Automatic Goal Extraction From User Actions To Accelerate The Browsing Of Software Libraries

by

C. Drummond

A THESIS

submitted to the School Of Graduate Studies and Research

in partial fulfillment of the requirements

for the degree

Master Of Applied Science

in

Electrical Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering

Faculty of Engineering

University of Ottawa

OTTAWA, ONTARIO K1N 6N5

©1992, Chris Drummond, Ottawa, Canada

Acknowledgments

There is only one author to a thesis, perhaps there should be more. It is hard to imagine that any research, especially at the graduate level, is done in isolation. I would like to thank all the people who have contributed. I particularly thank my supervisors: the pragmatism of Electrical Engineering courtesy of Dr. D. Ionescu to temper the imagination of Computer Science courtesy of Dr. R. Holte. I thank the researchers of the Ottawa Machine Learning Group Of Computer Science particularly Dr. S. Matwin, Dr. D. Duchier, Dr. G. Fouque, Dr. P. Clark, H. Brahim, T. Rovel, D. Charlebois and S. Lapointe for all their suggestions.

Abstract

This research addresses the problem of locating software items in extensive libraries. It aims to increase the speed and accuracy with which a user may browse software libraries for reusable code. The method proposed for this is called active browsing. The system monitors user actions, made within a normal browser, to infer an analogue representing the user's search goal. A relevancy measure is constructed from this analogue and used by the system to scan the library independently of the user and to evaluate potentially interesting components. The results affect the browser display to emphasize relevant components and thus aid search.

Although the main interest is software reuse, the approach has a much wider applicability. This is inherited directly from the broad applicability of browsing. Browsing is an important methodology particularly in search tasks where the target is not well defined. This thesis discusses a model of active browsing applicable in general to browseable libraries.

An implementation of this model, based around a browser used to explore libraries of object oriented code, is described. In this implementation, an inference engine forward chains on rules to generate the analogue from the user's actions. A relevancy measure, in the form of a template, is constructed , used to assign a score to library items and to produce a ranked list of classes of potential interest to the user. This implementation is used both to illustrate active browsing in a particular application and to experimentally validate the model.

Contents

Acknowledgments . . . . . **ii**

Abstract . . . . . **iii**

List of Tables . . . . . **viii**

List of Figures . . . . . **viii**

1 Introduction **1**

1.1 Software Reuse 3

1.2 Locating Software In Large Libraries 4

1.3 Improving Basic Browsing 5

1.4 Research Contributions 7

1.5 Thesis Overview 8

2 Reuse And Browsing: An Overview **9**

2.1 Generative Approaches 9

2.2 Compositional Approaches 11

2.2.1 Indexing 13

2.2.1.1 Manual Assignment Of Indices 13

2.2.1.2 Automatic Assignment Of Indices 15

2.2.1.3 Limitations Of Indexing 16

2.2.2 Browsing 17

2.3 Conclusions 19

3 Active Browsing: An Abstract Model: 21

3.1 View Of Normal Browsing 21

3.2 View Of Active Browsing 23

3.3 Generalized Architecture 25

3.4 Form Of The Analogue And Relevancy Measure 26

3.4.1 The Relevancy Measure 27

3.4.2 The Analogue 28

3.5 Computing The Analogue And Relevancy Measure 29

3.5.1 Inferring A System Analogue 30

3.5.1.1 Incremental Additions 31

3.5.1.2 Generalizing Subgraphs 32

3.5.1.3 Pattern Specific Inferences 33

3.5.2 Inferring A Relevancy Measure 34

3.6 Influencing Search 35

3.7 Obstacles To Active Browsing 36

3.7.1 The Problem Of Ambiguous Actions 36

3.7.2 Goal Maintenance 39

3.8 Conclusions 40

4 An Implementation For An Object Oriented Program Library 41

4.1 The Standard Browser 43

4.1.1 The Object Oriented Library 43

4.1.2 The Browser 46

4.1.2.1 Class Relationships 47

4.1.2.2 Lower Level Relationships 50
4.1.2.3 Program Understanding 52
4.2 Independent Search Mechanism 55
4.2.1 The Inference Process 55
4.2.1.1 Processing Browsing Actions 56
4.2.1.1.1 Data Processed 56
4.2.1.1.2 Implementation 58
4.2.1.2 Inferring Template Predicates 61
4.2.1.2.1 Data Processed 61
4.2.1.2.2 Implementation 68
4.2.2 Template Matcher 70
4.2.2.1 Data Processed 71
4.2.2.2 Implementation 73
4.3 Conclusions 77
5 Experimental Results 79
5.1 Overview Of Experimental Rationale 79
5.2 Plausibility 80
5.2.1 Focusing The Search 81
5.2.2 Expanding The Search Area 84

5.3 Generality and Effectiveness 89
5.3.1 The Simulator Of Human Browsing Patterns 91
5.3.1.1 The Black Box 91
5.3.1.2 The Heuristic Agent 93
5.3.1.3 The Experiments 95
5.3.2 Experimental Results 97
5.3.2.1 Assigning Wins 97
5.3.2.2 Alternative Performance Measures 100
5.3.2.2.1 Performance Against Steps Into Search 102
5.3.2.2.2 Performance Against Length Of search 104
5.3.3 Discussion And Analysis Of Results 108
5.4 Conclusions 110
6 Conclusions 112
6.1 In Summary 112
6.2 Limitations 114
6.3 Useful Machine Learning Techniques 117
6.4 Future Work 119
7 Bibliography 120
Appendix A Meta-Rules 126
Appendix B Search Rules 129
Appendix C Program Understanding Rules 132

List of Tables

Table 1 Upper Bound On Wins 98

Table 2 Lower Bound On Wins 98

List of Figures

Figure 1 Normal Browsing 22

Figure 2 Active Browsing 23

Figure 3 Generalized Architecture 25

Figure 4 Types Of User Actions 30

Figure 5 Levels Of Abstraction 32

Figure 6 System Architecture 41

Figure 7 Human Interface To The Active Browser 42

Figure 8 An Objective-C Library Item 43

Figure 9 The Inheritance Hierarchy 44

Figure 10 Control/Data Flow Graph 45

Figure 11 The Browser Interface 47

Figure 12 Class Menu Options 48

Figure 13 Classes With Similar Names 48

Figure 14 Classes With Similar Interfaces 49

Figure 15 A Single Node For A Class' Source Code 50

Figure 16 Search At A Lower Level Of Abstraction 51

Figure 17 Low Level Search Operators 52

Figure 18 Options In Programming Understanding 54

Figure 19 A Sequence Of Browsing Actions 57

Figure 20 Action Processor 59
Figure 21 Code For Filling The Schemata Slots 60
Figure 22 Meta-Rules 63
Figure 23 Search Rules For Incremental Additions 65
Figure 24 Search Rules For Pattern Specific/Generalization Inferences 66
Figure 25 Program Understanding Rules 67
Figure 26 The Inference Engine 69
Figure 27 Code For Task Predicates 70
Figure 28 Code To Select Subgroup of Search Rules 71
Figure 29 Matching Templates To Classes 72
Figure 30 Template Matcher 74
Figure 31 Code To Cycle Through Template Class Names 75
Figure 32 Code To Calculate Match Score On Class Names 76
Figure 33 Suggestion Box 78
Figure 34 The Experimental Methodology 79
Figure 35 Exploring Two Different Classes 82
Figure 36 Predicates After Exploring Class “String” 82
Figure 37 Predicates After Exploring Class “Layer” 83
Figure 38 Suggestions 84
Figure 39 Class Hierarchy 85
Figure 40 Exploring The First Sibling 86
Figure 41 Predicates After Visiting The First Sibling 87
Figure 42 Exploring The Second Sibling 88

Figure 43 Predicates After Visiting The Second Sibling 88
Figure 44 Suggested Classes 89
Figure 45 The Simulator Of Human Search Patterns 92
Figure 46 Heuristic Search Algorithm 94
Figure 47 Experimental Procedure 96
Figure 48 Record Of Ranking During Search 96
Figure 49 Wins For Zero Noise 100
Figure 50 Wins For Medium And Highest Noise 101
Figure 51 Zero Noise: Steps Into Search 102
Figure 52 Medium Noise: Steps Into Search 103
Figure 53 Highest Noise: Steps Into Search 104
Figure 54 Difference In Rank: Medium Noise 106
Figure 55 Difference In Rank: Zero Noise 107
Figure 56 Difference In Rank: Highest Noise 108

Chapter 1 Introduction

The aim of this research is to reduce the cost of locating reusable software in large libraries. The speed and accuracy of browsing is increased by a method termed active browsing. The user browses the library in the usual manner while the system searches in parallel for items of relevance to the user's search goal. The novelty of this approach is that this goal is inferred directly from such actions requiring no other input from the user.

The thesis presents a general model of active browsing. A specific implementation is described and validated using a library of software written in Objective-C, an object oriented programming language. This model is applicable to many browsing tasks. Browsing is an important means of search that has been applied to knowledge bases, document bases, object oriented and even relational databases, as well as more inhomogeneous structures within hypertext. This approach aims at maintaining the broad applicability of browsing while increasing its effectiveness.

The exact role and importance of browsing, and therefore active browsing, in locating items in a library can be best understood by comparing it to the main alternative — indexing. Indexing requires the construction of a query in an indexing language, possibly a natural language. The query is used to locate the relevant component, usually by moving through an indexing structure and matching against an item's index. Some queries are in the form of boolean equations, others use a more flexible matching systems using weighted vectors of terms. The latter, by ranking returned items according to the degree of match, overcomes a problem associated with boolean queries of returning too few or too many items. These methods can be further enhanced by relevance feedback where the user is required to critique

the items returned by the query [31]. The system can use this information to adjust the matching procedure to find more relevant items.

Indexing methods are appropriate only if the user has a good goal definition. Without a good goal definition the user must browse. Fischer et al. state [22], when describing natural language interfaces for information retrieval, " ..... they do not assist users who are unable to describe precisely what they want at the beginning of an information seeking process". Poor goal definition also occurs when the library does not contain exactly what the user wants. In this case the aim of the search is to find an item that best matches the user requirements. So even if the requirements are well defined the goal is not.

Studies [8] have shown that when given the choice, browsing is preferred over indexing by many users. Overall, browsing is seen as a more natural and effective process when the user is uncertain of the target description. This is further supported by Fischer et al. [22] in the statement "This (human remembering) theory postulates that people naturally think about categories of things not in terms of formal attributes but in terms of examples". The two methods, browsing and indexing, are by no means mutually exclusive. If the user can accurately describe the whole or even a significant part of the target, indexing is the more powerful approach. Ideally indexing and browsing facilities would be incorporated in the same system.

Active browsing, being based around a normal browser, is applicable in the same situations. It takes as input normal user actions. Its output is subtle changes in the user's display to highlight items of potential interest to the user. The user has access to the full facilities of the standard browser and is required to take no actions over and above the normal browsing actions required for search. The justification for extracting all the information from the user's actions rather than, say, by asking questions is one of cost. As the user is not

required to do anything special there is no additional cost in user input. By basing the system around an existing browser the user is not required to become familiar with a new tool.

1.1 Software Reuse

Active browsing promotes the rapid location of relevant software and thereby encourages reuse. Reuse is widely accepted to be a significant way of improving software productivity by making use of already existing design information. Much research has been concerned with reusing code itself. The notion of code reuse is seen by many as culminating in the production of software components analogous to integrated circuits [3,46]. Certainly research into the design of reusable software has led to considerable activity in creating large software libraries. A prime example of which is object-oriented programming. In general the reuse of code encourages the development of high quality, well tested and even optimized software as the initial investment is repaid each time the code is reused.

To support code reuse large libraries are essential. Some have grown large by natural evolution. This previously designed software represents considerable investment and thus there is strong financial pressure for reuse. Others have been designed specifically to support reuse. As new libraries of reusable software become available these will be combined with older ones to form even larger libraries. Large libraries are desirable to give the maximal coverage of future requirements. They should ideally cross many domains as there may well be a great deal of common core software

Actual code reuse is felt by some to give limited returns [5,34]. They propose the reuse of much higher level software design information. There is still however a requirement for large libraries. Libraries of these higher level forms, including specifications, diagrammatic representations and other forms of documentation, are likely to be increasingly prevalent.

An important idea within the knowledge base community is the "Knowledge-Based Software Assistant" [51]. This approach aims to formalize the whole process of software development and use correctness preserving transforms to aid the user in producing executable code from specifications. As W. L. Johnson [35] points out "Reuse is essential at the requirements level, just as it is at the program level". Thus there is no reason to be restricted to libraries of code: libraries of formal specifications, of transforms, or even of free text would be of undoubted value.

1.2 Locating Software In Large Libraries

The existence of large libraries does not, by itself, guarantee effective software reuse. Biggerstaff and Richter [5] highlighted three central issues: finding, understanding and modifying components. Many researchers consider the locating of software the most critical of the above issues. P. Freeman [25] stresses the importance of locating items: "Our continued work in this area, however, has convinced us that we had "assumed away" one of the more critical aspects of reusability — locating an appropriate set of information to reuse". W. Frakes et al. [23] make the statement: ".... there is strong support for the belief that to effectively promote software reuse we must develop tools to aid in the process of locating software components that are candidates for reuse". Libraries should and will increase in size. Finding a relevant component is likely to become increasingly difficult and this is the main motivation of this research.

Research that has looked at adapting existing methodologies — associated with database querying, information retrieval etc. — for use with software libraries has focused principally on indexing. Normally an index is manually assigned but there are approaches that automatically extract indices from the particular item. Other approaches aid the user in modifying the query if the retrieved items are not of interest. Browsing is particularly relevant to code

libraries which are often sparse and may contain nothing that would even closely match a query. In addition the individual items may only offer partial solutions. The final solution depends on finding and combining multiple items in the library, so a considerable amount of search is required. Further, if the emphasis is on reuse, the requirements may, themselves, change to accommodate what is available. So in these cases browsing should be a more effective means of locating items.

The task of browsing is commonly viewed as a user navigating through a graph where the nodes represent library items and the arcs connecting relationships. Thus for a library to be browseable it must be expressible as a graph. Many different types of information can be expressed in this form. Ideally the graph consists of items and relationships natural to the domain but the relationships must also be useful in terms of search. In object oriented software, many such relationships are part of the natural programming structure. Nevertheless extra relationships can be added including those between associated documentation, formal specifications and knowledge frames. Many other contemporary languages naturally include relationships relevant to browsing of which different forms of inheritance and polymorphism are good examples. In addition application of certain design practices particularly with respect to naming conventions can greatly increase the readiness with which software can be browsed. Even in more conventional languages there are relationships which can be usefully explored. There is for instance the calling structure, reference to external variables and naming conventions. Other relationships not specific to code could also be added. These might be defined by documentation, by modification histories etc. or learnt from user interaction with the library.

1.3 Improving Basic Browsing

Even with a well structured library normal browsing can be a time consuming process.

It is also knowledge intensive as the user must have some idea of not only what is in the library but also of the structure. Given that browsing is an essential means of search, an important issue is how to remove these limitations and generally enhance its effectiveness. By extracting information solely from the user actions and by changing the user interface to the standard browser in a minimal manner there is little additional cost in usage. Thus any gains in performance from active browsing do not have to be offset against losses due to extensive user interaction.

Active browsing aims to maintain the advantages of normal browsing by allowing the user to search in the usual way. This takes advantage of the fact that users may be already familiar with browsing. This is particularly evident in object oriented programming where browsing is essential to exploring program structure and therefore is a natural part of program development. Even if it required to learn the browsing operators this is similar to learning the syntax of an indexing language. The actual terms of the "browsing language" exist within the items being browsed and are usually natural to the domain.

Active browsing makes use of the primary advantages of a computer, its speed and memory. During browsing the user will spend considerable time studying information displayed on the screen to asses an item's relevance to the requirements. As long as the system does not distract the user, this time can be used to search for relevant items. In addition the system is able to maintain in memory more features that might be relevant to the search goal than a user. B. Curtis [17] , with reference to G. Miller [41], describes this problem by the statement "Short term memory is a limited capacity workspace which holds and processes information under our attention. It has been characterized by Miller (1956) as holding approximately seven items of information." This surprisingly small size of a human's "working memory" shows one advantage of a computer system as a search aid.

In active browsing the user actions, collected over the whole history of the search, are utilized in inferring an analogue representing the search goal. The analogue, in combination with further browsing actions, is used to construct a relevancy measure consisting of terms describing an item. With this measure the system scans the library, independently of the user, looking for potentially interesting components. The results are both cached to improve the system response time to anticipated actions and indicated to the user to influence search direction. Active browsing will improve normal browsing by searching a particular area more rapidly. It will also extend the area of the library being examined to improve the coverage and thereby accuracy of the search.

1.4 Research Contributions

- A novel approach to browsing termed active browsing has been defined, where an analogue of the user's search goal is inferred solely from normal actions.
  - This definition includes how this analogue is used in locating relevant library items, ones the user may not have found as quickly, if at all.
- A model of active browsing has been produced that is generally applicable to browseable libraries.
  - This model includes three forms of inference.
  - A representation for the extracted information is proposed: an analogue of the user's search goal and a relevancy measure to evaluate library items.
  - Based on this model, an architecture to support active browsing, using a standard inference engine, has been developed.
- An implementation of the architecture, for browsing object oriented code libraries, has been produced and demonstrates how a standard browser may be enhanced by active browsing.

