical levels. The 1985 ICP edition has been expanded considerably. The major breakdown of the 1985 edition has seven subdivisions:

1 Systems Software
2 General Accounting Systems
3 Management and Administration Systems
4 Banking, Insurance and Finance Systems
5 Manufacturing and Engineering Systems
6 Specialized Industry Systems
7 Microcomputer Systems

The change in classification from the 1982 edition to the 1985 edition is illus-

trated below:

1982		1985	
35	Finance	41	Banking and Finance
35.1	Banking	41.1	Bank Management
35.1.1	Bank Management	41.1.2	Operation Support
35.1.1.1	Bank Planning		
35.1.1.2	Bank Operations and Support		

Explosive growth of commercially available software packages is responsible for the continuous revisions of the classification schemes. For example, The Software Catalog—Microcomputers, fall, 1983 edition [SOFT83] listed 9740 program descriptions compared with 15,500 for the winter 1985 edition [SOFT85], a 63% increase in only 15 months. Even though, classification schemes for software catalogs have been expanding continuously, they remain highly enumerative and the listed classes are still very general. Several programs are usually listed under each class. For example, the winter edition of the Software Catalog—Microcomputers lists 232 programs under class 1.03—Accounting–General Ledger, leaving to the user the task of inspecting each one during the selection process.

A flexible and adaptable classification scheme is needed that can accommodate new classes without major changes in the classification structure and is capable of higher precision in describing program classes. For integrated programs aimed at the *end user* (e.g., use as is), classification by application area seems appropriate. An end user is interested in using the system packages listed in these directories—not in modifying them. There is no need from the user perspective to know about implementation details. For software components intended for modification and adaptation, as would be required by a *software designer/builder*, a more specific classification is required.

Classification by subject area or by application, complemented by detailed

catalog description is apparently sufficient for the end use type of application. The designer/builder, on the other hand, wants to know *what* is inside the package. He or she wants to have a software collection organized by characteristics relevant to reusability factors like what a functions does, how it is implemented, in what environment it operates. Different classification approaches must be explored to satisfy these needs.

In reusing software, a designer/builder needs access to code fragments within software systems. Current software classification schemes do not classify code fragments; they stop at the system or package level. In the book domain, this requirement would be equivalent to classifying chapters of books or even sections within chapters or paragraphs within sections. Current library classification schemes organized by subject areas would be too general to classify at that level. Other attributes particular to chapters or sections should be added to the general subject areas. A software reuser in this analogy would be a book writer (author) generating books by compiling and adapting existing chapters, sections, and paragraphs from different sources. A software classification scheme must be capable of providing very specific classifications by collecting terms with different descriptive characteristics.

Summary

Current software classification practice resembles the early stages of library classification. The schemes are highly enumerative, too general, rigid, and each one is different. There is a different classification scheme for each collection of programs or systems. There is no unique classification scheme for classifying all programs or all systems. Current software classification is aimed at the end user not at the software designer/builder.

Conclusion

This chapter presented the basic concepts of classification as conducted in library science. Five major classification schemes were surveyed illustrating how library classification has evolved in the last 100 years from an essentially enumerative approach to an almost purely faceted approach. It was shown that enumerative schemes are extensive, rigid, difficult to expand and cumbersome to compile while faceted schemes are compact, flexible, easy to expand and easy to make.

The state of the art in software classification was surveyed. It was found that current software classification schemes are enumerative, rigid and not well suited for expanding collections. Software classification practice has been evolving in an ad–hoc fashion as the field develops. Initial enumerative schemes have been proposed and expanded continuously as the discipline grows. Such an approach seems to satisfy the needs of the users considering the current size of the collections. As the collections become larger, however, current enumerative schemes would be impractical.

Classification in software seems to be repeating the same development process experienced in library classification but at a much faster pace. DDC for example, is in its 18th edition since the first 1876 edition while ICP is in its 53rd edition in less than 15 years. Software classification as library classification began with enumerative schemes, but, in contrast with library classification, software classification has remained essentially enumerative. There is no need to wait for collections to become unbearably large to start thinking about a different classification approach.

A faceted scheme is better suited for a large, continuously expanding collection of code fragments requiring high precision in classification and for minimizing retrieval during the reusability process. A faceted scheme provides ease of expansion, high precision, and is well suited for retrieval implementation.

CHAPTER 4

A Classification Scheme

The classification of software components requires an extensive analysis of the reusability–related attributes that are considered when reusing a particular component. This analysis consists of tracing the process of component reuse and recording the important attributes considered. The process of constructing a faceted scheme by ***literary warrant***, as explained in the previous chapter, consists of collecting terms and grouping them by their relationships. When the analysis is conducted properly, the set of collected attributes form a vocabulary of terms that can be used to describe software components by their reusability–related attributes. Classification is the systematic organization of this vocabulary to display the relationships between the items considered.

This chapter presents the development of a coherent scheme for organizing descriptive terms for classification and retrieval of software components. The first section explains the advantage of using a flexible-facet classification scheme for software. Then a model for classifying software with examples showing how to use the model is presented. A short discussion appears in the final section.

Flexible Facets in Software Classification

The purpose of a classification scheme is to show relationships by collocation—that is, to keep related classes more or less together according to the closeness of the relationship. Defining facets, citation order, and their internal ordering is a critical step in the design of classification schemes. In typical libraries, citation order is 'hard–wired' into the schedules resulting in particular subject groupings

reflecting the criteria of the schedule designer. Library users do not always agree with the classification criteria used because their order of relevance of the facets may be different.

A cross-referenced index partially solves the problem. There are different kinds of cross-referenced indexes for different types of users. For each new item entered in the library, a set of cross-references for that item must be defined. This requirement increases the overhead of the classification process. It also increases the probability of missing a cross-reference since no systematic process is followed when doing it. Cross-referencing is more of a patch than a solution.

One major advantage of an open library is the browsing capability provided to its users. Subjects located closer together in the schedules result in books being physically closer in the stacks. Browsing capability, however, is severely restricted when the schedules are hard-wired as in enumerative schemes.

An ideal solution would be a library where books could be reordered at will. A user would walk in, revise the classification schedules, define a particular citation order and term ordering, have the library reordered, and browse. Since books are physical objects with obvious restrictions imposed on their physical reordering, this idea is not practical.

Reordering can be simulated by keyword searching, provided the user knows what keywords to use. Refering to the example of citation order from last chapter where (WATER INVERTEBRATES) was a class, if the user did not know that there were entries in the habitat facet under RIVER, LAKE, and SEA, keyword browsing would be ineffective. Browsing can be conducted independently in each facet, or by selecting a term in one facet and browsing along another. If the user selects invertebrates in the zoologist–taxonomy facet for example, browsing could be performed along all habitats. Usually browsing is conducted in alphabetical rather than conceptual order (e.g., DESERT, LAKE, MOUNTAIN, RIVER, SEA, TROPIC,

WATER). The relationship among water, river, lake, and sea are conceptual. There is no syntactic relationship among them that could be derived from the language.

An effective classification scheme capable of preserving its browsing capabilities must allow for an adjustable citation order and an adjustable term ordering. Different schedules, adjusted for different kinds of users, could be used on the same collection.

Software components are not physical objects and flexible reordering can be applied without major inconvenience. As long as consistent descriptors are used to identify them, different citation orders can be used to group components by different criteria. A characteristic of code fragments collections is the large numbers of very similar items. High precision in classification is thus required. One way to increase precision is to have a quantitative measure of conceptual distance between any two terms in a facet. Replacing linear lists of terms by some kind of conceptual structure would significantly improve the quality of browsing by broadening the search along a precise conceptual scale.

This dissertation proposes a model of classification that supports adjustable citation order, adjustable term ordering, and a conceptual distance metric among terms.

A Formal Classification Model

Some basic assumptions must be stated before presenting the model. These assumptions are based on the perception of the world of software components as evolving towards a widespread application of reusability in all phases of the software development cycle. As more 'reusable' components become easily accessible at the code level, reusers will try to find an available component before trying to code their own. The need for reusability provides the basis for proposing a classification

scheme for software. The assumptions are the following:

- Collections on the order of thousands of software components at the code level exist.
- There are large groupings of components that perform the same function under slightly different environments.
- Users will prefer to classify their own works into the library rather than work with a specialized librarian.

Given these assumptions, we propose a classification scheme with the following characteristics:

- The scheme is based on *facets*. Facets offer flexibility, extensibility, and precision in classification.
- The scheme proposes a component description format based on a standard vocabulary of terms.
- The scheme imposes citation order.
- The scheme provides a *conceptual metric* to measure conceptual distances between terms in each facet for more effective discrimination among similar items. Rather than extend the number of facets to increase precision, a conceptual distance approach provides precision with a constant number of facets.
- The scheme keeps the ordering of facets and distance measurements among the terms independent of the scheme. This provides a high degree of customization of the scheme in different environments and isolates the collection from these changes.
- The scheme provides a mechanism to evaluate the similarity of items helping the user choose the best.

Concepts and implementation of the mechanism mentioned in this last point are presented in Chapter 6. This mechanism is required because, even if classification offers high resolution, it is possible to have several distinguishable components in the same class. For this purpose an evaluation mechanism that compares features between programs against user needs is included to provide a finer selection

of candidates. This mechanism orders a group of functionally similar components based on a criteria of potential reusability.

A formal description of the classification model is presented below. Component descriptors are defined first. This is followed by a description of two metrics: one for measuring the relevance of facets, the other for measuring the conceptual distance among terms within the same facet.

Component Descriptors

Define a facet F_j as a set of attribute values $\{a_{j1},\ a_{j2},\ ...,\ a_{jm}\}$ where m is finite. Every software component σ has a characteristic descriptor d_σ that is defined as an n–tuple

$$d_\sigma = \langle v_1,\ v_2,\ ...,\ v_j,\ ...,\ v_n \rangle$$

such that σ is an instance of d_σ and each v_j is an attribute value from a facet F_j. Each $v_j = a_{jk}$ for some k.

We define our universe of descriptors as the set D of the Cartesian product in the n facets:

$$d_\sigma \in D = F_1 \times F_2 \times ... \times F_j \times ... \times F_n$$

As the number of facets n gets larger, the descriptive power or resolution of the descriptor increases; that is, there are more elements to describe software components at a greater level of detail. When n is small, precision decreases. Note the inverse relationship between the number of facets used in a descriptor and the number of entities to which it applies. For example, the type DOG applies to fewer entities in the real world than its supertype ANIMAL, but more attributes are required to describe it. This inverse relationship, first noted by Aristotle, is called *duality of intension and extension.*

Let U be the universe of all available components; a function $X\ :\ d_\sigma \rightarrow\ 2^u$

of a particular descriptor $d_\sigma = \langle v_1, v_2, ..., v_n \rangle$ is defined as

$$X(d_\sigma) = \{x \mid x[1] = v_1 \ \wedge \ x[2] = v_2 \ \wedge \ ... \ \wedge \ x[n] = v_n\}$$

where $X \subseteq U$, σ is an instance of some member of X, and U and D are independent. Elements in D are used to describe elements in U. We denote the set produced by X and the function as the same in this section.

In general, X may return the empty set if there are no components in the data base that match a particular description.

To increase the probability of retrieving a non–empty set X, a metasymbol called 'any' written as $\{*\}$ that matches any a_{ij} is introduced. This metasymbol has the effect of reducing the number of elements in the descriptor. If, for example, v_1 is replaced by $\{*\}$ in X above then the new X becomes

$$X = \{x \mid x[2] = v_2 \ \wedge \ ... \ \wedge \ x[n] = v_n\}$$

In particular, the match criterion is:

$$\forall j, \ \ x[j] = v_j \ \text{ or } \ x[j] = *$$

resulting in a reduced specificity of the descriptor. The extreme case when all $v_j = *$ makes $X = U$. Usually, the size of X increases when metasymbols are introduced in the description. This is the typical approach used in information systems retrieval based on keyword indexes where an increase of the 'hit' ratio is obtained at the cost of a loss in precision.

Keyword retrieval has the disadvantage of returning samples where relevant items are mixed with non–relevant items, and there is no way to separate them except by examining each item of the sample and determining its relevance. The KWIC (Key Word In Context) approach simplifies this task by providing a context for each item matching the keyword.

When U is very large, the size of sets X matching different descriptors also becomes large, making the task of discriminating between relevant and non–relevant items more difficult.

One way to obtain groupings of relevant items is by imposing some *arbitrary partial order* on the facets based on their relevance to the user. For our purposes, this is not enough. We will define ordering on facets to be as follows:

$$F_i \succ F_j \quad \text{whenever} \quad i < j$$

One can notice that this defines a full ordering in a set of facets. The symbol "$\succ$" means "more relevant than."

If $n = 4$ and a target description $d_t = \langle v_1, v_2, v_3, v_4 \rangle$ returns $X(d_t) = \{\}$ it can be said that there is a need to 'generalize' the descriptor to make $X(d_t) \neq \{\}$. A further assumption can create two different descriptors based on d_t by introducing the 'any' symbol $*$,

$$d_1 = \langle v_1, v_2, v_3, * \rangle \quad \text{and} \quad d_2 = \langle v_1, *, v_3, v_4 \rangle$$

thus d_1 is a *more relevant generalization* of d_t than d_2 is, and, in fact, d_1 is *the most* relevant generalization of d_t since it preserves the remaining most relevant terms of the descriptor. As a result, the set $X(d_1)$ is more relevant than the set $X(d_2)$.

The positioning of the metasymbol $\{*\}$ thus provides a mechanism for controlling order of relevance desired during retrieval. When more than one metasymbol is introduced the relevance of the generalizations is determined by the position of the leftmost positioned metasymbol. If $d_3 = \langle v_1, v_2, *, * \rangle$ is introduced, the resulting order of relevance of the three cases is $d_1 \succ d_3 \succ d_2$.

Measuring Relevance Using Facets

A measure of relevance may be computed from the positioning of the leftmost metasymbol $\{*\}$ in a descriptor. The relevance of a descriptor $R(d_\sigma)$ may be

computed by:

$$R(d_\sigma) = \min\{j \mid x[j] = *\} \tag{4.1}$$

Applying this equation to the descriptors $d_1 = \langle v_1, v_2, v_3, * \rangle$, $d_2 = \langle v_1, *, *, * \rangle$, and $d_3 = \langle v_1, v_2, *, * \rangle$ we get:

$$R(d_1) = 4, \quad R(d_3) = 3, \quad R(d_2) = 2$$

so $R(d_1) > R(d_3) > R(d_2)$.

This is one formula to compute relevance based on metasymbol positioning that provides an intuitive notion of the role played by the metasymbol in the descriptor.

When the collection becomes very large, sets produced by descriptors with the same degree of relevance also grow leaving the user with the previously mentioned problem of individual inspection of the sample. Precision can be increased by expanding the number of facets but this approach has two major drawbacks. The first has to do with the reclassification of items already in the collection. If a new facet is added, then all existing elements of the collection have to be defined in terms of the new facet. For example, if the facet SIZE is added, each component in the collection must be measured and its size entered under that facet. The second problem has to do with the increasing size of descriptors. It would be desirable to keep the descriptor size short as long as it succeeds in describing the relevant attributes of the object. Extending the number of facets to increase precision would result in long, difficult to use descriptions. If the description of a program is as large as the program itself and cumbersome to use, then the purpose of reusability gets lost as already mentioned in Chapter 2.

There is a clear trade off to be observed here: resolution *vs.* descriptor size. One way to increase resolution without increasing descriptor size is to introduce a metric for terms within the same facet.

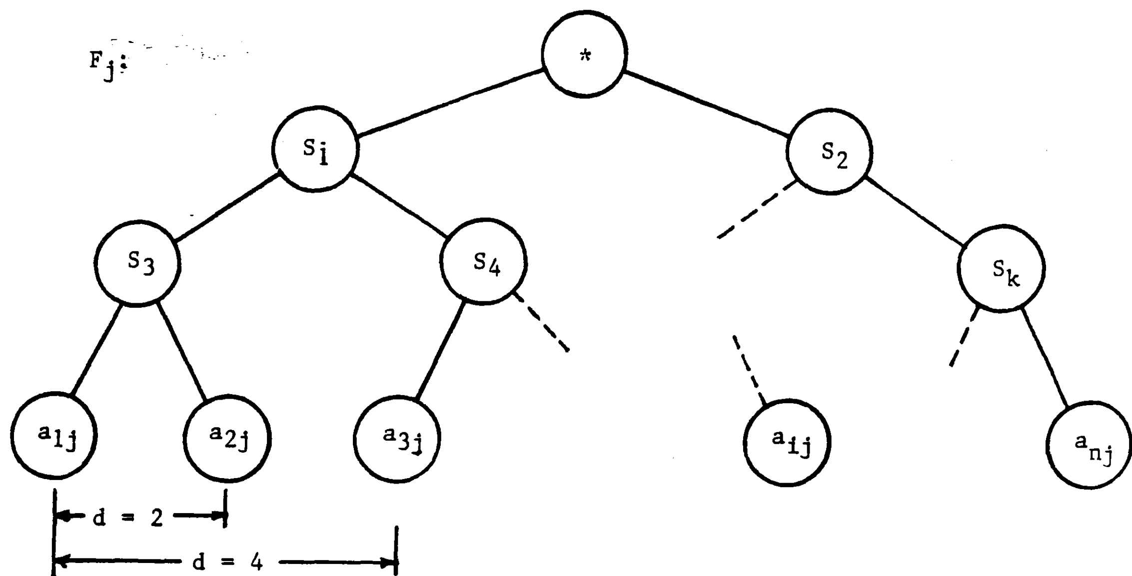


Figure 4.1: Measuring Conceptual Distance in a Tree Hierarchy

Measuring Closeness Between Attributes

A logical approach to measuring closeness among attribute values in the same facet is to associate a type hierarchy with each facet. Each type hierarchy has attribute values at the leaves, $\{*\}$ at the root, and attribute supertypes at the intermediate nodes. If we mark the arcs with the conceptual distance between a node and its parent, the conceptual distance between any two attribute values (in a hierarchy with tree structure) can be computed by locating their *minimal common supertype* (i.e., least common ancestor) and summing the arc weights along the paths between the two as shown in figure 4.1. Here, all arcs are assumed to have unit weight. This approach works in hierarchies that have a tree structure. When the hierarchy structure is a directed acyclic graph (DAG) as is usually the case, we have leaves connected to different sets of nodes at different levels in the hierarchy. Figure 4.2 shows a case of a DAG hierarchy. An arc with no weight on it has an implied conceptual distance of 1. In the case of a DAG hierarchy, a measure of conceptual distance must take into account a more complex notion of closeness.

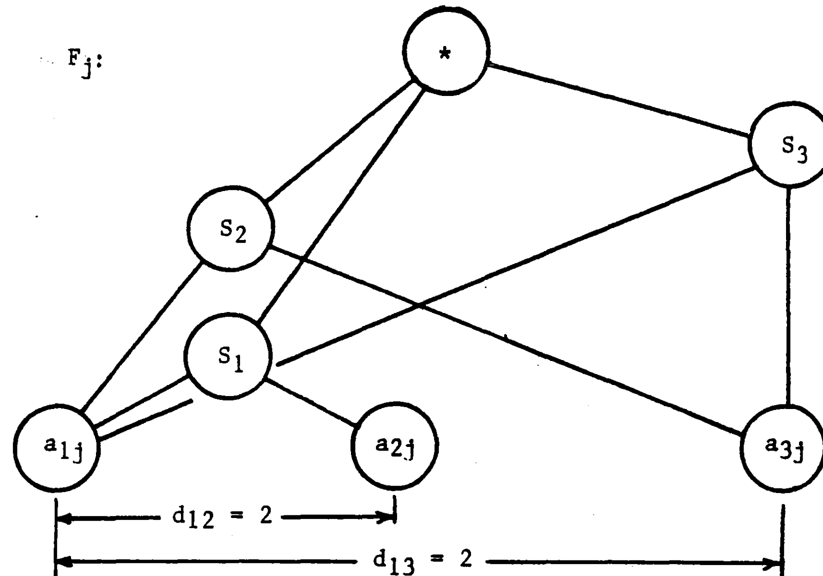


Figure 4.2: Measuring Conceptual Distance in a DAG

Using the same structure in figure 4.2 to construct figure 4.3 but assigning specific labels to nodes, it appears that the semantics specifies the conceptual distance between attribute values and supertypes. These values are user–interest dependent. The level of a supertype in a weighted DAG hierarchy can be associated to the concept of relevance of that supertype as seen from a particular perspective. In this example, arbitrary weights were assigned assuming the perspective of a software designer. The supertype *recursive* is more relevant than the supertype *early HLL* (High Level Language) thus making the conceptual distance between Algol68 and Pascal (2) closer than the conceptual distance between Algol68 and FortranIV (45). We assume in this example that the only supertypes used in the hierarchy are the three shown. A scientific programmer would probably assign a different weighting scale reflecting the relevance different supertypes would have for him.

This example illustrates the intuition upon which the metric is based. A hierarchy of this type can be represented by associating a set of pairs within each attribute value. Each pair consists of a supertype name and a conceptual distance

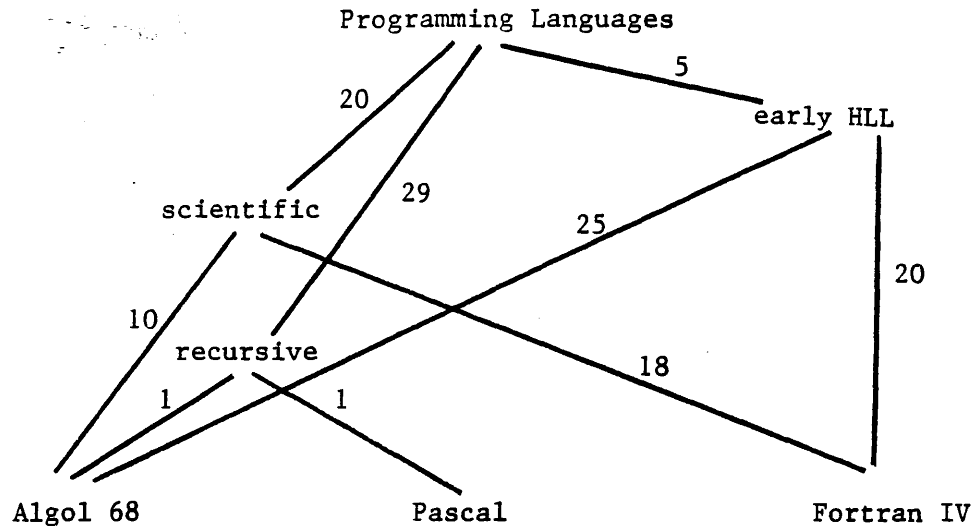


Figure 4.3: Labeling the Nodes in a DAG Hierarchy

between the attribute value and the supertype name. A metric for measuring closeness between any two attribute values consists of listing the set of their common supertypes, computing some function of the weighted path length so determined, and selecting the one with the shortest conceptual distance.

Let s_k be a member of an outside universe of supertypes S, define $Q_{a_{ji}}$, a subset of supertypes satisfying the condition that a_{ji} is an instance of each supertype in the subset as:

$$Q_{a_{ji}} \equiv \{s \,|s \in S \;\; \wedge \;\; a_{ji} \diamond s\}$$

where "$\diamond$" means "is an instance of".

Define $\mathcal{P}(a_{ji},\ s)$, the primitive conceptual distance between any attribute a_{ji} and any supertype in s, as:

$$\mathcal{P}(a_{ij},\ s) \equiv \text{the weight of the shortest path from } a_{ji} \text{ to } s.$$

This definition holds only if $a_{ji} \diamond s$

In this model, a single measure of conceptual distance between any concept

and any of its instances is important. The objective is to measure conceptual closeness between any pair of attribute values. There may be different paths between any attribute and its supertype (s); however, computing a single quantity resulting from $\mathcal{P}(a_{ij},\ s)$ is intuitively appropriate.

There may be several constraints one would like to impose on the assignment of weights to edges. These constraints may be a function of the facet being described or of user dependent characteristics.

The actual conceptual distance between any two attributes a_{ji} and a_{jl} (i.e., within a facet) could be defined by the following algorithm:

1. Compute the set of common supertypes.

$$C = Q_{a_{ji}} \cap Q_{a_{jl}}$$

2. Compute conceptual distance to be the minimum pair of conceptual distances from all the common attributes.

$$\Delta_{i,l} = \min\{\mathcal{P}(a_{ji},\ s)\ +\ \mathcal{P}(a_{jl},\ s)\ \ \forall\, s \in C\}$$

To keep the hierarchy consistent, a rule must be observed that, if a new *supertype* is added to the descriptor of an attribute value, then that supertype must be added to the descriptors of all members of the denotation. The addition of 'imperative' to a programming language descriptor, for example, necessitates the addition of 'imperative' to all programming language descriptors that belong to that class.

