1

OF/DE

1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

CASE-BASED SOFTWARE REUSE
THROUGH
COMMENT ANALYSIS

by

AFAQ AHMAD, MS., M ENG., PH. D

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Science in
Information and Systems Science

School of Computer Science

Carleton University

Ottawa, Ontario

November 5, 1993

copyright

1993, Afaq Ahmad

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-13864-X

Name

AFFA AHMAD

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

0984

U-M-I

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0591
African 0305
American 0352
Asian 0355
Canadian (English) 0593
Canadian (French) 0311
English 0312
Germanic 0315
Latin American 0313
Middle Eastern 0314
Romance 0579
Slavic and East European 0581

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0777
Accounting 0454
Banking 0338
Management 0285
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0378
Geography 0378
Gerontology 0378
History 0378
General 0378

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and Relations 0617
Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
General 0938
Criminology and Penology 0631
Demography 0628
Ethnic and Racial Studies 0629
Individual and Family Studies 0630
Industrial and Labor Relations 0700
Public and Social Welfare 0344
Social Structure and Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Biology 0306
General 0287
Anatomy 0308
Biostatistics 0309
Botany 0379
Cell 0329
Ecology 0353
Entomology 0369
Genetics 0793
Limnology 0410
Microbiology 0307
Molecular 0317
Neuroscience 0416
Oceanography 0433
Physiology 0821
Radiation 0778
Veterinary Science 0472
Zoology 0786
Biophysics 0760
General 0425
Medical 0996
Earth Sciences 0425
Biogeochemistry 0996
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0427
Palynology 0368
Physical Geography 0415
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0748
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Human Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry 0485
General 0749
Agricultural 0486
Analytical 0487
Biochemistry 0488
Inorganic 0738
Nuclear 0490
Organic 0491
Pharmaceutical 0494
Physical 0495
Polymer 0754
Radiation 0405
Mathematics 0605
Physics 0986
General 0606
Acoustics 0608
Astronomy and Astrophysics 0748
Atmospheric Science 0607
Atomic 0798
Electronics and Electricity 0759
Elementary Particles and High Energy 0609
Fluid and Plasma 0610
Molecular 0752
Nuclear 0756
Optics 0611
Radiation 0463
Solid State 0346
Statistics 0984
Applied Sciences 0346
Applied Mechanics 0984
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned recommend to the Faculty of Graduate Studies and
Research acceptance of the thesis

"CASE-BASED SOFTWARE REUSE THROUGH COMMENT
ANALYSIS"

submitted by Afaq Ahmad, MS. (Southern Illinois), M Eng. (Carleton),
Ph. D (City University of New York)

in partial fulfilment of
the requirements for the degree of
Master of Science in
Information and Systems Science

Professor Stan Matwin,
Joint Supervisor

Professor Franz Oppacher,
Joint Supervisor

Professor John Pugh,
Director, School of Computer Science

Carleton University

November 23, 1993

ABSTRACT

Ruben Prieto-Diaz [1989] proposed a simple scheme for software indexing and reuse based on the faceted classification system used in library science. Recently Fouque and Matwin [1992] pointed out the similarity between case base technology and compositional software reuse. They developed a case-based reuse method that extracts knowledge about software from analysis of its code. The present work extends this line of inquiry by combining techniques from information retrieval and CBR and applying these to the reuse problem.

It employs a domain-dependent description of software components developed from the noun phrases appearing in programmer's comments. We start with comments taken from a domain-specific library of well-documented software (LINPACK). Noun phrases extracted from LINPACK comments form the basis of simple domain models for linear systems. The process of constructing a reuse system is broken into three steps. A file containing comments from all LINPACK routines is processed by the AZTEC text analysis package developed at University of Ottawa. This program is invoked mainly to yield a list of technical noun phrases. The second step involves building domain models based on an analysis of these technical phrases and then indexing cases according to these models. Finally, tools provided by the REMIND CBR shell are used to create a case library incorporating this domain knowledge.

Results support three conclusions. Noun phrases extracted by automated text analysis systems can provide the building blocks for domain modeling. The resulting simple domain models are usually sufficient for software reuse applications. And finally, standard CBR technology is a viable means of constructing compositional software reuse libraries.

Table of Contents

1. INTRODUCTION	5
Motivation.....	5
Historical Background.....	6
Types of Reuse.....	7
Building a Reuse Library.....	8
Steps in the Reuse Process.....	9
Prieto-Diaz Reuse Model	10
Figure 1: Prieto-Diaz Reuse Algorithm.....	10
Prieto-Diaz Software Classification Scheme.....	11
The Conceptual Distance Graph.....	12
Prieto-Diaz Implementation.....	13
Case-Based Reasoning	14
Nearest-Neighbor Matching	15
 2. INDEXING, CLASSIFICATION AND DOMAIN MODELING	 16
Indexing and Organization of Knowledge for CBR	16
The Information Retrieval Problem	16
Domain Modeling	17
Change of Representation	18
Term Independence.....	19
Relevance	19
Weights	20
Clustering	20
Term Classification.....	21
 3. DESIGN OF THE EXPERIMENT	 22
Comment Analysis	23
Text Analysis Using AZTEC.....	24
Input Text Preparation.....	26
Cleaning up and Modifying the List	28
Expert Verification.....	29
Analysis of the Noun Phrase List.....	29
Construction of A Reuse Library.....	30
Mapping NPs to Individual LINPACK Subroutines.....	31
Field Selection	31
The Symbol Tree	32
REMIND Data Import Facility.....	36
Creation of Facets	37
Types of Retrieval and Importances.....	37
Testing and Calibration	38

Evaluation Criteria.....	38
The Retrieval Evaluation Experiment.....	38
LINPACK Classification Scheme and the Relevance Table.....	39
Single Level Tree vs. Keyword Matches.....	42
Two Level Symbol Tree and Faceted Classification.....	42
Multi- Level Tree and Conceptual Distance Graph.....	42
Browsing and One Stage retrieval.....	42
Other Implementation Issues.....	43
Reusability Metric.....	44
Two-stage retrieval.....	45
Case-base Management.....	46
New Case Additions.....	46
Synonyms.....	47
4. ANALYSIS OF RESULTS	48
Results of AZTEC.....	48
AZTEC and Manual Lists.....	48
Using AZTEC Technicality Ratings.....	49
Main Results.....	51
Critique of AZTEC as A Reuse Tool.....	51
Evaluation of the Software Reuse Library.....	52
Major Retrieval Results.....	52
Additional Experiments.....	55
Comparison of CBR and Prieto-Diaz Approaches.....	57
5. CONCLUSIONS	58
Major Results.....	58
Additional Work.....	60
Identifying Well Commented Text.....	60
Storage for Efficient Retrieval.....	60
Non-Homogeneous Collections.....	61
Improved Text Analyzer.....	61
Attribute Analysis.....	61
Normalization of Similarity Function.....	62
Other Measures of Effectiveness.....	62
Expanded Text Mining.....	62
Combining Code and Comment Analysis.....	62
Enlarged Case Base.....	62
Related Work:.....	63
REFERENCES	66

APPENDIX I.....	70
AZTEC	70
AZTEC Objectives	70
The Text Base Builder.....	70
Extract_Text_Sentences Module	72
Sentence Parser	72
Syntactic Category Analyzer within AZTEC	73
Classification Module.....	73
 APPENDIX II	 76
REMIND.....	76
REMIND Datatypes	76
Symbol Tree.....	76
Remind's Matching Algorithm.....	77
Qualitative Modeling.....	77
Clustering.....	78
 APPENDIX III.....	 79
Matrix Algebra and Linear Systems.....	79
Libraries of Linear System Software and their Organization.....	79
LINPACK Subroutine Library.....	80
Historical Background	80
Comment Standards	80
Subroutine Classification Scheme	81
 LIST OF FIGURES.....	 84
 LIST OF TABLES.....	 85
 GLOSSARY OF TERMS	 86

1. INTRODUCTION

Almost as soon as they first start to write code, programmers have the objective of reusing software. Although this intention is initially limited to their own code it soon broadens to include the code of others. Good programming shops promote the practice by establishing subroutine and function libraries and encouraging their use by all programming staff. Notwithstanding how common this situation is, there is no generally acceptable definition of reuse and many researchers define it slightly differently. We follow Krueger [1992] in defining it as:

"... (the) process of creating software systems from existing software rather than building software systems from scratch".

The definition covers the use of program generators as well as the modification of existing software packages to produce routines which either stand alone or are integrated into the user's own source code. In contrast to Wegner's [1989] more liberal definition, it excludes operating systems as an example of reuse—Wegner considers UNIX to be one of the best examples of a successful reuse system. While most definitions apply the concept of reuse to artifacts produced in all phases of the system life cycle, the present work is focused on the reuse of code alone.

Motivation

The basic motivation for reusing software is to increase programmer productivity [Horowitz and Munson, 1989]. Additional financial benefits will occur if the return on an investment in software can be maximized by amortizing it over multiple uses. Computer programming is one of the few corporate assets that does not lose value or deteriorate with use; its performance is quite independent of number of times its code is reused. Recognizing this, a number of studies have identified opportunities for reuse. Frank [1981], for example, has shown that a large percentage of code is repeated in various programs within an organization. Given the high cost of developing software, it is only natural to look to at exploiting this commonalty of code by reusing software. Horowitz and Munson [1989] have argued similarly that the potential for increasing productivity by reuse is far greater than by any other means, including the use of higher level languages.

However, before undertaking a decision to reuse, its benefits must be weighed against the cost of the effort needed to accomplish it. *The basic problem to make reusing code significantly less resource intensive than writing it anew.*

The calculation is not straightforward. Even if marginal time savings are predicted from reuse, social and psychological factors may tip the balance in favor of writing code from scratch. To add to the difficulty in making a decision, there is no clear way to measure the effort involved. Although we would really like to quantify what is needed to modify code to specification, often the best we can do is to rank various candidates in relative terms. Finally we must also consider the substantial initial resources needed to start a reuse program up. This effort must be amortized over the potential life span of the reuse system. A reuse program should thus be seen as an opportunistic activity in the process of software development. This places the emphasis solely on any life cycle savings achievable through reuse.

Historical Background

Current reuse activities are only the most recent manifestation of a movement to optimize the software development process which started long ago. McIlroy [1967] was among the first to propose reuse as a means of overcoming the software development bottleneck. He suggested a manufacturing approach to software production. In his method of reuse one constructs programs by selecting appropriate components from a software 'parts catalogue'. This approach was facilitated by higher level languages like FORTRAN and C that support modular programming via subroutines and functions. McIlroy's concept was first employed to build subroutine or function libraries of widely-used mathematical algorithms in fields such as linear algebra, statistics, numerical analysis and finite mathematics. Although in some cases the libraries allowed programmers, scientists and engineers to reuse the source code exactly as is, more often than not adaptation was necessary to code slight variations of the algorithms and to make I/O statements conform to the local installation.

In parallel to these developments, object oriented programming systems (OOPS) emerged that have reuse of code and design as a basic goal. The advent of viable OOPS programming environments has resulted in the availability of a number of libraries mainly in Smalltalk and C++. OOPS promote the reuse both of complex data structures (objects), as well as valid operations on them (methods). This extends the concept of reuse to include non-executable parts of a program. In a related but separate development, the US Department of Defense funded the development of the ADA programming language with reusability as its major design goal. Booch [1987] has developed an ADA programmer's tool kit, thereby carrying the 'parts catalog' paradigm of McIlroy [1967] into the present.

Types of Reuse

Within the scope of his definition Krueger [1992] identified eight different methodologies that have been tried for software reuse. These approaches range from unstructured and informal 'design and code scavenging' to highly structured transformational systems and software architectures. Biggerstaff and Richter [1989] classify the different types into *compositional reuse* and *generational reuse*. Although the boundaries between the two categories are not sharp, we can contrast them as follows:

COMPOSITIONAL	GENERATIONAL
Building Blocks	Patterns
Atomic / Immutable / Passive	Diffuse / Malleable / Active
Component Libraries	Languages or Transformer

These criteria divide the eight methodologies of Krueger as follows:

COMPOSITIONAL	GENERATIONAL
Design and Code Scavenging	Software Schemas
Source Code Components	Application Generators
Software Architecture's	Transformational Systems
	Very High Level Languages
	High Level Languages

In the context established by the above descriptions, the present work deals with compositional reuse, and more specifically, with the reuse of source code components. It involves the construction of an indexed library of reusable components along with a retrieval facility. Given the specification of a software component in some retrieval language, our goal is to identify candidates for reuse in the library. The retrieved candidates should be ordered by the effort needed conform to the specification.

Building a Reuse Library

The first step in setting up a reuse library is to describe the given set of routines in a suitable language. Ideally the language used for software description or indexing should be the same as the user's retrieval language. It should be simple to learn and easily understood by the user. The basic syntax should be domain independent so the language is usable by all programmers. For example a software component could be represented by the familiar 3-tuple:

(input, process, output)

Prieto-Diaz [1989] applied a similar domain-independent top level description to software components. This top level description hides the hierarchies of input and output objects and the relationships between processes from the user. Prieto-Diaz assumed that users do not need the taxonomic and relational details of the domain model. These details are however accessible to the domain engineer who initially designs and subsequently modifies the domain model underlying the system. In addition to domain engineering, a librarian function is also needed to control additions and deletions of the library items. To accomplish these tasks we need specialized tools and facilities. The two most important are:

1. a storage and retrieval facility for an indexed set of software components,
2. tools to build the domain model.

The first functionality is very similar to that provided by a database system and one might look to database technology to provide it. However database systems cannot easily meet the second requirement, which must then be satisfied by tools external to the database [Prieto-Diaz, 1989]. Moreover database retrieval is exact and deterministic, and is likely to result in zero hits when applied to software reuse. One must then resort to external means to expand the original query and get output from the system. This makes the process cumbersome. Fouque and Matwin [1993] considered Case-Based Reasoning (CBR) as an alternative storage and retrieval facility for reuse applications. They found that CBR can provide the functions needed for compositional reuse within a single framework.