- Specific modifications have been made to the browser's user interface and rules designed to facilitate inference.
- Various algorithms have been developed that evaluate the similarity of features between two object oriented classes.

□ The experimental method proposed in this thesis is an alternative to the normal method of evaluating browsing systems.

- A rule based heuristic searcher is used to simulate human search patterns. This reduces the requirement of using multiple users to validate performance and allows greater control during experimentation.

1.5 Thesis Overview

In chapter 2 a broad overview of related research will be presented. A critique of this research will be used to argue the case for active browsing. Chapter 3 describes a model of active browsing, developed during this research, which is applicable to browseable libraries. It addresses the crucial issues in producing such a system, illuminating details of the approach with examples from software and information retrieval. Chapter 4 takes the generalized architecture developed as part of the model and produces a specific implementation that is used to search libraries of object oriented code. Chapter 5 presents the experimental methodology, the results obtained and how they validate the implementation and the model. Finally chapter 6 includes the conclusions drawn and the potential directions for continuing this research. In addition the limitations will be discussed, how these might be overcome and, in general, ways to enhance the approach.

Chapter 2 Reuse And Browsing: An Overview

This chapter serves two purposes. Firstly it gives a broader context to this research and secondly it presents an argument in support of active browsing. The review focuses primarily on the research into software reuse. Other areas such as information retrieval, hypertext and user models are also discussed, particularly as they relate to browsing.

The research field associated with the topic of software reuse is divided into two broad sections as discussed by Biggerstaff and Richter [5], the generative and compositional approaches.

2.1 Generative Approaches

The generative approaches to software reuse use program generators to convert a very high level, or specification, language into a lower level form. Here it is design knowledge that is reused. The generator uses this knowledge to produce executable code, or the intermediate stage of a conventional language. This can be seen as a natural extension of the principle of high level languages. There is little argument that such languages have produced a dramatic improvement in program productivity over assembly level or machine code. The goal is to produce gains of similar magnitude with specification languages. One dimension on which they can be characterized is that of domain specificity. At one end are application generators. At the other are the general purpose specification languages and their generators.

In application generators the specification language is often expressed as a series of user menu selections. These selections guide the production of an executable application. For this approach to work successfully the domain has to be well defined and understood. The original design must predict the vast majority of future user requirements and is thus reliant

on extensive domain analysis. Many application generators have been produced but their success has usually been achieved only in very narrow domains.

General purpose specification languages come in many forms such as SETL [20] which can be converted with some user interaction into an efficient executable form. SETL is procedural but still reduces the code required significantly. Other research like Prywes [44] looks at the use of a non-procedural specification language. Not requiring procedural information further reduces the size and complexity of the program specification and the amount of design effort. On the other hand the lack of procedural definition makes it harder to convert to a conventional language, particularly to an efficient implementation.

Occupying the central position in specificity is Neighbours et al. [43] where domain specific languages are constructed during domain analysis and can be converted by transformations into intermediate forms. These are either conventional languages or forms that can themselves be converted into conventional languages. Neighbours' research aims to capture the design expertise that goes into domain analysis. It also promotes languages that are much more natural to domain experts and so are more easily understood and modified. An important aspect of this approach is the reuse of the transformations used to convert the domain language into a more concrete form. A domain expert should reuse them when creating a new domain language. A side effect of this approach is that not only are there collections of specifications but also collections of transformations. If the system is to grow libraries of transformations and specifications may well be needed and must be searched.

The goal of the above research is to be able to automatically convert a specification into executable code. A crucial issue in this area is the ability to produce code that is efficient and many researchers have concentrated on this (for instance see Cheatham [11,12]). This is problematical particularly with the most high level forms. To produce efficient code usually

requires extensive user interaction. Thus although considerable design effort is saved in the generating the specification some of this saving may be lost in the effort to produce efficient code.

Even if the automatic conversion of specifications becomes feasible, it will still be necessary to produce the formal specifications in the first place. A lot of design effort is required to translate an informal specification into a formal one. It is likely that there would quickly develop large libraries of specifications. Thus the notion of reuse of "code" albeit at a much higher level is still useful. Even if the ultimate goal was achieved of being able to automatically produce programs from full natural language specifications, reuse of such specifications would still be less effort than creating one from scratch.

In conclusion it would have seemed at first glance that the generative approach obviates the need for libraries. New software is synthesized not composed from already existing code. In practice, however, there are likely to be libraries of software artifacts even if they are of a radically different form from code. These could certainly include formal specifications, transformations, design plans and free text requirements.

2.2 Compositional Approaches

The compositional approaches to software reuse includes three main tasks: location, comprehension and modification (Fischer et al. [21], Biggerstaff et al. [5]). One approach to the comprehension of software is the browsing of data and control flow graphs. This is a useful method of exploring program structure and helps with understanding the function particular modules. This approach is used extensively in object oriented programming [18]. This use of a graphical forms can be extended to represent a broad spectrum of software artifacts such as hypertext [4,14,15,47] or a knowledge base [19].

Automating the modification and composition of components is a difficult process without additional information. Much research has concentrated on the reuse of design plans that contain the reasoning and design decisions which went into the original process. This work has been based on the design replay notion introduced by Carbonnell [10]. Although not exclusively for software, a review of various design replay algorithms is given in [6]. These approaches and others using code templates [50] or using abstract data types raise the issue of what is a useful software artifact. This is still largely orthogonal to the issue of location. Nevertheless systems that support reuse should ideally be capable of dealing with artifacts other than code.

In the compositional approaches most research has concentrated on reuse of code. Many companies now produce libraries of specific purpose components and these may be combined to produce much larger ones. As these libraries grow the issue of locating code quickly and accurately becomes of paramount importance and different research has looked at ways of using existing techniques and applying them to software.

One field that has addressed general retrieval issues is database research. Certainly, if the software can be described by values for a limited set of attributes, the relational model can be used. Although the actual storage of software would be in conflict with the atomicity stipulation, a simple direct index for a software component could be included as an attribute. Still, the heterogeneous nature of software suggests that the contemporary approach of object oriented databases is more relevant. The close relationship with object oriented programming should ensure the suitability of such databases for storing software. Most research concerning reuse has not, however, drawn inspiration principally from database research. Rather information retrieval has been the most fertile source of valuable methodologies. The following discusses research related to the two most common techniques of locating

components, indexing and browsing.

2.2.1 Indexing

The most common focus of research has been index based retrieval for software. To support queries, useful indices for items must be either manually or automatically assigned.

2.2.1.1 Manual Assignment Of Indices

The first step in manual assignment of indices requires the selection of attributes to best describe, and partition, the software components of a particular domain. A set of values is then assigned to each attribute and any component added to the database will be described by a tuple of these values. The simplest version of this method is to use attributes like the designer, the date produced, the general application area etc.

In Prieto-Diaz's paper [45] a more extensive and useful set of attributes including those that describe the function and input data types are used. In his method the initial stage requires extensive domain analysis to select attributes and their value sets. Further analysis is used to construct a conceptual graph, defining the similarity of the different attribute values, used during query expansion. In addition a thesaurus is produced which defines concepts, their canonical values and synonyms. A human librarian assigns a descriptor — a conjunction of canonical values — to each component before inclusion in the library. A query can then be formed as a tuple of attribute values. One potential problem is that even with extensive domain analysis it may be difficult to define a single useful classification structure. This may only become apparent when new items are added at a much later time, an important consideration with a dynamic library.

A more rational foundation for the choice of attributes is proposed by M. Wood et al. [53]. Their research extends the single classification structure for libraries by introducing multiple

tuples of different arity to describe different "basis functions". Founded on conceptual dependency in natural language, their approach is built around three basic types of concepts: nominals, actions and modifiers. In software, usually actions correspond to functions. Nominals correspond to the object that performs the function, the object it acts upon etc. Modifiers might be a property of one of the objects that affects the way the processing is carried out. Again a domain analysis is required to identify the appropriate concepts. One "basis function" is associated with each classification of conceptually similar functions and assigned a canonical term. Each "basis function" forms a descriptor frame and each component is associated to a particular instantiation. Each frame has associated with it a number of questions. If the user enters a function name the system first attempts to classify it with respect to a basis function. Assuming a successful match the system then asks the questions associated with the particular frame until the appropriate match is found.

In both these methods there is no formal method to carry out domain analysis but it is important that the domain is well understood. A poor choice of attributes may lead to difficulty in classifying new components. More importantly incorrect classification will make it difficult to find useful components at a later time. An additional problem arises if the classification is not apparent to the user or is inappropriate for the user's purpose. Careful domain analysis and the use of very general attribute types should help to maintain robustness. The additional flexibility of multiple arity tuples should also decrease the brittleness.

One type of index is particularly powerful in locating components. A number of researchers have looked at assigning a formal specification to each component. In S. Meggendorfer and P. Manhart [40] preconditions and postconditions are used to define a reusability relationship. This defines a partial order over the set of components. An automatic theorem prover can be used to locate the appropriate component. In M. Harandi

and S. Bhansali [30] a similar specification is used but in this case as an index into a plan library. Various heuristics are used to determine the degree of match to a new specification. Although this may be a powerful indexing method there is still the considerable effort of producing the formal specification needed for indexing. So as discussed in sections 2.1, there would be a strong motivation to have a library of specifications.

2.2.1.2 Automatic Assignment Of Indices

One approach based on principles from information retrieval is the use of keywords. These may be defined by the author, or a librarian, in free text or extracted from a dictionary of acceptable terms. This idea is further extended in S.D. Fraser et al. [24] where keywords are automatically extracted from text associated with software, the component documentation. They are extracted from the "brief text", the text before each method, in the Smalltalk library. By removing common English terms and splitting concatenated words a set of keywords are produced. They can be joined by logical connectives to form a query.

Y. S. Maarek et al. [38] automatically extract indices, through a more complex analysis of text, in the form of lexical affinities. In linguistics, a lexical affinity is the relationship between two words and represents the correlation of their appearance in a phrase. It has been shown that lexical affinities convey information of both a syntactic and semantic nature. From blocks of five words from manual pages all pairs of open-class words (nouns, verbs etc. but not pronouns, conjunctions etc.) are extracted. The word pairs are then ranked by the number of occurrences, an important indicator of their importance, in the document. The score is adjusted to remove bias due to frequently occurring single words. A query is a free text sentence and the lexical affinities in that sentence are extracted by the same method as for the text. These are then matched against those associated with the different components using a

distance measure and are ranked according to the score. The distance measure can also be used to cluster components and this allows browsing.

2.2.1.3 Limitations Of Indexing

Even a good indexing scheme is not effective if the user is unfamiliar with the indexing language. G. W. Furnas et al. [26] demonstrate the large amount of inconsistency that occurs when people assign names to items. Thus a user may have difficulty guessing the correct value of an index. In Prieto-Diaz's approach this is partially answered by the addition of thesauri and special conceptual graphs. In M. Wood et al.'s approach simple questions are asked and the user is able to copy and paste from different lists of possible terms in response to such questions.

This notion is taken one step further in J. F. Gilmore et al. [28] where an expert system asks questions of the program developer. The answer at each stage is used to select further questions to be asked and thus focus the user's search and narrow down the most likely choice of components. As in the other manual approaches extensive analysis is required in which a domain expert must design a set of questions that have a good discriminatory power between components. Ideally a set of optimal questions would be arrived at that allow the fastest location of a particular piece of code.

Even if the user is familiar with the language, it may be difficult to formulate an accurate description of the requirements. Although to a lesser extent, this problem also arises when the user is asked questions. The user may be uncertain of exactly what an appropriate answer should be. Any match on the query is therefore unlikely to be exactly what is wanted. Further after reviewing the retrieved components it may be apparent that they are not what the user is looking for. To answer this Maarek's system includes limited browsing. The use

of browsing is further extended, in later research, by combining the relationships defined by lexical affinities with those in object oriented code [32].

One approach that aims to address both these problems is retrieval by reformulation. In G. Fischer et al. [21,22] and S. Henninger [33] a query returns an initial set of items. The user can then modify the query using information from one of these items. In response to the new query another set of items is returned and the process repeated. The matching process uses spreading activation through a graph of keywords and library items in which the arcs represent mutual association. The items with the highest activation are returned plus the keywords that were activated during the process. The user can use these keywords in the modification of the query and continue search. One advantage of this method is that if a query is not sufficiently discriminating additional keywords can be added. Although this has been categorized as indexing, the progressive form of search is similar to the notion of browsing.

In summary, problems arise in indexing from the user's lack of familiarity with the indexing language. Even a user who is familiar with the language, may find it the difficult to formulate an accurate description of the requirements. Assuming a good description is generated, it is likely that when the library is sparse there will be few items that even partially match the query. These items may well not be relevant to the requirements so a new description will be required.

2.2.2 Browsing

The argument presented here is that the problems of indexing are not negligible. This is particularly true in certain types of search when there is no good goal definition. The amount of search required is much greater than the previous systems support. Browsing is an attractive alternative. It has been used extensively in information retrieval [29,48], knowledge bases [39] and hypertext [14,47,49] and even relational databases [42].

R. Thompson et al. [48] offer support for browsing in information retrieval. They make the statement with reference to Bates' paper [2] "Bates (1986) points out the advantage of browsing by showing how it takes advantage of two cognitive capabilities. The first one is the greater ability to recognize what is wanted over being able to describe it. ..... The second capability is being able to skim or perceive at a glance." They propose an intelligent text retrieval system, based on a blackboard architecture using multiple experts, to aid in browsing. There is, for instance, a domain knowledge expert that suggests concepts that might be relevant to the user. This is determined by asking the user to indicate concepts of interest. This information is used to search the global domain knowledge and the user domain knowledge model for other related concepts. Another example is a browsing expert. It suggests paths the user might take that are likely to lead to relevant information. Here various heuristics are used to decide which links to show to the user. This is based in part by information gathered in a user model in the form of stereotypes obtained by asking the user questions.

G. Boy [7] presents the idea of changing the response to user's request depending on context information. If the user is browsing a hypertext system and selects a menu option this information is used to filter and order the items returned. The determination of context comes from knowledge built up about a specific user. This knowledge is accumulated by the user indicating if items returned during previous searches were of interest.

Hypertext is a very flexible way of representing informal knowledge and a number of researchers have looked at it to store software information [4,14,15,47] P. Constantopoulus et al. [14] describe a software information base consisting of software artifacts and their connecting relationships. The base is not homogeneous. One view is as a hierarchy based on application frames. At the lower levels are application specific frames. At the highest level

are generic application frames. Another view is as an interconnected set of descriptions. They represent requirement, design and implementation information associated with a particular application specific frame. The paper describes an extensive list of links that can connect the different artifacts. The implementation allows graphical browsing by filtering the links that connect to the current node.

These papers show that browsing is considered a useful alternative to indexing by many researchers in many diverse situations. Of particular interest is that many people are looking to hypertext as a useful means of storing information about software. In M. Creech et al. [15] work is discussed on how to automatically construct hypertext graphs by clustering existing code.

A critical problem however is that in large graphs of data the user can become disorientated. This problem is often called "lost in hyperspace" in the hypertext community. In R. Trigg [49] guided tours, paths predefined by the author, is one way to try to overcome this problem. Another way is by adding intelligence, and preferably learning, to the browser. The learning in the R. Thompson et al. paper [48] is principally by assigning a user to a particular stereotype by asking questions and keeping this information in a user model. In G. Boy's paper [7] learning was concerned getting feedback from the user about the relevance of items.

The information learnt by Thompson and Boy's systems is used over a long period of time i.e. over many searches. The intent of active browsing described in the rest of the thesis is to acquire useful information within a single search. Further this is done from the existing actions, the user is required to do nothing not done within normal browsing.

2.3 Conclusions

In this chapter, alternative methods to support software reuse have been discussed. It

was argued that generative methods do not eliminate the need for large libraries so locating items is still a critical task. It was also argued to support software reuse, at its most general level, it is important to be able to deal with many different types of item. These might include formal specifications, transformations, design plans and free text requirements.

It was further argued that extensive browsing facilities are required in a system that aids the user in locating items. This is required to overcome significant indexing problems that frequently occur as the user does not have a good goal definition. But, there are problems too with browsing. It is a time consuming process and in a large highly interconnected library, a user can become disorientated. The solution proposed to these problems is to add a learning component to the browsing system so that it acquires the user's search goal and aids in the search for relevant items.

Chapter 3 Active Browsing: An Abstract Model:

In this chapter an abstract model of active browsing is discussed. Reference will be made to examples of software and document retrieval to illustrate the ideas.

Section 3.1 discusses the view of browsing used in this thesis. Section 3.2 discusses how this view is augmented with the addition of the active component. Section 3.3 introduces a generalized system architecture suggested by this view. Section 3.4 describes how the inferred information is represented. Section 3.5 shows how this information is inferred and section 3.6 deals with how the user's search is influenced by the results. Section 3.7 discusses some general issues which can affect the inference process.

3.1 View Of Normal Browsing

Browsing is considered to be search where the goal is not well defined a priori because, for example, the goal is ultimately dependent on what is discovered during the search. Browsing is viewed as a user navigating through a graph where the nodes represent library items and the arcs connecting relationships. Thus for a library to be browseable it must be representable as a labelled graph. In addition these relationships should be instances of a limited set of common types which are relevant to the domain and to search.

Figure 1 shows an abstract rendering of the normal browsing task. The user is looking for an item that best meets a set of requirements, possibly a specification. The user converts these requirements into a search goal which describes the expected form of the target item. This search goal is part of what Fischer et al. [22] denote as the "situation model" which is all the knowledge the user brings to bear on the search problem. In addition to the goal it contains problems that motivate the tasks, the plan to find the target etc.