Example

To illustrate how the model works, assume a collection where each item can satisfactorily be described by the function it performs and by the objects manipulated by that function. A classification schedule with only two facets is sufficient.

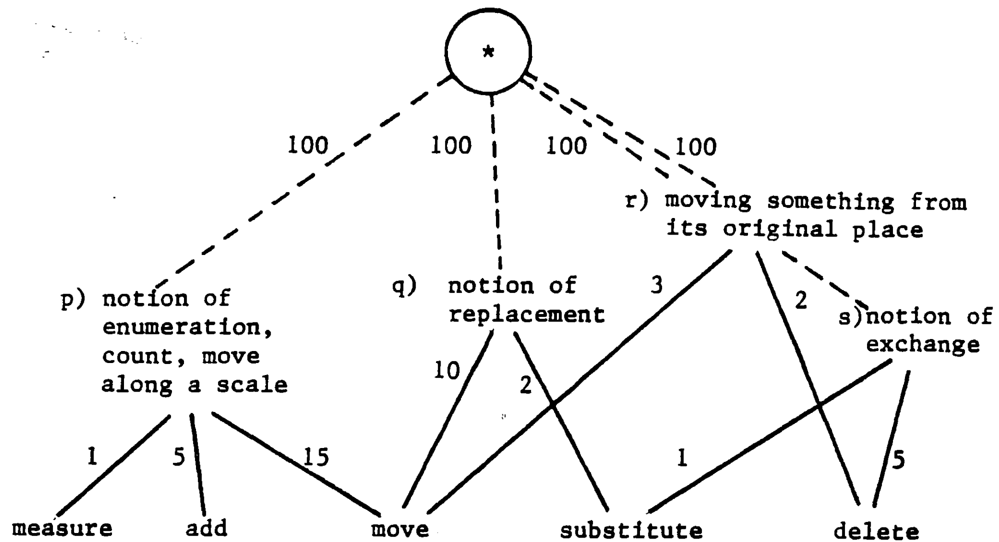


Figure 4.4: A Partial Conceptual Weighted Graph for the Function Facet

Assume further that the schedule is as shown below:

FUNCTION	OBJECTS
measure	trees
move	backspaces
add	files
substitute	lines
delete	tabs
	words

Here $n = 2$, $|D| = 30$, the name of F_1 is FUNCTION, and the name of F_2 is OBJECTS. Citation order is FUNCTION $\succ$ OBJECTS. Some attribute values are a_{11} = measure, a_{12} = move, a_{21} = trees, and a_{22} = backspaces. A partial conceptual graph for FUNCTION is shown in figure 4.4 and a partial conceptual graph for OBJECTS is shown in figure 4.5.

Term selection (i.e., add, move, tabs, etc.) is the result of program description analysis; supertype selection (e.g., a notion in figure 4.4) is user–defined. In this example, supertype selection and weighting is based on the author's criteria.

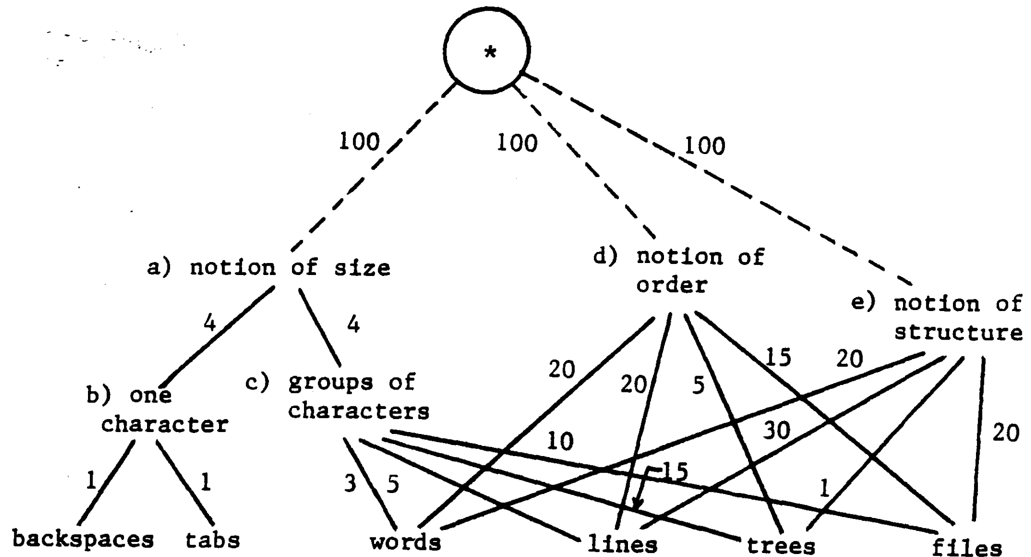


Figure 4.5: A Partial Conceptual Weighted Graph for the Objects Facet

Using the following descriptor as given

$$\langle v_1 = \text{substitute}, v_2 = \text{backspaces} \rangle$$

the goal is a list of the most relevant descriptors based on terms conceptually closer to *backspaces* and to *substitute*. Since FUNCTION $\succ$ OBJECTS for this case, a descriptor with a different objects term is more relevant than a descriptor with a different function term. For example, ⟨substitute, words⟩ is a more relevant descriptor than ⟨add, backspaces⟩. To list the most relevant descriptors, the objects term is broadened starting with backspaces and expanding towards the conceptually closer terms. The computation of the conceptual distance between backspaces and each object in the facet is:

a_{i2}	a_{l2}	C	Paths	$\Delta_{i,l}$
backspaces	*tabs*	[a, b, $*$]	[2, 6, 106]	2
backspaces	*words*	[a, $*$]	[12, 112, 225, 225]	12
backspaces	*lines*	[a, $*$]	[14, 114, 225, 235]	14
backspaces	*files*	[a, $*$]	[19, 119, 220, 225]	19
backspaces	*trees*	[a, $*$]	[24, 124, 210, 206]	24

Order by Relevance	Descriptor
0	*substitute/backspaces*
1	*substitute/tabs*
2	*substitute/words*
3	*substitute/lines*
4	*substitute/files*
5	*substitute/trees*
6	*delete/backspaces*
7	*delete/tabs*
.	.
.	.
29	*measure/trees*

Table 4.1: An Ordered List of Descriptors

The ordered set of objects as they relate to backspaces is therefore:

[backspaces, tabs, words, lines, files, trees]

The same procedure is applied to function.

a_{i1}	a_{l1}	C	Paths	$\Delta_{i,l}$
substitute	*delete*	[s, $*$]	$[6, 106, 207, 204]$	6
substitute	*move*	[q, $*$]	$[12, 112, 204, 211, 216, 217]$	12
substitute	*add*	[$*$]	$[207, 206]$	206
substitute	*measure*	[$*$]	$[203, 202]$	202

The ordered set of functions related to substitute is:

[substitute, delete, move, measure, add]

The set of most relevant descriptors would be those conceptually closer to ⟨substitute, backspaces⟩. From the above results, the ordered list of descriptors is shown in table 4.

If, instead of backspaces, the objects were words, the conceptual distance

computations would be:

a_{i2}	a_{l2}	$\Delta_{i,l}$
words	*tabs*	12
words	*backspaces*	12
words	*lines*	8
words	*trees*	18
words	*files*	13

resulting in the ordered set:

[words, lines, tabs, backspaces, files, trees]

Discussion

It is important to note that the proposed concept structures are usually incomplete. The set $Q_{a_{ij}}$ of supertypes associated to a particular term may be very large. A schedule designer (e.g., librarian) may be familiar with only a few supertypes for each a_{ij} considered. An initially–defined concept structure may look complete to the schedule designer but may be an incomplete structure for another person. A complete concept structure may be difficult to obtain since it must consolidate the perspectives of all the users involved. The flexible weight assignment compensates for this handicap of the scheme. The user–assigned weights provide an acceptable conceptual distance among terms even if only a few supertypes are considered. Weight assignment may be driven in many cases by intuition rather than by considering all possible supertypes.

Weight flexibility provides for schedule adjustment and environment adaptation. A given concept structure with different weight arrangements could serve different types of users; however, different structures may be required if additional new supertypes are considered.

One approach to schedule adjustment is the 'cow path' idea. An initial concept structure is defined for a given set of terms. As the schedule is used, a user's

satisfaction or dissatisfaction with the results is used to adjust weights and, if necessary, to add new supertypes.

In summary, the main features of this classification model are:

- A flexible–facet scheme with user defined citation order.
- A definition of a standard descriptor format for software components.
- A mechanism to increase precision on retrieval.
- A scheme to measure closeness among terms within a facet.

The next chapter shows how this classification scheme is implemented in a computerized library system for software components.

CHAPTER 5

Classification of Software Components

Implementing a classification scheme in a particular domain requires a thorough analysis of that domain. Specialized classification schemes are usually domain specific and are designed around the characteristics of the objects or documents classified. Examples of specialized classification schemes are the work of Simon and Tansey on slide classification [SIMO70] and the work of Moys on law books [MOYS68].

The scheme presented in the last chapter is general enough to be used in almost any domain. Customizing it for the requirements of software components, however, requires an analysis of their characteristics.

In this chapter, the classification scheme is implemented showing how to classify software components, how to start a collection of software component descriptors, and how the scheme and collection can be implemented in a library system. A data base prototyping tool (TROLL [WASS82]) is used to make the prototype library system.

Descriptor Synthesis

Software components can be described by different attributes, such as the function they perform, how they perform it, and their implementation details. These descriptors fall naturally into facets that can be ordered by their relevance to reusability.

The component descriptor proposed has been defined as a tuple of terms. Each term is an attribute value of a selected facet. This section explains why a tuple

format was selected for the descriptor and which facets are required to describe a typical software component. The objective is an effective classification and retrieval of reusable code.

In describing software components for effective classification, its *genus* and its specific difference or *differentia* must be stated. Genus corresponds to 'what it does' which, in turn, is a description of its functionality, and differentia corresponds to 'how it does it'. The 'how' could be reduced to a description of the implementation details.

Several program descriptions and source listings were inspected in order to determine the structure and contents of program descriptions. The sample included over 200 descriptions of commercially available programs and systems from different software directories [ICP83, IPS83, SOFT84], over 100 source listings of modules selected from local programs such as a line editor (2500 source lines of Pascal code), a payroll system (4000 source lines of Cobol code), and a pattern recognition program (1000 source lines of Fortran code). The sample also included several programs from published sources like [WELS80] and [PEMB82] and from programming textbooks.

This exercise led the author to the following observations:

- Program descriptions and comments in source listings are characterized by describing the function performed by the program but usually leave out the details of the implementation.
- There are several diverse programs that perform the same function.
- Description of implementation details are to be inferred, usually, from the source itself, or, if made explicit, the description is a PDL–like listing difficult to capture in a single statement.

There is a need to consolidate in a single descriptor both the what and the how of a program. The descriptor should also be brief and succinct in order to use it as a classification code and as a retrieval key.

Programs performing the same function and designed for the same application have more similarity in their implementation details than programs performing the same function but in different applications. Intuitively, a 'search' program used in a compiler may be functionally similar to a 'search' program used in a data base system. Both may even be implementations of the same algorithm. What is more likely is that the compiler search deals with a particular data structure often found in compilers like symbol tables while the data base search deals with data structures common to data base systems like B–trees. In fact, four of the five compilers analyzed use a table, implemented as an array, for symbol look–up. Three of the five payroll programs analyzed use files with fixed record lengths to store employee information. A sequential search algorithm was used once in a compiler and once in a payroll system. Although functionally equivalent, implementations were very different—mainly because of their data requirements.

In most of the programs analyzed, a close relationship was observed between the similarity of implementation details and the similarity of application. The same similarity argument may be extended to the type of variables used, design strategies, and other implementation details. Semantically rich terms like 'compiler' or 'data base system' usually imply certain approaches to making programs that are typical of the systems they represent. We can say that these terms denote particular ways of implementing programs—some of them by definition, others by common practice.

Environment is another determining factor for implementation details. A particular operational environment may impose certain implementation constraints on a program. There are two environments affecting any program: the internal environment where the program is executed and the external environment where the application is conducted.

Internal environmental differences address the problem of software portability such as interface adaptation. Functional requirements and program structure are

usually left intact. Resolving internal environmental differences means having exactly the same system in a new operational environment—a problem in portability. Portable software design is an area of research in itself, and experiences in the conversion effort of particular systems have been reported extensively in the literature (e.g., *Software Practice and Experience*). Some formalizations to characterize the problem have been proposed [TANE78, STAN76], and new approaches to achieve automatic conversions have been tried [ARAN85].

External environmental differences, on the other hand, are those that require modifications of the design or specifications of a program. External environmental changes typically occur when a new set of requirements are proposed for an existing system. This type of activity is known as software evolution and involves the continuous adaptation of software systems to a changing environment. Modifications at the design or specifications level require program modifications more typical of reusability. External environments are usually synthesized in single 'semantically rich' terms generally used to denote different external environments such as 'payroll', 'general ledger', or 'materials control'.

These observations motivated the decision to use 'application' and 'external environment' as an approximate descriptor of the 'implementation details' relevant to the reuse process.

Functionality

If a program's functionality denotes 'what it does' (e.g., compare two files for equality), and its environment[1] denotes 'how it is different from others', (e.g., requires files open, compares line by line, component of a file management system) then, for classification purposes, the facets *functionality* and *environment*, capture what is essential for a description.

[1] Environment here means developmental, implementational, and operational.

Functionality and environment alone are still too general for an accurate description of a software component. Single, semantically rich terms may be ambiguous if used alone. Some context dependency is required to make the descriptor more accurate.

Imperative statements are generally used to describe functionality of a program fragment. From a sample of more than 250 source programs from different sources such as programming books and software catalogs, it was observed that all program descriptions had at least one imperative statement describing the main function of the program. This is a natural result considering that most commonly used programming languages are imperative. Typical programming techniques and design methods are based on imperative constructs.

An imperative statement is defined by the triple [*action*, *object*, *agent*]. The same format is adopted here using three facets to represent functionality:

function, *objects*, *medium*

Thus, functionality contributes three facets to the descriptor.

Function is the name of the specific primitive function or action performed by the program, (e.g., move, start, compare). *Objects* refer to the objects 'manipulated' by the program (e.g., characters, lines, variables.) *Medium* refers to entities that serve as 'locale' where the action is executed. These entities are sometimes the supporting structures for the functions. *Agent* (the conductor of the action), as applied to programs, does not change. The program always carries the action. In describing programs, the definition of agent is irrelevant. It is more relevant to describe what supporting structure is used to carry the action or 'where' the action is executed because it tells more about the details of the action being described. Examples of medium terms are: line, table, file, keyboard. In the sample analyzed, *medium* was not usually included in the description; 45% of the descriptions ana-

lyzed mentioned what medium was used. In several cases, direct code inspection was required to determine what medium was used.

In most of the programs analyzed it was found that the triple [*function*, *objects*, *medium*], was adequate for describing "what" a program does. The list below shows some program descriptors.

$\langle$input, characters, buffer$\rangle$
$\langle$substitute, tabs, file$\rangle$
$\langle$search, root, B – tree$\rangle$
$\langle$compress, lines, file$\rangle$

The idea of describing program functionality with an imperative triple is supported by the work of Sugarman [SUGA81]. In his dissertation, Sugarman presents a model to describe dissertation abstracts with a triple of the form

[*operation*, *objects*, *properties*]

His model is intended for dissertations dealing with the presentation of experimental results where an experiment is performed to demonstrate a hypothesis. The operation facet deals with the execution of the experiment, the objects facet describes the objects of the experiment, and the properties facet is the description of the results. He found that 65% of all sentences found in a sample of 167 dissertation abstracts were written in the triplet format. A classification and retrieval test showed better performance than techniques based on keywords.

Sugarman's model is similar to this one in that the item described performs an action on some objects. The difference is that, in the case of dissertation abstracts, the results are analyzed while, in the case of programs, a medium is used to perform the action.

Environment

As indicated before, knowing a program's intended application and its external environment provides *indirect* knowledge of its general characteristics. Some

lyzed mentioned what medium was used. In several cases, direct code inspection was required to determine what medium was used.

In most of the programs analyzed it was found that the triple [*function*, *objects*, *medium*], was adequate for describing "what" a program does. The list below shows some program descriptors.

$$\langle \text{input, characters, buffer} \rangle$$
$$\langle \text{substitute, tabs, file} \rangle$$
$$\langle \text{search, root, B} - \text{tree} \rangle$$
$$\langle \text{compress, lines, file} \rangle$$

The idea of describing program functionality with an imperative triple is supported by the work of Sugarman [SUGA81]. In his dissertation, Sugarman presents a model to describe dissertation abstracts with a triple of the form

[*operation*, *objects*, *properties*]

His model is intended for dissertations dealing with the presentation of experimental results where an experiment is performed to demonstrate a hypothesis. The operation facet deals with the execution of the experiment, the objects facet describes the objects of the experiment, and the properties facet is the description of the results. He found that 65% of all sentences found in a sample of 167 dissertation abstracts were written in the triplet format. A classification and retrieval test showed better performance than techniques based on keywords.

Sugarman's model is similar to this one in that the item described performs an action on some objects. The difference is that, in the case of dissertation abstracts, the results are analyzed while, in the case of programs, a medium is used to perform the action.

Environment

As indicated before, knowing a program's intended application and its external environment provides *indirect* knowledge of its general characteristics. Some

implementation details can be inferred from these factors. Thus, environment description is an indirect characterization of classes of programs. An environment (in the general sense), usually determines:

- Certain programming practices, style, and structure.
- Types of data structures used.
- Type of information manipulated.
- Particular computational methods and algorithms.

From the previously mentioned examination of over 300 program descriptions and source listings, environment related words were selected and grouped into class groups. Three major groups resulted: 1) terms describing the type of system, 2) terms describing the functional area, and 3) terms describing the location or setting of the application. Environment, therefore also contributes three facets for the descriptor: *system–type, functional–area,* and *setting.* System–type descriptions usually refer to one or more of the following:

- A functionally identifiable module.
- An application independent module.
- A unit larger than a single component.
- A group of components that perform an identifiable function.

Examples of types of function descriptors are report–formatter, lexical–analyzer, scheduler, retriever, expression–evaluator, and interpreter. In our term–grouping experiment, system–type terms were seldom found in descriptions of software packages. Most system–type terms were obtained from source code documentation.

Functional–area describes a particular identifiable function performed in an application area. It is application dependent and may be implemented with different types of programs. Functional–area is usually defined by an established set of

procedures on an area of application. Examples are general–ledger, cost–control, operating–system, and purchasing.

Setting describes the location where the application is exercised. Setting captures details of how to conduct certain operations. For example, a manufacturing–control program in a print–shop may have a slightly different implementation than a manufacturing–control program in a chemical–plant. The differences here are mainly in the units used and in the objects handled. The functionality does not change.

Six terms are therefore required to describe a code fragment—three terms to describe functionality and three terms to describe environment. Each one of the six classes of terms is grouped under a facet. Refering back to the formal model of Chapter 4, $n = 6$, $F_1 = function$, $F_2 = objects$, $F_3 = medium$, $F_4 = system-type$, $F_5 = functional-area$, $F_6 = setting$. Relevance order among facets was assigned based on the type of information they provide for reusability purposes. For example, program function is considerably more relevant than where in an application the program is used. This particular relevance ordering is based on the author's criteria.

Schedule Construction

The typical procedure for making a faceted scheme in library classification is by *literary warrant* [BUCH79, VICK60]. From a representative sample of titles of the collection to be classified, descriptive terms are extracted. For example, from the title "Animals of the Mountains" the terms 'animals' and 'mountains' are separated. Terms are then grouped by classes to form facets and, if necessary, subfacets are created to group more specific sets of terms.

The approach taken in this dissertation, although based on literary warrant, has some subtle differences. Facets are defined *a priori* based on a 'descriptor'

criteria that minimizes the number of terms required for an 'adequate' component description (as described above). Adequate means that it includes reusability–related attributes. This *a priori*–defined facets approach is supported by the work of Austin on the PRECIS system [AUST71]. The PRECIS system is a computerized system for document retrieval based on a predefined semantic structure of terms used for queries.

With a predefined skeleton of reusable–relevant facets, the task is how to flesh it out with appropriate terms. The three complementary approaches used are:

1. Reliance on functional models of well–defined systems (top–down).
2. Strict bottom–up approach by analysis of existing programs.
3. Analysis of existing software classification schemes.

In the first approach, a well known system model, such as compilers, is selected. A preliminary schedule is constructed by analyzing some instances of the model. To make the compiler preliminary schedule of figure 5.1, for example, five small compilers were examined. An analysis of the tasks performed by each component resulted in an unordered list of names (terms). After an iterative process of grouping common terms several times, facet names were given to each defined group. *A priori* knowledge of the general compiler logical structure was essential in this process. The break–down of a compiler into well known and widely accepted subfunctions (lexical, syntactical, and semantic analysis followed by code generation) provided a global structure for a *top–down* approach for making the schedule in the example.

This preliminary breakdown identified some of the terms that could be grouped into the proposed six facets of our scheme. For example, terms such as insert and search describe the function facet; terms like integer, character, loop–statement,

```
COMPILER COMPONENTS
  (By Compiler Subfunction)
    scanning
    lexical analysis
    syntax analysis
    semantic analysis
    code generation
    symbol table handling
       initialization
       identifier handling
          insert
             search
          retrieve
             search
          search

  (By Type of Instruction Generated)
    program control
    procedure control
    storage allocation
       data structure definition
    storage management
       data structure operation
    variables
    expressions
    assignment
    label generation
    address assignment
          .

  (By Method Used)
    top-down
      recursive descent
      predictive parsing
    bottom-up
      shift reduce
      operator precedence
             .
             .
  (By Type of Character Analyzed)
    identifier
    constant
       real
       integer
       boolean
       character
           .
           .

  (By Type of Statement Analyzed)
    block, procedure, program
    declaration
    control
       looping
       conditional
       unconditional
    I/O
           .
           .
```

Figure 5.1: A Preliminary Faceted Schedule for Compiler Components

and block, go with the objects facet; and lexical–analyzer and code–generation are terms fitting the systems–type facet.

An advantage of this approach is that only a few instances of programs have to be analyzed as long as a widely recognized logical structure of those programs is identified.

As more models are analyzed, many terms would repeat and fewer new terms appear until all possible terms are defined. In the end, each facet would have most of the terms used by most of the program descriptions of programs in that particular facet. The objective would be to produce as complete a schedule as possible. This objective is practically unrealistic, mainly because of the amount of effort required to produce a complete schedule and because of the continuous expansion of knowledge. Book libraries, for example, have been expanding their schedules continuously for several decades and there is no evidence of change in that trend. The objective in this classification scheme is not to provide a complete schedule but rather to provide one that is easy to expand and to provide the guidelines to do so.

The second approach is called "bottom–up" because there is no predefined structure to help in grouping terms. Program descriptions are examined for terms that fit in any of our six facets.

In less standard applications, like business, schedule making may require large samples of programs to come up with representative terms. Fortunately, some of this work has been conducted previously by Goodell [GOOD83]. He identified twenty–four "primary functions" after studying 1,338 business applications programs. Lanergan and Poynton conducted a similar study [LANE79] on a sample of 5,454 business applications programs. They identified six different categories of "program types" that were later reduced to three. Goodell's study was aimed at finding *common functions* while Lanergan and Poynton's objective was to find *common programs*. The availability of this initial set of function terms was very

helpful in constructing the preliminary schedule proposed in this thesis (Appendix B).

In other areas, program descriptions were collected, mainly from software directories, to build a preliminary schedule by literary warrant. Terms were then selected and assigned to the six facets already defined. Appendix A shows a preliminary schedule for the applications area of Communication and Media. It was extracted from descriptions of programs in software directories.

In the case of inspection of code fragments where no program descriptions existed, a description had to be generated. A criteria was established during this process to determine what is actually a code fragment: Code fragments are pieces of code embedded in existing software systems. To consider a code fragment as a classifiable item, it must perform a readily identifiable and potentially modular function that can be described by the [function, objects, medium] triplet of our descriptor. We call these requirements 'functionally coherent' and 'functionally describable'. Selecting code fragments from a software system may be a time consuming task. This is a one–time investment that is paid off during reusability.

The problem of description generation from embedded fragments can be substantially reduced if software systems were originally designed for reusability. By this we mean proper modularization and proper documentation. Most existing programs, however, have not been designed with reusability criteria in mind.

The third approach was, basically, to reuse some of the terms in other software classifications. Software directories such as International Computer Programs [ICP84] and International Directory of Software [IDS84], offer very extensive and well–defined classification schedules. The objective of these directories, and consequently of their schedules, is to classify integrated software packages for the user and not for the reuser. Software packages are intended to be used as delivered and not to be reused as components of new packages. Placing this difference aside,

the classification schedules analyzed were rich in environment description terms—in particular, terms describing functional areas and settings.