Describing the characteristics of reuse components in a specification language is a precondition for building a case library. This important indexing step will be dealt with in detail in the next Chapter. For the moment we will discuss implementation issues.

Steps in the Reuse Process

Once the necessary tools are at hand, the steps necessary to set up a reuse library can be taken. These steps are:

- 1) *the domain engineer builds a model of the domain,*
- 2) *the librarian adds cases to this framework.*

In an optional final step,

- 3) *the librarian and the domain expert collaborate to organize the cases in a domain hierarchy to make retrieval easy.*

Programmers can then start retrieving components that match their specifications. Employing the reuse process involves its own series of steps [Biggerstaff and Richter, 1989]. The following table enumerates these steps and lists the tools or features necessary to implement them in a reuse system:

REUSE STEP	TOOLS NEEDED
finding components	matching function, storage scheme, indexing/query language and tools
understanding components	browsers, code access
modifying components	interface to full programming environment
composing components	interface to full programming environment, testing tools

The first step retrieves potential reuse candidates in order of their closeness to a specification. The reuse system provides facilities for browsing through this list to select a case for closer examination. Text editing tools let the user inspect code to confirm the order suggested by the system. The outcome of this code review is selection of one or more candidates that will require the least amount of effort to reuse. As noted above, most candidates need some modification to meet the

specification. This re-compiling and debugging cycle must be done within a full programming environment. The last step in the table is needed since a component usually satisfies only part of the overall requirement. The user must then integrate the retrieved software fragment with other code to compose the total application.

When no component in the library addresses the specification sufficiently to be retrieved a modular matching process can be tried. This involves dividing the original requirement into logical modules and attempting to match each one independently. The retrieved sub-components are then combined to assemble the final component specification.

Prieto-Diaz Reuse Model

Prieto-Diaz' [1985] early effort in constructing a compositional reuse system applied a methodology to a set of simple system commands. His reuse philosophy (described in Figure 1) can be seen to be quite similar to that of Biggerstaff and Richter [1989] discussed in the pervious Section. The Prieto-Diaz compositional reuse algorithm is quite general and adaptable to many different environments. In addition to developing this algorithm, Prieto-Diaz designed a new classification scheme and a domain model for the reuse systems.

```
Given a Functional Specification For a Desired Component
Begin
  Search Library
  If Identical_match Then Terminate
  Else
    Collect Similar Components
    For Each Component
      Compute Degree_of_match
    End
    Rank And Select Best
  Fi
  Modify Component
End
```

Figure 1: Prieto-Diaz Reuse Algorithm

Prieto-Diaz Software Classification Scheme

Standard software libraries employ well-established classification practices. Libraries such as SPSS [1988], IMSL [1987] and the IBM Software Catalog need classification schemes that allow easy access to software identified by application type [Prieto-Diaz, 1989]. For the most part their classification schemes were enumerated types borrowed from library science. These schemes employ fixed, pre-defined classes that cannot easily be changed without rebuilding the library. The Dewey Decimal and Library of Congress systems work well for applications where classes are essentially fixed. This condition is met by book collections and to a lesser extent by commercially available software libraries, but not necessarily by software reuse libraries.

Searching for more flexibility, Prieto-Diaz found a classification system popular in libraries in India and Europe that met the needs of reuse much more fully. The *faceted* classification proposed by Ranganathan [1967] differs from enumerated systems in one essential way. It allows classification codes to be built up from a list of facets analogous to relational tuples. Prieto-Diaz applied these tuples to the description of software components:

<function, object, medium>

where the three 'process' facets, function, object and medium, can take on various values from a controlled list of terms. For example,

<input, character, buffer> and <search, root, B-tree>

might refer to routines involved in buffered input and B-tree search respectively. Prieto-Diaz went on to augment the component description by introducing three 'environment' facets producing this descriptor:

PROCESS FACETS | ENVIRONMENT FACETS

< input, object , medium , system_type , functional_area , setting >

System_type refers to broad classes of systems such as assemblers and matrix invertors, while 'functional_area' and 'setting' might refer to the department and industry respectively. An example may clarify the concept:

< delete, directory, mouse, *editor*, *programming*, *publishing* >

represents a mouse-based directory deletion routine used in publishing industry programming departments. The user specifies facets he wants by selecting terms from lists, so limiting the size of these lists is essential. To accomplish this Prieto-Diaz controlled the vocabulary of facet terms and augmented it with a thesaurus. The user can also determine the (citation) order in which the search considers facets to ensure that it is directed to the to the more important facets first.

The Conceptual Distance Graph

The Conceptual Distance Graph is a central idea of the Prieto-Diaz scheme. This graph is needed the faceted representation is implemented as a relational database application and cannot retrieve cases which match the queries partially. Such a situation often results in no hits, requiring the query to be expanded. The traditional way to expand a query in information retrieval is to generalize one or more of its facets. This results in more cases being retrieved; unfortunately often many are non-relevant. The Conceptual Distance Graph allows the search to be expanded to related terms without resorting to query generalization. Figure II shows how such a graph can describe relationships between terms for the function facet. Various terms are related to *super types* or *notions* which describe the concepts behind the terms in a general way by a measure or weight. During retrieval the query is modified according to the Conceptual Distance Graph to access those components most closely matching the requirement. The graph should be viewed as an essential part of the domain model, in this case the domain of computer programming.

Figure II: Conceptual Distance Graph

Figure III: Prieto-Diaz Implementation

Prieto-Diaz Implementation

Figure III displays various components Prieto-Diaz used to implement a reuse library. The components are stored in a relational database format. Searching first proceeds by looking for components which match the *classification schedule* exactly. This schedule consists of the terms allowed to describe the specification;

it is equivalent to a controlled vocabulary in information retrieval. The user employs the schedule and a thesaurus to construct the initial query interactively. If an exact match, is not found the Conceptual Distance Graph is invoked to modify the original query externally. This modification can be a semi-automatic process where the system suggests related terms to the user as possible candidates. After searching has concluded, the system ranks retrieved components on the basis of the effort expected to reuse them. Although first implemented as an academic project [Prieto-Diaz, 1985], the methodology was subsequently used in an industrial setting where a large library of telecommunication software was built. An evaluation of the cost and benefits of this application showed significant productivity gains [Prieto-Diaz, 1989].

Case-Based Reasoning

Prieto-Diaz's application of relational database technology to build a reuse library met with success. Recent work of Fouque and Matwin [1992] suggests that CBR technology offers an even better platform for implementation of reuse libraries. They pointed out the similarity between compositional software reuse and certain areas of CBR. The authors went on to exploit this similarity to construct a software reuse system [Fouque and Matwin, 1993]. The current work applies a similar CBR-based approach to the reuse problem using a CBR shell called REMIND from Cognitive Systems, Cambridge, MA. Details of the functionality of REMIND appear in Appendix II. In what follows we will describe how a case base allows storage and retrieval of software components.

The concept of a case base is rather simple. It is a library or database of cases indexed on descriptors of their domain. The descriptor attributes or fields can be any structure (including other cases) which can be used to define a normalized similarity measure $\text{Sim}(i, j)$ between two cases i and j with the following properties;

$$\text{Sim}(i, i) = 1, \quad 1 \geq \text{Sim}(i, j) = \text{Sim}(j, i) \geq 0$$

We can then express $\text{Sim}(i, j)$ as:

$$\text{Sim}(i, j) = \text{sum}\{C(k, v(k, i), v(k, j)) * w(k)\} \quad (1)$$

where the sum extends over all k features and $w(k)$ is the weight assigned to the k th feature. $v(k, i)$ is the value of the k th attribute of the i th case. The function $C(k, x, y)$ then measures closeness between two values x and y of the k th attribute: C must be defined for all possible values of the attributes. In

real life some values will be unknown and the closeness measure must reflect this. Although some flexibility exists, c_i is mostly determined by the types of the attribute data. (1) reveals that we compute similarity $Sim(i, j)$ by simply adding up the weighted closeness values of individual fields [Cain et al 1991]. Retrieval from a case base then involves selecting that subset of cases that are close to the input case as determined by this measure.

Nearest-Neighbor Matching

Nearest-neighbor matching functions employ a distance measure akin to the similarity function. These are used to match the attributes of the input case to those of all cases in the library. The nearest-neighbor of a case, $near(n)$, can be defined as the component j which maximizes the similarity function:

$$Sim(n, near(n)) = \max \{ Sim(n, j) \} \quad (2)$$

over all values of j except $j=n$.

The k th nearest-neighbor of n can be similarly defined. A k th nearest-neighbor algorithm retrieves the k cases closest to the input specification. In practice a threshold T is applied below which the similarity value is deemed unreliable due to noise. Various matching functions have been proposed in the literature and search for an optimal nearest-neighbor function is an active area of research [Dasarathy, 1991].

Once a similarity function has been settled on we can proceed to the actual retrieval itself. In CBR retrieval is a four step process [Fouque & Matwin, 1993]:

1. Specify Input Case
2. Match Cases from the Case Base
3. Select the Best Case(s)
4. Adapt to the Current Problem

This work describes experimental results for the first two steps with a limited treatment of the third. For this step we describe a methodology to rank and evaluate the candidates retrieved. We hypothesize that the flexibility of CBR makes it a suitable technology for setting up a reuse library.

2. INDEXING, CLASSIFICATION AND DOMAIN MODELING

Indexing and Organization of Knowledge for CBR

In order to store software components in a CBR we must develop an indexing scheme appropriate to describe them. In this Chapter we review indexing and classification issues in the context of CBR. In the literature these discussions often revolve around the selection of a language to describe software. Although many interesting issues are related to this task, we start by looking at different language requirements for different reuse functions. A language is needed for at least two distinct reasons:

- a) description of components to be added to the library, and
- b) description of the sought component.

We prefer to use a single, easy to learn, and fairly precise language for both indexing and query. Formal specification languages like SETL can describe software unambiguously. They are usually part of a generational reuse system [Krueger, 1992]. These very high level languages satisfy both of the above purposes but are not commonly known or widely available to practicing programmers. At the other extreme, natural language interfaces allow anyone to get a desired component from a reuse library without knowing much about the library. Although a natural language interface is useful, it may not be necessary. Most programmers want to specify a query more quickly than a natural language interface allows. Natural language is also inherently ambiguous due to the freedom of expression it allows. Fortunately, the information retrieval community has developed a compromise suited to describing user queries for document searches [van Rijsbergen, 1979]. They found that a controlled vocabulary specification facility with a built-in thesaurus provides the user with sufficient power and flexibility. Prieto-Diaz has applied this idea to the reuse problem by modifying it to include a domain model.

The Information Retrieval Problem

To profit from this technique we must understand the type of issues it deals with. The basic problem involves selecting the minimum vocabulary sufficient to describe all the documents in a given collection. Terms for the vocabulary are typically extracted from a sample of the collection in a process called *literary warrant* [van Rijsbergen 1979]. The user then augments this list with terms likely to occur in retrieval but not in the sample in a *user warrant* process. While there

are obvious similarities between documents and reuse components [Fraser and Duran, 1989] information retrieval systems do not usually encode domain knowledge. This is understandable because most retrieval systems are designed for applications spanning more than one domain and therefore cannot contain domain specific knowledge. For use in the reuse problem the basic information retrieval techniques need to be enhanced to incorporate domain knowledge [Prieto-Diaz 1989].

Domain Modeling

Describing software components in terms of keywords derived automatically from processing text is only the first step in setting up a reuse library. This knowledge must be organized in a way suitable to a case base. A domain model must be constructed from the concepts identified by these keywords and cases then indexed according to that model. Figure IV depicts some of the steps involved in domain modeling.

Figure IV: Domain modeling Methodology

Change of Representation

The first phase in modeling a domain is to change the representation. This operation typically involves deriving a new set of descriptor terms from the original set. It often results in more concise descriptions of cases. Simple examples are combining singular and plural forms, or generally merging terms that have the same stem.

Term Independence

This phase addresses the problem of picking out terms that are truly independent. This is done by identifying those that co-occur with others in some combination. Any attribute which always or nearly always co-occurs with certain other attributes can be dropped without much loss of information. This issue has been studied by many researchers, for instance Russell [1989] in the context of the Theory of Determination, and is known to be NP-complete. All known solution procedures are exponentially complex, and inducing determinations therefore becomes unwieldy except for domains with few attributes. However Schlimmer [1992] recently computed meaningful determinations quite efficiently by constraining the process with heuristics. He restricted the number of attribute combinations to 2 or 3 and employed some simple pruning rules.

Relevance

Many authors have discussed the importance of choosing proper descriptive terms [van Rijsbergen, 1979]. Including irrelevant ones introduce noise, slows down the retrieval process, and can lead to completely wrong results. Excluding relevant ones can produce equally poor results. A term list derived automatically or by indexers unfamiliar with the domain typically contains many irrelevant terms. Given such a list, the goal must be to reduce it to one containing only relevant terms. Often the librarian or indexer does this with little expert knowledge. An alternative is to design an automatic *relevance filter* to exclude attributes that are not meaningful.

Much work has been done [Almuallim and Dietterich, 1992; Salton, 1975] to develop automated tools for determining relevance. Automated selection of relevant indices in machine learning usually occurs in the context of supervised learners [Pazzani, 1990; Aha, 1989]. These methods require that an outcome field be specified against which the utility of various attributes in prediction can be tested. Software reuse provides no obvious outcome field; hence predictability criteria cannot be derived. Other measures to identify relevant attributes must be found. One possibility is to measure how much each feature contributes to distinguishing individual cases in the instance space. This can be done by removing an attribute from the description and measuring the effect on the average similarity between various cases [Salton 1975]. This can be formalized as the difference $D(i)$ in average similarity S before and S' after the removal of an attribute i :

$$D(i) = S - S'$$

All features i with positive $D(i)$ increase the average similarity between various members of the collection and therefore are not relevant.