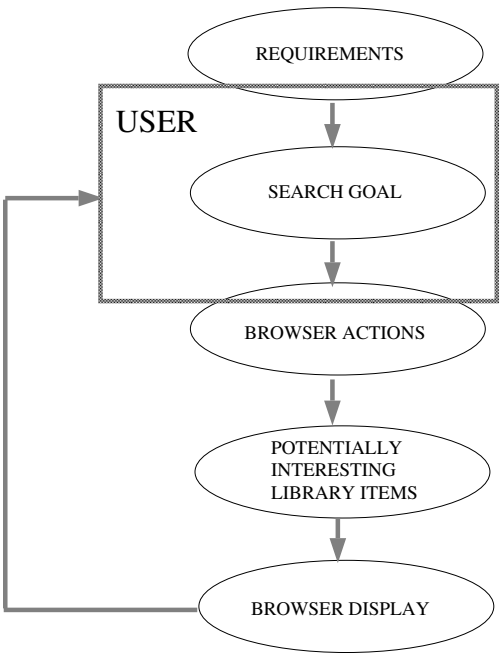


Figure 1 Normal Browsing

Browsing starts by selecting an initial item in the library. The user compares the item to the search goal and uses browsing actions to control navigation through the library to locate items that better match the search goal. Each action, or series of actions, results in a set of potentially interesting items. These are displayed to the user who reviews them, selects one and continues the search from there. This process continues until the user is satisfied that one or more items meet the requirements or no such items can be found.

Although the requirements should remain constant over the course of the search, the goal may well change. This is a natural consequence of the discovery element of browsing. The searcher, after reviewing new items, finds properties which better represent the requirements. In the simplest case this is just a refinement process where the initial search goal does not change in substance but detail is added. A more complex case arises when the user discovers that parts of the search goal are not the best representation of the requirements. In this case the search goal undergoes a more radical revision.

3.2 View Of Active Browsing

In active browsing (Figure 2) an additional branch is added to the process. The browsing actions are mapped to an analogue of the user's search goal. That the actions implicitly carry information about the search goal is a consequence of the fact that the actions have been deliberately chosen, to the best of the user's ability, to serve the user's interests. The analogue is built up over time from successive browsing actions and represents what the system believes to be the search goal. The analogue is mapped into a form that can be readily used to measure the relevance of an individual item to the user. This relevancy measure returns a numerical value representing the degree with which any particular library item matches the analogue. The measure is used to evaluate library items and the result of the evaluation can be used to change the display and thus influence the user's search direction.

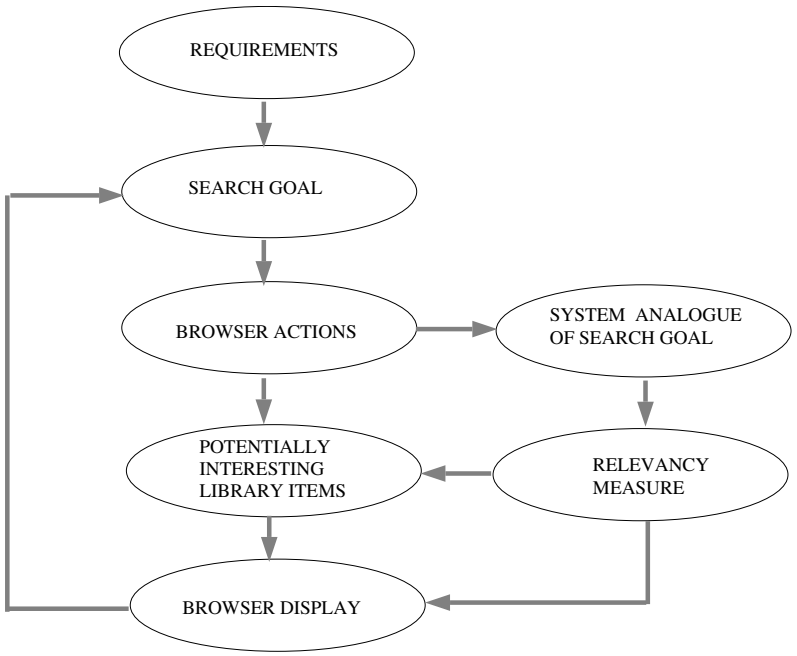


Figure 2 Active Browsing

How will active browsing accelerate the user's search?

Firstly, being unclear of the search goal, the user is likely to explore many different properties of the items not all of which will be in the final target. Nevertheless on average it is expected that the user will inspect mostly those properties that are relevant. Thus active browsing combines and filters user selections to find items that best match the common properties. A user is unlikely to remember all the relevant properties. Certainly the user is unlikely to compose a complex matching request including all the relevant properties seen during the search.

Secondly, a naive user may not be aware of all the different operators available or when they might be relevant. For instance many libraries have a primary structure e.g. in Object oriented software there is the inheritance tree. A user may use this as the main means of search. For instance, the user may principally navigate parent-child links. Classes with a common parent inherit identical method implementations, so sibling classes are likely to perform similar roles. Other items that also perform similar roles, but with different method implementations, may be well separated within the hierarchy. The user may not appreciate this or may not be aware of an operator to locate such items. The system can expand the area of search by retrieving items with similar roles from different areas of the library's primary structure.

Thirdly, the automatic extraction of properties reduces the amount of actual user selection required. For instance, if the user investigates particular properties of an item it seems reasonable to infer that other items with these properties are potentially of interest. Thus the system can locate such items while the user is studying the original one.

3.3 Generalized Architecture

The view of active browsing, presented in the preceding section, leads to the generalized architecture given in Figure 3. Rules are used to map user browsing actions to the analogue. A rule's antecedent involves user actions and existing features of the analogue and its consequent causes new features to be generated or old ones updated. In this way the analogue is updated to reflect the user's current interest. The general form of such a rule is :

IF ((Browsing Action) ((Analogue Term)....))

THEN ((Analogue Term)) Confidence

The confidence factor describes the certainty that the system has that the analogue term describes a feature of the search goal.

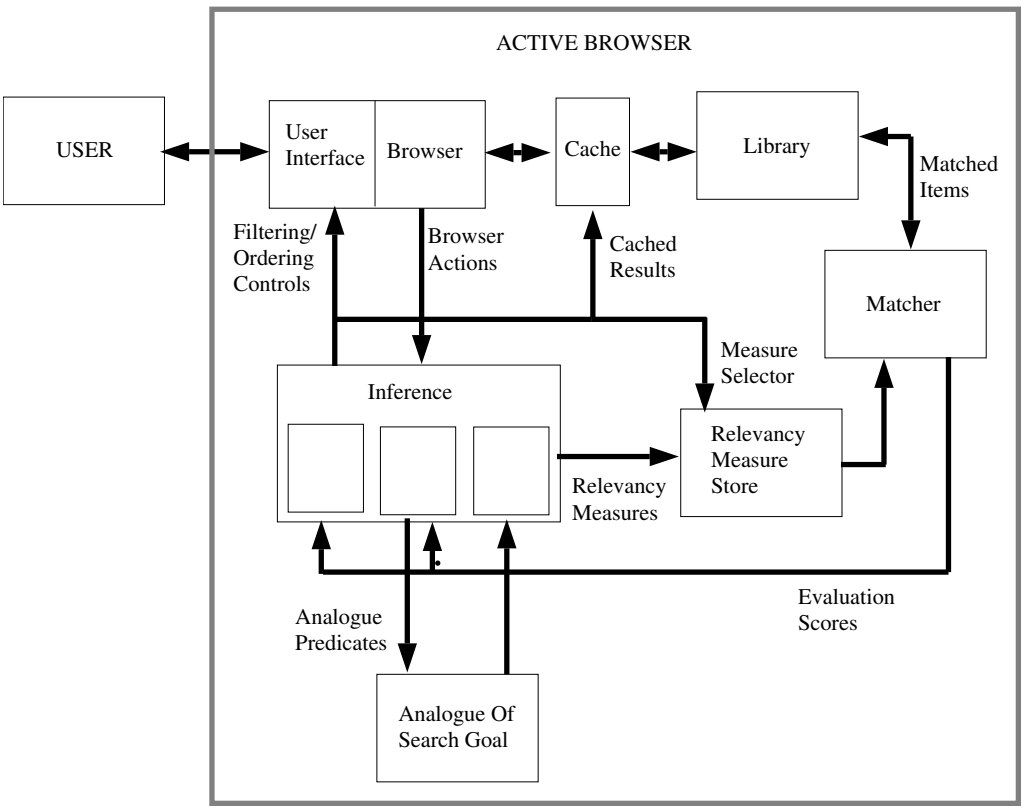


Figure 3 Generalized Architecture

The inference process is further used to produce relevancy terms. These are converted and combined to form the relevancy measure which can be matched against the library items. The general form of these rules is:

IF ((Analogue Term)....)

THEN ((Relevancy Term)) Confidence

A store of relevancy measures can be maintained to allow the user to pursue sub-goals of the overall search. The selected one is compared to each of the items in the library and the value associated with the match is a measure of its relevance. The matching values are used to alter the browser display by the relative enhancement of those items of most interest. Matching values on individual items can be used both to adjust the analogue and to select the appropriate measure. By storing the results of matching in a cache, a user requested match can be serviced more rapidly.

3.4 Form Of The Analogue And Relevancy Measure

In choosing a form for the system analogue and relevancy measure, the first consideration is, what are the units within the library in which the user is interested and for which relevance is evaluated. The assumption made is that the library consists principally of one level of granularity and that the user is attempting to locate one such item. For example in object oriented programming the items might be classes. In document bases the items might be conference or journal papers.

Nevertheless this approach extends gracefully to libraries in which the same entity can simultaneously be an individual library item and also a component in a larger item. In this case, relevance is evaluated of both the isolated entity, and the collection that contains it. This situation arises in software within the object-oriented paradigm, where each class is an

individual item and may also be a part of a "framework". Alternatively a document base might contain papers which are themselves parts of conference proceedings. One might return a conference proceedings to the user if it contains many papers that are of interest. This still assumes that the user is principally interested in items at a particular level. If the library has multiple levels of granularity it would be necessary to infer not only what properties of an items are important but also what level of item interests the user.

The next two sections describe the form of the relevancy measure followed by that of the analogue. They are presented in this order because the form of the relevancy measure affects the choice of form of the analogue.

3.4.1 The Relevancy Measure

The relevancy measure should be in a form allowing easy comparison with library items. Ideally it should be an abstraction of an item. The measure should contain terms, themselves combinations of atomic terms that are readily measurable in the item. For example, if an item in the library contains information about its author, then a relevancy measure can include terms referring to author. These might include the author's name and job title. In the case of documents these terms might involve keywords, lexical affinities etc. In object oriented software typical atomic terms might be method and instance variable names. In addition the measure might include relationships between terms within the item or between terms and parts or the whole of another item. For example, the fact that one method uses another in object oriented programming or that one class is the child of another might be included. In document libraries one might add relationships like *one document references another* or *two words are adjacent*.

Active browsing is looking for the best match to the user's search goal in the library. Therefore the relevancy measure should contain the minimum amount of information needed

to discriminate between items. This will minimize the time required to evaluate a single item. It is also necessary to determine how well an item matches the user's search goal. One way of addressing both concerns would be to first use the discriminating measure to determine the best items. This much smaller list could then be evaluated to determine the degree of match.

The relevance of a library item is evaluated by comparing its contents with each of the terms of the relevancy measure. The comparison of a term to the item's contents is computed by a matching function that is specific to the type of the term. In document bases, a matching function might compute values based on lexical affinities or logical combinations of keywords. In software libraries which include formal specifications, a matching function might involve proving that one specification implies the other. In object oriented programming it is standard practice that method names consist of concatenated words. These words can be matched individually to the words in other method names.

Each term should be assigned a value, or weight, that is a measure of its criticality to a match. Some terms represent features that are strongly desirable in any match, others are less important. This may be dependent on the particular term or its type. For instance, in an object oriented programming library it may be more important to match well on instance methods than on class methods. The instance methods would be assigned larger weights. The numerical values representing the degree of match returned by the matching functions would be combined according to these weights.

3.4.2 The Analogue

The form of the analogue is dependent on two factors. Firstly the ease with which it can be inferred from user actions. Secondly the ease with which it can be converted to a relevancy measure. The analogue could for instance be a literal record of the user's browsing actions but this would make it difficult and costly to infer the relevancy measure. It is better

to filter out the redundancy in the user's actions. There should be a reasonable degree of compression of the information extracted from the actions. The rules used in the inference process represent knowledge on how this information might best be abstracted.

If the general form of the target item is known this can be used to reduce redundancy. For instance if the user's action was to look at a keyword in two different documents, should two copies be kept (one for each document) or just one? Knowing the form of the target and that the same keyword is not going to appear twice it should be stored only once. Of course the fact that the user looked at more than one document with the same keyword is significant. This information should also be stored. If the relevancy measure is only sufficient to discriminate between items, the amount of information required for this task will vary during the search. Generally the analogue needs to have sufficient information to produce all possible relevancy measures that might be needed.

Like the relevancy measure this seems best served by being in an abstract form of the item. One or more terms within the analogue would represent a feature of the item. The scheme chosen for the implementation was to use many weighted terms to represent a single feature. This is not the only method that could be used. A stronger compression could be applied to remove more of the redundant information but at the risk of losing important detail.

3.5 Computing The Analogue And Relevancy Measure

The previous section outlined the form of the analogue and relevancy measure, this section discusses how they might be inferred. The system uses information from the user's browser actions to form a system analogue, representing the user's search goal. Information stored in the analogue is then extracted and formed into a relevancy measure and used to evaluate library items.

The user may express direct interest in an item, or a specific property of an item, by studying it in greater detail. For instance in object oriented programming the user may study the code that implements a particular method. This direct interest indicates it is likely to be a property of the target class. The user may also have knowledge about the library that is important in terms of the final item chosen but is not specifically used as part of search. Three methods are used to extract such knowledge: firstly by encouraging the user to make explicit choices by means of operators, secondly by making the choice have a specific meaning and thirdly by inferring the likely extra knowledge the user has of the library.

3.5.1 Inferring A System Analogue

There are three basic types of operators used to browse a library (Figure 4). Expansion is showing a node in greater detail, in the form of a subgraph. Exploration is moving between nodes at the same level of abstraction. Consolidation is combining a group of nodes and arcs and moving to a node at a higher level that has a similar group in its subgraph.

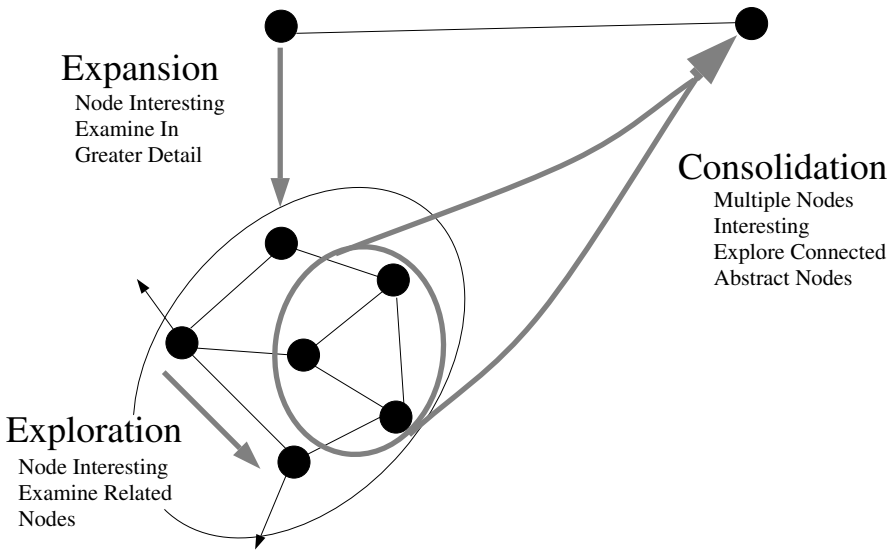


Figure 4 Types Of User Actions

This thesis proposes three distinct methods of inference in forming the analogue from these actions. The first is the adding of terms incrementally. Each time the user passes through a node the information at that node is extracted. The second is by generalizing the subgraphs of two nodes visited if the system determines they represent the same feature of the search goal. The third is pattern specific features. Here the system uses domain specific knowledge of the meaning of traversing one or a series of relationships to add extra information to the analogue.

3.5.1.1 Incremental Additions

The first principle in inferring a system model is the assumption that if the user search is continued after evaluating a node the information at the node is interesting and relevant to the search goal. Even if the user backtracks later, the information obtained from that node is still useful. The user made the decision to continue the search based on the information displayed at a particular node. Thus at best this will be important information, at worst it is not bad enough to overcome positive information obtained earlier in the path.

This idea is particularly apparent when the user travels down a series of abstractions (Figure 5). Starting at a potentially interesting node the user expands it to view its subgraph representing the properties of the node. One particular property, or subnode, is itself expanded to show greater detail. On reviewing this subgraph the user decides the path is uninteresting and backtracks up one level of abstraction. The user repeats this process on other nodes then finally moves back to the original level.