Other software classification schedules inspected were those for functional collections [BOLS75, GAMS81, BOIS83]. These schedules are abundant in function-related terms, particularly for mathematics, statistics, and numerical methods.

With an initial schedule of terms available to generate descriptors, the next logical step would be to use it for classification and retrieval. However, the schedule tended to grow very large in number of terms and many terms described the same concept. To keep the size under control, each facet was listed as a thesaurus.

Vocabulary Control

Describing code with a tuple of specially selected terms is not free of problems. Synonyms can produce different descriptors for the same component. In the case of functionality, for example, the descriptor $\langle move,\ words,\ file \rangle$ and the descriptor $\langle transfer,\ names,\ file \rangle$ may be two different descriptions of the same program.

To avoid duplicity of descriptors a *controlled vocabulary* is required. A terms thesaurus is needed to group all synonyms under a single concept. The term that best describes the concept is selected as the *representative* term. The term thesaurus is used for vocabulary control and to broaden existing index vocabulary. The result is an enhancement in recall performance because several terms are assigned to a single concept. A thesaurus may also be used as a technique to control the size of the schedules either, by increasing the number of terms assigned to a particular group or by breaking up groups of terms.

Thesauri can be constructed either manually or automatically. Automatic thesauri construction is mainly used for documents and is based on term frequency analysis and/or on statement construction analysis. Weights are assigned to terms

in proportion to their frequency of occurrence in a document. Some document retrieval systems use this concept with relative success [SALT83, SPAR71] although at a relatively high initial cost. Full texts have to be converted to machine readable form. More recently, this approach has been demonstrated not to be cost–effective in very large collections [BLAI85].

Automatic thesaurus construction might not be cost–effective when applied to code fragment descriptions because program descriptions are usually short, in–line comments are different from continuous text, and source code is context free. On top of these characteristics, collections of code fragment descriptors are expected to be increasingly large—on the order of several thousands. Another major drawback to automatic thesaurus construction is that, in many cases, code fragment descriptions and, in particular, descriptions of embedded modules, are not available. The assigned librarian has to inspect code and available documentation in order to extract an appropriate description. Therefore, manual thesaurus construction is advocated here.

Below, how to manually construct a thesaurus for code fragment descriptions and how to approach the problem of term groupings for both sets of facets, functionality and environment, is discussed.

Manual Thesaurus Construction

For the function facet, terms were collected that describe actions performed by a program. Sources for term selection included program descriptions, comments in source code, computer science books, software directories, and English language thesauri. Terms describing the same function or action were grouped into what is called a "thesaurus class" [SALT83]. In contrast with the accepted thesaurus construction practice of assigning a unique code to each thesaurus class, a term from each group was selected as the *representative* term of its respective group.

Criteria for representative term selection was based on the author's judgment. For each identified group, selection of representative terms was reduced to answering the question: Which of the terms in the group best describes the desired functional concept?

The selected term for a concept was entered as the schedule entry for the function facet. For example, the term ***substitute*** was selected to represent the group:

replace/transliterate/convert/change/map/crypt/update

For the objects facet, as well as for the rest of the facets, the same procedure was followed. Names of objects used by programs were searched for in computer science books, program descriptions, and software catalogs. Terms were then grouped and a representative was selected for each group.

In the objects facet, two classes of terms were identified: those of names given to the actual objects manipulated by the programs (e.g., arrays, variables, words) and those of names given to the objects being represented (e.g., paycheck, invoice, answer, volume). The author was interested in terms describing the actual objects not their representations. For example, in a check printing program, the desired descriptor is not the pair ⟨*print*, ***checks***⟩ but rather ⟨*transfer*, *buffer*⟩. 'Transfer' is the selected term for the group where 'print' is a member and 'buffer' is the actual object manipulated by the program. A check printing program is just an output program for a particular application (e.g., payroll). The keyword 'payroll', as will be shown below, is a term in one of the environment facets.

For the medium facet, terms were not usually available from program descriptions. Code inspection was necessary in most cases. Terms usually considered as objects in some programs played the role of medium in other programs. Typical examples are "compress words in a line" and "compress lines in a file". The

functionality descriptors for these two cases are: ⟨*compress*, *words*, *line*⟩ and ⟨*compress*, *lines*, *file*⟩. The term 'line' is a medium for the first program while 'lines' stands for the objects manipulated by the second. In the resulting schedule, many terms in the medium facet are also listed in the objects facet.

For the facets that make the environment (system–type, functional–area, setting), synonyms were seldom found. Most terms in this class consist of compound words such as pattern–matcher or file–handler for the system–type facet; order–entry or cost–control for the functional–area facet; and catalog–sales or car–dealer for setting. A set of partial classification schedules as described in these sections is shown in Appendix B.

Implementation

The required infrastructure, as proposed in Chapter 2, for effective code reuse consists of: 1) a specialized library, 2) a classification scheme, and 3) a support system. A specialized library may be built with the proposed classification scheme of Chapter 4 and with the set of classification schedules available from the previous section. A data base of component descriptors can be considered as the software catalog since the terms in the descriptors are software attributes. The support system may be seen as a group of procedures that help in query construction and in the evaluation of the retrieved sample for potential reusability. By implementation we mean integration of these parts in a single system we call *The Library System.*

The functional objective of the library system is to return a set of potentially reusable software components given a set of terms that describe the attributes of a required component. Figure 5.2 shows an SADT[2] context diagram of the library system.

Multiple objectives were sought when implementing the library system:

[2]SADT is a registered trademark of SofTech Inc.

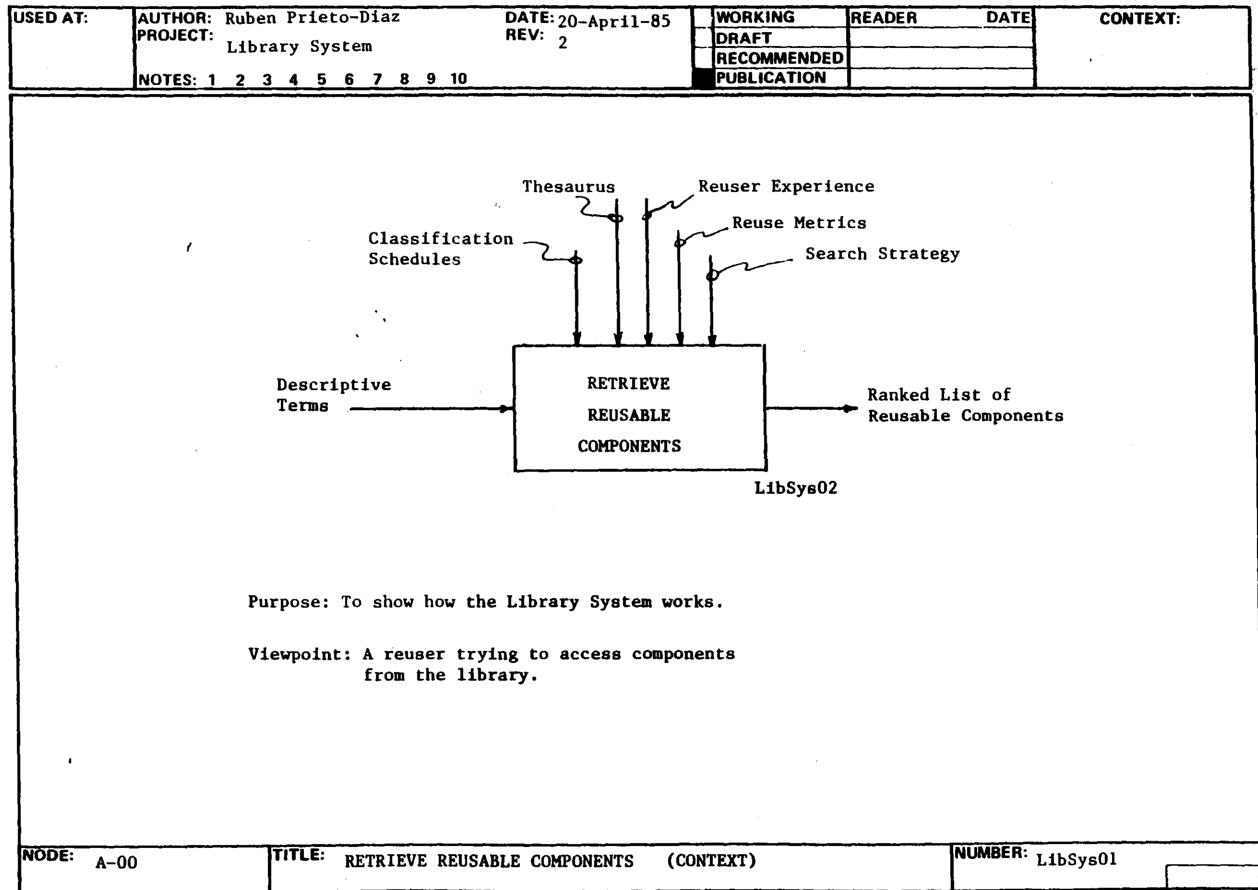


Figure 5.2: SADT Context Actigram for The Library System

- To have an integrated library system for reusable software components.
- To test the effectiveness of a computerized library system based on a classification scheme.
- To use the classification scheme and schedules as a guide for query construction.
- To test the usefulness of a library system that retrieves closely related items rather than exact matches.
- To test the idea of ranking potentially reusable candidates and verifying their usefulness.

The library system was implemented as shown in the SADT diagram of figure 5.3. Chapter 7 presents results on the evaluation of this system and Chapter 6 discusses the selection technique for ranking potentially reusable components (see box A05).

The query system (see A01, A02, A03) can generate one or more component descriptors. The system guides the user in selecting valid terms from the classification schedules and enforces a citation order of the terms based on the established relevance order of the six previously defined facets. A query is a six–tuple descriptor of a component.

A query may be modified by insertion or removal of 'any' metasymbols $\{*\}$, in a prescribed order. A generalization or specialization of the query results.

A query may also be expanded. Queries of closely related terms are constructed based on their relevance distance and on their conceptual distance. Conceptually closer terms are selected first for the new queries. Groups of queries are ordered by their relevance to the original query. The result is an ordered set of queries from most to least 'relationship' to the original query. Scope of expansion is controlled by the user.

Expansion is used when the original query does not retrieve a component description. This is the typical case. Recall from Chapter 4, that the set of possible

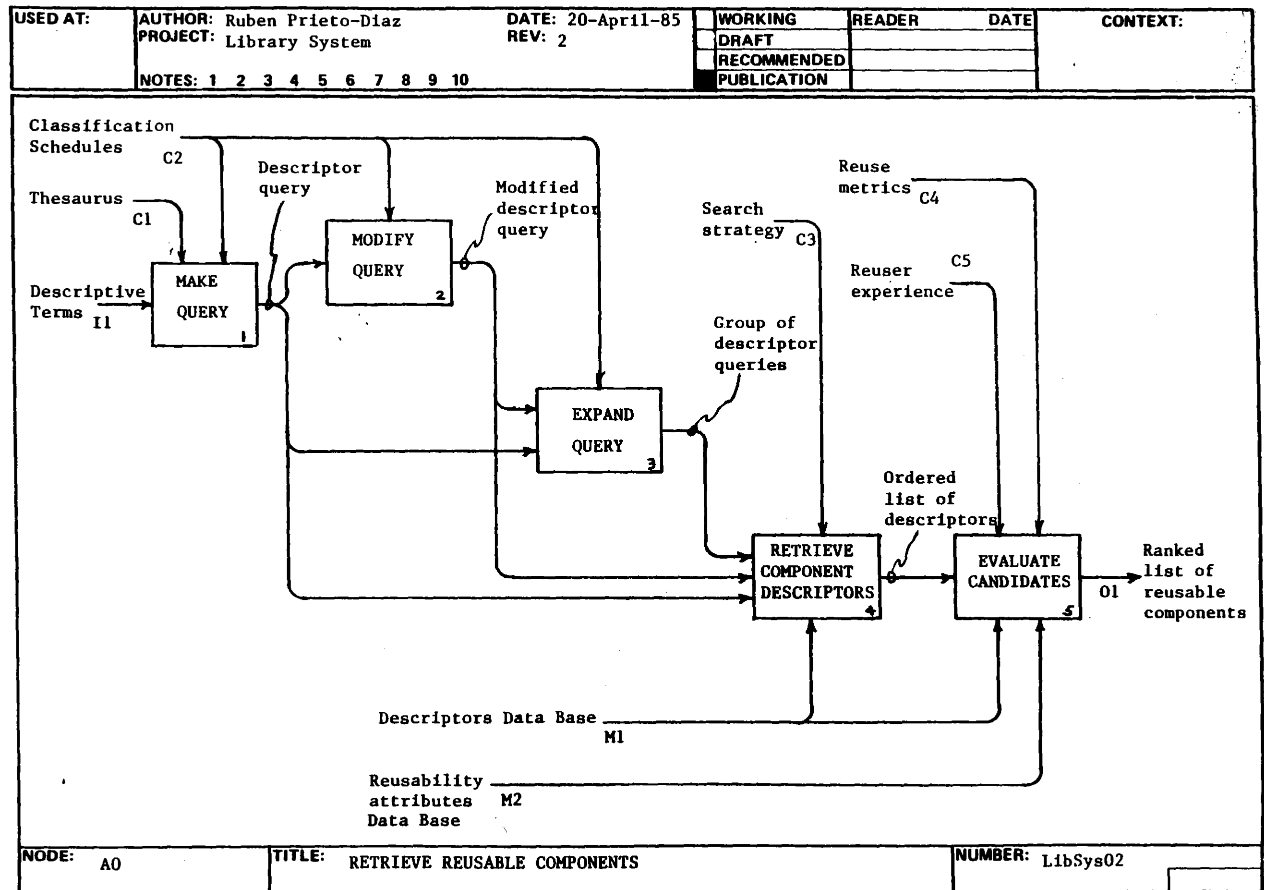


Figure 5.3: SADT Level 0 Actigram for The Library System

descriptors, D, is very large. It may be even larger than the collection itself, thus increasing the probability of a miss. Query expansion is central to the library system. Here is where the features of the classification scheme for measuring relevance and conceptual distances are used.

Retrieval (see A04) is implemented by a relational data base system. The TROLL system was used for this task. TROLL is one of the tools that support the User Software Engineering (USE) methodology for systematic development of interactive information systems [WASS82].

A software prototyping tool based on a relational data base, TROLL was selected mainly for its flexibility to interface with other USE tools, the C language, and the UNIX[3] environment. TROLL uses a compact procedural language that makes available the operations required in relational data base systems. Although initially small, TROLL relations can be easily expanded to a very large size. A major drawback, as with most prototyping tools, is performance. Most of the overhead is generated by its multiple interfaces and by its performance monitoring subsystem.

For implementation of the library system, which is a prototype system, performance was not important compared to the features offered by TROLL. The performance monitoring subsystem, for example, is an excellent feature for experimentation during prototyping.

A query in the library system is interpreted as a collection of keywords to be matched against a relation of program descriptors. Each tuple in the relation consists of the group of six descriptor terms, a group of reusability related metrics, the programming language used, a pointer or reference to the source code and documentation, and a brief description of the component. Reusability metrics and the programming language name are used by the evaluation system (Chapter 6). Relation level operations for queries and tuple level operations for data editing are

[3]Unix is a trademark of AT&T Bell Laboratories.

implemented directly in TROLL. The output of the retrieval system is a formatted table of selected component descriptors in the same order as the submitted queries.

Computing Conceptual Distances

One important aspect of the implementation is the representation of the conceptual structures proposed by the classification scheme. To determine conceptual distances among terms a preliminary concept structure consisting of terms, supertypes, and arc weights is postulated (like the ones shown in figures 4.4 and 4.5). Conceptual distances between terms are computed as proposed in Chapter 4. Path weights are adjusted several times before they are satisfactory.

Addition of new terms to a preliminary conceptual structure requires significant readjustment effort. New supertypes are added and path weights are adjusted again. It was observed that conceptual distances between new terms and terms in the initial set could be 'estimated' by the user through intuition and common sense outside the conceptual structure. Conceptual distances between terms can be nicely represented in a *closeness matrix*. A closeness matrix for the function facet of our example of Chapter 4 is shown below.

	measure	add	move	substitute	delete
measure	0	6	16	200	200
add	6	0	20	200	200
move	16	20	0	12	5
substitute	200	200	12	0	6
delete	200	200	5	6	0

For the prototype library system of this thesis, a closeness matrix was used mainly for rapid implementation. An initial closeness matrix derived from a preliminary structure offers a frame of reference for estimation of new distances. The closeness matrix was adopted for conceptual distances computation because of its simplicity and extensible properties compatible with the proposed conceptual struc-

ture. The explanatory power of a conceptual structure is the main reason for its use in the classification model.

Conclusions

The main contribution of this chapter is a preliminary classification schedule for software components derived from the classification model of Chapter 4. Another contribution is the detailed discussion of how each part of the schedule was derived.

Schedule construction is an elaborate and time consuming procedure, and, rather than aiming for a complete schedule, the objective should be an easy to expand schedule. Ease of expansion is provided by the faceted structure of the classification scheme proposed. Expansion consists in a controlled addition of terms to the schedules. The preliminary schedule listed in Appendix B was derived to classify functionally identifiable code fragments of medium size (e.g., 50 to 200 source lines of code).

Implementation of the library system, in particular the construction of the queries, is based on the structure of the classification scheme and implements the idea of conceptual distance. The library system as specified in the SADT diagrams of figures 5.2 and 5.3 has been implemented in the C language with the TROLL prototyping system and tested in a limited environment. Chapter 7 discusses the results of these tests.

The next chapter presents the idea of component evaluation based on the estimated reusability effort and the integration of this technique into the library system.

CHAPTER 6

Evaluation of Selected Components

After retrieving a sample of component descriptors that are closely related to the given query, there is a need to reduce the amount of reuser effort by helping the reuser select the component that would require the least amount of effort to reuse.

A software component, like any object, may have several intrinsic properties to consider during an evaluation of any kind. A book has properties other than those related to the information it carries such as size, weight, thickness, cover type, cover design, paper texture, paper color, number of pages, number of chapters, and font type. These latter are 'inherent attributes'. For the typical library user, these attributes are usually not relevant. For an editorial house, attributes such as number of chapters or number of pages may be more relevant. For the print shop, attributes like paper texture, paper color, and font type would be more relevant. For the marketing division, cover type and cover design may be the most relevant. For an engineer designing library book shelves, weight, size, and thickness are very important.

Attempting to provide all inherent attributes for each book to satisfy any type of potential user of the book would be impractical. Catalog cards with all that information would be very expensive to produce. Books in libraries are intended to be used by library clients; thus only the information important to them is highlighted, leaving aside less relevant attributes.

If a library user were planning to take a book on a trip and found two books with the same title and same author listed in the catalog, then he or she would look in the catalog cards for information regarding traveling requirements such as

size, weight, and cover type. Both books meet the content requirements, but the user wants to know if there are any differences in these inherent attributes that are relevant for traveling and if these are the only attributes relevant for traveling.

We can carry this example even further, to the hypothetical case of a writer wanting to reuse chapters, sections, or paragraphs from previous work on a given subject area to write a book. In this case, the reuser would collect the materials that meet the content requirements, and evaluate them on their reusability–related attributes such as average size of paragraphs, writing style, and difficulty of subject area. This evaluation would be completed by reading through each book, then selecting the best for the objective—a time consuming process based on subjective evaluation.

What would happen if there were a large number of items to be evaluated? How would cognitive strain affect the evaluation process? These are the questions faced by a software reuser when attempting to select among several functionally and environmentally compatible components. Which inherent attributes should be considered for differentiating among similar components? Can these attributes be used to estimate reusability effort?

One of the objectives of this chapter is to define the minimum number of inherent attributes that can be used to discriminate among similar components (assuming the point of view of the software reuser) and to estimate reusability effort. Another objective is to formalize the evaluation process in order to minimize cognitive strain. The approach presented here ranks the selected sample in order of potential reusability based on multi–dimensional evaluation. Reusability–related attributes of each component are rated and weighted and the sample is ranked.

In the first section, evidence is presented that a small set of attributes may be used as discriminators and as estimators for reusability effort. The next section shows how to conduct a typical multi–dimensional evaluation and how this process

can be modified to make it partially automatic. The last section introduces fuzzy set concepts as a normalization technique for the dimensions considered during evaluation.

Reusability Related Attributes

Evidence is shown in this section that program size, program structure, the programming language used, program documentation, and reuser experience play an important role in program understanding and in program adaptation. These are the reuse effort estimators, and they are a sufficient set of attributes for differentiating among functionally and environmentally identical components.

This research starts from the premise that program understandability and program maintainability are parallel concepts (i.e., the more difficult a program is to understand, the more difficult it is to maintain). Since maintenance staff must ultimately go to the source program, it would be desirable to quantify the relative magnitude of the task through an analysis solely of the attributes of the program. Evidence gathered on the validity of the proposed attributes comes mainly from empirical studies (referenced below) in the area of program maintenance.

The criteria for selection of these attributes were:

- Widely verified and assessed as reliable metrics, at least at the module level.
- Objective.
- Easy to define and easy to calculate.
- Directly related to program understanding and adaptation.

For the proposed software library system, metric simplicity is important in reducing manual measurement effort. A large variety of programming styles and programming languages are expected in the collection, making automatic measurement of the metrics impractical if not impossible.

Program Size

At the macro level (i.e., large programs), program size has been used as the basic estimator for programming effort and as an indicator of program complexity. Several software estimation models are based on program size. In Bohem's Constructive Cost Model (COCOMO) [BOEH81], for example, most estimating equations are based on the expected number of lines of code. Programming effort in man months (MM) is computed by the equation:

$$\text{MM} = 2.4(\text{KDSI})^{1.05}$$

where KDSI stands for "thousands of delivered source instructions". Total development time (TDEV) is also a function of size:

$$\text{TDEV} = 2.5(\text{MM})^{0.38} = 2.504(\text{KDSI})^{0.4}$$

In Walston & Felix's model [WALS77], effort (E) in man months is also a function of program size (L) in thousands of lines of code: $E = 5.2(L)^{0.91}$.

These empirical equations are derived from the analysis of several development projects. Many variations of these equations are proposed for different size projects. Size, therefore, is an important factor in estimating programming effort at the macro level. Programming effort is also related to program understanding and to program difficulty. At the macro level, project difficulty and size are usually correlated.

At the micro level, we have Halstead's software metrics [HALS77, HALS79]. He proposes the idea that there are some basic relationships between the number of unique operators and operands we use in solving a problem and the eventual effort and time required for development. Here, size is measured in terms of operands and operators. The model transcends methodology and environmental factors. There have been several studies that support these metrics as reasonable approximations of what they purport to measure. Most of these experimental tests deal with programs

or algorithms of module size rather than with entire systems. Halstead's metrics have been used mainly for comparison and evaluation of different implementations of the same algorithm.

In the language of software science, the measurable properties of programs are:

n_1 = Number of distinct *operators*,

n_2 = Number of distinct *operands*,

N_1 = Total occurrences of the *operators*,

N_2 = Total occurrences of the *operands*

The size of the vocabulary is defined to be $n = n_1 + n_2$ and the length of the program to be $N = N_1+N_2$. A suitable metric for the size of any implementation of an algorithm is called the program volume $V = N \log_2 n$. Equations for programming effort and programming time are derived as functions of program volume V. Assuming that the implementation of an algorithm consists of N selections from a vocabulary of n elements and that the selection is non–random, the effort required to generate a program is V mental comparisons. Each mental comparison requires a number of elementary mental discriminations where this number is a measure of the difficulty D of the task. D is measured as the ratio of the given implementation volume V to what would be considered the volume of the abstract representation of that implementation in a given language. It is called "potential volume" (V^*). Thus, the total number of elementary mental discriminations E required to generate a given program should be $E = V^*D = V^2/V^*$. This says that the mental effort required to implement any algorithm with a given potential volume should vary with the square of its volume in any language.

Implementation time T is derived from the difficulty equation by considering, after [STRO66], that humans perform between five and twenty mental discriminations per second depending on the individual. Halstead ran some experiments to

determine the number of mental discriminations per second S that programmers can make and concluded that $S = 18$ is a reasonable number. Thus $T = E/S = E/18$.

E has been used to measure program understanding effort. In [CURT79b], it was concluded that Halstead's E is related to the difficulty programmers experience in locating errors in code. It was suggested that "E may be used in providing feedback to programmers about the complexity of the code they have developed." Gordon [GORD79] has conducted a very detailed investigation to validate the use of E as a measure of clarity. He concludes that programming effort properly reflects "comprehension effort", the effort required to understand a given program.