Weights

Once irrelevant attributes have been removed, weights for the remaining terms can be specified. A domain expert can usually do this reasonably well in strong theory domains. Other techniques to estimate weights are based upon Explanation Based Learning [Cain et al 1991], probabilistic reasoning [Cost and Salzberg 1990] and Information Theory [Aha and Goldstone, 1990]. Most of these are only suitable in a supervised learning environment. Stanfil and Waltz [1986] applied a memory based reasoning technique for weighting in a speech synthesis task. Their method is based on nearest-neighbor matching and has worked well compared to neural networks on a variety of learning tasks. Although Stanfil and Waltz [1986] first proposed their method for supervised learning, it is easily extended to unsupervised learning [Aha 1993]. It also works in estimating weights for attributes with structured values such as lists.

A number of automated weighting methods popular in information retrieval [van Rijsbergen, 1979] are also suitable to the software reuse problem. A simple one based on *inverse document frequency measure* defines weight $w(i)$ of the i th term as :

$$w(i) = \log(N/n(i)) + 1 \quad (3)$$

where N is the total number of cases and $n(i)$ is the number of cases in which the i th term appears. The measure was proposed by Sparck Jones [1972], who showed that it significantly increases retrieval performance.

Clustering

In a flat case base, cases are stored under the root node and search involves matching serially against all of them. This may be acceptable if the library is small, but efficiency becomes increasingly important as the case base grows. However serial matching is only necessary if absolute certainty is needed about the best match. Because search time increases with N , we need an efficient storage and retrieval scheme for large case libraries.

Supervised learning environments use classification trees whose leaves store cases. These trees are normally built by inducing outcome field values using an algorithm such as ID3 (see Appendix III). During retrieval, induction trees classify input cases whose outcome values are unknown. Once a case's class is determined similar cases can be retrieved by looking just at the branch for that class. This inductive retrieval method is not immediately suitable for reuse libraries which lack target fields. However Arrango [1989] showed that a conceptual clustering algorithm such as COBWEB can be used to build a hierarchy of clusters of cases. Within a case library these clusters can be represented by *prototypes* or class representatives. Restricting case comparison to prototypes alone replaces serial search with tree search and speeds up component retrieval significantly.

A side benefit of clustering schemes is to reduce case base storage requirements by storing only the differences between a prototype and each case it represents instead of the whole case.

Term Classification

The next step in domain modeling is to identify any natural groupings within the attributes, an activity similar to Prieto-Diaz's definition of facets. Facets ease query specification, improve retrieval, and provide a basis for domain modeling. A single list covering all the terms needed to specify components in any nontrivial domain is usually greater than 100. Such a large list is not easy to browse when specifying a query. Logical groupings that divide it into sub-lists make browsing and selection more manageable.

Term groupings also serve as a starting point for domain analysis. However knowledge of more than facets is necessary to build a domain model. For example term classification can be carried to deeper levels to produce a taxonomy tree. Terms may also have associative relations within or outside their attribute groups. Although there have been efforts to automatically induce term groupings [Salton, 1975; Sparck Jones, 1971, Srinivasan 1993], this knowledge intensive final phase of domain modeling is normally performed by experts. A domain model can only be elaborated to the degree supported by the tools available to operate on it. The last step in Figure IV thus produces a domain model incorporating only as much available expert knowledge as the computational environment allows.

The model in Figure IV represents the generalized conversion of a simple term description to a domain representation suitable for CBR. Its steps are not all necessary or even possible in every instance. Individual steps are also not totally independent of each other—keyword relevance and weights are obviously related.

A weight can be viewed as spanning the continuum from 0 to 1, from not relevant to most relevant. Attribute relevance and independence are similarly related. One can argue that any attribute determined by others will also be identified as irrelevant in the attribute removal operation discussed in the previous Chapter [Salton, 1975].

3. DESIGN OF THE EXPERIMENT

A major premise of this work is that useful information can be extracted from the comments in a well-documented collection of software routines. An automated noun phrase extraction program will produce a list of descriptive phrases when applied to the comment text of such a collection. These phrases must then be processed as described in Chapter 2 to construct a domain model for the reuse library. Only then can the construction of the actual software library be begun. Therefore we formally divide our experiment into these three distinct parts:

1. Extraction of noun phrase list from the text of the comments.
2. Analysis of the list to assess weights and determine relationships, term groupings and taxonomy of the domain.
3. Design, implement and evaluate a software library from these terms.

Figure V displays the overall architecture of the system that executes the above process. The remainder of this Chapter will provide details of different parts of the system and identify their relationship to the three steps above.

Figure V: Semi-Automated Creation Of The Case Base

Comment Analysis

Good programmers provide algorithm and computing details in *boxed* and *inline* comments. The preamble or boxed comment contains general information about the program or routine it heads, such as its algorithm, variable names and descriptions, a list of calling and called routines, and tombstone data: the author's name, implementation and last change dates etc.. Inline comments explain steps in computation and are useful indicators of algorithmic detail. Although guidelines vary considerably, this practice is followed by most large programming shops.

There is however no standard for what constitutes well-documented or well-commented software. Applying these labels to a program remains a subjective determination. At a minimum, a well-documented program provides sufficient implementation details to allow its maintenance by programmers unfamiliar with it. The recent ISO 9000 initiative will in time hopefully establish documentation standards. Notwithstanding the current lack of standards, a number of widely-distributed source code libraries are good examples of well-commented software. The experiment uses a popular library of linear algebra routines: the C version of LINPACK [Dongarra et al, 1979]. Details are given in Appendix III. Comments were extracted from a set of single precision LINPACK subroutines and prepared for input into an automated text analyzer.

Text Analysis Using AZTEC

Automated text analysis is an active field of research [Salton, 1988; Ellis, 1990; Laurendeau, 1992]. Two distinct communities are involved in it. AI researchers wish to understand natural language, while the Information Retrieval community has an interest in developing proper descriptions for collections of documents. Comment text is not well composed and thus is not suitable for natural language understanding. In any case, semantic details can be better derived from an automated analysis of the code itself [Matwin and Fouque 1992]. This study applies text processing techniques to the more modest task of extracting technical noun phrases from comments. Our conjecture is that these terms describe the software accurately enough to allow its retrieval from a reuse library.

Noun phrase extraction was performed by the AZTEC syntactic text analysis package developed at University of Ottawa (Appendix I). As depicted in Figure VI, AZTEC performs a limited analysis of the text and identifies all of its technical noun phrases [Laurendeau, 1992]. AZTEC produces a list of noun phrases paired with a rating of their technicality in terms of the following ordered scale:

Technical > Rather Technical > Possibly Technical > Not Technical

Figure VI: AZTEC Algorithm

The boundaries between ratings on the scale are usually established by a domain expert and are not rigid. Terms which AZTEC cannot classify are flagged as Unknown.

- Comment text was previously described as not well composed. In particular,
- it does not conform to any strict grammatical standard;
 - it may be composed of incomplete sentences;
 - it may include variable and subroutine identifiers;
 - it may incorporate small explanatory fragments of code itself;

– it contains many generic computing terms.

To overcome these irregularities preprocessing and post processing functions were added to AZTEC to adapt it to analyze comment text. For instance, generic computing terms must be filtered out to identify domain-specific descriptors.

Most of these problems are not peculiar to AZTEC but would be expected whenever automated indexing methods are applied to text of poor quality. They leave unclear whether or not such methods are suited to poorly formed text. On the other hand, manual methods are tedious, costly, subjective and therefore prone to errors. They also require the indexer to have subject matter expertise. Surprisingly, speed is less of a consideration because current text analysis packages are quite slow, while manual identification of technical phrases without understanding can be quite fast.

One weakness peculiar to AZTEC is an inability to record when a term occurs in a negative context. Although this information is available from the parser, it has not been utilized by AZTEC because the question is not likely to be a major concern in technical text.

Input Text Preparation

Because AZTEC was not designed to accommodate the peculiarities of comments, substantial effort is required to prepare this sort of text for input to it. Figure VII lists the various steps necessary to convert the raw comments embedded in it to a form acceptable to AZTEC.

Figure VII: Pre-Processing Input For AZTEC

These six steps are performed through a combination of UNIX commands, shell scripts and vi editing functions. The first step is to extract comments from the source by recognizing their delimiters (`/*` and `*/` for C code). This is followed by visually checking the extracted text and deleting non-text areas like embedded code fragments. At this point tombstone data such as author name, date of last revision and called and calling routines could also be separated out for its own processing. Special characters not recognized by AZTEC's Text Builder module which could terminate processing must then be filtered out.

AZTEC's extraction of technical terms includes variable and subroutine names. If we include these in our noun phrase list, we must develop a thesaurus and translation table relating them to some meaningful domain term if we are to benefit from the information they contain. We decided that these identifiers are not sufficiently useful to justify the additional time needed to manually enter and classify them for the AZTEC pre-processor. We therefore used a spell checker to replace them with the generic identifier name. The text is now reasonably clean but still not acceptable to AZTEC. A final rearrangement of sentences is required so there are no extra linefeeds, periods and carriage returns. Appendix III contains the original text of a sample LINPACK routine and for comparison the same text after preprocessing.

The sequence described above is applied to each subroutine individually. AZTEC however needs a large body of text to do the frequency analysis needed to assign technicality ratings [Laurendeau 1992]. To get a body of text large enough to

produce statistically significant results from analysis, the individual comment files are concatenated into one large one. This process is similar to Prieto-Diaz' literary warrant [1989], where an indexer picks a sample from the document collection to be cataloged to build a term vocabulary. The LINPACK routines serve as the sample for the reuse library index.

Cleaning up and Modifying the List

The list produced by AZTEC needs some post processing before it can be used to index a case library. Chapter II described steps in an algorithm for this cleanup process. Figure VIII lists those implemented in the current study. They are performed by the same tools used in preprocessing: basic UNIX commands, simple shell routines, and manual operations.

Figure VIII: Building the Term List

Post processing involves correcting misspellings carried over from the original text, and merging singular and plural forms and terms like `triangle` and `triangular` that have the same root. Also, AZTEC only extracts two word noun phrases and many NPs are longer. However, a look at the term list often suggests an obvious extension to a domain expert. For example, a linear system expert would immediately recognize `value decomposition` as part of the three-

word phrase singular value decomposition. Most spell checkers do not flag single letter variable names so these must be eliminated from the list manually.

Comments extracted from software often contain description of the algorithm and the domain as well as implementation details. The present study uses only terms that describe domain related algorithmic details so we decided not to incorporate general computing-related descriptors. Generic computing terms such as loop, overflow and array are therefore eliminated from the list. Some of these computer-related terms might be utilized in the future to enrich the software description. Similarly, variable identifiers could shed some light on similarities among a homogeneous collection of routines. This view is in contrast to other researchers [Fraser et al, 1992] who did not find local variable names in Smalltalk methods to be a good source of information.

Expert Verification

The term list constructed by this automated process should be verified manually. If the comment text is not too voluminous the expert can review it in its entirety. Larger texts will need to be verified by review of a sample of the complete text. In either case, manually marked up text is scanned and the relevant noun phrases extracted. Using more than one expert can make this process less subjective; their different lists can be merged to produce a single more objective one. Such an expert-built list is very useful in assessing how valid automated indexing is for the task at hand. The manual list can be further augmented by user warrant to get a more complete list of noun phrases for the domain. This last step adds terms to the list that are not extracted from the sample but which in an expert's judgment are likely to apply to the domain. The next Chapter shows how a manual index can be merged with an automated one to obtain a complete list of noun phrases applicable to the library manual index can be merged with an automated one to obtain a comprehensive list of noun phrases applicable to the library.

Analysis of the Noun Phrase List

Noun phrase extraction produces a list of terms and associated technicality ratings. This list can be mapped against each one of the originating routines to provide a subset of keywords that describe the routine. Keywords can be seen either as attributes or as bibliographic terms. Taken as attributes, a software component is described with a binary vector rather than a list of bibliographic symbols. One advantage of the vector representation is that the fields can be easily assigned

individual weights in REMIND, something not possible without fuzzy logic when a variable-length list representation is used. Either way, the term list can serve as a controlled vocabulary for standard information retrieval methods [Salton 1975].

Chapter II discussed how a noun phrase list could be converted to a controlled vocabulary for retrieval purposes and how a domain model can be developed from the two. We can now turn our attention to the important issue of representation of software components. We start by noting that all terms in the list can be classified as either objects or processes. This suggests a modelling of software processes as 3-tuples:

$(inputs, process, outputs)$

where all elements are variable length lists. Complex processes could be split into two or more simpler sub-processes as in the following example:

$sub-process\ 1 \leq process \Rightarrow sub-process\ 2$

which shows the is-part relation. Applying the model to its first sub-process yields:

$(inputs\ 1, sub-process\ 1, outputs\ 1)$

Modelling a piece of software as an (inputs, process, outputs) tuple ignores its algorithmic details. To see these at least one level of decomposition into sub processes must be performed. We see below how this strategy provides us with the classification facets.

Construction of A Reuse Library

Figure IX lists the various steps involved in building a reuse library in REMIND.

Figure IX: Building The Case Base In Remind

Mapping NPs to Individual LINPACK Subroutines

The first step in the process is to map terms in the master list back onto the individual subroutines to build their descriptions. Mapping involves searching for the master list keywords in an individual subroutine's comment file and adding hits to the subroutine description. The final description depends on the domain model chosen.

Field Selection

A list of symbols is a natural encoding for the varied terms gleaned from a subroutine's comments. The list contains domain-specific terms that describe its algorithm. As illustrated above, three fields

inputs, process, outputs

provide a reasonable framework for software component descriptions which hide underlying algorithmic details and the domain model. Decomposing the process into sub-processes incorporates these details in the representation. A similar but more rigorous division of routines into functional 'slices' has been carried out by Fouques and Matwin [1993]. Our goal of analyzing descriptions without any

knowledge of their algorithms. forces us to distinguish the objects relating to a process from those relating to its sub-processes. This required the addition of a process-objects field. The two refinements produce the following process model

(inputs, process-objects, sub-processes, outputs)

where all elements of the 4-tuple are variable length lists. This is done automatically by dividing its terms into four sub-lists according to the facets that they belong to.

The Symbol Tree

REMIND's [1992] portrayal of a *Symbol Tree* can be used to depict this process graphically. Symbol trees for three domain models of increasing complexity appear below. Keywords appear as symbols in each. Figure Xa shows a partial symbol tree for a flat model where the description of a software component is a single variable-length list of all the terms appearing in its comments.