For example, the information at the top node might be the title of the proceedings of a conference. The conference seems interesting so the user expands the node. The next level could be titles of the papers in that conference. The user selects a title and expands this node to show the “abstract”. After reading this the user decides the paper is not interesting

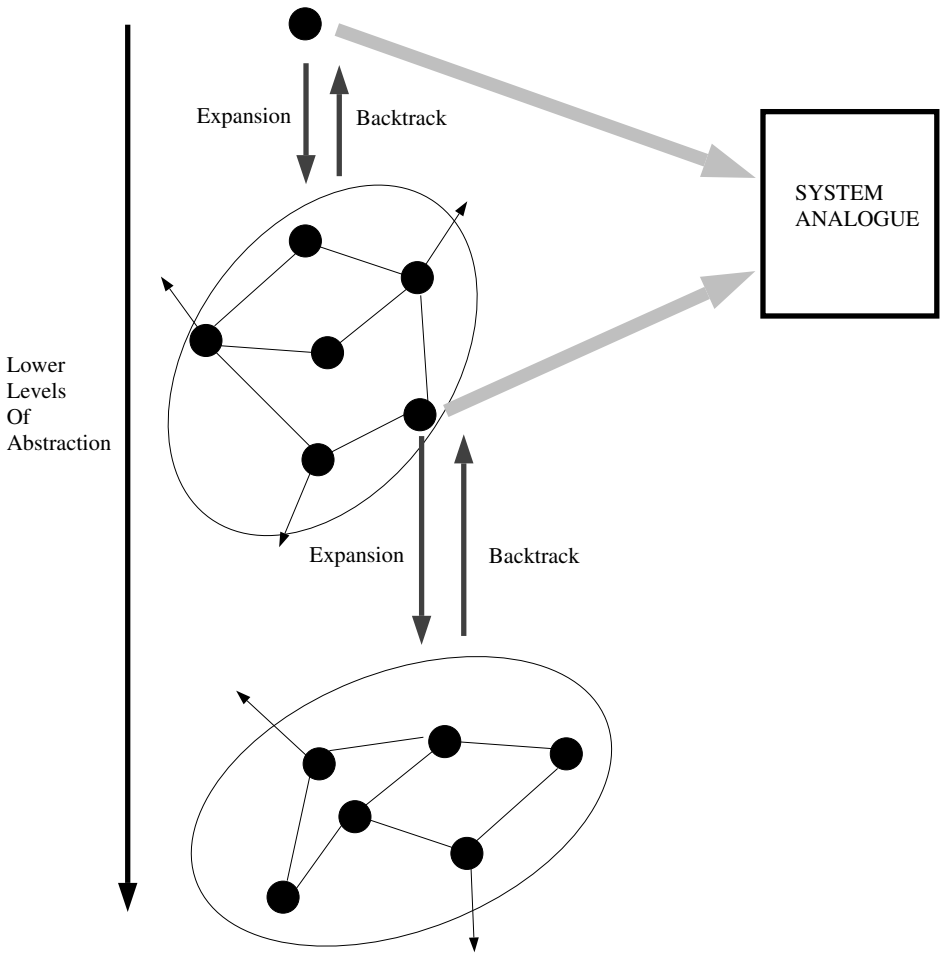


Figure 5 Levels Of Abstraction

and back tracks to the proceedings and reviews other papers. The user does not find the relevant paper and backtracks to the conference level. The system might scan the rest of the database looking for other papers that best match the selected titles and are in a conference similar to the one selected.

3.5.1.2 Generalizing Subgraphs

The preceding inference scheme requires that the user show specific interest in a node by passing through it to review a subgraph or other connected nodes. The user may have other knowledge which is not directly used in the search. For instance the user may review a number of papers in a conference proceedings. It would not be surprising if they are on

closely related subjects, reflected in certain properties of the papers, as they are part of the same conference. These papers are being explored precisely because the user expects them to be on closely related subjects. Even though the “subject “ is not explicitly used to guide the search. Properties that the papers have in common could be included in the analogue. A stronger conclusion might be drawn if only a small number of papers were reviewed and these had common keywords. The system might reasonably infer the keywords are highly relevant and add them to the system analogue marked to reflect this importance.

To make this generalizing inference effective, it is necessary to use some sort of statistical measure. It is important to know whether or not these common characteristics were likely to have occurred by chance. For instance, in the above case, if many of the documents in the proceedings contained the same key words they would only be considered important if they were unusual in the document base. Again if the particular key words in the selected documents are unusual in the particular proceedings they are likely to be useful.

One difficulty in this inference is deciding what nodes to generalize. With only a single level of library decomposition if the user looks at multiple items, it is likely they are alternatives. However suppose the user reviews similar properties in two items. Is the user interested in just one property or should the search target have both? The determination is largely dependent on the degree of similarity and the pattern of search. For instance, in object oriented programming, it is common to have two methods in different classes with the same name. Thus if the user deliberately searches for classes that implement a specific method and then explores the different versions it seems reasonable to infer these are alternatives i.e. the user wants one or the other but not both.

3.5.1.3 Pattern Specific Inferences

A third class of inference uses domain knowledge. As browsers are normally designed

for broad domains this knowledge can be quite general. In this case the fact that the user traversed a particular relationship may mean two things. The direct inference is that the user is interested in nodes connected by the selected relationship. The indirect inference is that nodes connected by other relationships, which domain knowledge determines are strongly tied to this one, might be relevant.

Using the example of the preceding section, the user is exploring papers in a particular conference. The direct inference is that the user is interested in papers in this or related conferences. The indirect inference, based on the knowledge that most conferences are focused on a narrow range of topics, is that the user is interested in the topic of the conference. This inference would extend the search looking for papers on the same topic even if they were from an unrelated conference proceedings.

3.5.2 Inferring A Relevancy Measure

Inferring an analogue from user's actions is much more complex than inferring a relevancy measure from the analogue. The inference of a relevancy measure takes one or more terms in the analogue and produces a term in the relevancy measure. The knowledge in these inference rules needs to take into account how to best discriminate between items and also how important particular terms are in finding useful items. For instance the analogue might describe a paper with a title that contains the word "inferring" plus has the keywords "machine learning", "browsing" and "software". How should the relevancy measure judge the importance of these terms? Knowing for instance that authors tend to be more systematic in choosing keywords than titles, the latter would be given greater importance and the title term might only be used to differentiate between items with the same degree of match on keywords.

Inferring the relevancy measure means removing a large amount of the detail of the analogue. It means changing the form of terms so that they can easily be matched and adjusting the significance or weights of the terms to best discriminate items in the library.

3.6 Influencing Search

Using the relevancy measure, items in the library can be evaluated. The results of the evaluation process are used to alter the display of information about the library. The general strategy is for the system to draw attention to the items that are judged most relevant, and suppress the display of the items of very low relevance. Similarly, the system can give prominence to the browsing commands that it judges to be most relevant — for example, by making them extremely simple to invoke — and suppress commands that are judged irrelevant to the user's current interests.

The main risk associated with using an active browser is a consequence of the inevitable errors it will make in evaluating the relevance of items in the library. The display of irrelevant items erroneously thought to have a high relevance will distract the user. And, more importantly, some highly relevant items, erroneously thought to be irrelevant, might never be shown to the user. The approach is to minimize the risks of active browsing and is based on the principle that the active aspects of a browser should not prevent, or greatly disrupt, normal browsing by the user.

There are a number of ways that the disruption to normal browsing is minimized while still exerting sufficient influence over the user to accelerate search. The simplest approach, the one used in the implementation, is to have a single, preferably small, extra window. In this way the user is not required to look at it at all and may only use it when problems have

arisen in normal browsing. If the window displays its information in an identical manner to one or more windows in the original display then it is easy to use when required.

In many cases lists displayed by the application of user operators are not sufficiently discriminatory. Many items may have the same score, so their respective ranks may be assigned arbitrarily. For instance, if a user requested all documents that included the keywords “machine learning” the list might be very long. The first paper in the list might be determined alphabetically by the browser; this gives prominence to items beginning with the letter "A”. The ranking of items on a list otherwise ordered in this way, using information inferred from a user’s previous actions, would be another way of influencing search. This would not significantly affect the user’s normal browsing actions.

Other ways to influence search include adding extra items to a list or highlighting specific items to focus the user’s attention on them. Perhaps the least disruptive option is precomputation. Here the system would predict the items the user will investigate during the search. It can then retrieve information about them ahead of time and cache the results. Thus when the information is requested by the user the response time is reduced.

3.7 Obstacles To Active Browsing

There are two obstacles to inferring an analogue in the manner described in the preceding sections. The first is that user browsing actions may be uninformative and thereby make it difficult to extract useful information. The second is that the user’s search goal may shift during browsing.

3.7.1 The Problem Of Ambiguous Actions

The primary effect of a browsing action is to display nodes related in a particular way to the one selected. A user chooses the actions because of their primary effects. In active

browsing the actions have a secondary effect that of communicating goal information to the system. As they were not designed to support this function, they tend to be ambiguous and so lack the expressive power needed for inference. New operators are required which are informative as well as useful search tools. Ideally the user will most frequently use the most informative actions. To encourage this, these actions should be the simplest and most useful in normal browsing. The aim is to increase explicit declaration of interest without restricting user search. To this end specific operators should be made easier to use than general ones and the scope of the latter reduced.

There are three main types of ambiguous actions. Non-descriptive actions are ambiguous because they do not explicitly refer to relations between items in the library. The system is unable to determine the reason the user moved between the two items. Nonspecific actions are ambiguous because they return a large amount of information containing many properties. The system is unable to isolate the specific property of interest to the user. Situation dependant actions are ambiguous because they have different meanings in different situations. The system needs to know the situation to extract the meaning.

Non-descriptive actions occurs when the user moves through the graph without traversing a specific link. The user jumps between two items probably based on internal knowledge of their relationships. To limit the likelihood of these actions it is important to make movement through the links as easy as possible and to have many links which correspond to the various notions of connection the user might have. A typical example of a non-descriptive action, with an object oriented program browser, is the selection from a complete list of classes. If the user takes a series of such actions it is difficult to infer the reason for these steps.

Nonspecific actions occur primarily when too much information exists at a single node in the graph. When the user reviews this information it is difficult to determine what specific

parts are of interest. For the purposes of inference it is preferable for the user to go through a series of progressively less abstract steps. This would allow the system to collect and compare other items against these abstractions. By splitting the nodes and including abstract descriptions at the root of each sub-tree the action is made more informative. Although this requires the user to take extra steps the introduction of abstract nodes accelerates normal search by aiding in the rapid analysis of interesting paths. Simple abstract terms can be more quickly and easily evaluated by the user than long detailed descriptions. As B. Curtis suggests " A process called "chunking" expands the capacity of our short term mental workspace. In chunking, several items of information are bound together conceptually to form a unique item". Many hypertext researchers have looked at abstraction as an important facility that aids the searching of complex graphical structures.[27].

There is, of course, an upper limit on the number of stages of abstraction. Too low a level of granularity would be a disadvantage to the user, introducing an unnecessarily large number of browsing steps. Perhaps a reasonable number of steps for a class in object oriented code could start with its name as the highest abstraction. The next level could be text describing the general characteristics of the class. The class could be then split into functional groups of messages, a requirement in Smalltalk. These would then split into the method names and with each having further text as a lower abstraction and so on.

Situation dependant actions occur when the same operator can be used under different circumstances for quite different reasons. This is particularly noticeable in graphical representations for programs. The structure can be explored both to locate a particular piece of code and to understand how it functions. Ideally the operators can be made distinct by allowing the user to select different views of the structure. One is most suitable for program exploration, the other for locating code. Another example might be in browsing knowledge

bases. Again there are two modes of operation i.e. finding and understanding a piece of knowledge.

3.7.2 Goal Maintenance

As discussed earlier the user's search goal may well change with time. The initial goal may be revised as a result of what the user sees during the search. Thus an important consideration is how to keep the analogue and the relevancy measure up to date. Changes in search direction are of a variety of types and occur for a variety of reasons.

The most common and gradual change is likely to occur as the user's requirements are reevaluated in terms of the language and items in the library. The information extracted from the user actions should get progressively more accurate as the search progresses. Certainly information obtained earlier in the search is less reliable than that obtained more recently. Changes may be just a refinement of parts of the existing search goal. For instance the user may be looking for a paper on learning then realizes that this must be machine learning to focus on the relevant topic. A more abrupt change is likely to occur if the user discovers that a property initially thought useful is irrelevant. An extreme case occurs when the user is forced to abandon an initially promising search path and starts a new search for a seemingly unrelated item. In this case the user might conclude that learning is not the issue at all but expertise is.

If the confidence in the value of data is inversely related to how long ago it was selected the simplest approach is to decay the confidence in the terms of the analogue and therefore the relevancy measure. Ideally terms should never completely disappear: even very old data might be relevant in breaking a deadlock based on more current information. Thus a function that decays the highest confidences the largest amount should be used. How can the system

tell if older information is still relevant? One approach is by comparing the analogue to the classes the user selects to see analogue is predictive.

Another issue in goal maintenance is when the user breaks the search task into different parts. Although the overall goal is to find something that meets particular requirements the user may focus on parts of these requirements at different times. Probably the most likely instance of this occurs when the user realizes that meeting the requirements requires multiple items. In this case the analogue may be considered to represent a single search goal. Multiple relevancy measures must, however, be produced and activated at different points of the user's search.

3.8 Conclusions

A general model of active browsing has been presented in this chapter. The model describes how an analogue representing the user's search goal can be inferred from normal browsing actions. It further describes how this analogue may be used in an independent search for items of relevance to the user's requirements. A generalized architecture, centered on a standard inference engine, was developed based on this model.

Three methods of inferring the analogue were proposed, namely incremental additions, generalizing examples and pattern specific inferences. It has further been suggested how, by the replacement of ambiguous browsing actions by functionally equivalent unambiguous ones, the effectiveness of the inference process can be improved.

Chapter 4 An Implementation For An Object Oriented Program Library

This chapter discusses an implementation of the general architecture given in the previous section. The system is built around a standard browser used for exploring libraries of object oriented classes. This browser is for Objective-C code and is very similar to the one used for Smalltalk.

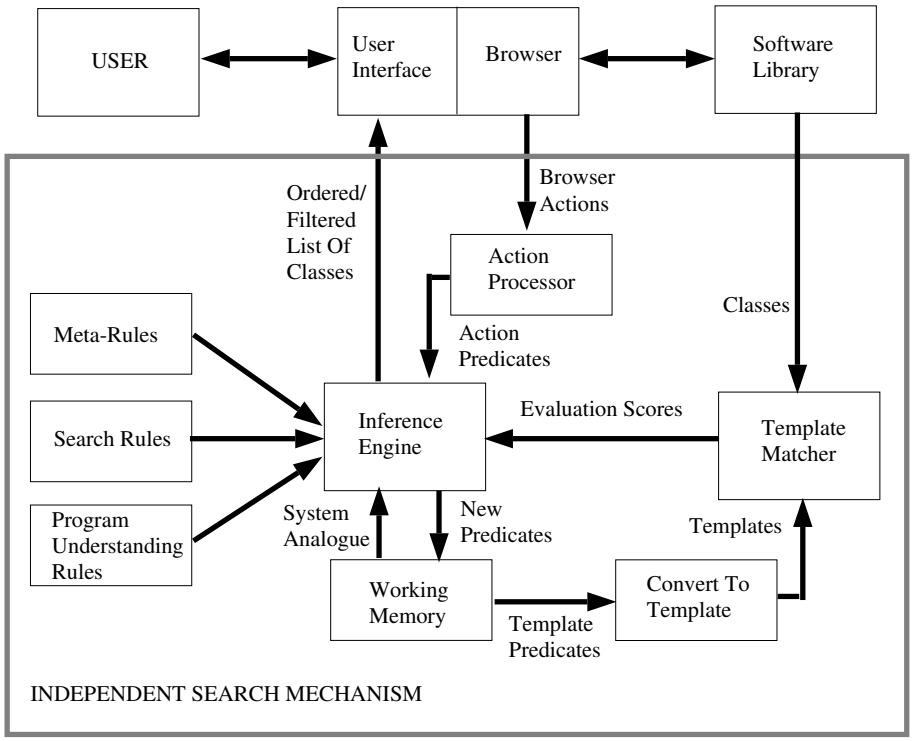


Figure 6 System Architecture

Figure 6 gives the architecture of the active browser. It consists of two main parts. The top section represents the standard browser. This allows the user to display items connected by specific relationships to the one selected and thus navigate through the library. The lower section is the independent search mechanism. This takes as an input the user browsing actions and outputs a list of potentially interesting classes. The general model, given in

chapter 3, discussed various ways of influencing search. At present this is realized in the display of the names of classes in descending order of their evaluation scores. A minimum filtering is done by removing classes whose matching score is below that of "Object" the most general class in the library.