Use of E as a reusability effort metric is very attractive. Computation of E, however, is difficult and language dependent; thus it would violate our proposed criteria of metric simplicity. Feuer and Foulkes [FEUE79] reported a strong correlation (.94) between E and statement count in a sample of 197 PL/I programs with an average size of 54 statements. Sunohara, et al. [SUNO81] have shown a high correlation between the number of source lines and other, more complex metrics.

There is strong evidence that program size is a good estimator for E and, given the relationship between program complexity and programming understanding, program size may also be a valid estimator for reusability effort. To meet the objective of computational simplicity, program size can be measured as the *number of executable statements in a program.* This includes declarative statements.

In a broad study by Evangelist [EVAN83] that relates complexity metrics to program structuring rules, Evangelist concludes that

> . . . in terms of the categories represented in this analysis, none of the metrics performs better than the number of program statements metric.

Program Structure

The structure of a program is often a good indicator of whether a program is well–designed, understandable, and easy to modify. Structure measures are often proposed as measures of the complexity of the product. At the macro level, structure of a system or of part of it, is the arrangement of its modules. A measure of structure depends on the level of modularization and on the interface complexity among modules.

Program or system understanding is related to appropriate levels of modularization. Woodfield, Dunsmore, and Shen [WOOD81] show evidence that programs are difficult to understand when they are highly monolithic and when they are highly modularized in a structure that is not well–defined. They found the best program understanding results in programs observing "useful" (i.e., reasonable) modularization under one logical structure.

One metric for a useful modularization, which is equivalent to measuring structure complexity at the macro level is Gilb's modularity and relative complexity [GILB77]. Modularity M is just a count on the number of modules, and relative complexity R_c is

$$R_c = \frac{\text{number of module linkages}}{\text{number of modules}}$$

where module linkages are the number of resources (e.g., arguments, data, and functions) that flow between the modules of a program. To measure useful modularization, this dissertation proposes a modularity index

$$M_I = \frac{M}{S}$$

where S is program size. M_I helps to differentiate reasonably modularized programs from those that are not. Examples of the latter are small programs with many modules or large programs with few modules.

At the micro level, the simplest control structure metric is the number of decisions as measured by the number of constructs that represent branches in the flow of control, such as *if then else* or *while do* statements [BASI80]. There is a basic belief that the more control–flow branching there is in a program, the more complex it is. A variation of this measure is the relative percentage of control–flow branching, i.e., the number of decisions divided by the number of executable statements. Early studies by Aron [ARON69] showed that varying levels of this type of complexity could account for a nine to one difference in productivity.

A more refined measure of control complexity is cyclomatic complexity as proposed by McCabe [MCCA76]. The cyclomatic complexity of a graph $\mathcal{V}(G)$ is defined as the number of edges minus the number of nodes plus twice the number of connected components

$$\mathcal{V}(G) = \sharp \text{ edges } - \sharp \text{ nodes } + 2(\sharp \text{ connected components})$$

and is equal to the minimum number of basic paths from which all other paths may be constructed. Given a program in which all statements are on a path from the entry node to an exit node, the cyclomatic complexity can be defined as the number of predicates plus the number of segments. A predicate is defined as a simple Boolean expression governing the flow of control, and a segment is defined as an individual routine, procedure, or function.

The measure originated as a count of the minimum number of program paths to be tested. This is one quantitative measure of a program's complexity. The measure is usually applied at the module level, and McCabe proposed a cyclomatic complexity of 10 as an upper bound for the safe range of module complexity.

Other measures of control complexity involve the weighting of various types of control structures as to whether they are simple or complex, where 'simple' means easy to read and easy to understand based on the graph structure. For example,

single–entry, single–exit program graphs containing a single predicate node are easier to understand and to abstract from than more complicated graph structures. Thus, one approach would be to weight various graph structures based upon this complexity. This type of measure requires a more detailed analysis of the program structure than does the cyclomatic complexity measure, but it tends to be a deeper measure of control flow and can include other complexity factors, such as nesting level. One such measure is "essential complexity", which assigns a complexity of one to every program using only structured programming control structures.

For single module programs, cyclomatic complexity measure may be computed using

$$\mathcal{V}(G) = \Pi + 1$$

where Π is the number of logical conditions in the program. This metric is computed simply by counting the number of conditional statements. The assumption that library components are single–entry, single–exit modules may be considered as true in most cases.

High correlation of this simpler version of the metric with program structure is reported in [EVAN83, CURT79b, GORD79]. In all three experiments, McCabe's $\mathcal{V}(G)$ is only one of the indicators of program structure. Other factors such as programming style, the structuring rules used, and documentation must be considered for a better indication of program structure. The cyclomatic number, however, is a good indicator of program complexity; "an indicator of the difficulty programmers experience in locating errors in code" [CURT79b]. Statistically, the cyclomatic number correlates very highly with program size (.83 in [SUNO81], .97 in [BASI84], and .95 in [FEUE79]).

It was found by the experiments conducted in this thesis that, although complexity and size are correlated, in 90% of the cases any two programs of approximately the same size had different cyclomatic numbers, and any two programs with

about the same cyclomatic number had different sizes. Therefore, program size and cyclomatic number perform roles both in estimating reuse effort and in differentiating attributes for similar components. Program size and program complexity are therefore two potential reuse metrics that complement each other.

Documentation

Documentation plays an important role in program understanding as discussed in [BROO78]. Documentation has two main objectives:

1. To bridge knowledge domains.
2. To provide information about a domain.

There are two kinds of documentation used to describe programs: *internal* and *external.* Internal documentation is internal to the program text. There are eight types of internal documentation [BROO78].

1. Prologue comments, including data and variable dictionaries.
2. Variable, structure, procedure and label names.
3. Declarations of data divisions.
4. Interline comments.
5. Indentation.
6. Subroutine or module structure.
7. I/O formats, headers, and device or channel assignments.
8. Action of statements, including organization.

External documentation is basically development information including a history of refinements. It may be presented at the following levels:

S A LEVEL — Specification document, requirement analysis document (e.g., SADT and DeMarco).

DESIGN LEVEL	Structured design charts, PDL, flowcharts, MIL descriptions.
CODE LEVEL	Cross reference listings (modules and/or variables), available references to algorithms used, samples and explanation of code execution and/or I/O data, error dictionary, data dictionary.
OPERATIONS	User manual, performance data (speed, memory requirements, accuracy, reliability), I/O data restrictions (type, size, number), interface requirements (OS, hardware), inter–version incompatibilities.

Internal Documentation

Some of the eight internal types of documentation proposed are not relevant to program understanding. Weissman [WEIS74], for example, reports that interline comments in programs did not significantly aid program understanding and significantly degraded performance in the hand–simulated tasks. There is evidence [SOLO83, JEFF81, CURT83] that program understanding, in particular by experienced programmers, results from recognizing familiar structures in programs in the form of "schemes". If schemes are contaminated with interline comments, they become more difficult to recognize. Weissman also reports an improved understanding of programs due to use of structured control constructs.

Indentation contributes partially to program understanding as reported in [MIAR83, WEIS74]. Proper labeling of procedures and proper variable naming contributes to program understanding, but source code paragraphing does not [LOVE77].

There is no specific evidence on how I/O formats, source statements, and data dictionaries contribute to program understanding other than general recommendations on accurate and useful comments.

External Documentation

Documentation at the design and requirements levels contribute substantially to program understanding. Requirements documents usually explains *what* the program is intended to do and its scope of operation. A design document usually explains *how* a program is implemented and presents a structure chart. Design and requirements documentation usually direct programmers to the appropriate places in code for identification of schemes.

Selby [SELB85] conducted some empirical studies to demonstrate that code reading is a significantly more powerful technique for software defect detection than functional or structural testing. He reports that through code reading, programmers were able to understand the program faster and better. For effective code reading, programmers relied heavily on design information to increase understanding speed.

Another problem with documentation as related to program understanding is that the reuser is left to decide which documentation to examine first, that is, the reuser must determine which level of documentation is more relevant for reusability purposes. If there are no clues where to look first, he may be overwhelmed by the amount of information provided in the documentation thus increasing the *cognitive strain* problem discussed before. With more decisions to make and more information to examine, reuse becomes less attractive.

An exhaustively documented program, if not examined at the right level or if examined top–down, may cause the reuser to look at unnecessary information. In [HOGA80] it is shown that the order in which information is presented can produce "primacy" or "recency" effects. That is, when a number of items of information are presented, sometimes the earlier documents dominate the individual's final opinion (primacy) and sometimes the latter (recency). In addition to primacy or recency effects, having too much information can reduce the consistency of a

person's judgment. Care must be observed in providing the right amount and type of documentation. Irrelevant documentation should not be included if reusability is in mind.

Program documentation, both internal and external, is an important factor in understanding code. The problem is choosing the metric for measuring documentation. Several software catalogs like [APPL83] have used a subjective rating on overall documentation for their listed programs effectively. In this thesis, a similar approach has been taken. Code documentation is given an overall score between 0 (lowest) and 10 (highest) based on the following general criteria.

For a score of 5 (i.e., average), a program must meet the following minimum requirements:

- Availability of design documentation.
- Availability of requirements documentation.
- Use of structured control constructs.
- Prologue comments on module functionality.
- Indentation.
- Proper labeling and variable naming.

Any deviation from the minimum shifts the score up or down accordingly. A program, for example, satisfying minimum requirements and with excellent documentation quality would bring the score up to 7 or 8. A program may miss some of the minimum requirements but make it up by showing excellence on the rest.

This approach is somewhat subjective but the simplicity of the metric is very important. Attempting to define an objective measure of documentation based on quantifying the several attributes possible to consider (e.g., writing style, line spacing, content, clarity and succinctness) would be impractical for the purposes of reusability. The proposed metric for documentation is, therefore, a quality scoring between 0 and 10.

Programming Language

The programming language used is another important factor in estimating code reuse effort. A library of software components as proposed in this thesis is expected to have components written in different programming languages. There is a need to define a metric to estimate the conversion effort from one language to another based on language attributes. This metric may also help to differentiate among similar components. Two components may have, for example, the same complexity and size but be written in different programming languages. A metric that estimates the conversion effort would show the difference between the two components.

The programming language used is considered an inherent attribute of software components rather than a major facet in our classification scheme. It can be used to estimate the reusability effort. The programming language used does not determine the functionality or environment. Although some clues about the environment where a software component is used could be derived from the programming language, the spread of computer applications has diffused the intended objective of domain specific programming languages. No longer are all business applications written in Cobol. Fortran, Basic, and Pascal are widely used in business applications as well.

The objective is to find the set of programming language features that are considered when translating from one programming language to another. Programming language surveys like [SAMM69, HORO83c] and language comparison studies like [SHAW81, HORO83b, BOOM80] show that the most relevant attributes for programming languages are the ones that reflect their most fundamental differences like *imperative* vs. *applicative*. Ability to recognize such attributes is very important for program conversion from one language to another.

The universe of existing programming languages can be divided, as proposed in [HORO83b], in two major classes: *imperative* and *applicative.* Our immediate concern for a library of reusable components is the use of existing software. Most existing software is written in imperative programming languages, therefore, we concentrate on language features characteristic of imperative languages.

A partial list of programming languages features is:

- Structure Support (type of flow structuring constructs used)
- Data Abstraction Support
- Typing
- I/O Power and Simplicity
- Recursiveness
- Compiled vs. Interpreted
- Separate Compilation
- Types of Parameter Passing
- Number of Reserved Words
- Comment Support

Based on a selected subset of programming language features, is it possible to estimate how difficult it is to translate from one programming language to another? To answer this question, analysis of the translation process is necessary. During reusability, a reuser will attempt to convert a program written in source language $\mathcal{A}$ to target language $\mathcal{B}$. The natural approach is to minimize the conversion effort by mapping constructs in language $\mathcal{A}$ into constructs in language $\mathcal{B}$ on a line by line (or 'chunk' by 'chunk') basis keeping as much as possible of the program structure. This implies that some features of the target language will not be optimally used and that several constructs that are characteristic of the source language will be 'literally'

translated into the target language. Translating from a source language that closely resembles features of the target language is more effective than translating from a language with very different features. What the reuser is actually doing is simulating the features of the source language with the target language. To convert an existing Fortran (source) program to Pascal (target) for example, the reusers would look for Fortran–like constructs available in Pascal before attempting any abstraction–specialization cycle.

With these considerations, a more natural 'closeness' measurement results when a given source language's relevant attributes are selected and ranked and the target languages are measured against these attributes. For example, Cobol's main features are:

1. I/O Interface
2. Data Structure Specification
3. Strong Typing
4. English–like Syntax

Distances from Cobol are defined along each feature for each language. For feature 1, Fortran is very close, Basic and Pascal are somehow farther and Algol may be the farthest; for feature 2, Pascal and Ada are closer than Fortran; for feature 3, Pascal is much closer than Algol; and for feature 4, Ada is much closer than Basic. The total distance between Fortran (the target) and Cobol (the source,) for example, would be the sum of all the distances for all the features.

A simplified way to implement this approach is to define a *closeness* matrix C of size $I \times J$. I is the size of the set $\{F\}$ of programming language features considered, and J is the size of the set of languages $\{L\}$ used in the library components. Each matrix element r_{ij} is a rating of language l_i along feature f_j in a predefined scale as shown below.

$$
\begin{array}{lccccccccccc}
\{F\} & & & & & & \{L\} & & & & & \\
 & l_1 & l_2 & \ldots & l_s & \ldots & l_j & \ldots & l_t & \ldots & l_J \\
 & & & & & & & & & & \\
f_1 & r_{11} & r_{12} & & r_{1s} & & r_{1j} & & r_{1t} & & r_{1J} \\
f_2 & r_{21} & r_{22} & & r_{2s} & & r_{2j} & & r_{2t} & & r_{2J} \\
\cdot & \cdot & \cdot & & \cdot & & \cdot & & \cdot & & \cdot \\
f_i & r_{i1} & \cdot & & \cdot & & r_{ij} & & \cdot & & r_{iJ} \\
\cdot & \cdot & \cdot & & \cdot & & \cdot & & \cdot & & \cdot \\
f_I & \cdot & \cdot & & \cdot & & \cdot & & \cdot & & r_{IJ}
\end{array}
$$

The rating is used to determine where a particular language is in relation to the other languages along each feature. (For this study, a scale from 0 to 100 relative points was selected.)

The closeness C_{ts} of a target language l_t to a source language l_s is computed by

$$C_{ts} = \left\{ \sum_{i=1}^{I} (r_{it} - r_{is}) \quad \forall r_{is} \leq r_{it} \right\} + \left\{ \sum_{i=1}^{I} w_i (r_{it} - r_{is}) \quad \forall r_{is} > r_{it} \right\}$$

where w_i is the *retroactive conversion factor* to compensate for conversion from a 'less powerful' language to a 'more powerful' language.

Analyzing the conversion mechanism in more detail reveals that a *more* powerful programming language may simulate most of the features of a *less* powerful language but not vice versa. For example, if the target language is Pascal and an available module is written in Basic, then the new Pascal module can be constructed in a line–by–line translation process because most Basic statements can be written directly to Pascal. The specialized I/O statements that are usually more powerful in Basic can not be translated directly into Pascal. Translating from Pascal to Basic would not be as direct. Usually Basic does not have powerful enough control and structure statements nor the data typing capability needed to perform a line–by–line translation. Most Pascal statements would be simulated by groups of Basic statements, thus making a direct translation from Pascal to Basic more difficult. We can extend this argument to translation between very distant languages like

FEATURES	LANGUAGES						
	FortIV	Basic	Fort77	Cobol	Pascal	C	Smallgol
Structure Support	35	20	80	75	95	93	90
I/O Power	85	50	90	92	35	92	40
I/O Simplicity	30	90	40	70	75	80	95
Comment Support	50	55	55	80	78	60	50
Typing	78	60	75	80	95	90	90

Table 6.1: A Programming Language Closeness Matrix

Pascal and Assembly and to translation between very close ones like Fortran IV and Fortran 77.

Two major advantages of this feature rating approach are *expandability* and *flexibility*. New features and more languages can be added with relative ease. Rating values can be adjusted for fine tuning to the environment where the library system is to be used. Table 6.1 shows the closeness matrix derived by the author and used in the prototype library system proposed in this thesis. For this matrix, a global retroactive conversion factor of -1.5 was used for all features. If, for example, Pascal were the target language, then for structure support:

$$\text{Pascal} - \text{C} = 95 - 93 = 2, \quad \text{Pascal} - \text{Basic} = 95 - 20 = 75, \quad \text{etc.}$$

for I/O power:

$$\text{Pascal} - \text{C} = 35 - 92 = (-57) \times (-1.5) = 85,$$

$$\text{Pascal} - \text{Basic} = 35 - 50 = (-15) \times (-1.5) = 22.5, \quad \text{etc.}$$

The final rating for this example is shown in figure 6.1. It shows the relative closeness of each language to Pascal. This distance may be interpreted as a relative estimator of the translation effort. Translating from C to Pascal implies, as shown, relatively less effort than translating from Fortran IV to Pascal. The major contributing factor to the relatively larger separation between Fortran IV and Pascal is the difficulty in changing the powerful and highly specific Fortran I/O statements.

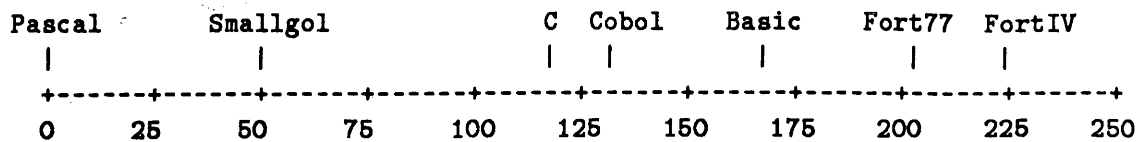


Figure 6.1: Relative Closeness to Target Language Pascal

This approach for a language closeness metric is relatively simple and objective. It is based on the premise that programming languages have well–defined features which can be used to differentiate them.

User Experience

User experience is a very important factor in program understanding. Jeffries, et al. [JEFF81] and Shneiderman [SHNE77] confirm that more experienced programmers understand programs faster than novices. Programmer experience however is usually confined to certain languages and domains of application. It is reported in [SELB85] that experienced programmers had similar difficulty as novices in understanding programs written in an unfamiliar language.

Soloway [SOLO83b] reports that programming language characteristics definitely affect programmer "cognitive" strategies in programming and in code understanding. It was observed that experienced programmers relied more on language characteristics than novices for program understanding.

Mainly at the system structure level, experience in an application domain is also a contributing factor in program understanding. At the design level, analysts with more experience in the domain of the application performed better in design understanding than analysts with similar levels of experience in programming and design but in a different domain of application [JEFF81].

How can user experience be measured? An experienced programmer will have

an overall advantage in reusing a software component over a novice programmer. A precise and objective metric is difficult to define. For the purpose of this work, however, the experience level for a programmer was divided into two areas: *programming language experience* and *domain of application experience.* A level of experience was assigned for each programming language with which the programmer was familiar and a level of experience for each familiar area of application. Levels of experience were defined and rated as:

10	8	6	4	2	0
Expert	Advanced	Intermediate	Knowledgeable	Novice	Ignorant

Each reuser of the library system is assigned a *reuser profile* that defines his or her level of expertise in a set of programming languages and domains of application. The reuser profile influences the values of all the other program attributes. A reuser with a higher reuse profile would require less effort to reuse a given program than a reuser with low profile. Attribute values for a given program are modified by the reuser profile. The next sections discuss how the reuser profile interacts with the other program attributes to estimate reusability effort.

Conclusion

In this section we have proposed five program attributes as the most relevant indicators of reuse effort and a metric for each:

PROGRAM SIZE	Lines of code.
PROGRAM STRUCTURE	Number of modules, number of linkages, and cyclomatic complexity.
PROGRAM DOCUMENTATION	Subjective overall rating (1 to 10).
PROGRAMMING LANGUAGE	Relative language closeness.
REUSER EXPERIENCE	Six levels in two areas: programming language and domain of application.

For each attribute, substantial evidence of its validity as a reuse effort estimator was presented and its metric selection was discussed.

Multi–dimensional Evaluation

Multi–dimensional evaluation is a method widely used in decision theory. It is used for evaluation and selection of feasible alternatives to a given decision problem. The core of the method is the utility function which assigns utility values to the factors involved in the decision. One typical technique [KAHN82] consists in assigning "worth" values to the factors considered when making a decision. Some attributes for a class of entities may be more relevant than others. Thus, the worth value of an attribute (usually represented as a numerical weight) is its degree of relevance for the particular selection problem considered.

Athey [ATHE82] has developed a systematic approach for the selection problem. Evaluation attributes are selected first. When selecting a car, price, safety, economy, size, and power may be some of the most relevant attributes. When selecting a university, tuition costs, prestige, and location may be the relevant attributes. Attributes are treated as dimensions for evaluation, and each one is rated according to its "relative worth" for the problem solution. A preference chart is constructed to compare each dimension with each of the other dimensions, one at a time. The outcome is a ranked list of dimensions based on their relevance to the particular selection problem. Tuition costs may be more important than prestige for a student with a limited budget, while prestige may rank highest for a student with no financial problems.

After all the dimensions are ranked by relevance to the problem, a "utility function" is defined for each dimension. It is necessary to specify the relative desirability of different levels of performance in each of the selected dimensions.

CRITERIA	NORMALIZED UTILITY RATING										
	0	1	2	3	4	5	6	7	8	9	10
Tuition Costs ($/year)	10k	7k	5k	4k	3k	2k	1.5k	1k	500	300	100
Location	East Cst.	Mid West	South East	North West	West Cst.	Nrtrn Calif	Soutrn Calif	San Diego	LA	OC	Irvine

Figure 6.2: Two Hypothetical Utility Functions

Explicit measures of performance are defined. Each dimension is assigned a normalized score, usually from 0 to 10, according to its "desired" performance. For example, tuition costs in the university selection case, could be rated from low to medium to high. A low score would be given to high tuition costs while a high score to low tuition costs. The extreme or boundary values for each attribute are given from the constraints or initial conditions of the problem. The student selecting a university may have a restricted budget that determines the maximum that can be spent on education. Two hypothetical system utility functions for the university selection problem are shown in figure 6.2.

The final step is to evaluate each candidate based on the criteria established by rating them along each dimension. The score is multiplied by the relative worth of the dimension and the weights are added. The best one will be the alternative with the highest score.

We use this methodology to select the best from a sample of functionally similar components. The criteria are the four attributes: program size, program structure, program documentation, and programming language. Selection of these four attributes, as mentioned earlier, was based on their relevance to code reuse and their properties as code reuse effort estimators. All four are considered equally relevant for the purpose of this study. Although, under certain circumstances, some

REUSE EFFORT UTILITY FUNCTION

Component Size (lines of code)		5	10	15	20	30	50	80	120	200	300
relative worth	EXPERT	10	10	10	9	8	5	3	2	1	0
	NOVICE	10	9	8	7	5	3	2	1	0	0

Figure 6.3: Two Reuse Effort Utility Functions

differences in their relevance to the reuser could exist, the differences are minor and do not seem to affect the selection process. Therefore, no ranking of dimensions was done.

The next step was to derive utility functions for each attribute. Utility functions are derived each time an evaluation process is conducted. Utility functions are dependent on the user and the situation. The main modifier of the utility function is reuser experience in both the programming language and the domain of application. Intuitively, an expert reuser may invest less effort in understanding a relatively complex program than a novice reuser. For an expert reuser, a particularly high complexity measure may have less weight than for a novice. Two hypothetical reuse effort utility functions to rate program size for different levels of expertise are shown in figure 6.3.

Deriving utility functions each time a sample of functionally equivalent components is retrieved is a time consuming effort that could easily offset the benefit of reusability. An approach to mechanically derive utility functions for given reuser profiles and reuse situations is needed.

The first step towards a mechanized approach is to define the ideal criterion that minimizes reuse effort for each of the four attributes defined so that a general attribute can be defined. The best component for reusability (i.e., the one that

would require the least amount of effort to be reused) would be one with:

Size	=	small
Structure	=	simple
Prog. Lang.	=	same (i.e., closeness = 0)
Documentation	=	excellent (i.e. rating = 10)

Each reuser has particular criteria, based mainly on experience, for determining what is meant by small, simple, and excellently documented. In some cases, what is considered the same programming language differs from having the same ancestor (e.g., members of the Algol family), to being dialects of the same language (e.g., Fortran IV and Fortran 77) or versions of the same dialect (i.e. Basic Plus on PDP11/70 and Basic Plus on Prime 250).

One approach is to define a reuser experience profile and determine how these criteria change with level of experience. Since these criteria do not have precise values, and are, therefore, fuzzy concepts, Fuzzy Set Theory is used to evaluate them.

Conclusion

A systematic evaluation methodology for the selection of equivalent alternatives has been presented. This methodology is applied to the selection of functionally feasible components by identifying four evaluation attributes and defining a comparison criterion for each. Utility functions may be derived based on these criteria.