Figure Xa: Partial Symbol Tree for Model I

In Figure Xb, the terms have been divided into four facets without any associative or taxonomic relationships between them. This is Model II.

Figure Xb: Partially Expanded Symbol Tree For Model II

Figure Xc: Partially Expanded Symbol Tree for Model III

Figure Xc shows that Model III contains additional organizational layers derived by introducing new relations expressing the knowledge in basic linear algebra texts. This last model exploits REMIND's capacity to record associations across categories via multiple parentage. More and more layers could be added to our symbol tree with additional term associations and deeper taxonomy. All of this domain knowledge is made available to REMIND's symbol list matching function when it calculates similarity values.

Although the Symbol Tree is the basic knowledge structure used in the three models used in the current study, REMIND offers other features to enhance model construction. These features include *Qualitative Modeling*, *Case Adaptation* and *Clustering*. They are discussed in Appendix II of later Chapters. Table I highlights the differences in the domain models including the binary attribute model discussed earlier.

MODEL	ATTRIBUTES	SYMBOL TREE	WEIGHTS
Model I	List of Symbols	Flat, One level	No
Model Ia <i>not implemented</i>	Binary	No	Yes
Model II	List of Symbols	Two Levels	Yes
Model III	List of Symbols	Multiple	Yes

Table I: Definition Of The Domain Models

With three domain models in hand, we can proceed with mapping the term list into subroutine descriptions. For the single level tree of Model I the process is directly analogous to keyword identification in an information retrieval system. For the faceted classifications used in Model II and Model II, a further step is required to place terms in the right slots in each subroutine's frame.

REMIND Data Import Facility

The batch data import facility of REMIND automatically converts the ASCII file of concatenated subroutine descriptions into case representations according to a pre-defined import map. This map relates fields in the data file to REMIND case fields. It must also include a formula to translate the file's ASCII text strings into REMIND's symbol datatype.

The current study includes an experiment testing retrieval performance for various models of the case base with three different symbol trees described above. This means that unique import maps are required for each representation.

The single layer Model I is the simplest domain model. When all attributes belong to the same group, mapping simply involves taking the list of unique keyword describing the routine and representing it as a list of symbols. A representational variant (Model Ia) treats all of the terms as binary attributes. The advantage to this is that we can then assign different weights to the attributes in REMIND. When all binary attributes have the same weight this representation of the library reduces to the first one.

To produce a faceted structure such as Model II, the single list of Model I must be sorted into separate lists corresponding to the facets. Case descriptions must then be modified to put terms in the appropriate facet slots, and a new import map

designed to reflect these modifications so REMIND will add cases to the base with descriptors in the proper fields. Moving from Model II to Model III is simpler. Only the symbol tree need be changed. With case descriptions unaffected, the import map need not be modified. This is a major benefit of making the top level of the description domain independent and hiding the details of the domain model within the symbol tree. This opacity, together with REMIND's data import facility, lets different domain models be tested just by changing the symbol tree and re-importing cases. Top level domain independence is particularly useful in fast-changing fields such as computer science because it allows the underlying domain model of the reuse system to be modified without affecting the user.

Creation of Facets

Different symbol trees permit more and more complex reuse domain models to be tested. This normally requires an expert defining relations between objects and processes in different facets. Such a change of representation is only the first step in building a domain model. While the organization of well-established fields like linear systems is well understood, many domains have weak formal theory. In such situations term groups must be extracted by automated techniques [Sparck Jones, 1971; Salton, 1975]. The process is somewhat similar to that of identifying determinations—trying to identify groups of terms that appear together. Here we wish to find terms that are mutually exclusive or least likely to co-occur.

Types of Retrieval and Importance

REMIND can perform retrieval with or without fuzzy matching. The type of exact matching used in database systems is not of particular value in software reuse, where the interest is in retrieving multiple candidates, most of which only approximate the query.

REMIND supports both inductive and nearest-neighbor retrieval via fuzzy matching. Inductive retrieval is efficient but requires that routines be pre-classified according to an outcome class. No such field can be identified in advance to use to build an induction tree for reuse. Nearest-neighbor retrieval, on the other hand, does not require an outcome field. It operates by serially comparing the input case with all other cases in the base. Their similarity to the input case is evaluated according to (1). Details of the formula used to calculate similarity $S(t, s)$ between a target t and source s is explained in detail in Appendix III.

The selection of term weights is usually done by an expert who uses these to assign importance of the attribute. Alternately, (3) can be used to compute individual term weights $w(1, k)$ of the 1th value of the kth attribute. These are then summed to arrive at the attribute weight $w(k)$ for the kth attribute.

$$w(k) = \text{sum}(w(1, k)) \text{ over all } 1 \quad (4)$$

Attribute weights are input into REMIND directly.

Testing and Calibration

Evaluation Criteria

The experiment employs a measure based on the information retrieval notions of *recall* R and *precision* P [Sparck Jones 1972]. These are formally defined for a query as:

$$R = n/N \quad \text{and} \quad P = n/n';$$

where

N = total number of cases relevant to the query,
 n' = total number of cases retrieved by the query,
 n = number of relevant cases retrieved by the query.

The Retrieval Evaluation Experiment

To help evaluate the results of the experiment the LINPACK library has been organized into a table whose columns list the various operations and whose rows identify the types of matrices that are input (see Appendix III). The table is utilized by following these steps for all routines:

- 1 - pick a routine
- 2 - retrieve its 8 nearest neighbors
- 3 - select all relevant neighbors above the similarity threshold
- 4 - calculate recall and precision

This comparison test assumes the computed similarities of retrieved components bear some resemblance to the actual similarities between the LINPACK routines. (For this to be true, the underlying assumption is that similarity in comments implies similarity in routine's function.)

LINPACK Classification Scheme and the Relevance Table

The above definitions of recall and precision are based on the concept of relevance. While it is up to each user to decide what is relevant and what is not for his particular need, in software reuse relevance measures the effort necessary to reuse a case. For the purposes of this experiment a definition of this complex issue has been based on the classification scheme of the LINPACK library itself. Table II displays part of this scheme.

	Condition (CO)	Solve (SL)	Factor (FA)	Determinant (DI)
General Band (GE)			X	
Positive Definite (PO)	X	X	X	X
Positive Definite Packed (PP)			X	
Positive Definite Band (PB)			X	
Symmetric Indefinite (SI)			X	
Symmetric Indefinite Packed (SP)			X	
X indicates an instance of the routine SPOFA				

Table II: Relevance Table

Table II, the Relevance Table, shows how LINPACK routines are classified by a simple two dimensional scheme according to:

- 1) the input matrix type
- 2) the kind of operation performed

This classification is reflected in LINPACK's naming convention for subroutines (Appendix III). Table II also shows the routines selected from the LINPACK library for our experiment. This LINPACK subset identifies 24 routines: 4 kinds of operation, for each of 6 types of input matrix.

We interpret Table II as indicating similarities between routines and therefore their relevance for reuse. For example, when retrieving cases similar to the routine SPOFA (factoring a positive definite matrix), we consider relevant only those routines that have a positive definite matrix input or do factoring. The relevant cases are the eight single-precision routines occurring along the horizontal and vertical lines marked by X's. Matches for SPOFA are: a) three routines which have positive definite (PO) matrices as input;

SPODI, SPOCO & SPOSL

and b) five routines used for factoring (FA),

SGBFA, SPPFA, SPBFA, SSIFA, & SSPFA

The experiment involves matching each routine against all others. Assume that when SPOFA is matched against all other routines in Table II, the following eight nearest neighbors are retrieved:

SPPFA	SPBFA	SPOCO	SPODI	SSIFA	SSIDI	SPPCO	SPPDI
67	52	50	46	42	42	40	39
							▲
similarity threshold for the matching function							

The five neighbors on the left are similar to SPOFA in input matrix (PO) or in function (FA) and have similarity values above the 40% threshold. They are therefore considered relevant matches to SPOFA. Although the sixth SSIDI routine has a similarity above the threshold, it is not considered relevant because it does not have a PO or FA in it. If we increased the similarity threshold to 50% SPODI and SSIFA would be excluded. The recall and precision of this particular query are easily computed:

$$N = 8, \quad n = 7, \quad n' = 5$$

giving

$$R = 7/8, \quad P = 5/7.$$

The actual experiment involves the following steps:

- 1 – running the comparison for all 24 cases in Table II;
- 2 – computing R and P for each query using thresholds of 30%, 40% and 50%;
- 3 – averaging the P and R values for each case to get a composite value;
- 4 – repeating steps 1-3 for Models I, II and III

Choosing different threshold values varies recall and precision for the models. While each is a desirable quality in a retrieval system, it is not possible to optimize both recall and precision simultaneously. For this reason a composite *effectiveness* measure E is often used. Many such measures appear in the information retrieval literature [van Rijsbergen, 1979]. An example is

$$E = 1 - \frac{(1 + b*b)*P*R}{b*b*P + R}, \quad (5)$$

where b is a measure of the relative importance, to the user, of recall and precision. A related concept more suited to reuse is the cost-benefit function f . If, there are N relevant cases in the library and if n are retrieved out of which n' are relevant, then cost-benefit is defined as the ratio of cost over benefit:

$$f = \frac{C1*n + C2*(N-n)}{B*n'} \quad (6)$$

where $C1$ is the cost of reviewing a component, B is the benefit of retrieving a relevant component, and $C2$ is the cost of not retrieving a relevant component. Relating cost-benefit to recall and precision,

$$f = \frac{C1 + C2*(1/R - 1)}{P*B} \quad (7)$$

The matching routine can then be adjusted to maximize cost-benefit by changing the relevance threshold. Other authors have applied decision theory concepts to calculate query costs for information retrieval systems [van Rijsbergen, 1981].

Single Level Tree vs. Keyword Matches

The single level model is interesting because it allows us to compare our results to keyword matching used in traditional information retrieval systems. One difference is the current system's use of fuzzy matching to broaden search. Traditional information retrieval only relaxes criteria through generalization if exact matching produces no hits. Another difference is the current system's capacity to match on two word phrases. Most information retrieval systems use an inverted list of single words and can only identify multi-word phrases by proximity analysis of the two or more terms within documents at retrieval time [Salton, 1975].

Two Level Symbol Tree and Faceted Classification

The two level representation resembles the Prieto-Diaz model (Figure Xb) with its fields similar to his facets. However it lacks an equivalent to his conceptual distance graph. This lack is rectified by the addition of further layers to the tree in Model III.

Multi- Level Tree and Conceptual Distance Graph

Conceptual distance is represented by the symbol tree itself. REMIND offers two means of doing this: by ordering siblings within a layer and through its multiple parentage feature. The program's matching function takes these factors into account when calculating conceptual distance or similarity.

Browsing and One Stage retrieval

Once the case base has been set up retrieval of a list of candidates to match a query specification proceeds fairly simply. The specification is developed by selecting entries from a controlled vocabulary of terms, which in case of single level model is the total list;

{term 1, , term n}

A query specification might thus be:

[term 2, term 5, term 6]

Specifying a query in the single level model is tedious because it often involves making selections from a large list. This typically requires passing over the list a number of times to ensure that the right terms have been chosen. Specialized retrieval templates can make this process less onerous but it can only be made fundamentally easier by grouping the descriptors into facets as is done in the two-level system. Specification then involves selecting sets of terms from each facet's list, resulting in one list of terms per facet. Consider a faceted classification for a system with $n = 6$ and two facets:

```
facet 1 = [term 1, term 5, term 6],  
facet 2 = [term 2, term 3, term 4]
```

The same query is now in the form of a list-of-lists:

```
[[term 6], [term 2, term 5]]
```

These lists now become the target against which all of the cases in the base are tested by REMIND's nearest-neighbor matching function. Correctly specifying the matching function's parameters directly affects retrieval performance, and a user can achieve his retrieval goals by varying these parameters for different queries. The nearest-neighbor matching function has two parameters:

- values of importance levels
- number of nearest neighbors k

The similarity threshold can only be changed indirectly by varying k . Future release of a C library for REMIND may permit this matching function to be customized.

Other Implementation Issues

The present study is concerned with building a description of a software component from its comments. Descriptions built in this way are domain-oriented and algorithmic in nature. They lack much important implementation data [Prieto-Diaz 1989] that cannot be extracted from comments:

- lines of code
- quality of user documentation
- language
- speed and/or memory requirements
- code quality

- currency of the code (when last used or changed)
- hardware / software environment
- application area

Some or all of the above factors may influence code reusability and should be used as secondary criterion for selection in the final stage of evaluating a component after the user has accepted it as functionally similar to his specification. Initial acceptance and evaluation together form a two-stage retrieval process that is conceptually similar to the one used by Prieto-Diaz [1989]. Before turning to it let us first define reusability more formally.

Reusability Metric

We can start by defining reusability in terms of reuse effort:

$$\text{reusability} = 1 - \text{reuse effort} / \text{rewrite effort} \quad (8)$$

where by *rewrite effort* we mean the effort needed to implement the routine from scratch. The closer a software component's reusability is to 1, the more likely a programmer is to use it. When reusability drops below a certain threshold (~ .5) most programmers will opt to rewrite the code if given a choice. This attitude is understandable and has been discussed by other authors [Fraser et al, 1989]. One difficulty in determining reusability is in estimating the effort needed to reuse a component prior to making the choice between reuse or rewrite. While estimating *rewrite effort* is also difficult, at least some standards and prior experience apply.

Lacking quantified relationships between reuse effort and the factors listed above, we might seek qualitative ones. A qualitative model can be used to rank candidates according to their associated reuse efforts. Prieto-Diaz [1989] considered similar factors in evaluating components' reusability ranking. A qualitative model of reuse effort can be build in REMIND by describing the degree to which reuse effort depends on various factors. Figure XI portrays such a model. Its two *virtual nodes*, 'algorithmic quality' and 'code quality', both influence reuse effort. These non measurable factors exist only as intermediates which are in turn influenced by factors such as lines of code and stability.