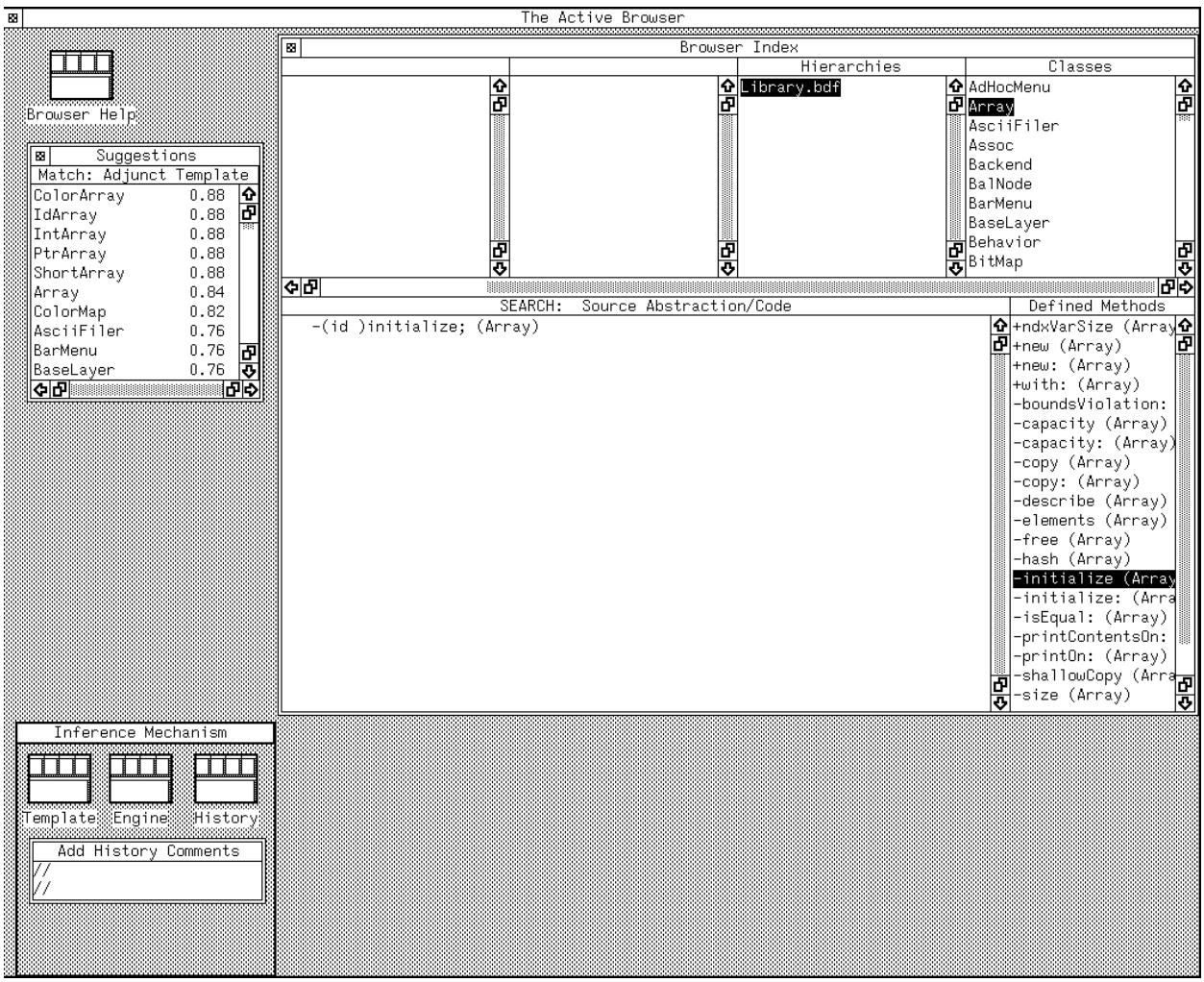


Figure 7 Human Interface To The Active Browser

The human interface to the system is shown in figure 7. The large area marked "Browser Index" represents the standard browser. In the lower left corner is the interface to the independent search mechanism. The attached three icons can be opened to examine the

workings of the inference process during debugging. Ultimately however this part would not be seen by the user. The only part that would be displayed is the “Suggestion Box” to the left of the standard browser. This shows the list of classes proposed by the active browser as potentially interesting to the user.

4.1 The Standard Browser

The standard browser is the first part of the active browsing system. It consists of two parts, namely the software library of object oriented classes and the browser itself.

4.1.1 The Object Oriented Library

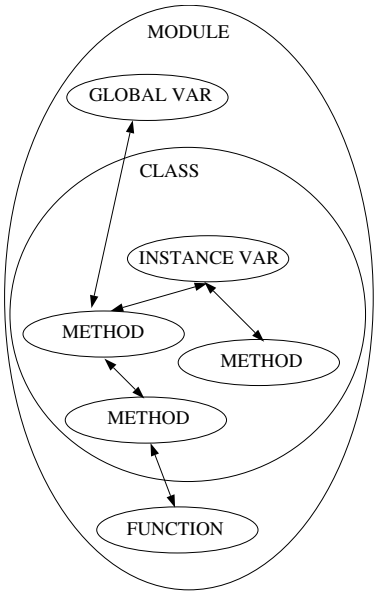


Figure 8 An Objective-C Library Item

An object oriented library principally consists of classes. However, some languages do not follow the object oriented principles as rigidly as for instance Smalltalk. Objective-C is preprocessed into C before it is compiled and this allows the inclusion of pure C code. This has the advantage that general object oriented principles can be followed but when

efficiency is required the code can be written in C. The disadvantage, apart from allowing the side stepping of the object oriented methodology, is that it also produces a more complex structure as seen in figure 8.

So although for the sake of simplicity the library is treated as if it consists of classes, strictly it consists of modules. A module contains a class but may also contain function definitions. In addition there may be variables over and above the instance variables defined in the class. These variables are of the traditional C type in that they can be defined as public and accessed directly by other modules. In Objective-C all variables of this type, public or otherwise, are termed global variables. The class itself consists primarily of methods which may access the instance variables also defined within the class. Usually documentation is included with each class and as a leader to each method.

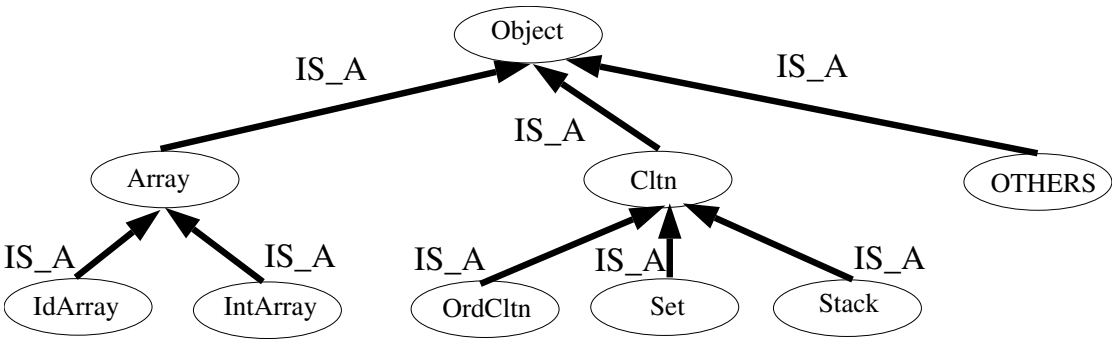


Figure 9 The Inheritance Hierarchy

There are a number of intrinsic relationships between the individual items and between their sub-parts. These define the basic graph of the library. The most important in object oriented programming is the IS_A or inheritance hierarchy. Figure 9 shows part of a commercially available library used in experiments discussed later. Each class is a special type of another class, its parent. The only exception is the root of the inheritance tree, normally designated “Object”, the most general type of class. For instance IdArray, which

stores Id's (Id's are a special type of pointer used in object oriented programming), and IntArray, which stores integers, are both types of Array. In fact Array is an "abstract class", which is not usually instantiated but defines the general behaviour of types of Array (see (R Wirfs-Brock [52]) for a discussion of current object oriented ideas). Each class inherits all the methods and instance variables from its parent and therefore from all its ancestor classes. At each stage, moving down the tree, more methods and instance variables can be added to increase the specific functionality of the type. It is also possible to block the functionality of a method of the parent by overwriting i.e. using an identical name.

The other principal relationships within this type of programming are those associated with control and data flow (Figure 10). Methods use other methods, of either the same or another class, as part of their processing. Instance variables also can be read or changed by methods in the class. In addition in Objective-C global variables and functions can be accessed by methods.

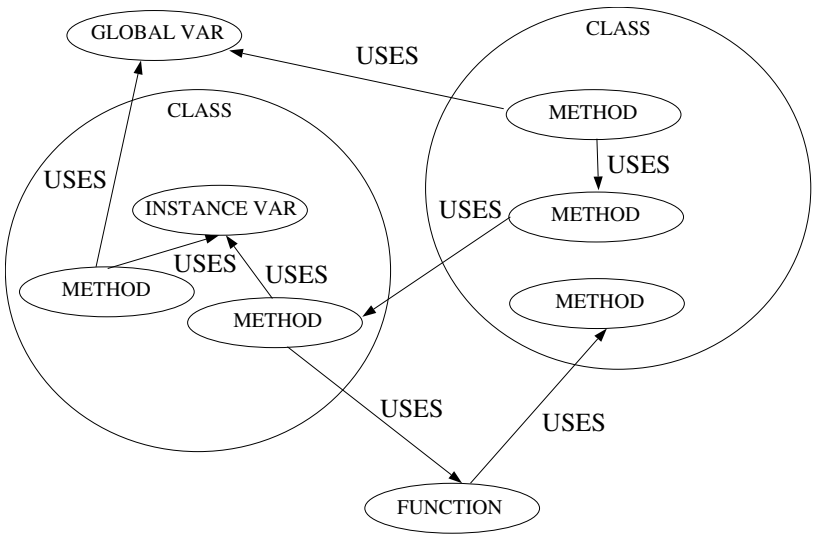


Figure 10 Control/Data Flow Graph

These are not the only relationships that might exist in a library of such code. Further

relationships are effectively defined by different operators available to the user in the browser. One example is methods with the same name (polymorphism). In object oriented programming browsers, there is typically an operator that finds other methods whose names are the same as the one selected. Thus it can be considered that a "same name" relationship links these methods.

4.1.2 The Browser

The browser allows the user to navigate the graph that represents the library of classes. This is achieved by displaying information about classes, methods etc. and their interrelationships. Application of operators results in the display of all nodes joined to the selected node by a particular relationship. The nodes are displayed in a list in one of the browser windows. Other operators may be applied to items in the new list and so on producing multiple lists.

Extra operators were added to the existing browser for a number of reasons. The most important was to reduce the ambiguity of operators as described in section 3.7.1. Also the independent search mechanism required more complex matching operators than were implemented in the original browser. All additional operators were made available to the user through the browser interface.

The standard browser interface (Figure 11) can be divided into three parts which represent three levels of abstraction. The top series of windows normally contains lists of classes, the highest level of abstraction. There are two exceptions to this. The first is the list titled "Hierarchies" which shows the name of the library being explored. The second occurs in a special mode called program understanding, discussed later, when lists of methods instance variables etc. can be displayed. The lower right hand side gives the expansion of the class into the methods that it implements. The methods themselves can be further expanded, in lower left hand window, to show greater detail. Here the individual methods can selected.

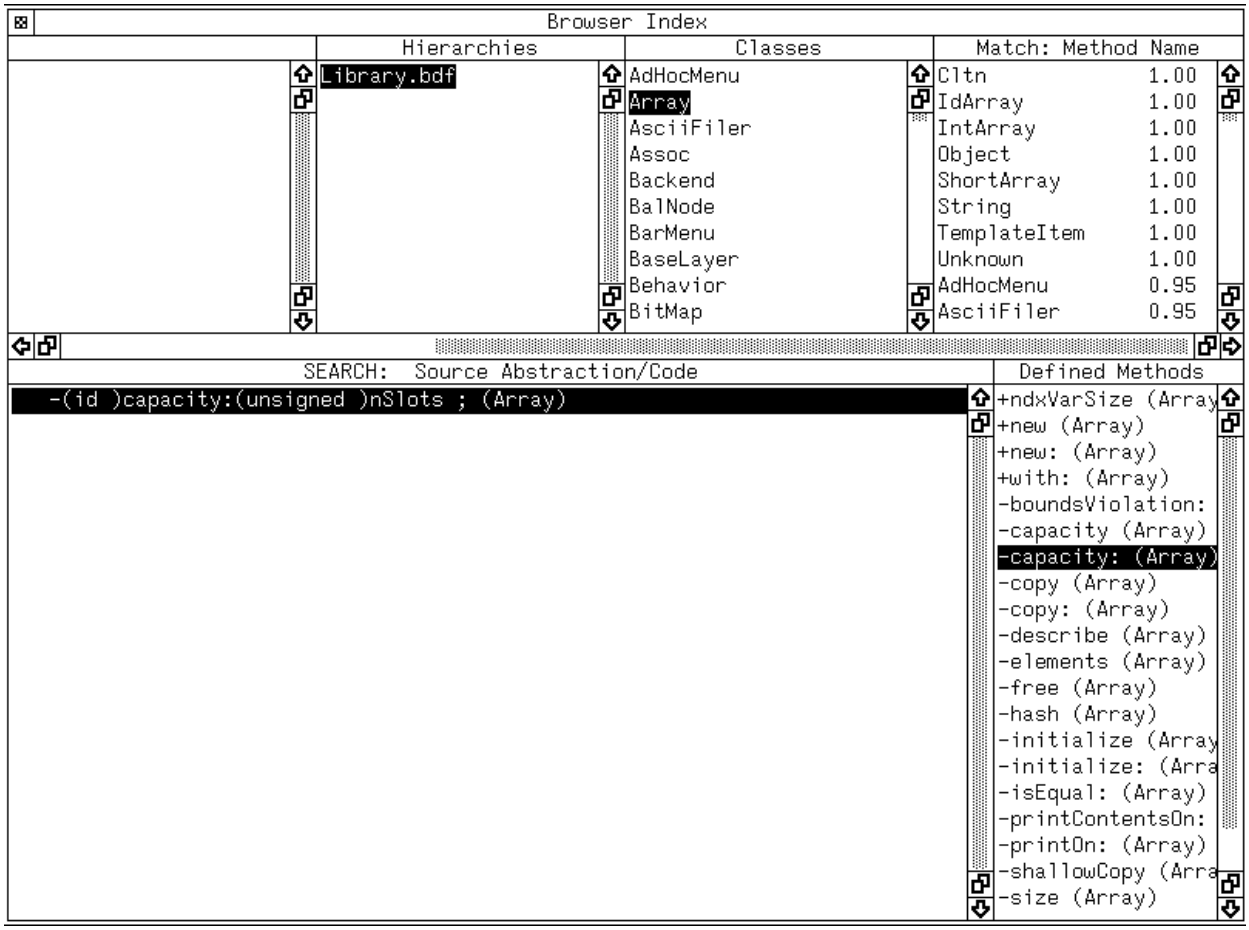


Figure 11 The Browser Interface

These can be used in a matching process which returns a list of classes ranked according to the match score. Each time an operator is applied to an element in a list, the element is highlighted.

4.1.2.1 Class Relationships

Search can take place at the different levels. At the most abstract, the user selects from relationships joining classes. Three new operators were added at this level “Similar Named Classes”, “Similar Classes (Defined)” and “Similar Classes (Inherited)”. The addition of these operators express the idea of reducing ambiguity by including links that express useful notions of connection (as discussed in section 3.7.1.). The first expresses the idea of related

classes having similar names. The latter two express the idea of related classes having similar interfaces and therefore potentially able to perform the same role.

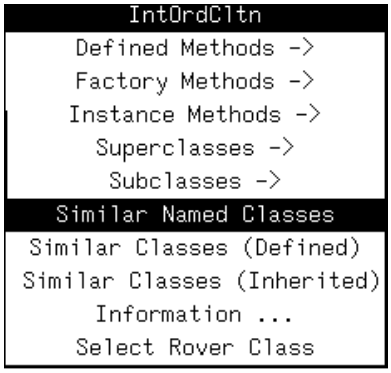


Figure 12 Class Menu Options

Figure 12 shows the menu used to explore class relationships. The top of the menu shows the name of the class selected "IntOrdCltn" in inverse video. When the user selects the desired relationship, in this case "Similar Named Classes", it is highlighted. The name of the selected class is itself highlighted in the original list and the appropriate list returned to its right (Figure 13). Adjacent to each class name is the matching score which is used to rank the returned list.

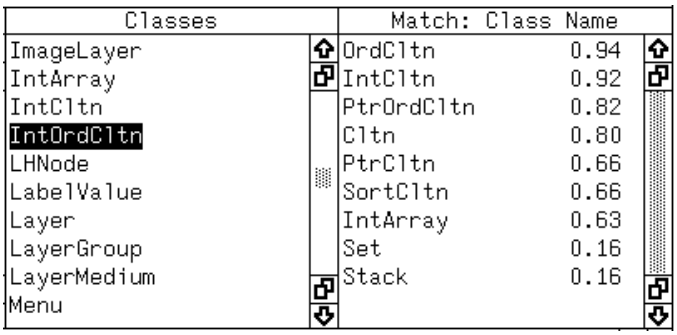


Figure 13 Classes With Similar Names

The way the matching score is computed produces a bias towards words that share the same position in each name counting from the last word. This gives a simple grammatical

form of matching. The last word is effectively a noun and the preceding words adjectives. In the example shown in figure 13 “IntOrdCltn” describes a class that is an ordered collection of integers. The best match on its name is the general class of Ordered Collections, “OrdCltn”. “IntCltn” is a collection for integers, “PtrOrdCltn” an ordered collection of pointers and so on. At the bottom of the list are two classes that are similar due to the fact that their parent is “Cltn” and they have “inherited” a portion of its score.

The other two new operators display a list of classes based on similarity to the selected class’s interface. This is defined by the method names of the class. There are two different options for this similarity. One is based only on the methods defined in the class “Similar Classes (Defined)”. The other ,“Similar Classes (Inherited)”, is based on all methods , i.e. including the ones inherited from ancestor classes. The list returned due to the application of this operator is shown in figure 14.