Dimensional Normalization with Fuzzy Sets

In this section, the concepts of fuzzy logic and fuzzy sets and functions are introduced. Fuzzy functions and function modifiers are used in this work to determine the degree of membership a given component has in each of the four reuse

effort attributes. Reuser experience functions are used as modifiers for the attribute functions.

Fuzzy Logic Concepts

When making decisions, computers usually compare attributes that have precise values: Is the height over 5.7'? Is the temperature in excess of 72.3° F? Humans do not generally reason in such a precise manner. Rather, most humans reason in categories with fuzzy boundaries: John is short, today's temperature is pleasant, or a short program is easier to understand than a large program. Fuzzy logic, thus, is:

> . . . a kind of logic using graded or qualified statements rather than ones that are strictly true or false. [ZADE84]

Fuzzy sets are a concept that can bring the reasoning used by computers closer to that used by people. Whereas a conventional, or "crisp", set has sharp boundaries, the transition between membership and nonmembership in a fuzzy set is gradual rather than sharp. The degree of membership is specified by a number between 1 (full member) and 0 (full nonmember).

A fuzzy set is a class with fuzzy boundaries like the classes of expensive cars, small numbers, high mountains, or blond men or women. Such a class may be characterized by associating a grade of membership in the class with every object that could be in the class. For our evaluation problem, the candidate components will be associated to the class 'small' when looking at size, to the class 'simple' when looking at structure and so on. A 20–lines–long component, for example, may have a degree of membership in the 'small components' class of perhaps .7 while a 40–lines–long component may belong only .4 to the set.

A fuzzy set is an association between numbers, a correspondence that assigns to a given value one and only one number in the unit interval. Each element z_i of

a fuzzy set Z is defined as an ordered pair

$$z_i = \langle v_i,\ m_i \rangle$$

where v_i is a value in the range of values that Z represents, and m_i is a value in the unit interval $[0, 1]$. If Z denotes 'small program' in number of lines in source code then

$$\text{smallprogram} \ : \ \text{lines} \rightarrow [0, 1]$$

A fuzzy set is actually a function that maps a set of values in a given domain to the interval $[0, 1]$. So, if

$$Z = \{\langle v_1,\ m_1 \rangle,\ \ \langle v_2,\ m_2 \rangle,\ \ \ldots, \langle v_i,\ m_i \rangle, \ldots, \langle v_I,\ m_I \rangle\}$$

then a fuzzy function $f_z(x)$ for the set Z is defined as

$$f_z : V \rightarrow M$$

where $v_i \in V,\ \ m_i \in M,\ \ M = [0, 1],\ \ v_1 \leq x \leq v_I$, and $\ 0 \leq f_z(x) \leq 1$.

Fuzzy functions that denote the concepts of *small* or *large* are considered to be S–shaped. There is experimental evidence [NORW82] that the fuzziness of a concept varies along an S–shaped curve. S–shaped curves for the 'small' concept are computed by the function

$$f_z(x) = \begin{cases} 1 - 2\left(\frac{x}{v_I}\right)^2 & \text{if } \ 0 \leq x \leq v_I/2 \\ 2\left(\frac{x}{v_I}\right)^2 & \text{if } \ v_I/2 < x \leq v_I \\ 0 & \text{otherwise} \end{cases}$$

Basically, the degree of membership is subjective in nature; it is a matter of definition rather than measurement. In this work, as mentioned before, the degree of membership depends on how reusers relate a particular metric value to a given class. This association depends on reuser experience.

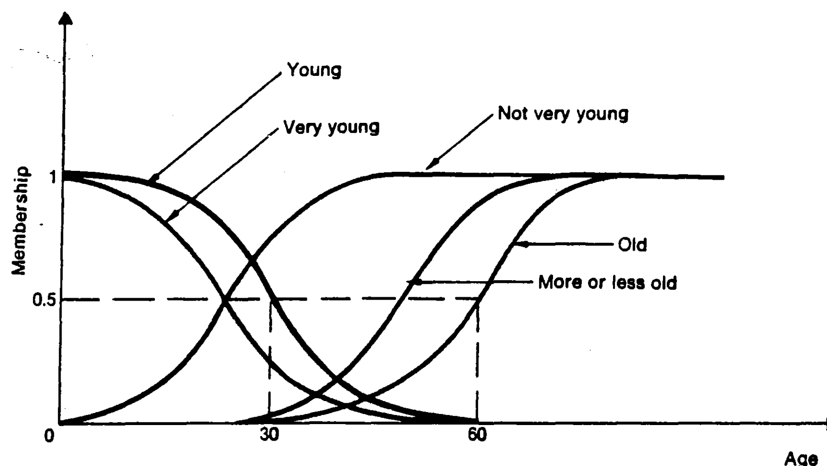


Figure 6.4: Fuzzy Functions for Young and Old with Some Modifiers [ZADE84]

Fuzzy Modifiers

Fuzzy modifiers are

> . . . operations that change the membership function of a fuzzy set by spreading out the transition between full membership and nonmembership, by sharpening that transition, or by moving the position of the transition region. [ZADE84].

Sharpening the transition is usually accomplished by squaring the membership function and spreading out the transition is obtained by computing the square root of the membership function. These operations are called ***concentration*** and ***dilation*** respectively [RAGA77].

Figure 6.4 shows the shapes of some fuzzy functions for the concepts "young" and "old", and their modifiers, "very young", "more or less old", and "not very young" for the "age" domain with values in number of years. The "very young" transition curve is a concentration of the "young" membership function and the "more–or–less old" curve is the dilation of the "old" function.

Reuser experience may be seen as a fuzzy modifier for the fuzzy functions of program attributes. Reuser experience can concentrate, dilate, or move the transition region of a program attribute's fuzzy function. Figure 6.5 illustrates

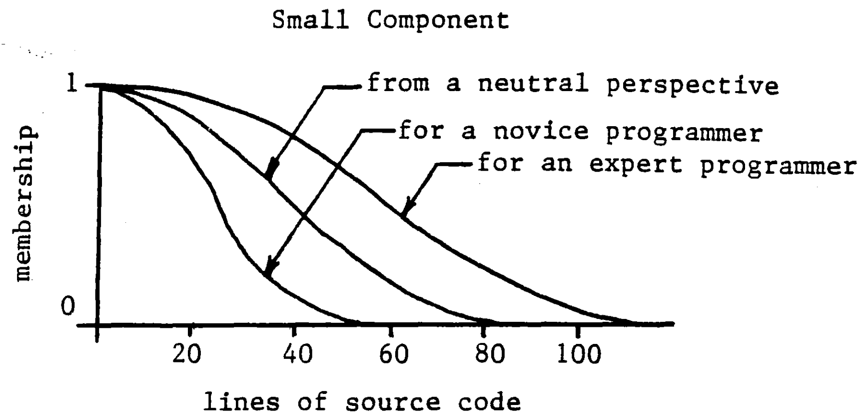


Figure 6.5: Reuser Experience as Modifier for Small Component

a change in the transition region of the fuzzy function for the *small component* concept.

Program structure and documentation quality are modified in the same fashion as program size. In the case of documentation quality, what is modified is its overall relevance to program understanding. For an experienced programmer a high documentation quality–value may contribute less to reuse effort than the same high rating has for a novice programmer.

Programming language used is another function modifier for size and structure. The concept of a small component may have different membership values for different programming languages used. For example, a large component in APL may have less lines of code than a small component in assembly language. Figure 6.6 illustrates this idea for the small component concept for different languages as function of lines of code.

Reuser experience was implemented in this work as a modifier for size, structure, and documentation functions by moving the positions of the transition regions proportional to reuser experience. For the evaluation subsystem, the range spread was assumed to vary from one half to twice the normal range as illustrated in figure 6.7a.

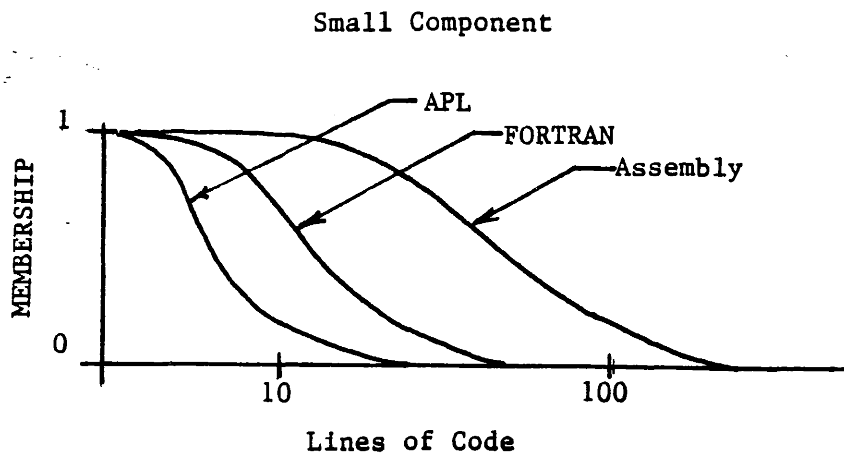


Figure 6.6: Programming Language as Modifier for Small Component

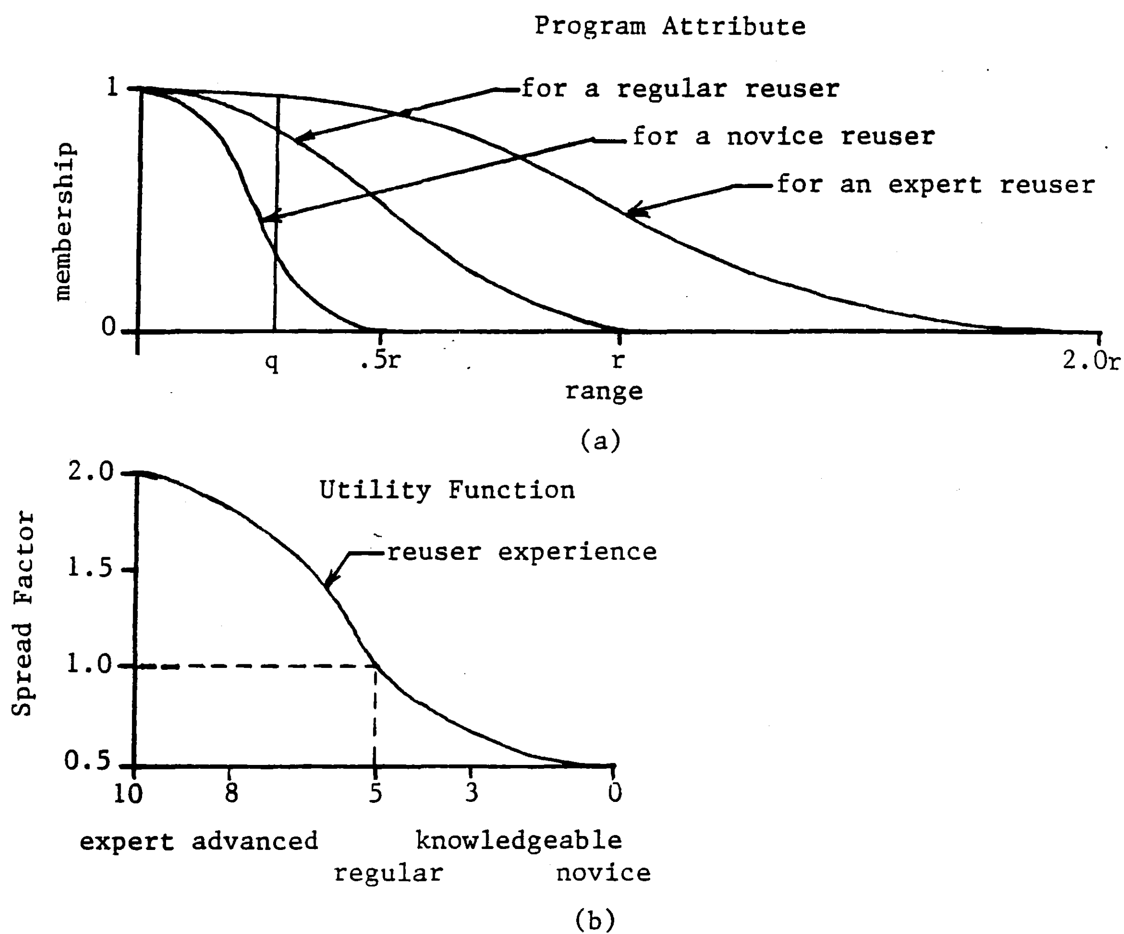


Figure 6.7: User Experience as Modifier for Attribute Functions

Figure 6.7b is a fuzzy function that maps experience level into a spread factor from .5 for beginners to 2.0 for experts. If we let R be range spread and x be reuser experience, then the curve in figure 6.7b is computed by:

$$R = \begin{cases} 2\left[1 - 2\left(\frac{10-x}{10}\right)^2\right] & \text{if } 5 \le x \le 10 \\ 2\left[\frac{1}{4} + \left(\frac{x}{10}\right)^2\right] & \text{if } 0 \le x < 5 \\ 0 & \text{otherwise} \end{cases}$$

A direct mapping from reuser experience to degree of membership for a given reuse metric may also be computed. Figure 6.8 shows this mapping for a family of normalized curves (range r has been normalized to 1). This figure shows the curves for points where a fixed value of r (illustrated by point q in figure 6.7a) crosses all membership curves generated from the spreading of a membership function as computed from the reuser experience function of figure 6.7b.

If point q of figure 6.7a is assumed to be .3 then membership will change from about 0.35 to about 0.9 following the pattern of the curve marked .3 in figure 6.8 Curve .3 results from plotting all cross over points (at q in figure 6.7a) for all membership curves generated by spreading the range of curves for all valid values of reuser experience (x).

This figure shows how to compute the change in relevance of the reuse metrics for different levels of reuser experience. The case illustrated in figure 6.8 has two components with size .6 and .5 of the established range. For an advanced reuser (e.g., value 8) the differential in membership for these two components is only 6% while for a knowledgeable user (e.g., value 4) this differential increases to 15%. For less experienced reusers, therefore, the evaluation subsystem is more sensitive to small differences in reuse metrics than for more experienced reusers.

Using a generalized function as a modifier does not apply in the case of programming languages. To do so, a standard language would have to be defined

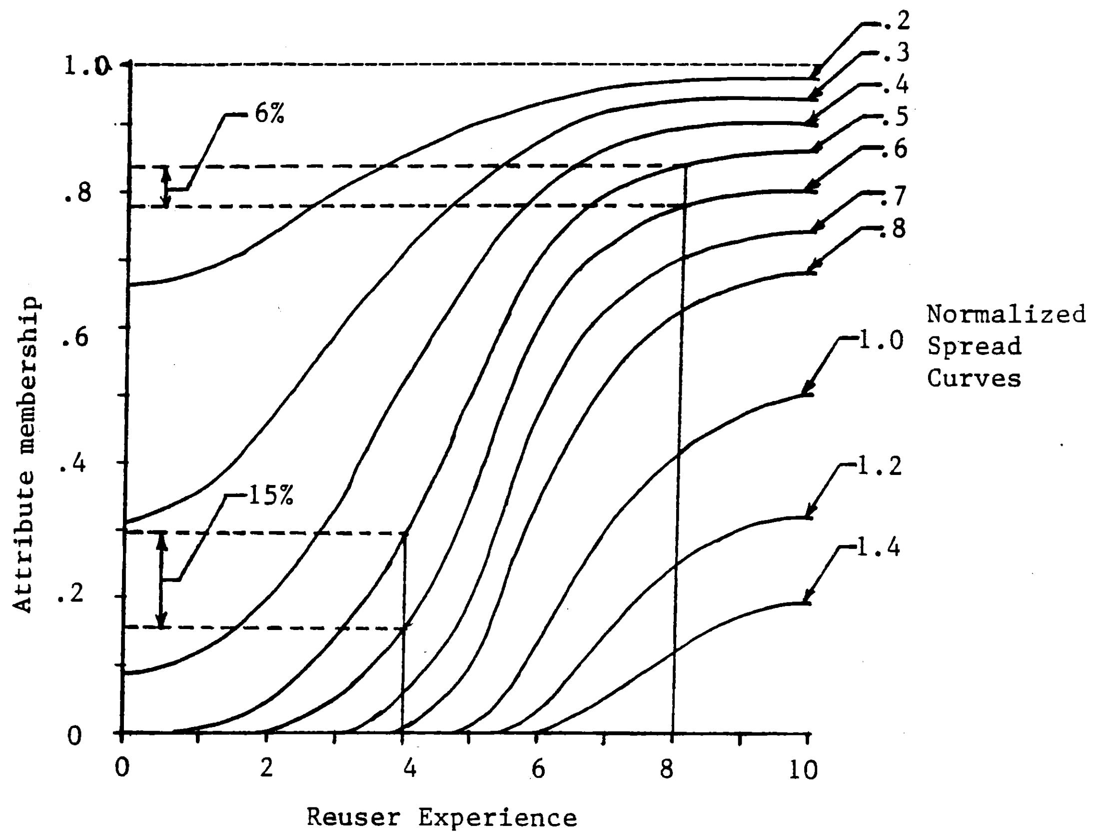


Figure 6.8: Membership as Function of Reuser Experience

so a membership function could be derived and then modified to fit different languages in proportion to their closeness to the standard. There is no such thing as a standard language, so one fuzzy function has to be defined for each programming language used in the library system as shown in figure (6.6). It was assumed that in most programming (software development) environments the set of programming languages used is small. A set of fuzzy functions is derived for each programming language in use. One function for each reuse attribute (e.g., size, structure and documentation).

Conclusion

The contribution of this chapter is the development of a technique to evaluate a sample of functionally and environmentally equivalent components in order to select the best, that is, the component that required the least effort to reuse.

Four intrinsic attributes (program size, program structure, program documentation, and programming language) and one external factor (reuser experience) were identified as the most relevant for code reuse effort estimation. A metric for each was defined. A special case is the metric to for estimating programming language conversion effort. This metric is based on a comparative rating of conversion related features for the imperative languages considered. The metric is a relative closeness measure.

A multi–dimensional evaluation method was proposed. It requires defining utility functions for each attribute considered. Weights are computed for each component based on a normalized scale as a function of each metric. The best selection is the one with the highest score.

Reuser experience is a major factor in shaping utility functions for the four attributes. Fuzzy functions were used as utility functions with reuser experience

as their modifier. Fuzzy logic is ideal for this application where subjective human judgment is involved.

This evaluation technique based on fuzzy functions and modifiers has been implemented and integrated into the library system as the *evaluation subsystem* of figure 5–3, box 5. Specific S–shaped fuzzy functions have been defined for attributes, programming languages, and reusers and tested in a limited environment. Chapter 7 discusses the experimental results of some tests of this subsystem in particular and of the library system in general.

CHAPTER 7

Library System Evaluation

Usefulness of a classification scheme is demonstrated by direct evaluation through implementation and testing. In library science, implementation may take many years, mainly because document collections take time to build up. Tests are conducted by surveying users on their opinion of library usefulness. How easy it is to find their documents? How easy it is to use the catalog? Librarians also participate in evaluating the classification scheme used. A certain library size is desired before conducting an evaluation. Success of classification schemes is due mainly to their capacity for adaptation rather than to the qualities of their original design. [MALT75]

With a computerized library system, evaluation can be conducted continuously from the early stages of implementation. A highly adaptable scheme should prove useful, especially in very dynamic collections. Evaluations at the prototype stage may prove essential for early modifications before committing to full implementation.

This chapter presents three evaluations of the prototype *library system* introduced in the previous chapters: 1) Retrieval effectiveness, 2) Effectiveness of the classification scheme, and 3) Reuse effort estimation.

Retrieval effectiveness is tested by comparing its performance against that of a typical data base retrieval system not organized around a classification scheme. Effectiveness of the classification scheme is tested by asking reusers to classify programs and compare their performance in classification (i.e., assigning the right program to the right class) against the performance of a professional librarian. This test also probes the scheme's ease of use. Reuse effort estimation is tested by asking

reusers to estimate reuse effort ranking a list of functionally equivalent components from least to most reusable. The outcome is compared to that of the evaluation subsystem. Information on their evaluation process is analyzed and fed back to the evaluation subsystem.

The results of these tests should be interpreted within the context of a prototype system indicating what could be expected in a fully operational library system. The main difference between this prototype and an operational system would be the size of the collection. The procedural mechanisms would remain essentially the same. Given these circumstances, the tests described are indicative of expected performance.

Retrieval Effectiveness

There are basically two tests of interest for evaluating the retrieval effectiveness of this library system:

1. Does citation order in a query improve retrieval performance? We compare retrieval performance between queries denoting most relevant generalizations and queries that are not the most relevant generalizations.

2. Is retrieval performance better with queries made of a selected set of conceptually related keys than with queries made of randomly ordered keys? We compare retrieval performance between a group of keys ordered by their conceptual distances and the same group of queries ordered randomly.

Before discussing the experiments and their respective results, a tour of the retrieval process implemented in this library system is presented followed by a description of the criteria used for retrieval evaluation.

Making the Key

The library system, using the classification schedules developed in Chapter 5, guides the user in selecting the appropriate terms for the key. The objective is to make a complete key. There is one schedule for each facet, and each schedule is listed in a thesaurus format.

Making the key is an interactive session started by a request from the reuser. The library system asks the reuser to select a representative term from the thesaurus of each facet until all six terms that form the key have been selected. For example, if what the reuser is looking for is a component to: 'change backspaces to multiple lines', then the library system will prompt:

```
Selecting--->      function:             Enter term  change
substitute    replace/transliterate/convert/change/map/crypt/update
exchange      swap/trade
2 entries match 'change'      SELECT ONE      Enter term  replac
1 match 'replac'    GIVEN NAME: - substitute -    (w)rong (CR)ok
```

The library system asks the reuser to enter a term for the `function` facet. The reuser enters the word *change*. In this case, there are two entries where the word *change* appears in the classification schedule for the function facet: One in a group of terms that denote the concept of change in the context of substitute (i.e., replacement) and the other in the word exchange under the context of swapping. The request 'change backspaces to multiple lines' fits the replacement context better, so the representative term `substitute` is selected by the system upon acceptance by the reuser.

Following the same dialogue format, the reuser is asked to provide and select

a term for each of the six facets.

A typical request, as the one illustrated here, usually does not specify all six facets. The dialogue is designed to make the reuser define a complete search key. When asked for an entry for a facet not originally specified, the reuser must provide an initial entry.[1] In this example we assume the reuser is working on a text–formatting system; thus, the prompt is answered as:

```
Selecting--->    system-type:        Enter term format
1 match 'format'  GIVEN NAME: - text-formatter -   (w)rong (CR)ok
```

Functional–area and setting terms must also be defined. With a complete search key defined, the library system prompts options for key manipulation. The key can be *modified* or *expanded.*

In key–modification mode, the key can be *generalized* or *specialized.* The generalization process consists of an orderly replacement of terms by the any meta-symbol $\{*\}$ to obtain the *more relevant generalizations* of the completely defined key. More relevant generalizations consist on removing facets in the established citation order introduced in Chapter 4. For example, if the complete search key for the request above were defined as:

substitute/backspaces/file/text–formatter/program–development/software–shop

then the most relevant generalizations are:

substitute/backspaces/file/text–formatter/program–development/*
substitute/backspaces/file/text–formatter/*/*
substitute/backspaces/file/*/*/*
substitute/backspaces/*/*/*/*
substitute/*/*/*/*/*

[1]Key modifications are made later in *modification* mode.

The specialization process consists of placing the original terms back into the key. Key generalization and specialization are conducted interactively by the reuser in any order or combination to generate the desired key. With a mechanism to modify keys such as this, the reuser is capable of testing different sample retrievals.

In key–expansion mode, the selected (original or modified) key is used as a reference for creating new keys of closely related terms. The criteria for term selection is their conceptual distance from the reference term. Conceptual distances are calculated from the conceptual matrix presented in Chapter 5. Terms are ordered from closest to farthest in relation to the selected term. Each ranking step is called a conceptual level. Expansion can be requested for any facet where a term is present in the key and the range of the expansion can be increased as desired. Each expansion creates a new key. A five level expansion for substitute/backspaces is:

substitute/quotes
substitute/blanks
substitute/digits
substitute/tabs
substitute/characters

Retrieval is executed by the library system only after the reuser confirms satisfaction with the resulting search key or with the set of expanded keys. In key expansion, the retrieval for each key is conducted in the same order as the expansion of the keys, that is, from conceptually closer to conceptually farther.

Retrieval Evaluation

In retrieval evaluation, two kinds of tests must be distinguished: those concerned with the system's *effectiveness,* and those concerned with the *efficiency* of the operations. The effectiveness of a retrieval system is the ability to furnish information services that the users need. On the other hand, efficiency is a measure

of the cost or the time necessary to perform a given set of tasks. We are interested in effectiveness. In prototype systems, efficiency is usually not an issue until it is transformed into a production system.