Figure XI: Qualitative Model For Software Reuse

A close correspondence exists between REMIND's Qualitative Model representation and the Conceptual Closeness Graph which Prieto-Diaz uses to broaden search. Virtual nodes are conceptually very close to the *notions* or *super types* of Figure II. A Conceptual Closeness Graph can readily be implemented as a Qualitative Model in which the function being calculated is simply the closeness between any two concepts which are leaves in the graph. Notwithstanding this, the Symbol Tree representation is better suited to domains like linear systems which have well defined relationships.

Two-stage retrieval

The two-stage retrieval process proceeds as follows:

Its first step involves retrieving cases that match the specification from the base. The matching function limits the number of cases retrieved to a preset value. A threshold value for similarity is applied to eliminate unlike cases. The remaining cases are ranked according to their reuse effort as computed by the qualitative model described in Figure XI.

Case-base Management

New Case Additions

One or more new cases can be added to the base using the same term list initially used to build the base. New terms used in comments in the added cases must first be added to the term list, however. This can be done in three ways:

1. resubmitting new comments for AZTEC to identify new terms;
2. manually reviewing the comment text for new terms;
3. resubmitting both new and original comments to AZTEC.

If the comment text is extensive or many routines are being added, invoking AZTEC is the best course. The program's overhead is fairly large and it is best reserved for major additions. Manual review is preferable for a single new routine. It can be partly automated by first using the mapping routine to identify, group and mark all existing terms. This makes new terms easier to identify. Reprocessing the whole comment text in AZTEC is only necessary if there is a major supplement to

the collection. It should also be done when the incremental addition of new terms has substantially changed the noun phrase list.

Once identified, new terms must be classified and added to the symbol tree before any new case is processed. REMIND facilitates this with a library function for batch additions to the symbol tree using the import map. If only a few cases are involved the symbol tree can also be updated interactively.

Synonyms

When additions come from another software collection many of the new terms they introduce may be synonyms for existing ones. Such terms could simply be indexed as siblings of existing terms, but two cases then differing only in synonyms are not guaranteed to match with 100% similarity. The correct way is to create a formula field to which all synonyms are equated and then perform retrieval via the formula field alone. If term 1 and term 2 are synonyms with meaning M their formula would be:

M :- term 1 or term 2

When defined in this way synonyms are transparent in any retrieval employing M or its ancestors.

4. ANALYSIS OF RESULTS

Results of AZTEC

The LINPACK collection yielded 25 pages of concatenated comments taken from 31 single precision routines. Cleaning up raw comments according to the process detailed in Figure VII reduced this by 43%—the final text consisted of 820 sentences in 15 pages. Text analysis of the edited file took more than 7 hour on a SUN 4. The main bottleneck is the relative slowness of AZTEC's sentence parser [Laurendeau 1992]. Final output showed the following distribution of terms:

Technical	Rather Technical	Possibly Technical	Not Technical
26	59	118	49

AZTEC failed to classify 36 phrases, flagging them as Unknown. A review of the output showed that computing terms and single letter variable names are quite common. The 26 phrases in the Technical category break down into:

Computer-Related Terms	Single Letter Variable Names	Noise Words	Good NPs
7	6	3	10

Applying the information retrieval measure of precision (% of relevant terms) to the raw output gives a value of 40% for the Technical category. This low value is caused by the presence of computer related terms and single letter variable names that can easily be filtered out by a preprocessor. This increases precision to 70%.

AZTEC and Manual Lists

Noun phrases were also manually extract from the same text to see in detail differences between the manual and automated approaches. After the AZTEC output has been cleaned up it can be compared to the manual selection and categorization of terms. The first point of comparison, shown in Table III, is the content of the two lists:

Aztec Derived	Manual Derived	Common	Total
72	65	53	84

Table III: Comparing Aztec And Manual Lists

Applying the information retrieval measure of recall (% of relevant words),

$$\text{Recall} = \frac{\text{AZTEC Derived}}{\text{Total}}$$

gives a value around 86%. AZTEC does quite well especially when, as Table IV shows, most differences can be accounted for in terms of multi-word phrases that AZTEC was not designed to extract.

	AZTEC	MANUAL
SINGLE WORD	37	26
2 WORD	35	27
> 2 WORD		12

Table IV: NP Distribution By Length

Both lists show approximately the same frequency distribution for single and two word phrases. It will be interesting to do these comparisons again when AZTEC is enhanced to identify longer noun phrases.

Using AZTEC Technicality Ratings

AZTEC associates a rating of technicality with each phrase it identifies. Ratings are calculated according to a formula based on a term's frequency of occurrence in the text and other factors [Laurendeau, 1992]. These technical ratings seem an obvious key to identify which descriptors would be most important for indexing routines. However this is not possible because in this study since AZTEC

processes comments in a concatenated file that does not take into account the distribution of NP's within the subroutines. A final question concerns the significance of the technicality ratings themselves. To what degree do they correspond with the terms LINPACK uses to describe its components? The following table lists the terms AZTEC puts in the Technical and Rather Technical categories:

TECHNICAL	RATHER TECHNICAL	
low triangle	definite matrices	residual vector
column pivoting	semi-definite matrices	pivot element
factor	singular vectors	partial pivoting
symmetric pivoting	diagonal permutations	output
computing	updated factor	vector
element	coordinate transformation	pivot indices
lead dimension	positive eigenvalues	pivot column
order	negative eigenvalues	pivoting
determinant	triangular factor	largest norm
diagonal element	updated problem	pivot block
	reciprocal condition	normal value
	transform rotations	upper triangle
	householder transformation	norm
	triangular matrices	pivot array
	off-diagonal element	matrix
	orthogonal transformation	

To answer this question, the above list is checked against the terms LINPACK uses in its index (Appendix III). In some cases this required making logical extensions to the terms extracted by AZTEC. Table V summarizes the results. Only a few of the AZTEC terms with the two highest technical ratings appear in the LINPACK index.

Technical	Rather Technical	Possibly Technical	Not Technical	Unknown
2	3	8	1	0

Table V: Distribution of LINPACK Terms in AZTEC Results

This result discourages using technicality ratings to establish term importance. AZTEC technicality ratings are difficult to relate to attribute relevance or weights because they are based on global frequencies and ignore the distribution of terms within individual routines [Laurendeau 1992].

Main Results

The following general conclusions can be stated:

1. AZTEC can pick out technical terms quite successfully;
2. the non-technical list shows no technically significant terms;
3. noun phrases classed as unknown can have technical content;
4. differences within the three technical categories of AZTEC are only marginally meaningful.

A review of the AZTEC algorithm revealed that phrases occurring only once in a text are classified as unknown. Notwithstanding 2. above, the fact that the criteria used to separate technical categories from non-technical ones are arbitrary leaves the possibility of misclassification open. A review of non-technical and unknown terms might reveal additional technical terms for relatively little browsing effort.

Critique of AZTEC as A Reuse Tool

AZTEC was designed to extract phrases from technical literature such as user manuals. Its usefulness for extracting and analyzing noun phrases in code comments is limited because of the substantial effort needed to preprocess input and to clean up output. This is due less to shortcomings of AZTEC than those of the Sentence Parser and Text Base Builder, two external programs which AZTEC uses. A number of enhancements would make the whole system much more useful for the task at hand. These include:

- (1) Phrases of more than two words are quite common and the sentence parser should recognize at word phrases of three words;
- (2) Unknown syntactic classes which AZTEC encounters must be added by the user interactively in a tedious process. A batch program to preprocess the text can help in identifying and marking unknown phrases. This would permit these classes to be added to the dictionary more efficiently off-line

- (3) Restricting parsing to syntactically complete sentences is too restrictive for comment text, much of which is fragmentary. Many suitable terms are not extracted because the Sentence Parser ignores these. AZTEC does however have a fallback mode that extracts a limited number noun phrases from incomplete or overly-complex sentences.
- (4) If they cannot be used by AZTEC, incomplete sentences should be flagged by a preprocessor with a grammar filter which could also flag and/or remove a) words with similar roots, b) singular and plural forms of the same word and c) spelling mistakes.
- (5) AZTEC is quite inflexible on the format of its input. Considerable effort may be needed to clean and reformat a text. The tools to clean up input were developed outside the system. These could easily be incorporated into AZTEC's user interface, making the task easier to accomplish. Easing of the strict input file requirements is a potential beneficial side effect .
- (6) The biggest problem, speed, can really only be improved on by changing the Sentence Parser.

Evaluation of the Software Reuse Library

The LINPACK library can serve as a tool to evaluate the reuse system. Because it is already pre-classified, we can use the description of each of its components as a query specification to test retrieval performance. Acceptance or rejection of retrieved components is according to the Relevance Table criteria discussed earlier.

Assume we wish to code a routine with the same specification as one in LINPACK. If we did not have the routine in our library, the best starting point would be one of the other LINPACK routines similar in function or in input matrix (as indicated by the Relevance Table). The test involved testing the similarity of each routine with all others and then averaging the results over all routines for various similarity thresholds. This experiment was repeated for all three models.

Major Retrieval Results

The first measure used was the average REMIND similarity score of all nearest neighbor matches. The test was repeated on all the three models using their different symbol trees. One objective was to see the effect on similarity of increasing knowledge in the system. This was accomplished by testing the three domain models individually. Results are as are shown in Table VI.

Symbol Tree Complexity	Average Nearest Neighbor Similarity Score %
Model I (keyword based)	80.4
Model II	83.6
Model III	84.3

Table VI: Results of Increasing Symbol Tree Levels

They suggest that a significant increase in similarity score occurs when terms are combined to get a faceted representation (Model II). However, additional knowledge introduced through a more detailed description of the linear system domain (Model III) does not seem to increase this score by much.

This result is not independent of the definition of the similarity function [Holte, 1993]. Ideally one must normalize it to compensate for spurious increases, if any, due to the structure of this function. One could test this effect of faceting by dividing the terms randomly in four (control) groups of sizes similar to the actual facets. The case descriptions can then be mapped into the four random facets for our experiment. We recompute the nearest neighbor similarity averages for the new case descriptions. These can serve as baseline with which one can compare the difference Model I and II observed in VI.

To assess the retrieval performance another test was run to compute effectiveness E as defined in (5) and cost-benefit f as in (7).

	RECALL	PRECISION	EFFECT	COST
<i>similarity threshold = 30%</i>				
Model I	0.94	0.42	0.71	1.11
Model II	0.89	0.45	0.70	.99
Model III	0.91	0.45	0.70	.82
<i>similarity threshold = 40%</i>				
Model I	0.77	0.54	0.68	1.12
Model II	0.61	0.65	0.69	1.20
Model III	0.60	0.66	0.69	1.36
<i>similarity threshold = 50%</i>				
Model I	0.48	0.79	0.70	1.61
Model II	0.42	0.78	0.73	1.62
Model III	0.39	0.77	0.74	1.70

Table VII Evaluation of Retrieval Performance

This experiment used the standard definitions of recall and precision and three values of the threshold for similarity. The $b=1$ was used for the van Rijsbergens evaluation function that corresponds to the situation where precision and recall are equally important for the user. Cost-benefit function requires specification of the cost factors C_1, C_2 and benefits B for evaluation.

Let us consider special case of cost-benefit function where:

$$C_2 = B = 10 \cdot C_1.$$

In rough terms it suggests that effort invested in review of 10 software components is paid back if it nets a good match. The equality $C_2 = B$ corresponds to the situation where the benefit of finding a relevant case is equal to the cost of not finding it. This situation is not universally true; in medical diagnosis the cost of not finding the right case could be very costly indeed. With these values of C_1, C_2 and B , cost-benefit reduces to:

$$f = (1 + 10 \cdot (1 - R)) / (10 \cdot R \cdot P)$$

The following observations can be made about the data in Table VII:

- 1 Recall is greater when similarity has a low threshold;
2. Increasing domain knowledge tends to reduce recall and increase precision;
3. Effectiveness correlated weakly if at all with the similarity threshold or domain model complexity;
4. Cost-benefit is quite sensitive to the similarity threshold but less so to domain model complexity;
5. Recall and precision have the predictable inverse relationship. It is not possible to maximize both simultaneously.

The first two observations agree qualitatively with the findings of other workers [Prieto-Diaz 1989]. The results in Table VI and VII suggest that reuse systems may not need very elaborate domain models. The functions used to evaluate information retrieval systems may be of some value for measuring reuse system performance if these can be specialized for user needs. For example, recall is more important than precision in reuse systems with few components. This fact may be expressed in the effectiveness function (5) via the value of b that in turn can be empirically related to the threshold.

A specialized cost-benefit function may be a better measure of the reusability. However, it requires knowledge of cost factors specific to the reuse environment and user need. The matching function can be fine tuned to provide any desired retrieval performance.

Additional Experiments

The issue of using only the most predictive attributes has been raised by many authors [Salton 1979]. The basic premise is that the informative terms are those that occur neither too frequently nor too infrequently in the case description. These content bearing terms fall in the middle frequency range [van Rijsbergen 1979] of term distribution. We arbitrarily selected 1/3 of the total terms that fell in the mid frequency range about the average value for term occurrence. The above tests were run again for the Model II described in terms of only these terms with 40% threshold.

	RECALL	PRECISION
All Terms	.61	.65
Selected Terms	.64	.81
<i>similarity threshold = 40%</i>		Model II

Effect of Using a Selected Number of Attributes

We note that when we use only the selected terms for indexing the routines the recall and precision both get better. Precision shows a great deal of improvement with only a small change in recall. This is quite consistent with the earlier work in information retrieval. The improved precision is expected since the noise terms have been removed.