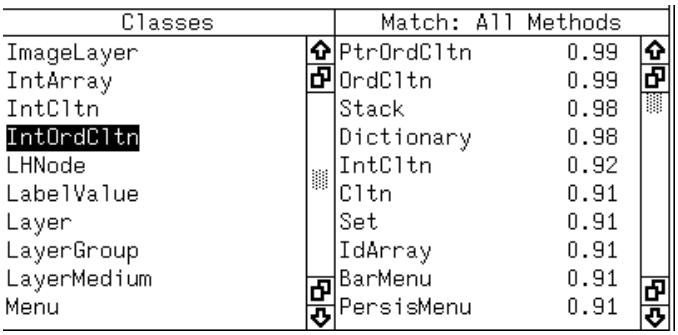


Figure 14 Classes With Similar Interfaces

The matching process takes all the methods in the class and compares them with all methods in other classes. The matching process is similar to that used for the class names. The main difference is the position bias starts from the front of the name. This produces a slightly different grammatical effect where the first word is considered the verb and the subsequent words adverbs. The other two class relationship options, shown in figure 12, are

the subclass and superclass operators which give lists of the children and ancestors of the class respectively. In this case the classes are ordered alphabetically.

4.1.2.2 Lower Level Relationships

After the user has expanded a class to show a list of its defined methods, methods can be expanded and operators applied at this finer level of granularity. This notion of moving down the abstract levels was an important part of the discussion of the general model, as a means of removing ambiguity. In the original browser, when the user selected a method to display its source code, the file where it is implemented was returned with the window positioned over the method (Figure 15). The user was then able to move through this file and look at the rest of the code by scrolling. Although it is possible to determine what is visible in the window a number of problems occur. If the window shows more than one method the user could be interested in any of one of them. In addition the scrolling might be stopped accidently on an irrelevant method which is of no interest to the user.

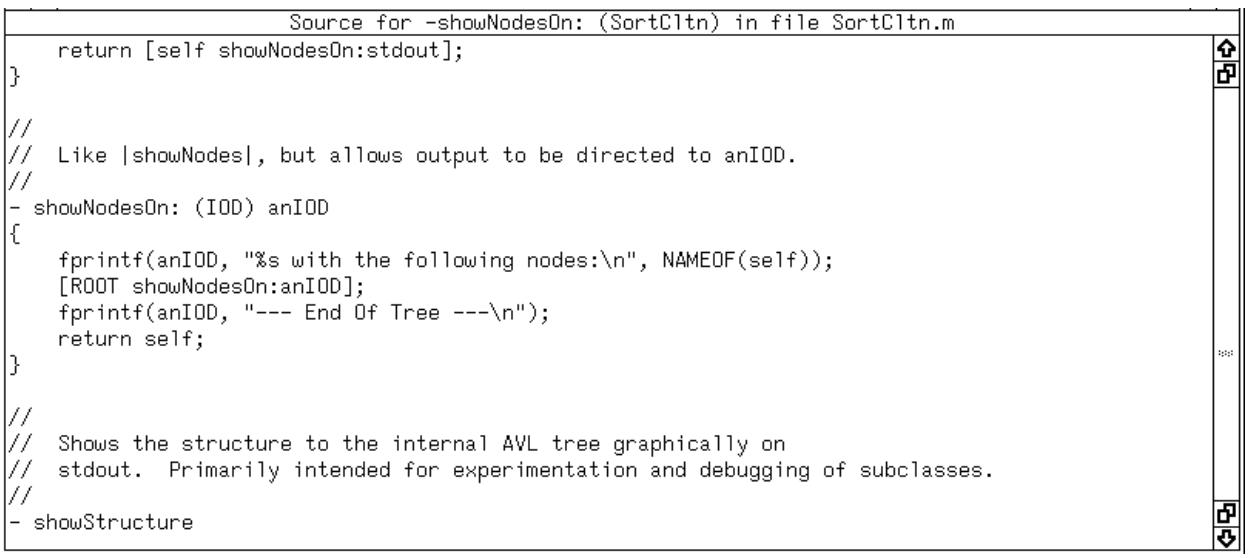


Figure 15 A Single Node For A Class' Source Code

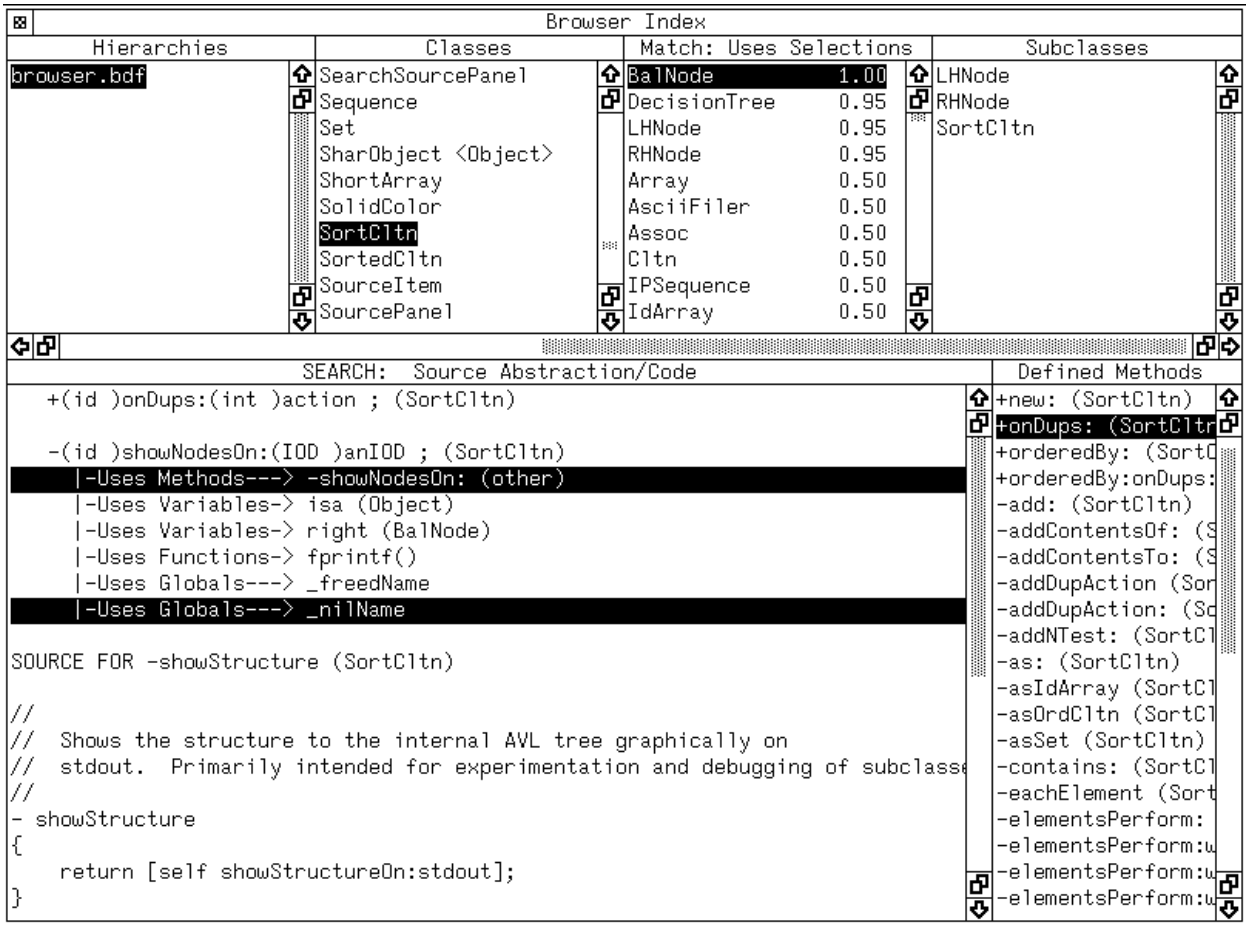


Figure 16 Search At A Lower Level Of Abstraction

The alternative approach has been to first generate a list of methods for a particular class. This is the list on the lower right hand side of figure 16. The selection of one of these will display firstly the method name with its argument types (“onDups:”). The method can then be expanded to a graphical form ("showNodesOn:") and then further expanded to show the actual code ("showStructure"). As discussed previously, the extra steps are important to communication with the independent search mechanism. In addition it aids the user’s rapid appraisal of a method, through a quick determination of its associated relationships, during normal search.

Another important advantage of this approach is that the user is able to request matches

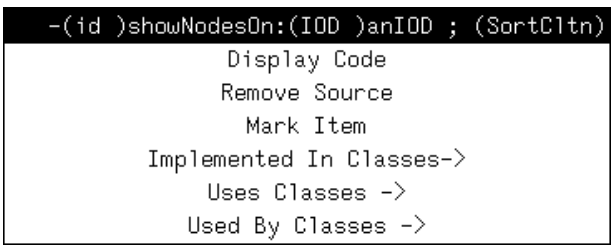


Figure 17 Low Level Search Operators

on more complex conjunctions of properties. In the original browser the only option was to show classes that implement one selected method. In the modified browser one of the options available (Figure 17) within this window is to mark items. Any of the last three options selected after this point refer to all the items marked.

One option "Implemented In Classes —>" looks for classes that implement all the methods selected. If such a class exists it will be given a score of one. Lower scores are obtained by a class that only implements some of the methods or has methods where only some of the word parts match with a selected method. A more complex case is shown in figure 16. The user has expanded class "SortCltn" and displayed one method in its graphical form. Here a user asks for all classes that use method "showNodesOn:" and use global variable "_nilName". At the top of the returned list is the class "BalNode" which matches on both selections. The next three classes do not use these items within their defined methods. Their parent classes, however, do and they inherit the methods and so are assigned a reduced score. In fact the subclass list shows that "LHNode" and "RHNode" are children of "BalNode" and in fact "DecisionTree" is the child of "SortCltn". "SortCltn" itself is not shown because it is deliberately excluded from the list by the matching process.

4.1.2.3 Program Understanding

The standard browser was originally designed primarily to explore program structure with searching for useful software as a secondary role. For the purposes of this research,

search is of most interest so much of the modification was done to enhance this role. Program understanding is then considered as sub-goal of search and is used to determine the utility of a specific component. In program understanding mode the user would normally explore the control/data flow structure. In object oriented programming this means tracing relationships of methods within a class to others, either in the same class or in another.

The effect of having two different modes of operation gives rise to ambiguity of "Situation Dependent operations" discussed in section 3.7.1. If the same operators are used for search as to understand program functionality, the user's intent is unclear and it is difficult to determine the important properties of an item. During search, traversing links is primarily a matter of locating or refining item properties. During program understanding, links are traversed to determine the meaning of a property i.e. what a method does.

The system needs to determine which of these two goals is being pursued. To establish this, search operators and programming understanding operators are made distinct and are related to two different views of the component library. In the search view the library is regarded as a graph of classes, where methods etc. form a subgraph. In the program understanding view it is regarded as a graph of methods. The system is able to determine the user's goal by the particular view selected.

Figure 18 shows the screen for "Programming Understanding" plus one of menus used in this mode. It is an exact copy of the search screen at the time the option for program understanding mode is selected. The only difference at this point is the title and the menu options available. One of the main facilities in this mode is the exploration of method chains. In this example starting with the graphical view of the method "onDups" the option "Source ...." was selected, by the user, for the line "|- Uses Methods—> +new {SortCltn}". This resulted in that method being displayed, which was itself expanded to the graphical

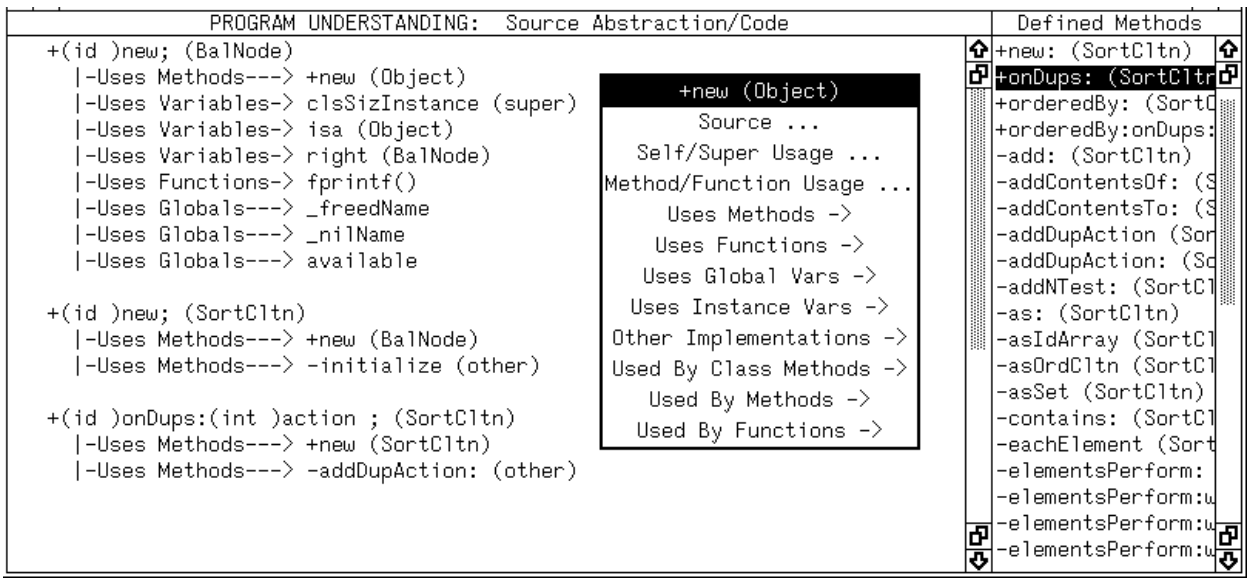


Figure 18 Options In Programming Understanding

form. The process was repeated for the line "|- Uses Methods—> +new {BalNode}" and the resultant method expanded. Thus the graphical representation, or the code if selected, of three connected methods can be viewed together. This has the advantage of being able to easily follow the results of one method using another as part of its functionality.

As can be seen from the menu options many other relationships can be explored which aid in the understanding of the functionality. The actual options given in this window depend on the type of the item chosen. The one shown is for methods in this case "+new (Object)". The selection of an instance variable would only allow the last four options. Not all options result in a display in the same window. Selecting "Used By Class Methods" on an instance variable would result in a list of methods that access it being shown in the rightmost top window of the browser interface. As program understanding is a sub-goal, to continue search this mode must be terminated. This is automatically achieved if any operation is performed on a class.

4.2 Independent Search Mechanism

The previous sections discussed the standard browser and ways of navigating the software library. Each time the user makes a browsing action, the operator used and various context information is passed to the independent search mechanism. It returns a ranked list of potentially interesting classes which are shown to the user.

The independent search mechanism can be divided into two parts. The first is the inference section itself. This takes in browser actions and outputs a series of template predicates. The second is the template matcher which assembles the predicates into a template that can be matched to the items in library. The matcher then compares the template to a class and returns its name and resultant score.

The template matching process can return scores for all classes in the library or one specific class. This is controlled by the meta-rules and depends on how the results will be used. Scores obtained for the whole library are used to produce the list in the "Suggestion Box". This is a display of the names of the classes in descending order of their evaluation scores. Scores from matching to individual classes are used to maintain the system analogue, and thus the template, current with the user's search goal.

Within the following sections particular parts of the architecture will be described in two ways. The first discusses the data that is used during processing and the resultant output. The second gives some detail of the software processing, highlighting the objects used and the messages that are passed between them.

4.2.1 The Inference Process

On receiving the browsing action, it is first checked to see if it is used in search or program understanding mode. A particular sort of action schema, determined by the item

type, is instantiated with the operator and context information. This information is assembled into predicate form, placed in working memory and the inference process initiated.

The engine uses a forward chaining inferencing technique plus Mycin like confidence factors [1]. The factor, for each fact or rule, is a number between zero and one representing the confidence that the fact or rule is true, where one indicates certainty. If a rule is fired after a successful binding with facts the confidence of the consequent is determined by the confidences of both the facts and the rule. The consequent produced is added to working memory as one or more facts. If the same fact already exists the confidence is adjusted using the Mycin approach for positive certainties.

There are three distinct sets of rules used by the inference engine, Meta Rules, Search Rules and Program Understanding Rules. Meta-rules serve two purposes: they control the selection of a subgroup of the Search and Program Understanding Rules (the schema type determines which one) and control when the matching and other tasks occur. After processing the meta-rules, the chosen subgroup of Search or Program Understanding Rules is used to process the action. During this stage, both the system analogue and the template predicates are updated. The latter are passed to the template matching process where they are assembled into a template.

4.2.1.1 Processing Browsing Actions

4.2.1.1.1 Data Processed The action is received by the independent search mechanism as a list comprising of the operator, the node it was performed on and various context information. The action is typically the result of an operator being applied by the user during search but may also be information sent from the browser the result of requests from the independent search mechanism. The output is the display of this information in the debugging windows,

the classification of the action as of a search or program understanding type and a predicate of the form :-

(Node type, node, operator, prior node, prior operator) Confidence

The node type and node define where the user is in the graph representing the library. The prior node and operator are the name of the node and operator that produced the list from which this node was selected. These are used to allow chaining of action predicates to identify particular search patterns. The operator is the menu selection applied to the node. At the end of each predicate is the confidence factor. For browsing actions this is always one. Some predicates that are not browsing actions may have confidences of a lower value. One example is the template matching predicates that are used to maintain the analogue. These are discussed in greater detail in section 4.2.1.2.1.