Retrieval effectiveness is directly related to the specificity and exhaustivity of the indexing language used. The classification schedules listed in Appendix B have a certain degree of specificity because of the vocabulary control imposed by the thesaurus. Vocabulary, when controlled by a thesaurus, as explained in Chapter 5, results in high specificity.

An exhaustive indexing language contains terms covering all items named in the collection. In our prototype, schedule development by literary warrant, ensures that all components have a descriptor. As the collection grows, however, index exhaustivity could decrease if the schedules are not expanded accordingly.

Retrieval effectiveness performance is usually measured by *recall* and *precision* [SALT83]. Recall is defined as the proportion of relevant material retrieved while precision is the proportion of retrieved material that is relevant as shown in figure 7.1. A high level of indexing exhaustivity tends to ensure high recall by making it possible to retrieve most potentially relevant items. On the other hand, a highly specific indexing language ensures high precision since most retrieved items may be expected to be relevant.

The concept of relevance is directly related to retrieval effectiveness. A carefully selected citation order and a careful design of an indexing language improves recall and precision values in a retrieval system.

Recall and precision measurements are used to compare retrieval performance of different retrieval systems on the same collection. For a given retrieval system, values of precision and recall are taken for different query sizes and their averages computed. The result is an average recall–precision curve for different size queries for each system compared. An ideal retrieval system would yield high precision

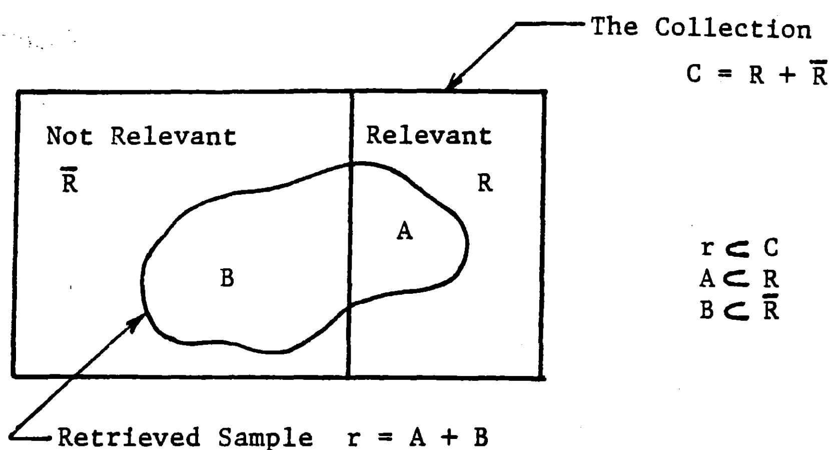


Figure 7.1: Recall and Precision in Retrieval

values for a wide range of recall values. Referring to figure 7.1, the size of B would be close to zero and $A \approx R$. Its average curve would be very close to the upper boundary of figure 7.2.

After introducing how retrieval is conducted and how it is evaluated, two experiments are discussed: the effect of citation order and the effect of conceptual ordering on retrieval performance.

The Effect of Citation Order

The effect of citation order on retrieval effectiveness was determined by comparing the retrieval performance of queries made of the most relevant generalization of a complete query with the performance of queries made of partial generalizations of the same complete query. The most relevant generalization of a query always generalizes the least relevant term in the original query while a partial generalization may generalize any of the more relevant terms in the original query. If we assume a complete query to consist of the descriptor $d_c = \langle v_1,\ v_2,\ v_3\rangle$, then the

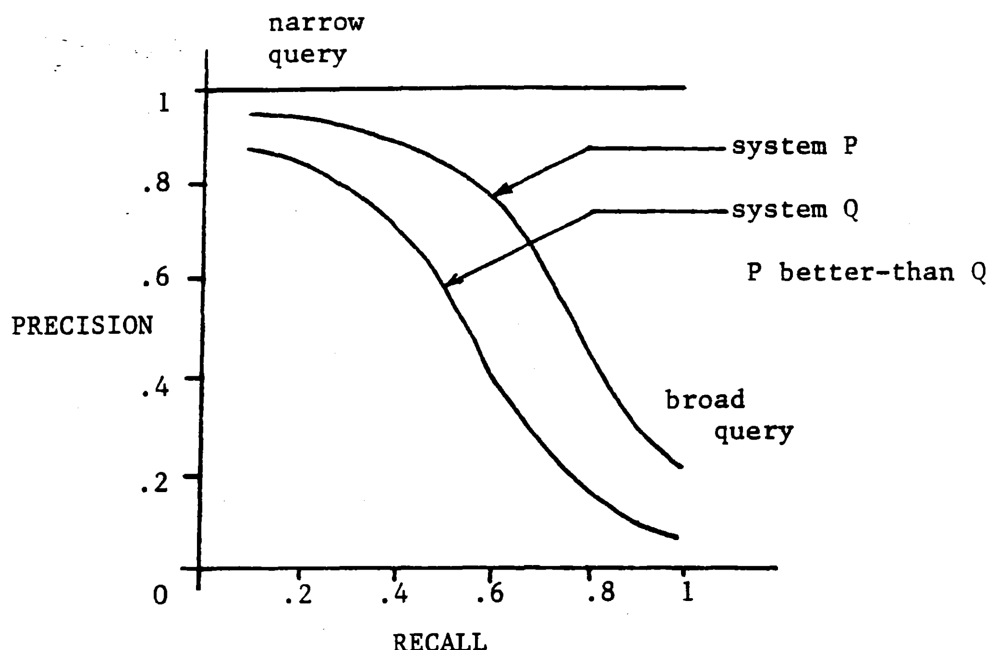


Figure 7.2: Use of Recall and Precision to Compare Retrieval Systems

most relevant generalization is $d_m = \langle v_1,\ v_2,\ *\rangle$ and a partial generalization may be $d_p = \langle v_1,\ *,\ v_3\rangle$.

An experiment was designed to compare the retrieval performance of both types of queries. The experiment was based on the concept that a sample retrieved by a most relevant generalization of a given query is more relevant than a sample retrieved by a partial generalization of the same query. The experiment was conducted as follows: for a given request (e.g., change backspaces to multiple lines in a file), first a complete search key, as illustrated above, was defined (e.g., d_c = ⟨substitute, backspaces, file⟩). Being a prototype library system, the size of the collection is limited. In order to have measurable samples, the size of the query was limited to the first three terms; those describing functionality.

The complete collection C (figure 7.1) was then inspected in order to identify the sets R and $\overline{R}$ for the given query. Identification of these sets was based on the author's interpretation of how close the descriptions in C matched the

original query. With a defined position of the boundary between R and $\overline{R}$ in C, recall and precision were computed for d_m and for a d_p. For the example d_c = ⟨substitute, backspaces, file⟩ the computed values based on the prototype collection of 150 descriptors (i.e., $|C| = 150$) were for d_m: $r = 1$, $A = 1$, $B = 0$, $R = 4$, Recall = 0.25, and Precision = 1.0 and for d_p: $r = 14$, $A = 4$, $B = 10$, $R = 4$, Recall = 1.0, and Precision = 0.28. Below are shown the average values of recall and precision for ten different queries analyzed by the outlined procedure.

	Most Relevant Generalization	Partial Generalization
Recall	0.5	0.8
Precision	1.0	0.4

Recall and precision values for retrieval using the most relevant generalizations are closer to the ideal performance of figure 7.2 than those values of partial generalizations. From these results we observe that citation order is important when designing a classification scheme. Retrieval performance may improve when terms used in queries are selected from a classification structure.

The Effect of Conceptual Ordering

The objective of this experiment was to evaluate the effect of query ordering on retrieval performance when queries are ordered by conceptual distances. For a given query, a set of 'conceptually related' queries is derived by the key expansion process described above, resulting in an ordered list of queries. The retrieved ordered sample from this list of queries is compared with a randomly ordered sample from the same set of queries. Precision and recall values were computed for each case.

Because of limitations imposed by the collection size, as in the previous experiment, query size was reduced to two terms. The experiment was conducted as follows: For each query $d_c = \langle v_1, v_2 \rangle$ tested (e.g., ⟨substitute, backspaces⟩), v_2 was expanded several levels (e.g., ⟨substitute, quotes⟩, ⟨substitute, blanks⟩, ⟨substitute,

digits⟩, etc.) until the retrieved sample was the same as that retrieved with the most relevant generalization $d_m = \langle v_1, *\rangle$. Two samples resulted: a conceptually ordered sample generated by the expansion of d_c and an unordered sample retrieved by d_m. Only the first third of each of the two samples was considered for evaluation to determine the difference between the ordered sample and the unordered one. This was also a criterion imposed by the limited size of the collection.

Retrieval performance was computed for ten different expanded queries. Values of recall and precision were calculated for the conceptually ordered samples and for the unordered samples. Values of R and $\overline{R}$ for each query were determined, as in the previous experiment, by the author's criteria. Below are shown the average values of recall and precision for the ten pairs of queries.

	Conceptually Ordered Sample	Unordered Sample
Recall	1.0	0.5
Precision	0.9	0.4

Samples of conceptually closer queries show outstanding retrieval performance compared to unordered queries. Relating terms by their conceptual distance in a classification scheme has an excellent payoff in retrieval performance.

Experiment Conclusion

These two limited experiments show the impact of a carefully designed classification scheme on retrieval performance. These experiments were designed not to provide precise performance measurements but rather to show the differences in performance between our approach and the approach followed by a typical data base retrieval.

These results should be interpreted within the context of a small prototype collection and used only as general indicators of performance. The significance

of these results is their ability to predict, within certain limitation, the expected retrieval performance in a larger collection.

User Classification

The complexity and extension of current classification schemes creates the need for professional librarians. Librarians are trained to use classification schemes to identify where in the schedule a particular title fits, to resolve ambiguities, and ultimately, decide where to classify an item that belongs to more than one class.

The reuser should be encouraged to contribute to the software library. A component produced from the reuse of an existing component should be added to the collection and classified. If several reusers actively participate in software development through the use of the library, the number of components added to the library may become large enough to require several librarians to maintain the collection. The effort librarians invest in becoming familiar with very specialized components must be added to the cost of maintaining the collection. It is preferable to have reusers classify their own programs so, the resulting classification scheme must be easy to use and unambiguous in its classes and vocabulary.

The classification development techniques used in this thesis ensure, to some extent, ease of use and precision in the resulting scheme.

- Literary warrant ensures coverage because there are terms in the vocabulary to classify every component in the collection.
- The thesaurus is an excellent technique to resolve contextual ambiguities.
- Partition of the scheme into only six facets with strict citation order makes the classification task relatively simple even for non–professionals in the area of library science.

Name	Short Description	Language	Size	Documentation Score	Documentation Size	Reference
fmove	move file name1 to file name2	Pascal	9	9	3 lines	[Kern83]
addfile	add file 'name' to archive	Pascal	38	9	1 page	[Kern83]
replace	replace substring old with new	Fortran	40	3	10 lines	[Fried77]
lookup	search node B–tree, rtrn parent	Pascal	30	4	3 pages	[Lewi83]
split	split node in B–tree	Pascal	19	4	2 pages	[Lewi83]

Table 7.1: Summary Programs for Classification Test

To test how difficult it is for a reuser to classify a new program using our classification scheme, a group of 13 potential reusers (graduate students in computer science) were asked to classify the same set of 5 programs. Each program consisted of a brief functional description, a structural context if it was a subcomponent of a larger program, and source code.

The average size of the components was 27 lines of source code. In order to have a common reference for evaluating the results, the programs selected for the classification exercise were from the same application area. The components considered are mainly the type of programs used in information processing applications as in file maintenance or data base management tasks. Table 7.1 summarizes the experimental sample. Program size is in lines of code, documentation score is measured on a quality scale of 0 (poor) to 10 (excellent) as explained in Chapter 6. Documentation size (measured either in lines of text or in pages of text) is noted to indicate how much the classification process relies on documentation.

Each participant was given the set of classification schedules in Appendix B. This is the same set of classification schedules used to classify the components in the prototype library system. A brief set of instructions on how to use them for classification was included. An example illustrating how a particular component was classified by the author was included with the experiment directions. Appendix C shows the example source listing, its documentation, and the classification code given by the author. Each participant was asked to mention any difficulties expe-

rienced during classification.

The objective of this exercise was twofold: to test classification simplicity and to test the accuracy and consistency of the proposed scheme. Simplicity was measured by evaluating the comments on classification difficulty. Accuracy was evaluated by comparing the selected classification codes to one another and with those of the author. Consistency was evaluated by corroborating how well those classification codes describe component functionality and environment.

Classification Difficulty

Concerning classification difficulty, there was common agreement on the following points:

- The classification task was not difficult. Once the participants read the instructions and looked at the accompanying example, they proceeded without any difficulty.
- Terms with more synonyms were easier to use for classification because the concept they denote was easier to grasp (i.e., more specific.) Terms without synonyms were considered sometimes as being ambiguous.
- Most of the effort was invested in understanding what the program does.
- Most of the information about a program was inferred from reading the documentation. Code description and interline comments were preferred over source code listings.
- Poorly documented components were difficult to classify.
- It was difficult to classify application and setting facets. Almost any program in the sample could be used in more than one setting.

- The medium facet was not well–defined.
- A written definition of each term in the schedules would have helped determine the accuracy of the terms selected.

From these points, we can draw the following five observations:

1) **Reusers can classify their own programs.** There was a consensus that the classification task is relatively simple. In addition, reusers already understand their own programs so it takes less effort for them to classify them. Therefore they should be encouraged and trained to classify their own components.

2) **Synonyms help classification.** Synonyms proved to be an effective technique to speed up conceptual focusing during classification. Thesauri have been used primarily for helping in query construction during document retrievals. In our library system, this is the primary objective as well. Evidence from this exercise, however, shows their potential for speeding the classification process. Synonyms help to quickly exclude from consideration classification terms that might otherwise had to be disambiguated by reading their definitions. Synonyms usually provide a general idea of the meaning of the terms to solve potential ambiguities.

3) **Schedules should be easy to expand.** The schedules were not complete and included some ambiguous terms. Participants suggested some term additions and some term accommodations that were implemented without any problem. This exercise reconfirms what has been said about the need for flexibility in the schedules. Rather than aiming for a complete and perfect schedule, capacity of expansion is more important. Users constantly need more terms.

4) **Generic components may be difficult to classify.** In this test, all of the five components classified were somewhat generic. They could be used in more than one application or setting. Participants mentioned some difficulty in trying to locate particular terms for the environment facets. The reason for this difficulty

was that the participants did not have all the information about the components. If the participants were classifying their own programs, however, this might not have been an issue.

5) **Documentation is more important than code listing.** Since most information needed for classification came from code descriptions, some code description standards would expedite the classification process by providing the right information in the right place. This problem would be solved by asking the reusers to classify their own programs. This would eliminate the need to transmit descriptive information to the librarian. Another problem is the information required by the prospective reuser. One way to cope with this problem is by providing precise code descriptions in catalog format.

Experiment Contributions

The information participants provided about the scheme's deficiencies was a valuable contribution. Most participants (70%) observed that the medium facet included some ambiguous terms. They had some difficulty finding an appropriate term from that facet. They suggested written on–line definitions of each term to speed up disambiguations. They suggested some term additions and deletions, and they wanted more synonyms in the environment schedules.

After the above suggestions were incorporated, some of the participants were asked to repeat the exercise. Their comments this time were substantially more positive. They had less difficulty finding the right terms. Such feedback should be encouraged by providing a mechanism for reusers to continuously make suggestions.

This exercise also tested the expansion capabilities of the scheme. When adding new terms to an existing schedule, parts of the collection may have items classified under conceptually close terms that should be reclassified under the new

term. This is a reclassification problem and is one of the main problems in any classification scheme, especially those organized around a hierarchical structure. This scheme, reduces this problem by using facets instead of a hierarchy. Use of synonyms in the schedules reduce this problem even further by ensuring, to a certain extent, that when a new term is added to the schedules it is truly a new term that could not be accommodated in the current schedule as another synonym. Reclassification, therefore, is substantially reduced.

Reclassification is usually a consequence of misclassification. When an item does not fit well in a classification bin, it is placed in one closely related. Then when new terms are added, those items that were classified with conceptually close terms have to be revised to see if the fit is better with the new terms. To reduce misclassification, the policy in this scheme is that if an item does not fit well in any available bin, then it is placed in an unclassified bin. As the set of unclassified terms grows, they are analyzed, new terms added to the schedules, and the items are classified with the new terms.

In this experiment, adding the suggested terms to the schedule did not require any reclassification of the collection. This positive outcome could be attributed to the small size of the collection. In a very large collection some reclassification should be expected. Of course, the misclassification and reclassification problems can not be eradicated completely, but at least the techniques used in this scheme provide better control.

This exercise also helped to define some classification facilities needed by the library system. Since the exercise was conducted manually, there were some complaints about searching difficulties. Classification is expected to be done through the library system, but it is currently done by searching the schedule files using the existing query subsystem. A richer interface is needed to support the classification process.

Classification Consistency and Accuracy

The second objective of this exercise was to test classification consistency and accuracy. Consistency measures the degree of consensus observed by a group of reusers in classifying a given component (e.g., To what extent are classification codes assigned to a component different?). Accuracy measures the match between a component or its description and the description provided by the classification code (e.g., How well does the classification code describe the component attributes?).

Consistency was checked by comparing the classification codes of each participant. Consistency was measured by computing the ratio of participants selecting the same terms to participants selecting different terms. The term selected by the majority of the participants was assumed to be the true term. Terms in the function facet were 100% consistent; all participants selected the same terms. The objects facet showed 95% consistency, and terms for the medium facet were 60% consistent. (An incomplete medium facet definition in the experiment directions was responsible for the 60% figure.) The environment facet showed about 50% consistency. This low rating was because:

1. Sample programs were too generic to be classified under a particular environment.
2. Environment schedules were incomplete and needed more synonyms.

Accuracy was checked by conducting a walkthrough of the classification exercise with some of the participants. There was 100% agreement in the accuracy of the function and objects terms. About 50% of the participants thought that some terms in the medium facet were not appropriate or were ambiguous in describing the medium where the components performed the function. There was complete agreement in the descriptive accuracy of the terms used in the system–type and functional–area facets. All participants agreed that the setting facet used in the exercise was limited to too few terms.

It can be observed that the schedules richer in synonyms were the better performers. From observations made by the participants, it was noted that a written definition for each term is essential in helping term disambiguation.

Test Results

Testing the scheme in a real classification situation, although limited in size and scope, showed three important characteristics of the proposed scheme:

1. Ease of classification. The reusers may classify their own programs.
2. Ease of expansion. New terms can be added to the schedules with minimal reclassification.
3. The scheme is adaptable. The feedback from reusers on their classification performance can be used to customize the vocabulary to meet the needs of a particular environment.
4. Accuracy and consistency. The most relevant facets proved to be precise and consistent when used for classification by reusers.

Reuse Effort Estimation

This section presents two tests conducted to evaluate the reuse effort estimation capabilities of the library system. The tests asked reusers to examine and rank, from least to most reusable, a list of functionally equivalent components by estimating reuse effort.

The degree of reusability of a code fragment is a function of the number of requirements it meets. It is the degree of matching between reuser requirements and program features. If the match is perfect, then the reuse effort should be negligible. It is called *use* rather than *reuse*. The term reuse implies that certain adaptations

are needed to make the match. The reuse effort is proportional to the number of mismatches between given requirements and available program features.

Our classification scheme groups functionally equivalent components, that is, components that have very similar functional requirements. The specific implementation details are not part of the classification scheme. Some, however, may be inferred from the operational environment of the components. Certain program features have been selected as indicative of implementation effort: size, complexity, programming language, and documentation.

The objective of this test was to determine to what extent these selected program features succeed in estimating reuse effort for a sample of functionally equivalent components, that is, to test the evaluation subsystem presented in Chapter 6.

Participants were given a set of functional requirements and a group of components that partially matched those requirements. Each participant was furnished with a set of requirements for a search program and five components that partially matched these requirements. The initial requirements and the corresponding complete query were:

THE INITIAL REQUIREMENTS

Function:	Search
Data Structure:	Files made of three field records (name, address, age)
Application:	Business
Language:	Pascal

THE COMPLETE QUERY

function objects medium system − type func − area setting

⟨search, records, file, report − generator, inventory − control, advertising⟩

It was further assumed that the library system did not find an exact match for the required component. Information for each of the five potentially reusable components included classification code, reuse metrics, catalog description with references, source listing, and documentation. Appendix D shows the information for one of the candidates. The classification codes for the five candidates were:

1. search/pointers/table/table–handler/program–development/software–shop
2. search/pointers/tree/table–handler/program–development/software–shop
3. search/numbers/array/*/*/*
4. search/patterns/file/line–editor/program–development/*
5. search/strings/list/line–editor/program–development/*

The task of each participant was to rank the components according to estimated reuse effort. Reusers were asked to explain their evaluation process and the specific component features on which their selection was based.

Table 7.2 shows the reuse metrics for the components in the experiment. Table 7.3 shows the outcome of the test. The first five columns are the ranking given by each participant. The last column shows the ranking as computed by the evaluation subsystem of the library system. Ranking correlation was low. Each of the reusers considered different criteria for their evaluation. Four of them did not look at the reuse metrics; their criteria were level of abstraction and implementation details such as, binary search vs. linear search, implicit tree structure vs. array, and main memory vs. disk residence of data.

A review of the results with the participants showed that the low correlation in their answers was due to the vagueness of the requirements definition. The requirements did not specify strict implementation constraints, thus allowing the reusers to select among the various implementations. During the walkthrough that followed the exercise, they agreed that if stronger requirements (e.g., ordered data

Component Number	Reuse Metrics: Size	Complexity	Explicit Interface	Implicit Interface	Documentation	Language
1	29	09	02	10	08	Pascal
2	41	07	03	03	10	Pascal
3	10	03	04	00	08	PDL
4	64	02	04	00	08	Pascal
5	13	01	07	00	07	Fortran

Table 7.2: Reuse Metrics of the First Experimental Sample

Rank	Participants 1	2	3	4	5	Evaluation Subsystem
1st	4	2	3	2	2	3
2nd	3	3	2	4	3	2
3rd	2	1	1	3	4	4
4th	1	4	4	1	1	5
5th	5	5	5	5	5	1

NOTE: Table entries are component numbers

Table 7.3: Ranked Components for First Sample

```
        Initialize
A:      Read the next employee card
        IF no more data
                THEN    calculate and print average hours worked
                        STOP
                ELSE    process the card and check
                        update for weekly average
                        go back to (A) for next employee
```

Figure 7.3: Specification for a Simple Payroll Program

in the files) or some indication of implementation constraints had been given, (e.g., files accessed only sequentially), they could have narrowed down their choices more precisely. In those circumstances, there was general agreement how to rank the components. When a tie was found, there was consensus that the reuse metrics would be a valuable source of discrimination information.

A second experiment was conducted to verify this hypothesis. The participants were given requirements for a simple payroll program and three versions of a payroll program. These samples were functionally equivalent and were implemented in the same fashion. The payroll program specification is shown in figure 7.3 and each version was implemented in a different language: Basic, Cobol and Fortran respectively. The requirements called for a Pascal implementation and functionally the three samples matched the requirements. The reuse metrics used to measure these components are shown in Table 7.4. Table 7.5 presents the result of the experiment.

The ranking correlation improved significantly among the participants and between the participants and the evaluation subsystem. It was confirmed that the reusers employed the reuse metrics heavily in their evaluation. It is expected that this will be the case in a rich library system where several functionally equivalent versions of programs will be available. In those circumstances, the criteria applied

Component Number	Reuse Metrics Size	Complexity	Explicit Interface	Implicit Interface	Documentation	Language
1	26	05	00	00	09	Basic
2	73	02	00	00	09	Cobol
3	22	02	00	00	09	Fortran

Table 7.4: Reuse Metrics of the Second Experimental Sample

Rank	Participants 1	2	3	4	Evaluation Subsystem
1st	3	3	3	3	3
2nd	1	1	1	1	1
3rd	2	2	2	2	2

NOTE: Table entries are component numbers

Table 7.5: Ranked Components for Second Sample

by the human reusers is expected to match that of the evaluation subsystem.

Experiment Results

During the first experiment, reusers indicated they needed more specific requirements to conduct a more effective evaluation. In the second experiment, when the sample of candidate components met most implementation requirements, reuse metrics were used for ranking.

This exercise showed the relevance of implementation details when conducting an evaluation. Implementation details play a vital role in the selection of a software component. For example, binary search can be used only if data access is random rather than sequential and if the input is ordered. Although the term *tree* from the medium facet may imply random access and ordered input when used in the classification of a search component, it is not sufficient evidence, that the compo-

nent meets these implementation requirements. The resolution of the classification scheme is not high enough to describe specific implementation details. One reason that specific implementation details are not derived from the classification code is because the classification schedules were designed to describe functionality and environment. Implementation details were assumed to be part of the code description in the documentation. It should be recalled that one major assumption during the design of the classification scheme was that, in general, certain implementation details can be derived from context–rich descriptors.