In another experiment we tested the correspondence between the terms that LINPACK uses to classify routines and the terms that we extracted from the comment text. We expect that the known terms of the LINPACK classification system (see Table in Appendix III) to occur in the comment text as well. This gives us an opportunity to validate the casebase. The experiment we have designed is based on the observation that for each of the cases in the Relevance Table there are 10 possible matches (including itself). If we extract 10 nearest neighbors then ideally we should have all of these in the retrieved set. Using this type of matching we found the average Recall and Precision to be:

$$\text{Recall} = \text{Precision} = 64\%$$

$$\text{Nearest Neighbors} = \text{Relevant Cases} = 10$$

Testing the Case Base Using LINPACK Terms

Note the fact that we have chosen to retrieve the same number of nearest neighbors as there are relevant cases. This choice forces recall and precision to have the same value. We also found that the theoretical closest neighbor, while

always on the list of ten and was the nearest neighbor in about half the cases. This is to say that if the relevance was defined in terms of the this theoretically closest neighbor the recall will be 100%. (Note that we apply the term 'relevant' in a generic sense to both the terms and the cases.)

Comparison of CBR and Prieto-Diaz Approaches

Many similarities and many differences exist between the Prieto-Diaz reuse methodology and one based on CBR. Figure XII compares the two approaches.

	PRIETO-DIAZ	REMIND / CBR
Representation	facet	case attributes
Structured Attributes	no	symbol tree, lists
Associative Relation	notions	multiple parentage symbol ordering qualitative modeling
Knowledge Addition Methods	conceptual distance graph	typicalities, symbol tree importances / weights qualitative modeling
Query	relational binary	nearest neighbor fuzzy
Access	DBMS	serial, tree

Figure XII: Comparing the Prieto-Diaz Method to REMIND / CBR

Facets as used by Prieto-Diaz are broadly equivalent to case attributes. However, the current implementation of his algorithm does not allow for structured facets. CBR, on the other hand, permits sub-attributes in symbol fields. This capacity is used in REMIND to build sophisticated hierarchies of attribute with multiple parentage for symbolic attributes.

Prieto-Diaz adds external knowledge mainly through the conceptual distance graph described in Chapter I. CBR has various equivalent techniques. These range from specifying attribute weights to qualitative modeling to developing a sophisticated Symbol Tree.

Prieto-Diaz retrieval employs a relational query language that uses the conceptual distance graph to broaden its search space when an exact match is not found. The match function always remains binary; fuzzy matching is never attempted. CBR retrieval is inherently fuzzy in that it looks for the k-nearest neighbors to the query. The simplest CBR retrieval technique is a serial match against all cases; however more sophisticated schemes that rely on inductive or prototypicality considerations are available.

5. CONCLUSIONS

Major Results

The two main objectives of the current study were:

1. to assess the usefulness of terms automatically extracted from source code comments for the indexing and retrieval of software;
2. to design and evaluate a software reuse system based on standard Case-Based Reasoning technology.

AZTEC was used for the first task of noun phrase extraction. It was designed for analysis of and tested on well-formed text found in technical manuals and other documents. Its usefulness in analyzing the fragmentary and rather cryptic text found in comments was not ascertained prior to the current study. Because AZTEC relies on syntactic analysis, extending it to work on poorly formed text was not trivial. The current study compared AZTEC's performance with a manual equivalent. Surprisingly, in spite of its various limitations, we found it to be adequate to the task. In addition to extracting technical noun phrases AZTEC assigns ratings of their technicality to each. Possibly because of the small size of the study text, the validity of these ratings was not confirmed by our results.

To accomplish the second objective, a reuse library was implemented indexed on terms in the comment text of the C version of the LINPACK linear systems library. The REMIND CBR shell was used for implementation. Three different models of REMIND's symbol tree were developed corresponding to differing depths of background knowledge about the domain. We found that:

- 1 – a flat undifferentiated description produces fairly good retrieval performance;
- 2– moving from a flat symbol tree description to a faceted one improves performance considerably;

- 3 – moving from a simple faceted model to one enriched with taxonomic and associative domain knowledge further improves performance only marginally.

Reasons for the third outcome are unclear but may be related to the manner in which REMIND's matching function uses the additional knowledge. Traditionally, case-based systems depend little on external or background knowledge. This is a natural consequence of CBR's emphasis on 'experience', resulting in reliance on its matching function and similarity metric. Additional knowledge therefore can only be useful to the extent that these elements are affected by it. This is in contrast to rule-based systems that depend strongly on domain knowledge and rely little on cases and prior experience. Hybrids have appeared in recent years which use both these strategies in a single system [Bareiss, 1989; Porter et al, 1990].

Various approaches to adding external knowledge to a CBR system have already been discussed. The following list summarizes how these influence the similarity metric:

METHOD OF KNOWLEDGE ADDITION	EFFECT ON SIMILARITY METRIC	
relevance of attributes	direct,	strong
term groupings	indirect,	strong
deeper taxonomy	indirect,	moderate
assigning weights	direct,	moderate
prototypes	indirect,	weak
qualitative modeling	direct,	weak

Additional knowledge has its strongest impact when used in the selection of relevant attributes, followed by term grouping. Augmenting the symbol tree taxonomy influences results only moderately, with increased depths contributing asymptotically less and less improvement to the similarity metric. However, this indirect addition of knowledge to the system reduces its brittleness and keeps the overall process fairly noise tolerant. Limited testing with facet weights also shows that these have marginal influence on the similarity function for our library.

In summary, results show that external knowledge and background domain theory can enhance the performance of a CBR reuse system to some extent. While it

seems a truism to incorporate whatever knowledge is available in a system, the present study suggests that returns diminish as additions continue.

In particular, this study confirms earlier work that found retrieval performance improved when terms were classified [Sparck Jones, 1971; Salton, 1975; Prieto-Diaz, 1989]. We note, however, that these results are influenced by the selection of the similarity function. Increased cost effectiveness is the major benefit of classifying terms into facets. In addition to improving recall precision, facets make retrieval easier. Most important, classification in facets is a necessary step to adding other knowledge to the system, whether in form of a symbol tree or as a conceptual distance graph. It can help provide explanations for selections made. Retrieval can also be sped up if classification are used to create clusters or other easily navigable structures. Although much of the classification must be done manually by a domain expert at present, some work has been done to automate this process [Sparck Jones, 1971].

Additional Work

Identifying Well Commented Text

Describing software based on terms found in its comments is a viable means of indexing a software library. The method used in the current study works when the documentation is well-written and somewhat homogeneous. This study has not attempted to define standards for the quality of comments needed to produce these results. The issue must be dealt with before this technique will enter widespread use. Esteva and Reynolds [1988] used induction on software metrics to identify candidates for reuse. If metrics can be developed to measure the quality of comment text, a similar approach can be used for reuse systems.

Storage for Efficient Retrieval

The study's serial matching scheme is very slow when the library is large. It could be sped up if a classification tree was used to store cases at its leaves. Retrieval would then involve traversing the tree to arrive at the desired case group, making it far more efficient than serial matching.

However a target attribute is needed as the basis on which to induce a classification tree; its values are used to classify cases. Unsupervised learning does not have a particular target field. Supervised and unsupervised learning

environments differ in this respect [Fisher, 1987]. Even lacking a target attribute, cases can still be organized according to any one of number of methods available for conceptual clustering [Fisher, 1987; Michalski and Stepp, 1983]. Clusters can then be analyzed to identify representatives or prototypes [Arrango, 1989]. These algorithms, applied recursively, create a hierarchy of clusters and attendant prototypes.

Non-Homogeneous Collections

The well-documented software library used in the current study is a best case. Real world software collections are likely to come from various sources with differing documentation standards and terminology, even for well established domains like linear algebra. The output of analyzing such heterogeneous texts will require much cleaning up. The problems raised by synonyms and common stem words should be addressed in a formal manner through a generalization of the thesaurus concept [Salton, 1975].

Improved Text Analyzer

AZTEC's Sentence Parser is slow, somewhat primitive, and is restricted to phrases of only two words. A new version has multiword phrase extraction and higher speed [Delisle 1990]. Ideally a custom NP extractor should be built, suited for the comment text and with some of the missing features we have described above.

Attribute Analysis

The significance and independence of attributes is an important issue that should be investigated in relation to reuse. The current study makes a small step in this direction by introducing weights for terms. The limited test of relevance of attributes that we did should be expanded to other measures of relevance. It would be interesting to see, for example, the effect of selecting relevant attributes according to Salton's [1975] discriminant function. Attribute independence could be tested by many methods, the most interesting of which is the calculation of determinations [Russel, 1989]. Schlimmer's [1992] algorithm should be tried to see if it would isolate independent terms satisfactorily. A generalization of this concept, change of representation, might produce more powerful and succinct descriptions of software.

The concept of a thesaurus can be extended to group terms which have certain relationship to each other, into classes [Sparck Jones, 1971; Salton, 1975, Srinivasan 1993]. These groupings create facets when applied to a list of NP's. The results of an automated system can be compared to classifications produced by an oracle. If automated classifiers show promise in well understood domains, they might be employed in weak-theory domains where they would really be useful.

Normalization of Similarity Function

The increase in the average value of similarity of nearest neighbors increases with model complexity. This increase is not entirely independent of the structure of the similarity function since it also depends on the structure of the symbol tree (see Appendix II). We must therefore correct for the possible increase due this effect.

Other Measures of Effectiveness

Simple cost and effectiveness measures were used to assess the performance of the current system. Other measures based on utility might be used for this purpose. It would be interesting to see if results are independent of the performance measure used [van Rijsbergen 1979].

Expanded Text Mining

In our efforts to cleanup the text, the non-domain type of terms were filtered out. These include author's name, dates of implementation and last revision, 'calling' and 'called' routines. These are usually specified in delimited fields in the boxed comment's area. Much of this tombstone data can be automatically extracted using very simple UNIX tools. Analysis of variables and subroutine identifiers can create additional knowledge useful for retrieval.

Combining Code and Comment Analysis

Code analysis done by Fouque and Matwin [1993] and comment analysis in the present work draw information from different parts of the source code file. It makes sense to combine the two to get a more powerful retrieval tool.

Enlarged Case Base

A much larger collection of software components is needed before most of the proposed improvements can be made.

Related Work:

A number of studies have dealt with the compositional reuse of software since the pioneering work of Prieto-Diaz [1985]. Differences between his methodology and that of the present work were discussed earlier. Their basic similarity lies in using keywords to describe cases, permitting the use of information retrieval techniques. Both are independent of the code and therefore the implementation language. However, while Prieto-Diaz takes a mainly functional or 'black box' approach to cases, we do consider the implementation details gleaned from preamble and inline comments.

Fouque and Matwin [1992] have demonstrated the feasibility of using dataflow analysis of source code as an aid to compositional reuse. In one experiment, their CAESAR system used CBR technology to store specifications and source code of linear algebra routines. The routines were cut into functional slices based on dataflow analysis of their code and the slices stored in the cases. CAESAR's goal is to compose actual code whereas the current work, is concerned only with retrieval. The program does this by retrieving the cases most similar to the user specification, sequentially composing a solution from them, and evaluating it with software testing techniques such as input/output analysis. In a recent enhancement to CAESAR [Fouque and Matwin, 1993] slices are combined into more complex modules by means of Inductive Logic Programming techniques to produce more efficient and succinct descriptions.

Frakes and Pole [1991] have developed a prototype of a reuse system employing four different types of representations. Retrieval strategies corresponding to them include information retrieval (IR), relational database, and one each for faceted and enumerated classifications. Their PROTEUS system is intended to provide a platform for the empirical evaluation of different methods for building software libraries. An interesting correspondence exists between the various representations used in PROTEUS and in REMIND, shown in Table VIII.

PROTEUS	REMIND
Attribute Value	Template
Information Retrieval	Model I Symbol Tree
Faceted	Model II Symbol Tree
Enumerated Graph	No exact equivalent

Table VIII: Comparing **REMIND** and **PROTEUS**

Although REMIND lacks an exact equivalent to PROTEUS' enumerated graph, many of the advantages of this representation are available elsewhere in REMIND. Both REMIND and PROTEUS allow the user to interact with the system through a common layer in which he or she selects different methodologies for testing through different views of the system. Speed rather than retrieval effectiveness is the major focus of preliminary experiments with PROTEUS.

Among other efforts to use comment text for indexing reuse components is one by Fraser et al [1992]. They extracted keywords from the brief text field in Smalltalk methods. Smalltalk/V was also used to build a Reuse Browser that supplemented the standard class hierarchy browser. The resulting keyword-based reuse system was equipped with a simple Boolean query language. Term extraction was limited to single words. This seems inadequate since many of the methods referenced have alphabetically joined names, e.g. sortBlock.

Components in the authors' system are represented by a four tuple:

```
identifier(i) = (class name, method name,
                method type, frequency)
```

These are indexed by keywords stored in an (inverted) Access Dictionary:

```
keyword(k) = {identifier(1), ..., identifier(i), ...}
```

Most queries in Fraser's study were described by one- or two-term Boolean expressions. Fuzzy matching was not attempted, but queries could be modified by generalizing or specializing the Boolean expression. This particular system is strongly tied to the Smalltalk environment, where it was found to be quite useful, especially by new users. One major goal of this research was to identify documentation criteria for reusable Smalltalk software; their results are not likely to be very useful for existing software libraries, most of which are in COBOL,

FRAN or C. The authors themselves point out that the logical extension of their ideas implies the establishment of documentation standards, only now beginning to emerge in conjunction with OOPS and ADA.

In another text analysis project, Creecy et al [1992] extracted industry and occupation descriptors from US census data in natural language. They used the memory-based reasoning (MBR) paradigm of Stanfil and Waltz [1986] to assign weights to various terms occurring in the free text descriptions. Their PACE system combines single words with three numeric descriptors to produce new conjunctive features that proved to be improved predictors. After processing a set of training examples, PACE was used to classify industry and occupational categories. Criteria were both accuracy and coverage, with coverage defined as the percentage of terms that could be classified with a predefined certainty threshold and therefore did not need manual intervention. When results were compared with those of a rule based expert system, they were found to be better both in terms of classification accuracy (by 13%), and in the time taken to set up the system (4 person months vs. 192).

Library cataloguers were the first users of automated indexing, on titles and abstracts [Ellis, 1990]. Many text analysis ideas have been drawn from the fields of library science and information/document retrieval. This suggests that the concepts we have applied to the software reuse problem are likely to have counterparts in information retrieval theory. (For a more complete description of the current information and text retrieval research see Frakes [1993] or Meadow [1992].)