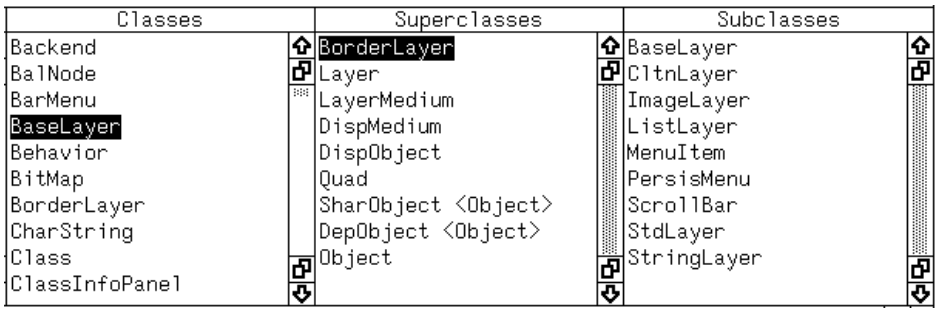


Figure 19 A Sequence Of Browsing Actions

An example of a predicate is produced is the sequence, shown in figure 19, starting with the selection of the menu option “Superclasses”. Applied to the class named “BaseLayer” (a layer used as a base for other layers) this lists its ancestor classes. Applying the menu option subclasses to the item in this list “BorderLayer” (a layer with a black border) would display a list of its children, the siblings of “BaseLayer” . The second predicate produced is :-

(Class BorderLayer Subclasses BaseLayer Superclasses) 1.0

The last action was an operation on the class “BorderLayer” to show the classes joined by the relationship “Subclasses. The prior node is “BaseLayer” and “Superclasses” is the prior operator and is identical to the node and operator of the previous browsing action.

For chaining to occur it is not necessary that these two operators be consecutive. The user may make other menu selections after the first and then return to the ancestor list for the next selection. The prior node and operator allow the system to determine this and thus detect the pattern of the two operators. The chaining of predicates can be extended indefinitely by this means. For instance in this example the user could have expanded “BorderLayer” and reviewed its methods before listing its subclasses.

4.2.1.1.2 Implementation The incoming action is first passed to the “Action List” (Figure 20) . This object first verifies that it is a known action and checks to see if it is used in search or program understanding. To achieve this the object constructs an “Action Type List” , during initialization, for each type of node i.e. Class, Method etc. Each list contains partially instantiated “Action Schemata” for each browsing action of that type. An example of a message that creates the initial schemata is given below. This schemata type is used for nodes that are classes, databases and global variables. In this case the schemata is instantiated with the operator described by the argument “itemName:” which is used during search mode indicated by the “search:” argument being “YES”.

```
[ClassDbGvAction
   itemName:"Similar Classes (Defined)"
   search:YES]
```

On receipt of the browsing action (message a) the type is extracted and the appropriate list (message b) is searched for the right relation (message c). When the action is found a copy of the schema is made and each of the slots filled with the relevant data (message d). An

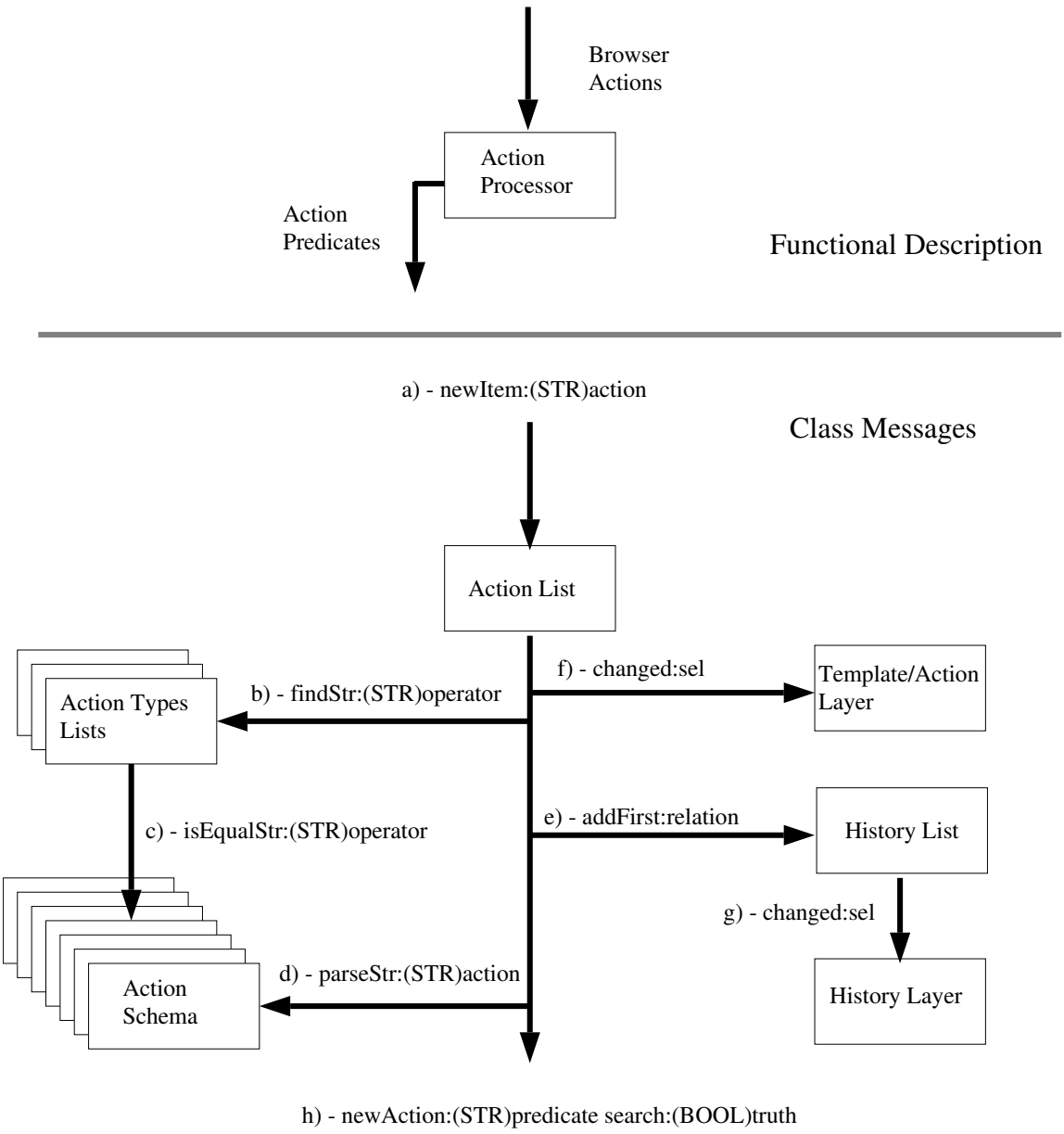


Figure 20 Action Processor

example of the code used to assign values to the slots is shown in figure 21. The majority of the slots represent the information shown previously in the browsing action predicate. Other schema have additional slots dependent on the type, plus the one that indicates whether or not this is a search operator.

The id of the instantiated schema is added to the “History List” (message e) where the

information is displayed on the “History Layer” (message f) in a readable form. It is useful as a record of user search and can be analyzed for interesting patterns. It is written to a file on exit from the browser. The resultant predicate form is added to its own internal collection and displayed on the “Template/Action Layer” (message g) . The output of this stage is a predicate and a flag indicating if it is a search action (message h).

```
// This method splits the browsing action string and uses the
// parts to fill slots in the schemata. The slots are :-
// itemId   : The id used by the browser for this node
// nodeType : The node type i.e. Class, Global Variable etc.
// operator : The operator applied to the node
// nodeName : The name of the node
// prevNode : The name of the previous node
// prevOp   : The name of the previous operator

  -parseStr:(STR)inStr {
     char buff[128];
     strcpy(buff,inStr);
     itemId = strCreate(strtok(buff,","));
     nodeType = strCreate(strtok(NULL,","));
     operator = strCreate(strtok(NULL,","));
     nodeName = strCreate(strtok(NULL,","));
     prevNode = strCreate(strtok(NULL,","));
     prevOp = strCreate(strtok(NULL,","));
     sprintf(buff,"%-12s%-22s%-22s%-22s%-22s",
       nodeType,nodeName,operator,prevNode,prevOp);
     listStr = strCreate(buff);
     sprintf(buff,"(%s %s %s %s %.18s) 1.00",
       nodeType,nodeName,operator,prevNode,prevOp);
     browserAction = strCreate(buff);
     return self;
    }
```

Figure 21 Code For Filling The Schemata Slots

4.2.1.2 Inferring Template Predicates

This section describes how the template predicates are produced with particular emphasis on the rules used for inference.

4.2.1.2.1 Data Processed The inference process uses two types of information to produce template predicates, namely rules and facts. The rules represent the system's knowledge of how to infer the user's intent from actions. These rules are of the following general form

IF (LogicalConnective (Clause1)....(ClauseN))

THEN ((Clause1)....(ClauseN)) Confidence

The rules allow a prefix notation of the two logical connectives "AND" and "OR" in the premise, or antecedent, of the rule. If no connective is specified "AND" is the default. The conclusion, or consequent, always uses "AND" by default. In addition NOT can be prefixed to each clause.

The facts represent what the system has inferred from the users previous actions, the system analogue, and the current action. These facts also have a confidence value and are of the form

(Predicate (Arg1)....(ArgN)) Confidence

If the rule fires after a successful binding with facts, the confidence of the consequent is determined by the confidences of both the facts and the rule. If the antecedent is a conjunction then the minimum confidence value from the bound facts is selected :-

$$\bigwedge_{0 \leq i \leq n} (\mathrm{Confidence_i}) = \min_{0 \leq i \leq n} (\mathrm{Confidence_i})$$

This is multiplied by the rule's confidence and assigned to the consequent. If the antecedent is a disjunction then the maximum confidence value is used :-

$$\bigvee_{0 \leq i \leq n} (\mathbf{Confidence_i}) = \max_{0 \leq i \leq n} (\mathbf{Confidence_i})$$

Any fact produced is added to working memory. If the same fact already exists the confidence is adjusted by the following formula.

Confidence = Old Confidence + ((1 — Old Confidence) X New Confidence)

This has the intuitive effect that confidence is increased proportionally to the difference between certainty and the old confidence. Thus only if the new fact is definitely true is the overall confidence set to one. The system only uses positive certainties but clauses do allow for the provision of "NOT". For the antecedent this condition is met if the fact does not exist in working memory and is assigned a certainty of one. In the consequent a "NOT" causes the fact to be removed independent of its certainty.

There are three types of rules: Meta Rules, Search Rules and Program Understanding Rules. A complete listing of all the rules is given in appendix A. The following discussion will highlight particular features of the rules by giving examples of each type.

Meta-Rules The first role of meta-rules is to select the subset of search or program understanding rules to use. In rule 1 of figure 22, browsing actions that affect methods are processed using search rules marked by the letter M. The designation "UserMethod" indicates that this method was selected by the user in a matching operation.

Rule 2 shows the selection of rules appropriate for a class. The second clause uses the special predicate "BOOL" which returns true on the specified conditions. In this case the condition is that the operator is not equal to "SelRovCla". In addition it generates a number

```
1) IF (OR (Method ?name ?op ?pnode ?pop)
          (UserMethod ?name ?op ?pnode ?pop))
   THEN ((Task rules: M)) 1.0

2) IF ((Class ?name ?op ?pnode ?pop)
       (BOOL NE ?op SelRovCla))
   THEN ((Task rules: C)(NewClass ?name)(NewClass)) 1.0

3) IF ((Method ?name ?op ?pnode ?pop)(NewClass))
   THEN ((NOT NewClass)(Task normalise)) 1.0

4) IF ((newTemplItem)(NOT matchOn))
   THEN ((Task: match)(matchOn)(NOT newTemplItem)) 1.0

5) IF ((EndTask check ?file))
   THEN ((Task readTemplateFrom:thr: ?file 0.45)) 1.0
```

Figure 22 Meta-Rules

of general facts which are used to control when certain processes occur. For instance the fact newClass is used by Rule 3 to degrade the elements of the analogue and template. If a user applies an operator to any class in the library then the firing of the previous rule generates this fact. If the user has opened a class and continues by expanding at least one method to explore its details, then the confidences of the analogue and template terms are reduced. The fact is cleared so that this process is carried out only once per class.

Rule 4 causes a match to take place each time a new template term is generated. A flag is set saying the match is in process to reduce the number of matches if many browsing actions are received in short succession.

Apart from the browser actions there is additional information passing between the browser and the independent search mechanism. The independent search mechanism requests

certain operations of the browser. On completion the browser dumps the information into a file and sends a message similar in form to a browsing action. Rule 5 is a control rule which responds to such a message by selecting a task to read in a template. This template was dumped to a file by the browser after checking the degree of match against the class under review.

Search Rules Search rules are divided into groups dependent on the predicate (Class, Method etc.) and each group assigned a letter. Figure 23 shows some of the two main groups of rules, those that deal with methods and those that deal with classes. On receipt of a browser action on a class, three actions occur. The analogue is updated with the name of the class. The template is updated with the name of the class both as a matching factor and as an excluded class. These inferences are not all parts of the same rule to allow different confidence factors to be used.

The first clause in the consequent of Rule 1 (Figure 23) denotes that the user is "interested in" the class to which the operator was applied. The second clause means that as the user has already shown an interest in the class it should be excluded from the matching process. The confidence reflects the certainty that the system can draw these conclusions. In the latter case as the confidence is not one whether or not it is actually excluded will depend on the evaluation score for the whole template. This will be discussed in greater detail in a section 4.2.2.2.

The consequent of Rule 2 represents the confidence a class's name should be include in the template for matching. Rule 3 increases the confidence the user is "interested in" the class each time one of the methods in that class are explored in greater detail. Rule 4 has the same effect with the matching term in the template.

```
1) C IF ((Class ?name ?op ?pnode ?pop))
     THEN ((interestedIn Class ?name)
           (Templ excludeClass: ?name 1.0)) 0.1

2) C IF ((Class ?name ?op ?pnode ?pop))
     THEN ((Templ className: ?name 0.5)) 0.02

3) M IF ((Method ?name ?op ?pnode ?pop)
         (NOT interestedIn Method ?pnode))
     THEN ((interestedIn Class ?pnode)) 0.05

4) M IF ((Method ?name ?op ?pnode ?pop)
         (NOT interestedIn Method ?pnode))
     THEN ((Templ className: ?pnode 0.5)) 0.01

5) M IF ((Method ?name ?op ?pnode ?pop))
     THEN ((interestedIn Method ?name)) 0.1

6) M IF ((Method ?name ?op ?pnode ?pop))
     THEN ((Templ impMeths: ?name 0.5)) 0.1

7) M IF ((UserMethod ?name SamMetNam ?pnode ?pop))
     THEN ((tried impMeths: ?name ?pnode)
           (Templ impMeths: ?name 0.5)) 0.2

8) T IF ((TempMatch className: ?type ?name))
     THEN ((interestedIn ?type ?name)
           (Templ className: ?name 0.5)) 0.02
```

Figure 23 Search Rules For Incremental Additions

Rules 5,6 are similar to the class rules but for methods. They add terms to the analogue to reflect the user's interest and to the template for matching. Rule 7 is fired when the action is of a particular type indicated by the predicate "UserMethod". Such an action occurs when the user applies an operator that returns a list of classes that match on this and possibly other

methods. The confidence that the template should include a term containing the method name is greater than in the previous rule because the user has shown very specific interest in classes implementing this method. Of course this may only be a transient interest so the confidence is not increased too strongly.

As discussed in section 3.7.2. there is a need to maintain consistency of both the analogue and the template with the user's current search goal. The normalize function, discussed with the meta-rules, reduces the confidences of all terms each time the user opens a new class. In figure 23, Rule 8 reinforces a term that matches with the class under review in proportion to its matching score. This is achieved by the confidence of the "TempMatch" predicate being dependent on how well the specified term matches with an equivalent term in the new class. Suppose that it is a method type with the name "addStr:". If the new class the user has just opened and started to explore, implemented an identical method the confidence would be one but lower if it implemented a similar one e.g. "addInt:". If there is no matching method at all the confidence would be zero.

```
1) C IF ((Class ?name ?op ?pnode Subclasses))
     THEN ((childrenOf Class ?pnode)) 0.2

2) C IF ((Class ?x ?op ?name Sub)
         (childrenOf Class ?name)
         (NOT interestedIn Class ?x))
     THEN ((childrenOf Class ?name)
           (Templ classInh: ?name 2.0)) 1.0
```

Figure 24 Search Rules For Pattern Specific/Generalization Inferences

The search rules are based on three basic methods of inference. Figure 23 shows search rules that involve only incremental additions. Figure 24 shows rules based on both the

pattern specific approach and the generalization of classes visited as part of search. Here the properties of two or more classes are generalized to form a single template. This occurs when the user explores classes that are the subclasses of a specific class. Rule 1 infers the fact that the user is interested in children of the class and this is stored as part of the analogue.