Resolution of the classification scheme can be augmented by including more specific terms in the schedules. For example, the term ***search*** in the function facet could be replaced by the terms ***binary–search*** and ***linear–search***. Each describing specific implementation information about the function search. The term *tree* in the medium facet can also be replaced by the terms *binary–tree, B–tree, linked–binary–tree, heap,* and *minimum–spanning–tree*. The classification code

binary–search/integers/B–tree

for example, is more implementation specific than the code

search/numbers/tree

Both are functionally equivalent, but the first carries more application specific information.

With such an increase in specificity from the classification schedules, the evaluation process is guaranteed to focus on reuse metrics rather than on implementation differences. In large collections, where large number of almost identical components is the rule, an automatic evaluation is very attractive.

Decomposing general terms into more specific ones, when applied to a schedule of a large collection, may generate some reclassification problems. An initial schedule, therefore, should include as many specific terms as possible. Literary warrant

and walkthroughs like the one conducted in this exercise will assure completeness in the schedules.

It can be concluded from these two exercises that:

- The evaluation subsystem is effective when differentiating among components with very similar implementations.
- The prototype classification schedules do not provide enough resolution to classify domain implementation details.
- Resolution can be augmented by expanding the schedules to include implementation specific terms.

Conclusion

Three different evaluations of the prototype library system were conducted:

1. Retrieval Effectiveness
2. User–conducted Classification
3. Reuse Effort Estimation

The results of these evaluations should be taken as indicators of what could be expected in a fully developed library system but not as conclusive general results.

The retrieval effectiveness test showed that a vocabulary organized around a classification scheme substantially improved retrieval performance on the collection. Retrieved samples ordered by their conceptual distances showed considerably better recall and precision figures than retrieved samples ordered randomly.

The test to determine the effectiveness of the classification scheme and its ease for reusers showed that:

- Reusers can classify their own programs.
- Schedules are easy to expand.

- The scheme is adaptable.
- The schedules show accuracy and consistency in classification.

The test on reuse effort estimation showed that the evaluation subsystem is effective when samples of very similar components were used and that, although the prototype classification schedules did not provide enough resolution for application specific information, they can be expanded with very specific terms to augment their resolution.

In summary, the tests described in this chapter confirm some of the evidence and arguments that were used as the basis for constructing the classification scheme and the library system. These tests showed the need for continuous revision of the classification schedules.

CHAPTER 8

Summary, Future Work, and Conclusions

Summary

This dissertation proposes a classification scheme for collections of code fragments intended for reusability.[1]

Reusability–related attributes are organized in a faceted scheme for ease of expansion of the scheme and to provide very specific definitions of the classes considered. Term synthesis is the method used to define new classes. The scheme can be expanded by adding new terms to the facets without disturbing the basic structure.

A library system is presented that integrates the proposed classification scheme with an evaluation mechanism based on reusability–related attributes, programming languages characteristics and reuser experience. The evaluation mechanism is used to select potentially reusable components from a sample of functionally equivalent components.

Tests with the prototype library system showed that, when the proposed classification scheme was used with the retrieval mechanism, retrieval performance improved significantly. Other tests showed that the scheme is easy to use by potential reusers and that the proposed evaluation mechanism is effective when used to select among very similar components.

The remainder of this section presents a summary of the results of our research as presented in this dissertation. Each point has been discussed in more detail above.

[1] Code fragments, in this thesis, are functionally identifiable components written in a high level language of size ranging from 20 to 200 lines of source code.

An Analysis of the Code Reuse Process

The code reuse process was analyzed in detail. As a result, an infrastructure was proposed as a requirement for a partial solution of the code reuse problem. This infrastructure is centered around a software classification scheme. The proposed infrastructure consists of a specialized library, a classification scheme, and a support system to reduce reuser cognitive stress and speed up the selection process.

A Formal Presentation of a Faceted Scheme for Software Classification

A formal notation to describe faceted schemes was introduced and used to express a measure of relevance based on the citation order of the facets. A measure of relevance at the facet level is essential to a retrieval system. Retrieval queries can be arranged to retrieve components by order of relevance.

A Measure of Term Closeness Based on Conceptual Distances

An approach to measuring closeness among terms in classification schedules was introduced. This approach consists on defining facet–relevant attributes for each facet. Facet terms are assigned distances to these attributes based on their conceptual closeness to them. A conceptual weighted graph results. This weighted graph is used to order the retrieved sample by their conceptual closeness, thus substantially improving retrieval performance.

Definition of Six Reuse-Related Facets

An analysis of the code reuse process together with the inspection of over 300 programs and program descriptions from different classes and from several software directories led to the synthesis of six facets that provide enough resolution to identify software components for reusability purposes. The six facets are divided into two

groups. Function, objects, and medium form the component functionality group and system–type, functional–area, and setting form the component environment group. These facets are prearranged in a citation order from more to less relevant to reusability.

A Top–down Approach to Define Terms for the Facets

A top–down approach to select terms for the facets was introduced to complement the bottom–up approach usually followed during literary warrant. This top-down approach consists of identifying widely accepted logical structures for certain classes of programs. Descriptive terms for these structures are extracted and added to the corresponding facets. This approach saves considerable time during schedule construction by selecting all the terms that describe a class of programs at once without having to analyze several of them as in literary warrant.

A Thesaurus as Vocabulary Control for Classification and Retrieval

A list of synonyms was added to each term in the schedules to speed up the classification process, to increase the precision (descriptive power) of the schedules, and to help define retrieval queries. Thesauri are typically used in retrieval systems to help define retrieval queries. The use of thesauri for classification was introduced. The use of a thesaurus significantly simplifies the classification process by providing the classifier an approximate but immediate definition of the terms used for classification.

Introduction of Six Reuse–Related Metrics

Analysis of the code reuse process resulted in the identification of six inherent program characteristics. These inherent characteristics play an essential role during

program understanding and during program adaptation. Metrics for each attribute and justification for their validity were presented.

The metrics are program size measured in lines of source code, program complexity measured in number of decisions, interface complexity measured in number of explicit and implicit arguments, program documentation measured by an overall subjective rating, programming language used measured by a closeness measure between the source and the target language, and reuser experience measured by a subjective rating. An important criteria for the selection of the metrics, besides their relevance during the code reuse process, was their simplicity to compute. Software components should be easily measured in order to be readily added to the collection.

Use of Fuzzy Functions to Normalize and Compare Reuse Metrics

Reuse metrics have to be normalized in order to obtain a reuse effort measure for each component considered for reuse. Fuzzy functions provide a technique for normalizing metrics that truly characterize the fuzzy nature of measuring reuse effort. Fuzzy functions, moreover, can be tuned to the environment where the collection is being used. An approach, also based on fuzzy functions, that considers reuser experience as an overall modifier of the metrics for a given situation was introduced. Reuser experience is considered as a meta–fuzzy function that modifies the characteristics of the fuzzy functions of each metric. The evaluation system proposed considers different levels of user experience.

Integration of the Concepts above into a Prototype Library System

The major contribution of this dissertation is the integration of all these concepts into an integrated library system. The classification scheme was used as a

knowledge structure in a data base management system to drive the query front–end and to guide the retrieval process. The reuse metrics and the fuzzy function concepts were used in a support system for the evaluation of similar components based on a multi–dimensional evaluation methodology.

The prototype system, although tested and evaluated in a limited experimental environment, showed encouraging results: 1) Use of relevance measures, conceptual distances among terms, and a thesaurus–based index resulted in very high precision retrieval values. 2) Classification is significantly simplified by the use of thesauri. 3) The evaluation subsystem arrives at the same ordering assigned by reusers when the components are very similar.

The prototype system can be used as an experimental tool to advance the knowledge of reusing software components. The prototype system could be used in a real environment to help the classification schedules. Another use of the prototype system would be in developing a larger library system in a software production environment. An incremental development would take advantage of the expansion capabilities of the classification scheme and would allow for a continuous evolution of the system.

Recommendations for Future Work

While the results of this dissertation are very encouraging, there is a need to expand the scope of the experiments before any meaningful generalizations can be made. This section presents a list of recommendations for specific problems that need more research and some recommendations about implementing the prototype library system in a real software development environment.

Suggested Problems

Standardize Schedule

A classification schedule is never complete. More needs to be known about schedule building in order to produce almost complete schedules. There is evidence that producing first–time classification schedules and making them complete enough to satisfy different kinds of users is difficult. The typical approach is to design one and impose it on the users. An experiment to produce more acceptable classification schedules would involve asking users to give their own definitions of each term used in the proposed schedule, analyzing all the definitions returned, and giving each term a single unifying definition that satisfies most users. The library subsystem could be used to check the performance of the proposed schedule in the collection.

The problem of schedule completeness could also be approached analytically. Probability theory could be used to calculate schedule completeness from a given incomplete schedule. The problem is difficult because schedule completeness is a function of how well it describes an unknown universe of objects. How complete it is depends on how much is known about that universe.

Identification and Standardization of Logic Structures

One use of classification is to identify often recurring programs in a collection as candidates for standardization. Logic structures of these programs can be analyzed to synthesize a standard logic structure for a given class of programs. This study would significantly help the software standardization process.

Validation of Reuse Effort Metrics

The proposed set of reuse–related metrics is but an initial general set. For any particular environment, other factors may be relevant for measuring reuse effort

and should be included in the evaluation subsystem. Different environments may select different sets of reuse effort estimation metrics. Validation of a general set of reuse effort estimation metrics would be possible only if there is a set common to several software development environments. The library system proposed here could be used as the medium to define that common set.

Integrate Prototype with Software Development Tools

A very interesting idea to explore is the possibility of integrating the library system as a tool into a software development system. The advantages would be enormous. Once a set of potentially reusable components is retrieved, a module interconnection system or a configuration management system would detect any incompatibilities between the available components and the given requirements, thus making the selection task easier and more precise. A documentation management system could be used to locate the proper level of documentation for a given selected component during the adaptation phase. Other benefits would be to enforce programs and documentation standards thus increasing the overall potential for reusability.

Testing the Prototype in a Real Environment

The tests performed on the classification scheme and on the prototype library system, although positive, were very limited in scope and can not be generalized. To conduct more comprehensive experiments, a real environment where code is available to reuse and where reuse could speed up the software development process is suggested. To transfer the present prototype into a software production environment, the following four steps are suggested:

1) **Expand and refine the prototype.** A mechanism to handle the storage and retrieval of documentation is needed along with a policy to determine how to select

and store that documentation. A performance monitoring mechanism is needed at the data base management system level to detect procedural bottlenecks as well as a system to monitor users performance and retrieval patterns while effectively implementing the cow–path approach mentioned in Chapter 7. Performance data can be used to modify and upgrade the system. A bulletin board should be available to capture user opinions. Other areas for refinement include an improved user interface, an increase in the data base capacity, and improvement in performance.

2) **Start with a small collection.** Select a small collection of components that seem to be used most often during development. Derive a classification schedule for that small collection. As users become familiar with the system, expand the collection and expand the schedules with terms preferred by the users. A set of refined classification schedules should result.

3) **Make system available to users.** Seminars should be given on how to use the system and contribute to the collection. As the number of contributors increases, so does the collection and with it the probability of reusing available components.

4) **Monitor Performance.** Careful monitoring of library use should be conducted from the early stages of implementation. Recall and precision should be computed, whenever possible, during retrieval and their values should be logged continuously. Schedule accuracy should also be monitored. Ambiguous, undefined, and conflicting terms reported by the reuser should be corrected at once. During classification of new components, reusers should report on the difficulties found during description, documentation, and measuring (e.g., computing the reuse metrics).

Reuse effort estimates should be compared, whenever possible, to those expressed by the reuser. After successful reuses, reusers should be interviewed about their reuse difficulties. Were reuse effort estimates useful? helpful? accurate? This information should be used for a continuous tuning of the fuzzy functions used to evaluate reuse-effort.

Conclusions

The main conclusion derived from this work is that the reuse of code fragments implies the reuse of their higher level representations. When reusing code, most decisions about its reuse are made by examining its documentation. Code is seldom inspected unless documentation is not good. What is being considered during the reuse process are high level representations in the form of documentation (specifications, design, code descriptions, and implementation details). Code enters the picture only during the selection of similar components and during the code conversion phase. Code reuse could be significantly improved if a formalism existed to describe higher level representations.

References

[AFIP80] *Taxonomy of Computer Science and Engineering.* Compiled by the AFIPS Taxonomy Committee, AFIPS Press, 1980.

[ANDE73] Anderberg, M. R. *Cluster Analysis for Applications.* Academic Press, New York, 1973.

[APPL83] Stanton, J. and Dickey, J. *The Book of Apple Computer Software.* The Book Co., Lawndale, CA 90260, 1983.

[ARAN85] Arango, G., Baxter, I., Freeman, P., and Pidgeon, C. "Maintenance and Portage of Software by Design Recovery." Submitted to *Conference on Software Maintenance-1985*, Washington, D.C., November, 1985.

[ARON69] Aron, J. "Estimating Resources for Large Programming Systems." *NATO Conference on Software Engineering Techniques*, M. Chartwer, N.Y., 1969.

[ATHE82] Athey, T. H. *Systematic Systems Approach: An Integrated Method for Solving Systems Problems.* Prentice-Hall, Englewood Cliffs, N. J., 1982.

[BALB75] Balbine, G. "Better Manpower Utilization Using Automatic Restructuring." *AFIPS Proceedings of the National Computer Conference*, pages 319-327, 1975.

[BALZ83] Balzer, R., Cheatham, T. E., and Green, C. "Software Technology in the 1990's: Using a New Paradigm." *COMPUTER*, **16**(11):39-45, November, 1983.

[BASI80] Basili, V. R. *Tutorial on Models and Metrics for Software Management and Engineering.* IEEE Computer Society Press, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, 1980. Initially presented at the IEEE Computer Society's 4th International Computer & Software Applications Conference, October 27-31, 1980.

[BASI84] Basili, V. R. and Perricone, B. T. "Software Errors and Complexity: An Empirical Investigation." *Communications of the ACM*, **27**(1):42-52, January, 1984.

[BLAI84] Blair, D. C. "The Data-Document Distinction in Information Retrieval." *Communications of the ACM*, 27(4):369-374, April, 1984.

[BLAI85] Blair, D. C. and Maron, M. E. "An Evaluation of Retrieval Effectiveness for a Full-text Document-retrieval System." *Communications of the ACM*, **28**(3):289-299, March, 1985.

[BOEH81] Boehm, B. W. *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[BOIS83] Boisvert, R. F., Howe, S. E., and Kahaner, D. K. "The GAMS Classification Scheme for Mathematical and Statistical Software." *SIGNUM Newsletter*, 18(1):10-18, January, 1983. A publication of the Association for Computing Machinery.

[BOLS75] Bolstad, J. "A Proposed Classification Scheme for Computer Program Libraries." *SIGNUM Newsletter*, 10(2-3):32-39, November, 1975. A publication of the Association for Computing Machinery.

[BOOM80] Boom, H. J. and DeJong E. "A Critical Comparison of Several Programming Language Implementations." *Software: Practice and Experience*, 10:435-473, 1980.

[BOWL83] Bowles, K. "Reusability in Ada." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 77-78. ITT Programming, Newport, RI, September, 1983.

[BROO76] Brooks, R. "How a Programmer Understands a Program: a Model." Technical Report 97, *University of California, Irvine.* Department of Information and Computer Sciences, 1976.

[BROO78] Brooks, R. "Using a Behavioral Theory of Program Comprehension in Software Engineering." In M. V. Wilkes, editor, *3rd International Conference on Software Engineering*, pages 196-201. IEEE, Atlanta, GA, May, 1978.

[BROW77] Brown, P. J., editor, *Software Portability.* Cambridge University Press, New York, 1977.

[BUCH79] Buchanan, B. *Theory of Library Classification.* Clive Bingley, London, 1979.

[COOM70] Coombs, C. H., Dawes, R. M. and Tversky, A. *Mathematical Psychology: An Elementary Introduction.* Prentice Hall, Englewood Cliffs, NJ, 1970.

[COOP82] Cooper, R. G. "MetaCAD: A Knowledge Based Software Support Environment." *Huges Aircraft Company*, Internal Report, Irvine, CA., 1982.

[COUL83] Coulter, N. S. "Software Science and Cognitive Psychology." *IEEE Transactions on Software Engineering*, SE-9(2):166-171, March, 1983.

[CURT79A] Curtis, B., et al. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics." *IEEE Transactions on Software Engineering*, 5(2):96-104, March, 1979.

[CURT79B] Curtis, B., Sheppard, S. B., and Milliman, P. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics." In F. L. Bauer, editor, *Proceedings of the Fourth International Conference on Softwarfe Engineering*, pages 356-360. IEEE, Munich, Germany, September, 1979.

[CURT83] Curtis, B. "Cognitive Issues in Reusability." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 192-197. ITT Programming, Newport, RI, September, 1983.

[DEWE65] Dewey, M. *Decimal Classification and Relative Index.* Forest Press, Inc., New York, 1965. Edition 17.

[EVAN83] Evangelist, W. M. "Software Complexity Metric Sensitivity to Program Structuring Rules." *Journal of Systems and Sofware*, **3**:231-243, 1983.

[FEUE79] Feuer, A. R. and Fowlkes, E. B. "Results from an Empirical Study of Computer Software." In F. L. Bauer, editor, *Proceedings of the Fourth International Conference on Softwarfe Engineering*, pages 351-355. IEEE, Munich, Germany, September, 1979.

[FITZ78] Fitzsimons, A. and Love, T. "A Review and Evaluation of Software Science." *Computing Surveys,* **10**(1):3-18, March, 1978.

[FREE68] Freeman, R. R. and Atherton, P. "Final report of the Research Project for the Evaluation of the UDC as the Indexing Language for a Mechanized Reference Retrieval System." In R. Molgaard-Hansen and Malcom Rigby, editors, *First Seminar on UDC in a Mechanized Retrieval System.* International Federation for Documentation Committee on Classification Research, Copenhagen, September, 1968.

[FREE76] Freeman, P. "Reusable Software." (Research Proposal), Irvine, CA. University of California, ICS Dept., 1976.

[FREE80] Freeman, P. "Reusable Software Engineering: A Statement of Long-Range Research Objectives." Tech. Report TR-159 Department of Information and Computer Science Technical *University of California, Irvine,* November, 1980.

[FREE83] Freeman, P. "Reusable Software Engineering: Concepts and Research Directions." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 2-16. ITT Programming, Newport, RI, September, 1983.

[FRIE77] Friedman, F. L. and Koffman, E. B. *Problem Solving and Structured Programming in Fortran.* Addison-Wesley Co., Reading, Massachusetts, 1977.

[GAMS80] "Guide to Available Mathematical Software." *Center for Applied Mathematics, National Bureau of Standars*, Washington, D.C. 20234, 1980.

[GILB77] Gilb, T. *Software Metrics*. Winthrop, Cambridge, MA, 1977.

[GOME79] Gomez, J. E., "An Interactive Fortran Structuring Aid." In F.L. Bauer, editor, *Proceedingds of the Fourth International Conference on Software Engineering*, pages 241-244. IEEE, Munich, Germany, September, 1979.

[GOOD83] Goodell, M. "Quantitative Study of Functional Commonality in a Sample of Comercial Business Applications." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 279-286. ITT Programming, Newport, RI, September, 1983.

[GORD79] Gordon, R. D. "Measuring Improvements in Program Clarity." *IEEE Transactions on Software Engineering*, SE-5(2):79-90, March, 1979.

[HAFN81] Hafner, C. D. *An Information Retrieval System Based on a Computer Model of Legal Knowledge*. UMI Research Press, Ann Arbor, Michigan, 1981.

[HALS77] Halstead, M. H. Operating and Programming Systems Series: *Elements of Software Science*. Elsevier North-Holland, Inc., New York, N.Y., 1977.

[HALS79] Halstead, M. H. *Advances in Software Science*. Academic press, New York, NY, 1979, pages 119-172.

[HOGA80] Hogarth, R. M. *Judgement and Choice: The Psychology of Decision*. John Wiley and Sons, Chichester, 1980.

[HORO78] Horowitz, E. and Sahni, S. *Fundamentals of Computer Algorithms*. Computer Software Engineering Series, Computer Science Press, Potomac, Maryland, 1978.

[HORO83A] Horowitz, E. "An Expansive View of Reusable Software." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 250-261. ITT Programming, Newport, RI, September, 1983.

[HORO83B] Horowitz, E. *Programming Languages: A Grand Tour*. Computer Science Press, 1983.

[HORO83C] Horowitz, E. *Fundamentals of Programming Languages*. Computer Science Press, 1983.

[IBMS83] *IBM Software Directory*. 1983 edition, IBM, 1 Culver Road, Dayton, NJ 08810, 1983.

[ICHB83] Ichbia, J. D. "On the Design of Ada." In R. E. A. Mason, editor, *Information Processing 83*, pages 1-10. International Federation for Information Processing, September 19-23, 1983.

[ICP 83] *ICP Software Directory.* International Computer Programs, Inc., 9000 Keystone Crossing, P.O. Box 40946, Indianapolis, IN 46240, 1983. Includes Sytems Software, Cross Industry Applications and Industry Specific Applications.

[IDS 83] *International Directory of Software.* 1983-84 edition, Computing Publications Ltd., First Federal Building, Suite 401, Pottstown, PA 19464, 1983.

[IMSL84] *International Mathematics and Scientific Library.* 10th edition, IMSL Inc., 7500 Bellairer Blvd., Houston, TX 77036, 1984.

[ITT83] *Workshop on Reusability in Programming.* Alan Perlis, editor, Sponsored by ITT Programming, Newport, RI, September 7-9, 1983.

[JEFF81] Jeffries, R., et. al. "The Process Involved in Designing Software." In J. R. Anderson, editor, *Cognitive Skills and Their Acquisition*, Lawrence Erlbaum, Hillsdale, NJ, 1981, pages 255-284, chapter 8.

[KAHN82] Kahneman, D., Slovic, P. and Tversky, A. *Judgement Under Uncertainty: Heuristics and Biases.* Cambridge University Press, Cambrige, 1982.

[KERN83] Kernigham, B. W. and Plaguer, P. J. *Software Tools in Pascal.* Addison-Wesley Co., Reading, Massachusetts, 1983.

[KIM 83] Kim, K. H. "A Look at Japan's Development of Software Engineering Technology." *COMPUTER*, **16**(5):26-37, May, 1983.

[KRUS78] Kruskal, J. B. and Wish, M. *Multidimensional Scaling.* Series: Quantitative Applications in the Social Sciences, Sage University Papers, Beverly Hills, 1978.

[KUMA79] Kumar, K. *Theory of Classification.* Vikas Publishing House Pvt. Ltd., New Delhi, 1979.

[LANE79] Lanergan, R. G. and Poynton, B. A. "Reusable Code: The Application Development Technique of the Future." In *Proceedings of the IBM SHARE/GUIDE Software Symposium*, IBM, Monterey, CA, October, 1979.

[LEWI83] Lewis, T. *Microbook: Database Management for the IBM Personal Computer.* Dilithium Press, Beaverton, Oregon, 1983.

[LIND72] Lindsay, P. H. and Norman, D. A. *Human Information Processing.* Academic Press, New York, 1972.

[LOVE77] Love, T. "An Experimental Investigation of the Effects of Program Structure on Program Understanding." *ACM SIGPLAN Notices*, **12**:105-113, March, 1977.

[LYON80] Lyon, M. J. "Structured Retrofit–1980." *Proceedings of SHARE 55*, pages 263-265, 1980.

[LYON81] Lyon, M. J. "Salvaging Your Software Asset (tools based maintenance)." In A. Orden, editor, *AFIPS Proceedings of the National Computer Conference*, pages 337-341, Chicago, Ill., May 1981.

[MALT75] Maltby, A. *Sayers' manual of Classification for Librarians.* André Deutsch Ltd., 105 Great Rusell St., London WCI, 1975.

[MATS80] Matsumoto, Y. "SWB System: A Software Factory." In H. Hunke, editor, *Software Engineering Environments*, pages 305-318. North-Holland, New York, 1980.

[MAYS68] Mays, E. M., *A Classification Scheme for Law Books.* Butterworths, London, 1968.

[MCCA76] McCabe, T. J. "A Complexity Measure." *IEEE Transactions on Software Engineering*, SE-**2**(4):308-320, Dec, 1976.

[MCIL76] McIlroy, M. D. "Mass Produced Software Components." In *Software Engineering Concepts and Techniques*, pages 88-98. Petrocelli/Charter, Brussels 39, Belgium, 1976. From the 1969 NATO Conference on Software Engineering.