Previous authors have exploited these parallels to build reuse systems that combine information retrieval methods with features from areas such as database and pattern recognition [Maarek et al 1991, Fraser et al 1989, Frakes and Gandel 1990, Frakes and Nejme 1988]. Maarek et al use comments as well as other text documents to extract multi term patterns of words by use of 'lexical affinity'. The current study has found that most information retrieval concepts can be implemented within the framework of a single, standard Case-Based Reasoning shell which provides many other tools and facilities to help build a reuse system with a complete domain model.

REFERENCES

- Aha, D.W. [1989]. *Incremental, Instance Based Learning of Independent and Graded Concept Descriptions*, Proceedings of the Sixth International Workshop on Machine Learning, 387-391, Morgan Kaufmann, Ithaca, NY.
- Aha, D.W. [1993]. Private Communication.
- Aha, D.W. and Goldstone, R.L. [1990]. *Learning Attribute Relevance in Context of Instance Based Learning Algorithms*, Proceedings of the XII Annual Conference of the Cognitive Sciences Society, 141-148, Erlbaum, L. (ed), Cambridge, MA.
- Almuallim, H and Dietterich, T.G. [1992]. *Efficient Algorithms for Identifying Relevant Features*, Proceeding of the Canadian AI Conference, 38, Victoria, BC.
- Arrango, G. [1989]. *Acquiring Knowledge for Specifying the Architecture of Libraries of Reusable Parts*, 3rd European Knowledge Acquisition for Knowledge Based Systems Workshop, 43, Paris.
- Bareiss, E.R. [1989]. *Exemplar Based Knowledge Acquisition*. Academic Press, San Diego, CA.
- Biggerstaff, T.J. and Richter, C. [1989]. *Reusability Framework: Assessment and Directions*. In Software Reusability Vol. I-Concepts and Models, 19, Biggerstaff, T.J. and Perlis, A.J. (eds), ACM Press, New York.
- Booch, G. [1987]. *Software Components with ADA*. Benjamin-Cummings Publishing, Inc., Menlo Park, CA.
- Cain, T, Pazzani, M.J. and Silverstein, G [1992]. *Using Domain Knowledge to Influence Similarity Judgments*, Proceedings of the CBR Workshop, Sponsored by DARPA Information Science and Technology Office, 191.
- Cognitive Systems Inc. [1992]. *REMIND Manual*. Cambridge, MA.
- Cost, S., & Salzberg, S. [1990]. *A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features*, Technical Report JHU-90/11, The Johns Hopkins University, Department of Computer Science. Baltimore, MD:
- Creedy, R.H., Masand, B., Smith, S.J. and Waltz, D.L. [1992]. *Trading MIPS and Memory for Knowledge Engineering*, Communications of the ACM, 35:9. 48.
- Dasarathy, B.V. [1991]. *Nearest Neighbor Norms- Nearest Neighbor Patterns Classification Techniques*. IEEE Computer Society Press, Los Alamitos, CA.

- Delisle, S. [1990]. *A Parser for Processing Technical Texts with a Large Coverage of English*, Technical Report TR 90-25, University of Ottawa, Department of Computer Science.
- Dongarra, J.J. [1979]. *LI. PACK: Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia.
- Ellis, D. [1990]. *New Horizons in Information Retrieval*. Butterworth, London.
- Esteva, J.C. and Reynolds, R.G. [1990]. *Learning to Recognize Reusable Software by Induction*, SPIE, v. 1293, Applications of Artificial Intelligence, V III.
- Fisher, D.H. [1987]. *Knowledge Acquisition via Incremental Concept Learning*, Machine Learning, Vol 2, 139-172, Kluwer Academic Publishers, Boston.
- Fouque, G. and Matwin, S. [1992]. *Compositional Software Reuse with Case-Based Reasoning*, Technical Report TR 92-30, University of Ottawa, Department of Computer Science.
- Fouque, G. and Matwin, S. [1993]. *A Case-Based Approach to Software Reuse*, Technical Report TR 93-4, University of Ottawa, Department of Computer Science.
- Frakes, W.B. and Nejme, B.A. [1988]. *An Information Systems for Software Reuse*, in Tracz W. ed. *Software Reuse: Emerging Technology*, IEEE Computer Society Press.
- Frakes W B [1993]. *Introduction to Information Storage and Retrieval*, Information Retrieval: Data Structures and Algorithms, Frakes W B and Baeza-Yates Prentice Hall, Engelwood Cliff, NJ.
- Frakes, W.B. and Pole, T.P. [1991]. *PROTEOUS: A Software Reuse Library Systems that Support Multiple Representation Methods*, SIGIR Forum, Vol. 24, Issue 3, 43-55.
- Frank, W.L. [1981]. *Software Productivity: Any Breakthroughs?*, Computerworld, July Issue.
- Fraser, S.D., Duran, J.M. and Aubin, R. [1989]. *Software Indexing for Reuse*, Proceedings of IEEE International Conference on Systems, Man and Cybernetics, 853, IEEE Computers Press, Cambridge, MA..
- Holte, R [1993]. Private Communication.
- Horowitz, E. and Munson, J.B. [1989]. *An Expansive View of Reusable Software*. In *Software Reusability Vol. I-Concepts and Models*, 19, Biggerstaff, T.J. and Perlis, A.J. (eds), ACM Press, New York.
- IMSL [1987]. *IMSL Math Library User's Manual*, 1st ed. Houston, TX.
- Krueger, C.W. [1992], *Software Reuse*, ACM Computing Surveys, Vol. 24, No. 2.

- Laurendeau, C. [1992]. *Automated Extraction of Noun Phrases from Technical Text*. Masters Thesis, University of Ottawa, Department of Computer Science.
- Maarek, Y.S., Berry, D.M., Kaiser, G.E. [1991]. *An Information Retrieval Approach for Automatically Constructing Software Libraries*, IEEE Transactions on Software Engineering, Vol. 17, No. 8, 800.
- Meadow, C. T. [1992]. *Text Information Retrieval Systems*, Academic Press Inc., San Diego, CA.
- McIlroy, M.D. [1967]. *Mass Produced Software Components*, In Software Engineering Concepts and Techniques, NATO Conference on Software Engineering, Buxton, J.M., Naur, P. and Randell, B. (eds), 88-98.
- Michalski, R.S. and Stepp, R.E. [1983]. *Learning from Observation: Conceptual Clustering*. In Machine Learning: An Artificial Intelligence Approach, Michalski, R.S., Carbonell, J.G., and Mitchell, T.M. (eds), Morgan Kaufmann, Los Altos, CA.
- Pazzani, M.J. [1989]. *Creating a Memory of Causal Relationships: An Integration of Empirical and Explanation Based Learning Methods*, Lawrence Erlbaum, Hillsdale, NJ.
- Pierce M, Goel A K, Kolodner J L, Zimring C, Sentosa L, Billington R [1992]. *Case-Based Design Support: A Study in Architectural Design*", IEEE Expert, p 14, October.
- Porter, B, Bareiss, R. and Holte, R.C. [1990]. *Concept Learning and Heuristic Classification in Weak Theory Domains*, Artificial Intelligence, 45:1-2; North Holland, Amsterdam.
- Prieto-Diaz, R. [1985]. *Software Classification Scheme*. Ph.D. Thesis, University of California-Irvine, Department of Computer Science.
- Prieto-Diaz, R. [1989]. *Classification of Reusable Modules*. In Software Reusability Vol. I—Concepts and Models, 19, Biggerstaff, T.J. and Perlis, A.J. (eds), ACM Press, New York.
- Ranganathan, S.R. [1967]. *Prolegomena to Library Classification*. Asia Publishing House, Bombay, India.
- Rasmussen E [1993]. *Clustering Algorithms*, Information Retrieval: Data Structures and Algorithms, Frakes W B and Baeza-Yates Prentice Hall, Engelwood Cliff, NJ.
- Russell, S.J. [1989]. *Use of Knowledge in Analogy and Induction*. Pitman, London.
- Salton, G. [1975]. *A Theory of Indexing*. Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, PA.

- Salton, G. [1988]. *Syntactic Approaches to Automatic Book Indexing*. Proceedings of the 26th Annual Meeting of the Association of Computational Linguistics: 206.
- Schlimmer, J.C. [1992]. *Efficiently Inducing Determinations.*, Proceedings of the Tenth International Conference on Machine Learning, 284-290.
- Sparck Jones, K. [1972]. *A Statistical Interpretation of Term Specificity and its Application in Retrieval*, Journal of Documentation, 11-21, 28.
- Sparck Jones, K. [1971]. *Automatic Keyword Classification for Information Retrieval*. Butterworth, London.
- SPSS [1988]. *SPSS-X User's Guide*. 3rd ed., Chicago, Ill.
- Srinivasan, P. [1993]. *Thesaurus Construction*, In Information Retrieval: Data Structures and Algorithms, 161, W. Frakes and R. Baeza-Yates Ed., Prentice-Hall, Englewood Cliffs, NJ.
- Stanfel, C. and Waltz, D. [1987]. *Towards Memory-Based Reasoning*, Communications of the ACM, 1213-1228, 29:12.
- van Rijsbergen, C.J. [1979]. *Information Retrieval*, Butterworth, London.
- van Rijsbergen, C.J. [1981]. *Retrieval Effectiveness*, Information Retrieval Experiment, K. Sparck Jones (ed), Butterworth, London.
- Wegner, P. [1989]. *Capital-Intensive Software Technology*. In Software Reusability Vol. I-Concepts and Models, 19, Biggerstaff, T.J. and Perlis A.J. (eds), ACM Press, New York.

APPENDIX I

AZTEC

The basic structure of the AZTEC system is displayed in FIGURE VI in the first Chapter. Here we describe, in little more detail, the part of functionality of AZTEC that is needed for the current work. The discussion in this Appendix is largely concerned with provision of details needed to understand this diagram. For additional details about AZTEC the reader should consult a copy of her thesis.

AZTEC Objectives

The stated objectives of the AZTEC system are automated acquisition of technical concepts from English texts. The AZTEC output contains the noun phrases extracted from the text with three grades of technical rating : 'technical', 'rather technical' and 'possibly technical'. Two additional categories, 'non-technical' and 'unknown', then complete AZTEC classification scheme. The AZTEC system consists of a number of separate modules integrated to perform this classification. In the following we will describe the functions of these modules in the order in which they are generally invoked by AZTEC.

The Text Base Builder:

One role of this module is to prepare the text in a form suitable for the Sentence Parser. In this process, it builds a text hierarchy with Chapters in the highest node and sentences as leaves. The output from each level of the hierarchy is stored in a separate file. For use in AZTEC the only file of interest is the sentence file. The version of Text Base Builder used for the AZTEC is very picky as to characteristics of the input text. The text extracted from comments, for example, must be extensively cleaned to be in a form acceptable to this module. As an example of the extensive pre-processing needed to perform the cleanup we produce a sample of the final AZTEC-ready input in Table II. The raw comments from where this is derived are reproduced in Table I.

TABLE I*

- * Subroutine * int sppfa_(ap, n, info)*
- * System generated locals *
- * Builtin function declarations commented out ---
- * supplied by f2cinc1.h instead
- * Local variables
- * sppfa factors a float symmetric positive definite matrix
- * stored in packed form *
- * sppfa is usually called by sppco, but it can be called
- * directly with a saving in time if rcond is not needed
- * (time for sppco) = (1 + 18/n)*(time for sppfa) *
- * on entry ** ap float (n*(n+1)/2)
- * the packed form of a symmetric matrix a the
- * columns of the upper triangle are stored sequentially
- * in a one-dimensional array of length n*(n+1)/2
- * see comments below for details
- * n int
- * the order of the matrix a
- * on return
- * ap an upper triangular matrix r, stored in packed
- * form, so that a = trans(r)*r *
- * info int
- * = 0 for normal return
- * = k if the leading minor of order k is not
- * positive definite
- * packed storage
- * the following program segment will pack the upper
- * triangle of a symmetric matrix
- * k = 0
- * do 20 j = 1, n
- * do 10 i = 1, j
- * k = k + 1
- * ap(k) = a(i,j) * internal variables *
- * begin block with exits to 40
- **
- * Parameter adjustments
- * 10 continue
- * 20 continue
- * linpack, this version dated 08/14/78
- * cleve moler, university of new mexico, argonne national lab.
- * subroutines and functions
- * blas sdot
- * fortran sqrt
- *

SAMPLE OF COMMENTS EXTRACTED
FROM THE LINPACK ROUTINE SPPFA

TABLE II

NAME factors a float symmetric positive definite matrix stored in packed form.
NAME is usually called by NAME but it can be called directly with a saving in time if NAME is not needed.
(time for NAME is equal to time for NAME .
NAME is the packed form of a symmetric matrix a .
the columns of the upper triangle are stored sequentially in a one-dimensional array of length $n(n+1)/2$.
n is the order of the matrix a .
NAME is an upper triangular matrix r , stored in packed form, so that a is equal to NAME r)
info is equal to 0 for normal return.
it is equal to NAME if the leading minor of order NAME is not positive definite.
packed storage.

TEXT EXTRACTED
FROM TABLE I
READY FOR INPUT TO AZTEC

The other function that the Text Base Builder performs is the creation of an augmented lexicon of all of the words appearing in the input text. In addition to the lexical information, the augmented lexicon contains the word frequencies as well as a 'dictionary flag'. The flag records, for each entry, presence (flag = 0) or absence (flag = 1) of the word in 'Longman's Dictionary of Common English Words'. (This dictionary contains all the words necessary to understand the meanings of entries in the 'Longman's Dictionary of the English Language'.)

Extract Text Sentences Module

The sentence file produced by the Text Base Builder contains many sentences fragments not acceptable to the Sentence Parser Module that is invoked next. AZTEC must first extract well-formed sentences from this file for the Sentence Parser. Towards this end the sentence file has to be processed by a utility called 'Extract_Text_Sentences' that creates a new file called 'Block_for_Classification'

Sentence Parser

The Sentence Parser Module in AZTEC was implemented for an earlier project (TANKA) by Delisle [1991]. The Parser is invoked first to validate the grammatical completeness of the sentence. Once the sentence passes this test its syntactic analyzer determines, from its word dictionary, the syntactic categories of its component parts. These categories are then used to come up with its syntactic structure that identifies the noun phrases. This requires us to ensure that each one of its words have at least one entry within the dictionary.