[MIAR83] Miara, J. R. *et al.* "Program Indentation and Comprehensibility." *Communications of the ACM*, **26**(11):861-867, November, 1983.

[MICH83] Michalski, R. S. and Stepp, R. E. "Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonomy." *IEEE Transactions on Pattern Analysis and Machine Intelligence,* (PAMI-5)4:396-409, July 1983.

[MILL56] Miller, G. A. "The Magical Number Seven, Plus-or-minus Two: Some Limits on Our Capacity for Processing Information." *Psychological Review*, **63**(2):81-97, March, 1956.

[MILL77] Mills, J. and Broughton, V. *Bliss Bibliographic Classification, Introdiction and Auxiliary Schedules.* Butterworth and Co., London, 1977.

[MILL80] Miller, J. C. "Structured Retrofit." *Techniques of Program and System Maintenance*, Etnotech, Lincoln, Nebraska, pages 85-86, 1980.

[MILL77] Mills, J. and Broughton, V. *Bliss Bibliographic Classification, Introduction and Auxiliary Schedules*, Butterworth & Co., London, 1977.

[MOHE81] Moher, T. and Scneider, M. "Methods for Improving Controlled Experimentation in Software Engineering." In S. Jeffrey, editor, *Proceedings of the Fifth International Conference on Software Engineering*, pages 224-233. IEEE, San Diego, CA, March, 1981.

[MORR79] Morrisey, J. H. and Wu, L. S. Y. "Software Engineering: An Economic Perspective." In F.L. Bauer, editor, *Proceedingds of the Fourth International Conference on Software Engineering*, pages 412-422. IEEE, Munich, Germany, September, 1979.

[NEIG80] Neighbors, J. M. *Software Construction Using Components*, Ph.D. thesis, University of California, Irvine, 1980. Also available as Department of Information and Computer Science Technical Report number 160.

[NILS80] Nilsson, N. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[NORW82] Norwich, A. M. and Turksen, I. B. "The Construction of Membership Functions." In Ronald R. Yager, editor, *Fuzzy Set and Possibility Theory: Recent Developments,* pages 61-67. Pergamon Press, New York, 1982.

[PEMB82] Pemberton, S. and Daniels, M. C. *Pascal Implementation: The P4 Compiler*, Ellis Horwood Ltd., Chichester, England, 1982.

[PRIE82] Prieto-Diaz, R. and Neighbors, J. "Module Interconnection Languages: A Survey." ICS Tech. Report 189, Dept. of Information and Computer Science, *University of California, Irvine,* August, 1982.

[PRIE83] Prieto Diaz, R. "Knowledge Representation Applied to Library Cataloging." In Salvador Perrotti, editor, *Anais do XVI Congresso Nacional de Informatica*, pages 42-47. SUCESU, Sao Paulo, Brazil, October, 1983.

[PRYW79] Prywes, N. S., Pnueli, A. and Shastry, S. "Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development." *ACM Transactions on Programming Languages and Systems*, **1**(2):196-217, October, 1979.

[RAGA77] Ragade, R. K. and Gupta, M. M., "Fuzzy Set Theory: Introduction." In M. M. Gupta, G. N. Saridis, and B. R. Gaines, editors, *Fuzzy Automata and Decision Processes,* pages 105-131, North-Holand, New York, 1977.

[RANG67] Ranganathan, S. R. *Prolegomena to Library Classification.* Asia Publishing House, Bombay, India, 1967.

[RICE83] Rice, J. R. and Schwetman, H. D., "Interface Issues in a Software Parts Technology." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 129-137. ITT Programming, Newport, RI, September, 1983.

[RICH83] Rich, C. and Walters, R. C. "Formalizing Reusable Software Components." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 152-159. ITT Programming, Newport, RI, September, 1983.

[RIGB72] Rigby, M. "The UDC in Mechanized Subject Information Retrieval." *Proceedings of an International Symposium held at the Center of Adult Ecucation, University of Maryland College Park*, pages 126-143, May, 1971. Greenwood Publishing Company, Westport, Connecticut, 1972.

[SALT83] Salton, G. and McGill, M. J., *Introduction to Modern Information Retrieval."* McGraw-Hill, New York, 1983.

[SAMM69] Sammet, J. E. *Programming Languages: History and Fundamentals.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.

[SAMM82] Sammet, J. E. and Ralston, A. (Editors) "The New (1982) Computing Reviews Classification System—Final Version." *Communications of the ACM.* **25**(1):13-25, January 1982.

[SAMM85] Sammet, J. E. "Introduction to the CR Classification System." *Computing Surveys,* pp 20-23 and pp 45-57, January 1985.

[SELB85] Selby, R. W. *A Quantitative Approach for Evolving Software Technologies.* Ph.D. Dissertation, Department of Computer Science, University of Maryland, December, 1984.

[SHAR63] "SHARE Reference Manual." *IBM Users Group*, 1963. With revisions.

[SHAR73] "SHARE Reference Manual." *IBM Users Group*, 1973.

[SHAW81] Shaw, M. et al. "A Comparison of Programming Languages for Software Engineering." *Software: Practice And Experience*, **11**:1-52, 1981.

[SHEN83] Shen, V. Y., Conte, S. D. and Dunsmore, H. E. "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support." *IEEE Transactions on Software Engineering*, **SE-9**(2):155-165, March, 1983.

[SHNE77] Shneiderman, B. "Measuring Computer Program Quality and Comprehension." *International Journal of Man-Machine Studies*, 9:465-478, 1977.

[SOFT83] *The Software Catalog – Microcomputers.* Elsevier, New York, Fall 1983.

[SOFT85] *The Software Catalog – Microcomputers.* Elsevier, New York, Winter 1985.

[SOLO83A] Soloway, E. and Ehrlich, K. "What do Programmers Reuse? Theory and Exeperiment." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 184-191. ITT Programming, Newport, RI, September, 1983.

[SOLO83B] Soloway, E., Bonar, J., and Ehrlich, K. "Cognitive Strategies and Looping Constructs: An Empirical Study." *Communications of the ACM*, **26**(11):853-880, November, 1983.

[SPAR71] Spark Jones, K. *Automatic Keyword Classification for Information Retrieval.* Butterworths, London, 1971.

[SPSS84] *Statistical Package for the Social Sciences.* SPSS, Inc., McGraw-Hill, New York, 1984.

[STAN76] Standish, T. A., Harriman, D. C., Kibler, D. F., and Neighbors, J. M. *The Irvine Program Transformation Catalogue,* Technical Report, University of California, Irvine, January, 1976.

[STAN83] Standish, T. A. "Software Reuse." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 45-49. ITT Programming, Newport, RI, September, 1983.

[SOWA84] Sowa, J. F. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley, Reading, Mass., 1984.

[STRO66] Stroud, J. M. "The Fine Structure of Psychological Time." *Annals of New York Academy of Sciences*, 1966.

[SUGA81] Sugarman, J. H. *The Development of a Classification System for Information Storage and Retrieval Purposes Based Upon a Model of Scientific Knowledge Generation.* Ph.D. Thesis, Boston University, School of Education, Boston Mass., 1981.

[SUNO81] Sunohara, T., et al. "Program Complexity Measure for Software Development Management." In S. Jeffrey, editor, *Proceedings of the Fifth International Conference on Software Engineering*, pages 100-106. IEEE, San Diego, CA, March, 1981.

[SWAR82] Swart, W. and Balzer, R. "On the Inevitable Interwining of Specification and Implementation." *Communications of the ACM*, **25**(7):438-440, July, 1982.

[TAJI84] Tajima, D. and Matsubara, T. "Inside the Japanese Software Industry." *COMPUTER*, **17**(3):34-43, March, 1984.

[TANE78] Tanenbaum, A. S., Klint, P. and Bohm, W. "Guidelines for Software Portability." *Software–Practice and Experience*, **8**(1):681-698, January, 1978.

[VICK60] Vickery, B. C. *Faceted Classification: A Guide to Construction and use of Special Schemes.* Aslib, 3 Belgrave Square, London, 1960.

[WALS77] Walston, C. and Felix, C. "A Method of Programming Measurement and Estimation." *IBM Systems Journal*, **16**(1):54-77, 1977.

[WASS82] Wasserman, A. I. "The User Software Engineering Methodology: An Overview." In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: A Comparative Review*, pages 591-635, North-Holland, 1982.

[WEGE83] Wegner, P. "Varieties of Reusability." In Alan Perlis, editor, *Workshop on Reusability in Programming*, pages 30-44. ITT Programming, Newport, RI, September, 1983.

[WEIS74] Weissman, L. M. "Psychological Complexity of Computer Programs." *ACM SIGPLAN Notices*, **9**(6):25-36, June, 1974.

[WELS80] Welsh, J. and McKeag, M. *Structured System Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[WINS81] Winston, P. H. and Horn, B. K. P. *LISP.* Addison-Wesley, Menlo Park, CA, 1981.

[WOOD81] Woodland, S. N., Dunsmore, H. E. and Shen, V. Y. "The Effect of Modularization and Comments on Program Comprehension." In S. Jeffrey, editor, *Proceedings of the Fifth International Conference on Software Engineering*, pages 215-223. IEEE, San Diego, CA, March, 1981.

[WOLB83] Wolberg, J. R. *Conversion of Computer Software.* Prentice Hall, Englewood Cliffs, NJ, 1983.

[ZADE84] Zadeh, L. A. "Making Computers Think Like People." *IEEE SPECTRUM*, **21**(8):26-32, August, 1984.

Appendix A

A Preliminary Schedule for Application Programs in Communication and Media

COMMUNICATION AND MEDIA

(Type of System)

- Information System
  - Administration-Business System
    - Resource Management System
      - Inventory Management System
        - Order Entry
- Purchasing
- Bookkeeping
        - Accounts Receivable
        - Accounts Payable
        - Billing/Invoicing
      - Transaction Analysis
    - Customer Information System
    - Accounting and Finance System
      - Cost Accounting
      - Payroll
      - General Ledger
      - Financial Reporting
      - Auditing
    - Management Information System
      - Decision Support
      - Planning
      - Project Control
  - Manufacturing Control/Production
    - Materials Control
    - Process Control
      - Work Scheduling
        - Labor Control
        - Equipment Control
  - Scientific and Technical Support System
    - Design
  - General Systems
    - DBMS
      - File Management/Maintenance

Subroutine Package

Turnkey System

(Setting of the Application)

(Entities)
- Associations
  - Unions
  - Clubs
  - Charities
  - Parish
- Foundation
- Chamber of Commerce
- Cemetery
- Library

(Establishments)
- Publisher
- Printshop
- Advertising Agency
- Broadcasting Station
  - Radio
  - TV
  - Cable
    - TV
    - Teletex/Videotex

(Objects the Application Deals With)

(Material Objects)
- Publications
  - Books
  - Reference Materials
    - Manuals
    - Catalogs
    - Directories
  - Periodicals
    - Magazines
    - Newspapers
- Tickets/Checks/Tabs
- Parcels/Packages
- Import/Export Goods
- Communication Equipment

(Symbolic Objects)
- Memberships
  - Subscriptions
- Timeslots

(Function Performed by the Application)

(by form)

- Remote
- Batch
- On-line
- Real Time

(by activity)

- Publishing
  - Translation
  - Printing
    - Typesetting
      - Composition/PhotoComp
        - Editing
  - Circulation/Distribution
- Indexing
  - Classification
    - Cataloging
- Advertising
  - Direct Mail Adv/Marketing
  - Classified Advertising
- Graphics Display/Plotting
- Mailing
- Broadcasting
  - Info/Message Transmission
  - Rating Analysis

Appendix B

Partial Classification Schedules for Software Components

This is a preliminary classification schedule synthesized from an sample of program descriptions by literary warrant. The process followed is outlined in Chapters 3 and 5.

FUNCTION FACET

add	increment/total/sum
append	affix/attach/concatenate/join/add
close	release/detach/disconnect
compare	test/relate/collate/match/check/verify/detect
complement	negate/invert
compress	reduce/condense/contract/crypt
create	produce/form/build/originate/generate/make/set
decode	multi_way/multi_branch/selector
delete	remove/erase/cancel/eliminate/extract/strip
divide	division/fraction/division_operation/arith_division
evaluate	
exchange	swap/trade
expand	decompress/spread/space/extend
format	arrange
input	data_entry/scan/enter/read/get
insert	include/push
join	link/connect/bind/concatenate
measure	count/advance/size/enumerate/list
modify	update
move	transfer/copy
open	connect/attach
output	data_output/print/echo/show/write/display/list/put
parse	recognize
pick	extract/select/choose/point_to
search	look_up/find/match/locate
skip	jump
sort	order/rank/arrange
split	separate/break_up/divide/part
start	initialize/define/set
store	save/keep/file/archive
substitute	replace/transliterate/convert/change/map/crypt/update
subtract	reduce/decrement
traverse	travel/move/follow/range

OBJECTS FACET

arguments	
arrays	
backspaces	
blanks	**spaces**
buffers	
characters	**alphanumerics/letters/ASCII/metacharacters/wild**
descriptors	**nodes**
digits	**numbers**
directories	
expressions	
files	
functions	
instructions	**commands**
integers	**numbers/digits/binary/octal/decimal/hex**
lines	**errors**
lists	**linked_lists**
macros	
pages	**page_delimiters/margins**
patterns	**configurations**
pointers	**line_numbers/file_names/addresses/locations/indexes/references/nodes/roots**
procedures	
quotes	
reals	**numbers/floating_point**
statements	**sentences/source_code/assembly_code**
strings	**patterns/groups_of_characters**
subroutines	
tables	**relations**
tabs	**spacers/markers**
text	**messages**
tokens	**semantic_units/symbols**
trees	**graphs/dags/heaps/b_trees**
variables	
words	**names/syllables/keywords/tokens/variables**

MEDIUM FACET

array	
buffer	
cards	
disk	hard_disk/floppy/cartridge
file	
keyboard	
line	
list	linked_list
mouse	
printer	
screen	display/scope
sensor	speech
stack	
table	relation
tape	mag_tape
tree	graph/dag/heap/b_tree

SYSTEM-TYPE FACET

assembler	
code_generation	
code_optimization	
compiler	
data_base_management	
expression_evaluator	
file_handler	
hierarchical_db	
hybrid_db	
interpreter	
lexical_analyzer	recognizer/parser
line_editor	forms_editor
network_db	
operator_precedence	
pattern_matcher	
predictive_parsing	
recursive_descent	
relational_db	
retriever	
scheduler	
screen_editor	forms_editor
semantic_analyzer	parser
shift_reduce	
space_allocator	
syntax_analyzer	parser
text_formatter	

FUNCTIONAL-AREA FACET

accounts_payable	
accounts_receivable	AR/
analysis_structural	
auditing	
batch_job_control	
billing	invoicing/
bookkeeping	
budgeting	
capacity_planning	
computer_aided_design	
cost_accounting	
cost_control	
customer_information	mail_listing/credit_references/
data_base_analysis	
data_base_design	
data_base_management	
data_base_optimization	
data_base_support	
data_dictionary	
decision_support	
design	
design_development	design_methodologies/software_design/
employee_benefits	health_insurance/retirement_plan/participation/
equipment_control	
equipment_maintenance	
file_maintenance	
file_management	
finance_analysis	
financial_modeling	
financial_reporting	
forecasting	
general_ledger	
inventory_control	
labor_control	
management_information	
manufacturing_control	
materials_control	
math_scientific_subroutine_package	
modeling	
numeric_control	
operating_systems	
operations_measurement	
order_entry	pricing/
payroll	
planning	
process_control	
production_control	

```
program_development      tools
project_control
project_scheduling
purchasing
quality_control
report_generation
run_time_job_control
run_time_subroutine_package
simulation
system_control
system_operation
system_development
tax_computation
transaction_analysis
transaction_control
transaction_retrieval
utilization_statistics
work_scheduling
```

SETTING FACET

DP_center	computer_center/DP_services/computer_operations
advertising	marketing/
appliance_repair	
appliance_store	retail/household_equipment/
association	union/club/charity/parish/chamber_of_commerce/council/ judicial/legislative/
auto_repair	body_shop/transmission/
barbershop	hair_stylist/beauty_saloon/
broadcasting	radio/tv/cable/videotex/teletex/station/
cable	broadcasting/programs/distribution/
car_dealer	used_cars/
catalog_sales	retail/mail/
cemetery	graveyard/
circulation	distribution/
classified	advertising/
cleaning	dry_cleaning/
clothing_store	retail/garment/
composition	photo_composition/press/printing/
computer_store	retail/software/peripherals
department_store	retail/general_merchandise/
flower_shop	
foundation	endowment/institution/
furniture_store	retail/office_equipment/
hardware_store	retail/construction_supplies/
hotel	lodging/motel/
library	collection/reference/
mail	post_office/parcel_post/packages/air_mail
mail_advertising	marketing/direct_mail_advertising
nursery	
printshop	printer/printing/press/
publisher	editor/editorial/
radio	broadcasting/station/programs/
social_services	
software_shop	software_engineering/software_production/programming/ software_development/software_labs
spare_parts	auto_parts/
teletex	broadcasting/station/programs/
television	broadcasting/station/programs/
type setter	press/printing/printer/
videotex	broadcasting/station/programs/

Appendix C

A Classification Example

This appendix shows the classification example included in the directions for the user classification experiment of Chapter 7. The program and documentation were taken from Kernigham and Plauger's "Software Tools in Pascal" [KERN83].

Classification Code

function → substitute; *objects* → files; *medium* → file; *system* → file – handler; *functional – area* → program – development; *setting* → software – shop

⟨substitute, files, file, file – handler, program – development, software – shop⟩

Program Listing

```
{ update -- update existing files, add new ones at end }
procedure update (var aname : string; cmd : character);
var
    i : integer;
    afd, tfd : filedesc;
begin
    tfd := mustcreate(archtemp, IOWRITE);
    if (cmd = ord('u')) then begin
        afd := mustopen(anme, IOREAD);
        replace(afd, tfd, ord('u'));   { update existing }
        close(afd)
    end;
    for i := 1 to nfiles do   { add new ones }
        if (fstat[i] = false) then begin
            addfile(fname[i], tdf);
            fstat[i] := true
        end;
    close(tfd);
    if (errcount = 0) then
        fmove(archtemp, aname)
    else
        message('fatal errors - archive not altered');
    remove(archtemp)
end;
```

Documentation

Updating an archive breaks cleanly into two stages: replacing existing members with new versions, and adding to the end any files named as arguments but not present in the archive. We assume that the only way to add data to the end of a file is to copy the existing information to a new file, add the new data to the end of that, then copy the whole thing back to the original. Even though some systems allow you to add at the end or rewrite in the middle of a file, it is unwise to do so. It is safer not to alter an existing archive until you're sure that the replacement is complete and correct.

The process of updating can be summarized as

```
open archive (create if new)
create temporary file
for each new file
    create header and copy it to temporary
    copy file to temporatry
if no errors
    move temporary back to archive
```

These operations are controlled by update.

Appendix D

Sample Library System Entry

This appendix shows a typical library system entry. The one shown here was used as one of the candidates for reusability in the reuse effort estimation experiment of Chapter 7.

Classification Code

search/pointers/tree/table-handler/program-development/software-shop

Reuse Metrics

Size	29
Complexity	9
Explicit Interface	2
Implicit Interface	10
Documentation	8
Language	Pascal

Catalog Description

```
PROGRAM NAME: SearchID- This procedure locates an identifier in the
                        identifier table. Local declarations are
                        searched first. Search is conducted in a tree,
                        and it checks for undeclared entries.
APPLICATION: Systems
  SETTING: Software-shop
  FUNCTIONAL AREA: Language Processor
    SPECIFIC FUNCTION: Compiler
      TARGET LANGUAGE: UCSD Pascal
      COMPILER SUBFUNCTION: Symbol Table Handling
        TASK: Identifier Handling
          GENERIC FUNCTION: search
          METHOD: Binary
LANGUAGE: Pascal
  DIALECT: UCSD
HARDWARE: n/a
DOCUMENTATION: Source- Pemberton and Daniels, "Pascal Implementation:
```

The P4 Compiler", Ellis Horwood Ltd., 1982.

ANALYSIS: n/a

DESIGN: Overall hierarchical structure of compiler on pp. 148-9.

CODE: Code description and explanation on p. 40 of [Pemb82]
Example of compiler output on p. 154.

OPERATION: n/a

CODE LISTING: p. 12 [Pemb82].

Source Listing

```
588  procedure searchid(fidcls: setofids; var fcp: ctp);
599    label 1;
590    var lcp: ctp;
591  begin
592    for disx := top downto 0 do
593      begin lcp := display[disx].fname;
594        while lcp <> nil do
595          if lcp^.name = id then
596             if lcp^.klass in fidcls then goto 1
597             else
598               begin if prterr then error(103);
599                 lcp := lcp^.rlink
600               end
601          else
602            if lcp^.name < id then
603              lcp := lcp^.rlink
604            else lcp := lcp^.llink
605      end;
606    (*search not successful; suppress error message in case
607     of forward referenced type id in pointer type definition
608     --> procedure simpletype*)
609    if prterr then
610      begin error(104);
611        (*to avoid returning nil, reference an entry
612         for an undeclared id of appropriate class
613         --> procedure enterundecl*)
614        if types in fidcls then lcp := utypptr
615        else
616          if vars in fidcls then lcp := uvarptr
617          else
618            if field in fidcls then lcp := ufldptr
619            else
620              if konst in fidcls then lcp := ucstptr
621              else
622                if proc in fidcls then lcp := uprcptr
623                else lcp := ufctptr;
624      end;
625 1: fcp := lcp
626   end (*searchid*);
```

Documentation

Routine *Searchid* lines [588-626]

This procedure is called to locate an identifier in the identifier table.

Pascal's scope rules require that first the local declarations be searched, then in the next surrounding block, and so on outwards. To effect this, *searchid* searches each tree in *display* from *top* down to *0* until the identifier is found, or it is discovered that the identifier was not declared, in which case a special 'undeclared' entry is returned.

592 *Disx* works down through the levels.

593–605 Search the tree at one level.

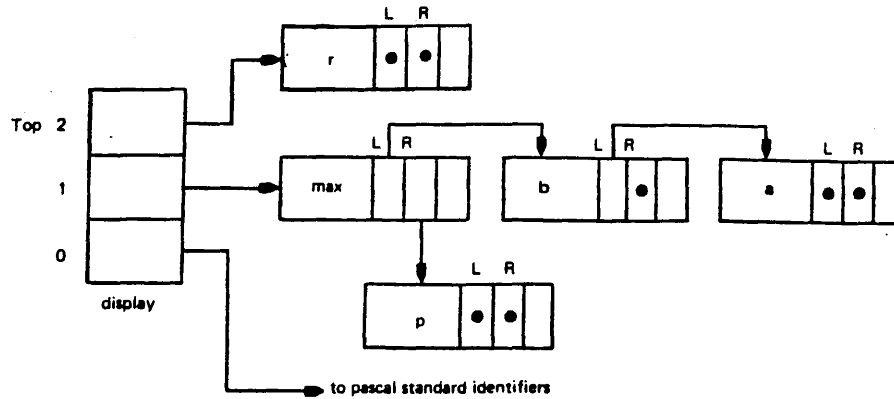
596–604 *Fidcl* is a set of *idclass* [137] representing the class of identifier acceptable, for example, variable, type, etc. At this point an identifier with the required name has been found, If it is of a suitable class, them this is the required identifier. Otherwise error 103 is reported, and searching continues. *Prterror* inhibits the error message during declarations, when a pointer type may be forward declared.

609 If the identifier was found, **goto** *1* [596] would have been executed. Thus if the loop terminates normally, the identifier was not found, and so error 104 is issued, and a pointer to a special undeclared entry is returned (these are initialized earlier). In this way the caller of *searchid* can be sure that the result is non–nil, and of an acceptable class.

IDENTIFIERS: At each level of declaration a tree is formed of the identifiers declared there. The first identifier declared at any level is pointed to by the field *fname* of the relevant element of the array *display*. The *rlink* and *llink* fields of type *identifier* then point to lower regions of this tree, *rlink* pointing to identifiers with alphabetically later names, *llink* for earlier names. For example with

```
program eg;
  const max=100;
  var b,a: real;
  procedure p;
     var r:real;
  begin . . . end;
begin . . . end.
```

while the body of procedure *p* is being compiled, the identifier tree looks like this:



(The variable *top* always points to the current top element of the display).

These trees are created by the procedure ***enterid***, which is called while compiling declarations each time a new identifier is declared. Then every time an identifier is used within the program being compiled, the trees are searched using the procedure ***searchid*** [588–626], and occasionally by ***searchsection*** [575–86].

Searchid works by searching the tree at each level of the display until the identifier is found, or until the whole structure has been searched without finding it.