Syntactic Category Analyzer within AZTEC

If the Sentence Parser comes across a word for which it has no syntactic category, it will prompt the user to provide it interactively. While this is quite acceptable for small text files, the larger files may take hours to process requiring the user to be 'baby sitting' the run. To overcome this problem AZTEC developed its own utility to preprocess the text for unknown words. This step allows the user to complete the syntactic categorization of all of the words before invoking the Sentence Parser.

Classification Module

The Classification module is the heart of the AZTEC program and the part specifically implemented for that project. AZTEC supports two modes of noun phrase classification namely:

- one sentence at a time,

and

- a whole block of text.

The algorithm for classification by sentence is quite different from classification by block. Since we only used the block classification algorithm in this work, we will not discuss the sentence classification any further.

The classification of noun phrases is a two-step process: extracting NPs and classifying NP's. The extraction is performed via an AZTEC interface to the Sentence Parser called 'NP Seeker' that extracts certain class of NP's from the sentence structure.

Once NP's have been extracted, the classification task can begin. The NP's are first assigned a tentative class by analysis of the frequencies of their occurrences and those of their head nouns. The final technical categories are determined from the tentative once by taking the Longman's dictionary flag into account. The tentative category is increase toward more technicality if flag is 0 and decreased otherwise.

To distinguish the five categories of noun phrases AZTEC needs additional parameters. These include the three thresholds that are needed to define the technical categories. As well we need a minimum NP occurrence level below which the Classification algorithm can not safely classify and must declare the term as of unknown technicality. These bounds are not known a priori and some tuning can be done for specific body of text to shift the boundary between the categories. The values chosen for our experiment were as follows:

Possible Categories	Term Frequency
Tech. Rather Tech.	<10 %
Tech. Rather Tech. Possibly Tech.	1.5%- 10 %
Rather Tech. Possibly Tech. Non Tech.	.75% - 1.5%
Unknown	Single Occurrence

These bounds represent percentage term frequency calculated as ratio of head noun frequency over the total term frequency. The noise threshold is a single word implying that AZTEC only processes the NP if it occurs more than once in the text. We note that multiple categories are possible for each frequency range. This is because actual category is determined by modifying the tentative category to account for the dictionary flag of the head noun.

There are features of AZTEC that were implemented to overcome some of the shortcomings of the Sentence Parser. The Parser represents the processing bottleneck of the AZTEC system that often takes minutes for parsing a single

sentence. To overcome this difficulty AZTEC sets a maximum time limit per sentence for Sentence Parser. This time out was set at 5 seconds for the current implementation. The sentence that does not get parsed in this time has to be parsed by a separate algorithm by AZTEC. This algorithm can also work on incomplete sentences that are also rejected by the Parser.

APPENDIX II

REMIND

For this project, we selected a commercially available Case Base Reasoning shell called REMIND. The package is available from Cognitive Systems for variety of platforms including UNIX, Macintosh and DOS. It has a number of features that have been exploited by other CBR researchers [Pierce et al, 1992]. In what follows we will describe those features that are deemed useful for reuse application. For a complete description of the shell we refer the reader to the 'REMIND User Manual' [REMIND 1992].

REMIND Datatypes

REMIND supports most common datatypes including Boolean, numeric, text and symbol. In addition REMIND supports a 'case' datatype. The datatypes can be single or in form of variable length lists. In particular, 'list of symbols' datatype is supported that is especially well suited for the description of the software components. We have shown in Chapter III, how various facets of software components are represented by a variable length list of keywords (symbols).

Symbol Tree

REMIND can represent various types of relationships that are possible between symbols. The symbols, for example, are allowed to have one or more children. In REMIND it is possible to represent a complete taxonomy of the domain via a 'symbol tree'. The 'symbol tree', however, is not an inheritance hierarchy but rather simply a family tree. As well, REMIND symbols can be descendants of multiple parents. This feature is useful to relate a concept to more than one higher level concept. We found that this multiple parentage is a natural way to represent the relationships between various descriptors of software artifacts. One shortcoming of a REMIND's symbol tree links is that they can not model partial relationships. This is unlike Prieto-Diaz's 'conceptual closeness graph' where the arcs have weights that determine the strength of the relation. The siblings in a symbol tree, on the other hand, can have ordering that in some cases are useful in representing intra-node relations. We selected 'list of symbols' datatype as the best representation of the domain related facets of a software component.

REMIND'S Matching Algorithm

Matching algorithms for various field types such as Boolean, text and numeric are described in the user manual. We only concern ourselves here with how the matching works for a 'list of symbols' field. Let us consider the matching for the list for j-th attribute for target T to another similar list for source S. The first step is to expand the T and S to include all of the ancestors of the members of S and T. This results in two expanded lists:

S' and T'

that have to be compared to get attribute similarity for the j-th attribute:

$$S(j) = 100 * \frac{2 * ||T' \cap S'||}{||T'|| + ||S'||}$$

where parallel bars indicate the list size and where $\cap$ represents the intersection. We note that this is a special case of Dice Coefficient used in information retrieval [Rasmussen 1993]. We also note that this is only an attribute level similarity. To get case-to-case similarity SS we must sum these with appropriate weights and normalization:

$$SS = \sum (w(j) * S(j)) \quad j = 1, \dots, N$$

where $w(j)$ is the normalized weight assigned to the j-th attribute and N is total number of attributes.

Qualitative Modeling

We can also exhibit relationships between attributes by use of another REMIND feature called 'quantitative modeling'. In building a qualitative model we define new 'virtual' non measurable attributes or concepts that depend on observable attributes. This facility is used in Chapter III (Figure XI) to define virtual attributes called 'code quality' and 'algorithm quality' that in turn depend on

known measurable attributes such as 'lines of code'. Figure XI shows how these 'virtual nodes' figure into calculation of reusability. We should note that qualitative modeling allows for conditional relations for example if the algorithm is iterative then the accuracy and speed are inversely related. The mode also allows for strengths to be associated with the relational arcs that is reminiscent of the Prieto-Diaz's 'conceptual distance graphs.

Clustering

The serial matching for the nearest neighbor retrieval is very slow as it increases linearly with the number of cases in the case base. Partly to overcome this problem, REMIND allows for a knowledge guided or inductive clustering of cases. REMIND applies well-known ID3 clustering algorithm to generate a classification tree that stores the cases in its leaves. Another way in which the defining clustering can be done is via use prototypes or class representatives around which various similar cases can be clustered [Arrango 1989]. These tree based storage schemes results in great efficiency of retrieval without sacrificing the accuracy by much. However, if one is looking for 100% certainty of retrieving all nearest neighbors then there is no alternative but to serial matching. In reuse scenario this type of certainty seldom if ever needed. Another disadvantage of the tree storage is that substantial new additions to the case library require re-clustering.

APPENDIX III

Matrix Algebra and Linear Systems

We have chosen a collection of subroutines from the domain of linear algebra for studying our reuse model. The application of linear algebra algorithms is universal, cutting across various fields in mathematical and social sciences and in engineering. The algorithms are well known and well understood. As well, the source code of various linear algebra libraries is easily available in machine readable form. For these reasons, other researchers in software reuse have focused on the linear algebra domain [Fouque and Matwin 1992, 1993]. However, since the field is quite vast we restricted ourselves to one of its branches namely the linear systems theory. The selection of a smaller sub-domain allows us to keep the study to a manageable proportion. Since the linear system theory can be treated essentially as a closed domain, we can model it within the limits of our hardware and software. Even with this restricted domain there is a multiplicity of algorithms in giving us an opportunity to draw from a rich source for modeling and testing.

Libraries of Linear System Software and their Organization

As mentioned in the Introduction, (mostly FORTRAN) libraries of scientific subroutines were available almost from the early days of the third generation languages. Most notable of early libraries was IBM's Scientific Subroutine Library developed in Zurich. All of these libraries have a substantial number of linear systems routines. The popular general purpose libraries of scientific/mathematical software include IMSL [1987] in the US and NGC in the UK. Both have separate modules containing linear system programs. In addition to these, many large commercial organizations and academic institutions have their own inhouse routine libraries with large number of linear system programs. From the many choices available to us, we selected 'C' version of one of the more popular packages, LINPACK [Dongarra et al, 1979]. While there were many other advantages in choosing LINPACK for testing the model, the major reason was quality of its comments. This quality is reflected in certain consistency and homogeneity across the collection that can be exploited for analysis of comments.

LINPACK Subroutine Library

Historical Background

In the early seventies the US government funded a large scale effort in developing linear algebra subroutine libraries. These included three major projects :

- Basic Linear Algebra Subroutines (BLAS),
- Eigensystem Subroutines (EISPACK)
- Linear Systems Routines (LINPACK)

The work was done in various computing laboratories with Argonne National Laboratories coordinating the overall efforts. The routines were developed in FORTRAN on one machine and then the source code was converted for distribution for other hardware. These routines have since been translated into other languages including 'C'. Over the past two decades, LINPACK routines have been distributed to a very wide population of users. These are available on different hardware platforms including personal computers. For the current project we have chosen the IBM PC version of LINPACK source code in 'C' distributed by Rogue Software. This library contains 31 core programs that have extensions for users needing double precision or complex calculations. The versions were generated by a program transformation tool called TAMPR from a single master version written for complex matrices [Dongarra, 1979]. This effort is one of the earliest examples of a generational reuse system.

Comment Standards

As we noted above, the authors of these routines have followed a fairly standardized comment methodology. Each of the routines is preceded by a comment box that describes the algorithm in free form text. The box also contains some fields including calling and called routines, variables' identifiers and other 'tombstone data'. In addition to the boxed comments there are inline comments that describe the flow of the program by using brief sentences. These comments expand upon the algorithmic descriptions that are contained in the boxed comment area. There is some commonalty between various comments that is indicative of similarity between the code.

Subroutine Classification Scheme

The LINPACK uses a classification scheme that is also basis for its naming convention (see Table I). The dimensions of classifications are based on the following:

- type of input matrix (inputs)
- the operations on input matrix (outputs)
- type of arithmetic used (single /double precision and real/complex)

There are two classes of routines that are needed for linear systems computations. The first kind is used to simplify the problem by either factoring or decomposing the original system (or input matrix). The routines of the second type are used to actually solve the linear system or to perform calculations on the input matrices. The terms that LINPACK uses to describe its routines are:

LINPACK DESCRIPTORS

Object	Processes
General	Factor
Band	Estimate Condition
Positive Definite	Solve
Triangular	Determinant/Inverse/Inertia
Packed	Decompose
Tridiagonal	Update
Indefinite	Downdate
Symmetric	Exchange
(Hermitian)	Cholesky Decomposition
(Complex)	Orthogonal Decomposition
(Real)	Singular Value Decomposition

For our experiments we only used the single precision subset of 31 routines so the bracketed terms do not apply. Table II is derived from the LINPACK manual [Dongarra, 1979] to show the list of all the single precision routines for real matrices that form part of the library. We also do not show the 7 decomposition routines in this table that were also analyzed for this project. The complete library LINPACK library contains double precision and two complex versions of the list. Note as well that, the LINPACK naming convention is derived from this table.

LINPACK SINGLE PRECISION ROUTINES

	CONDITION	SOLVE	FACTOR	DETERMINANT
	(CO)	(SL)	(FA)	(DI)
GENERAL BAND (SGE)				
POSITIVE DEFINITE (SPO)				
POSITIVE DEFINITE PACKED (SPP)				
POSITIVE DEFINITE BAND (SPB)				
SYMMETRIC INDEFINITE (SSI)				
SYMMETRIC INDEFINITE PACKED (SP)				
TRIANGULAR (STR)		NA		
GENERAL TRIDIAGONAL (SGT)		NA	NA	NA
POSITIVE DEFINITE TRIDIAGONAL (SPT)		NA	NA	NA

LIST OF FIGURES

Figure I	Prieto-Diaz Reuse Algorithm	10
Figure II	Conceptual Distance Graph	13
Figure III	Implementation of Prieto-Diaz's Method	13
Figure IV	Domain modeling Methodology	18
Figure V	Semi-Automated Creation of the Case Base	23
Figure VI	AZTEC Algorithm	25
Figure VII	Pre-Processing Input for AZTEC	27
Figure VIII	Building Term Lists	28
Figure IX	Building the Case Base in REMIND	31
Figure Xa	Partial Symbol Tree for MODEL I	33
Figure Xb	Partially Expanded Symbol Tree for MODEL II	34
Figure Xc	Partially Expanded Symbol Tree for MODEL III	35
Figure XI	Qualitative Model	45
Figure XII:	Comparing the Prieto-Diaz Method to REMIND / CBR	57

LIST OF TABLES

Table I	Definition of the Domain Models	36
Table II	Relevance Table	39
Table III	Comparing AZTEC and Manual Lists	49
Table IV	NP Distribution by Length	49
Table V	Distribution of LINPACK Terms in AZTEC Results	50
Table VI	Results of Increasing Symbol Tree Levels	53
Table VII	Evaluation of Retrieval Performance	54
Table VIII	Comparing REMIND and PROTEUS	64

GLOSSARY OF TERMS

AZTEC	24
Conceptual Distance Graph	12
Cost-Benefit Function	41
Data Import (REMIND's)	36
Effectiveness (retrieval)	41
Enumerated Classification	11
Faceted Classification	11
Importances (in REMIND)	37
IMSL Library	11
LINPACK Subroutine Library	80
Matching Algorithm (REMIND's)	77
Precision	38
Prieto-Diaz Reuse Model	10
Qualitative Modeling (in REMIND)	77
Recall	38
Relevance	19
Reusability Metric	44
Sentence Parser (in AZTEC)	72
Similarity Threshold	15
SPSS Library	11
Symbol Tree (REMIND's)	76
Technicality Ratings (in AZTEC)	24
Text Base Builder (AZTEC)	70

END

2 2-1 1-9 6

FIN