In figure 24 Rule 2 is fired if the user expands a child class. The system infers that the classes should be generalized. This is achieved in the domain specific manner of adding the methods of the parent class to the template. The first clause of the consequent of rule 2 means that the confidence that the user is "interested in children of" this class is increased as the user has expanded a child class. This means that if the user opens a second child, the confidence that the generalization should occur is increased both by the refiring of the same rule and because the confidence of this fact is higher than on the previous occasion. This inference, and therefore the increase in confidence, is repeated each time the user opens another child class.

```
M IF ((Method ?name ?op ?pnode ?pop)
      (interestedIn Method ?pnode)
      (BOOL NE ?name ?pnode))
  THEN ((Templ ?pnode usesMeth: ?name 0.5)) 0.05
```

Figure 25 Program Understanding Rules

Program Understanding Rules As program understanding is a subgoal of search different inferences are used. Paths taken while trying to understand the functionality of particular methods in a class are assumed to be of special interest to the user. For instance if the user looks at the source code of a method used by another method implemented in the class, it is inferred that the search goal may have a similar property. In other words the search goal may implement a method with the same name that also uses the former method. This name

and relationship are added to the analogue and template. The rule shown in figure 25 adds a template term for the implemented method using the other method as part of its processing.

4.2.1.2.2 Implementation On initialization the three sets of rules are read in from their respective files and formed into the internal representation, "Rules". Each rule is an element in one of a number of linked lists. During forward chaining "Facts" are produced and these are passed to "Working Memory" where they are stored as elements in the general, task or template list. The facts and rules are both constructed by the use of "Cons" (Figure 26) which form linked lists in a LISP like form (message k).

When the action predicate is received by the "Engine" it is passed to "Working Memory" (message a) replacing the previous action predicate at the start of the general fact list. "Infer" is then passed a pointer to the first rule in the meta type "Rules" (message b) and the id of "Working Memory" (message c) to start the forward chaining process. "Infer" progresses through each of the rules (message d) and tries to find facts from the general list (message e) that bind to each of the clauses in the premise of the rule (message f) . If a successful substitution is found, a conclusion is produced (message g) of one or more clauses and these are sent to the "Working Memory" (message h) . These are converted to facts (message i) and stored. In the case of meta-rules the facts produced are mostly of the task type and are stored temporarily in a list (message j).

Figure 27 shows the code for executing tasks. A "Task" predicate is broken down into its constituent parts. The first variable in the predicate is the query, a method name. Subsequent variables are used as arguments to this method. The task is started by the use of the general method "perform:with:with:with:" which executes the specified method with as many arguments as given.

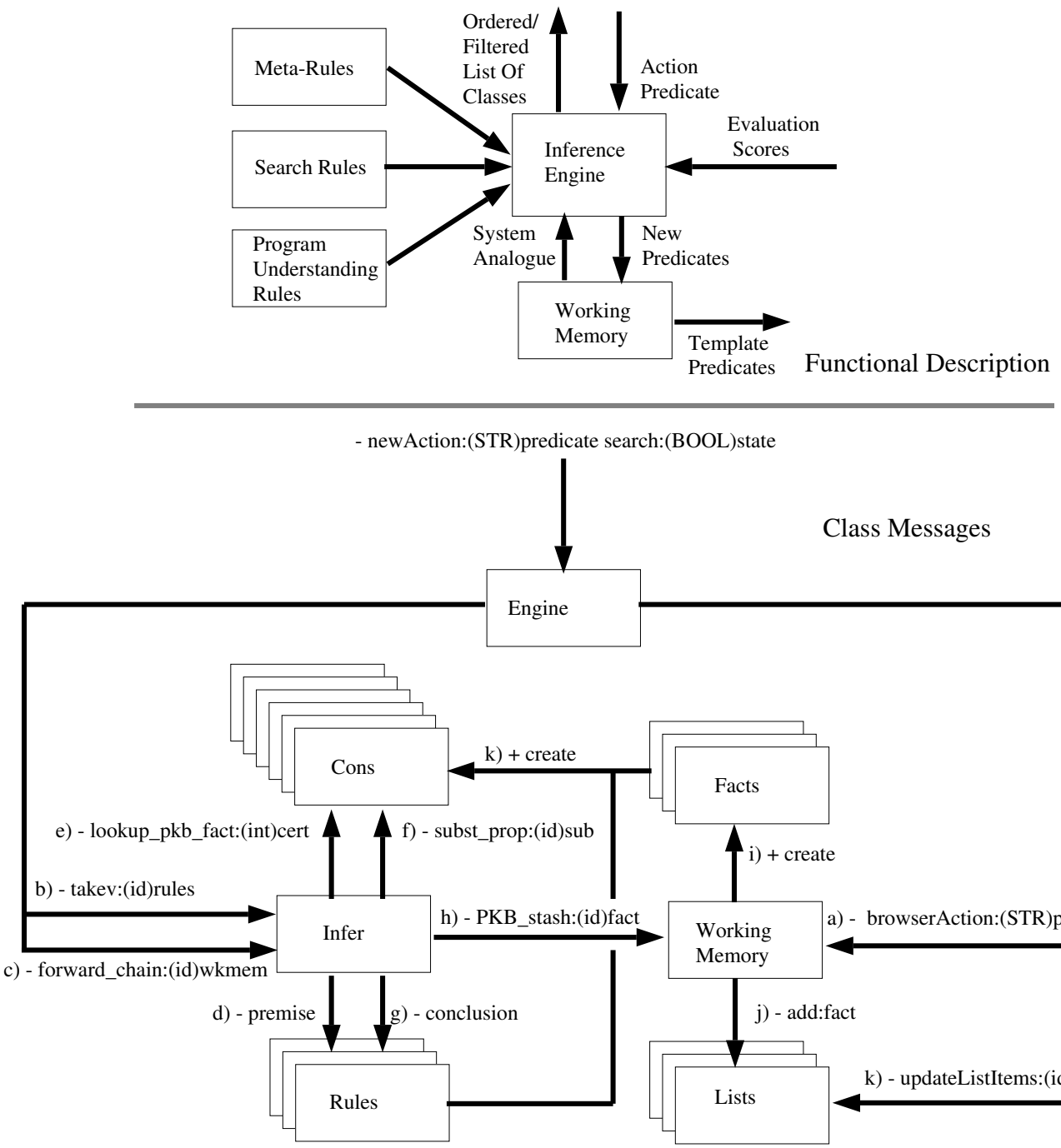


Figure 26 The Inference Engine

After forward chaining on the meta-rules halts, because all possible bindings have been tried, control is returned to "Engine". The list containing the tasks is emptied executing each task in turn. When the task is to select rules, if the action is a search type, a subgroup of

```
// Method to process task predicates.
// The last element in the list of tasks is extracted
// and executed. This process is repeated until all
// tasks have been executed.

-(BOOL)doTasks:(int)no {
  char  buff[128], *query, *arg[3];
  BOOL rules = NO;
  id cltn, task;
  cltn = [taskList listItems];
  while  (task = [cltn lastElement]) {
    sprintf (buff, "%s", [task listStr]);
    [cltn remove:task];
    [task free];
    strtok(buff, " ");
    query = strtok(NULL, " )");
    if (!strcmp(query,"rules:")) rules = YES;
    arg[0] = strtok(NULL, " )");
    arg[1] = strtok(NULL, " )");
    arg[2] = strtok(NULL, " )");
    [self perform:query with:arg[0] with:arg[1] with:arg[2]];}
   return rules;}
```

Figure 27 Code For Task Predicates

search rules is selected by using the method “rules:” (Figure 28). To process a browsing action “Infer” is passed a pointer to this new list. The same process, as described for meta-rules, is repeated but this time mainly analogue or template predicates are produced. These are passed to “Working Memory” and stored in their respective lists (message i). At the end of the process “Engine” updates the window display by asking the various lists used during the process to update themselves (message k).

4.2.2 Template Matcher

The template matcher takes in the template predicates and forms them into weighted

```
// Method to selects the appropriate subgroup of search rules

-rules:(STR)s {
  switch (*s) {
    case 'T': RuleDataBase = SearchRules[TempMatch1]; break;
    case 'S': RuleDataBase = SearchRules[TempMatch2]; break;
    case 'C': RuleDataBase = SearchRules[Class]; break;
    case 'M': RuleDataBase = SearchRules[Method]; break;
    case 'I': RuleDataBase = SearchRules[InstVar]; break;
    case 'F': RuleDataBase = SearchRules[Function]; break;
    case 'G': RuleDataBase = SearchRules[GlobalVar]; break;
    case 'L': RuleDataBase = SearchRules[Lists]; break;
  default:   break;}
 [[inference takev:RuleDataBase]
       forward_chain:workingMemory];
  return self;}
```

Figure 28 Code To Select Subgroup of Search Rules

terms which are added to the template. This template is then compared to each class term by term. The score for each term is calculated and added to the score of other terms. This is normalized so a perfect match would give a value of one. The classes and the evaluation scores are used to produce the list of suggested classes.

4.2.2.1 Data Processed

The main input to the template matcher is template predicates of the general form:

(Template, Property, Node, Scale Factor) Confidence

Property denotes the node type. In the example given below the term describes node "OrdCltn" which is a class name.

(Templ className: OrdCltn 0.5) 0.10

The template predicate also has two associated numbers. The confidence value is determined by the interest the user has shown in a particular node. The scale factor is a fixed value specified in the rule. The scale factor determines the importance assigned to this type of term in the template. These values are multiplied together to give the weight for the specific template term. In this example the weight of the template term would be 0.5 X 0.10 = 0.05.

There is one variant on the general form which allow the expression of relationships:

(Template, Related Node, Relationship, Node, Scale Factor) Confidence

In this case there is an extra argument, the related node. The "Relationship" expresses the link between the two nodes. In the following example the classes match strongly, on this term, will access the global variable "freedName" using a method called "—showNodesOn:"

(Templ —showNodesOn: usesGlob: _freedName 0.5)) 0.3

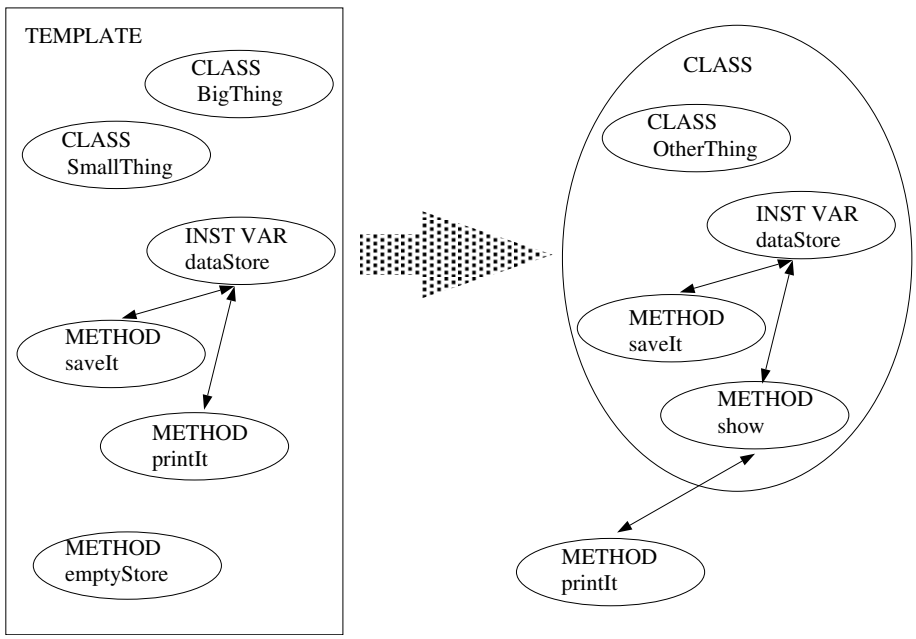


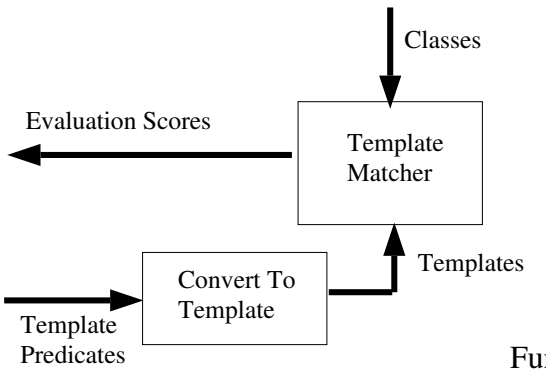
Figure 29 Matching Templates To Classes

The template is formed by combining the template terms together and is similar to a class. In a template a node type might have multiple values, even if this is not possible in an actual class. This can be seen in the following example (Figure 29). On the left is an abstract representation of a template. It contains two class names “SmallThing” and “BigThing”. In addition it contains terms for three method names “emptyStore”, “printIt” and “saveIt” and for the latter pair relationships to the instance variable “dataStore”. On the right is shown a class that would return a good matching score. Firstly the class name “OtherThing” matches with both the template class names due to the common subterm “Thing”. Secondly the method “saveIt” is implemented in the new class and has the same relationship to an instance variable with the same name “dataStore”. Lastly although “printIt” is not implemented in the class, the method “show” uses it as part of its processing. The matching algorithm returns a score inversely related to graphical distance.

The matching process is identical to that used by the operators added to the standard browser discussed in an earlier section. The actual matching software is describe in the next section.

4.2.2.2 Implementation

The process is started as the result of a fact being added to the list of tasks while processing the meta-rules. When this task is executed it tells the “Action Processor” to carry out a matching process (Figure 30). The action processor cycles through the “Template Facts” instructing them to add themselves to a selected template (message a). Each fact first creates a “Template Item” with the appropriate node, related node (if required) and weights (message b). It then adds this term to the appropriate matching lists, dependent on the term type (message c). After the template is complete it is placed in an ASCII file and a message sent through a pipe to the standard browser.



Functional Description

Class Messages

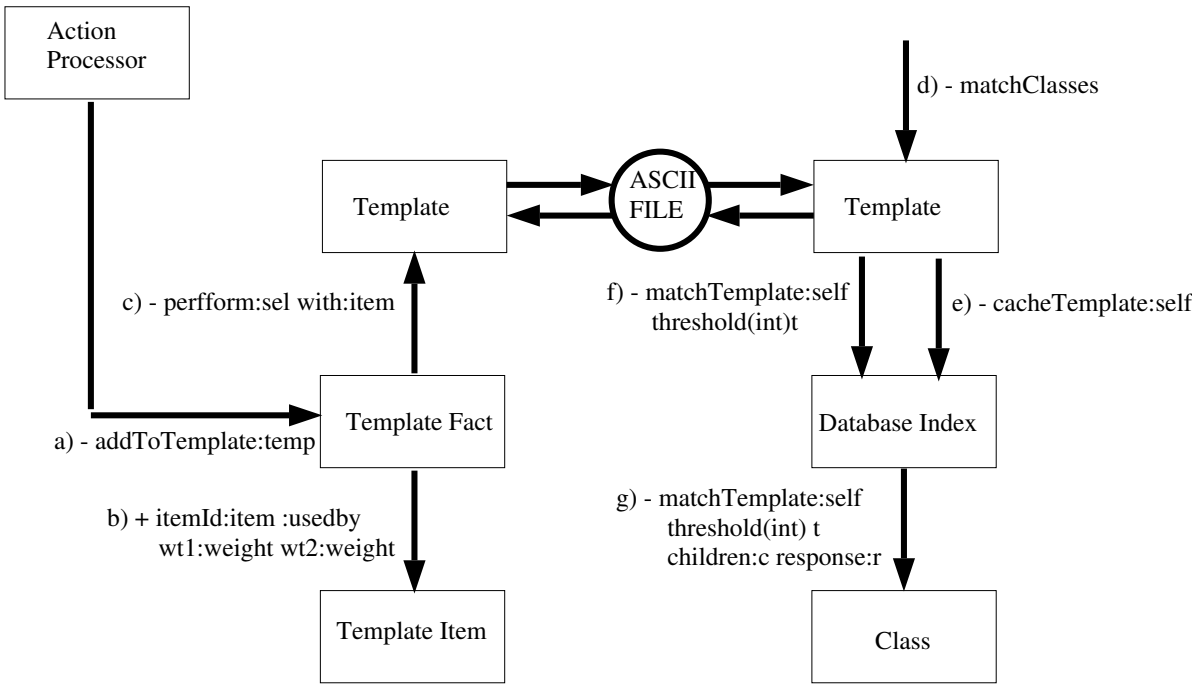


Figure 30 Template Matcher

Although the independent search mechanism is shown distinct from the standard browser, the actual matching process is carried out in the browser. When the browser receives the message it recreates the template from the ASCII file. The “Template” is directed to match itself against the library classes (message d). Firstly “Database Index” is asked if this term

has been checked before (message e). This will have occurred if it was used in a template matched against the library earlier in the search. If it has been checked before and is still held in the cache, the cached term, which includes all the previous matching scores, is used to replace the existing term in the template. If not the template term is added to the cache (the correct values will be added to the term during matching).

```
// Each class name is extracted in turn and compared to
// the name of this class. If the score is less than the
// parents multiplied by 0.4 the latter score is used.

    if (cltn = [tid className]){
      seq = [IPSequence over:cltn];
      while (item = [seq next]) {
        weight = [item wt];
        totalWeight += weight;
        if ((matchScore = [item scoreAt:myNo]) > 0.0) ;
        else {
          matchScore = [self matchClassName:[item itemId]
                                        and:self];
          if ((score = ([item scoreAt:parentNo] * 0.4))
                 > matchScore)
                 matchScore = score;}
          [item score:matchScore at:myNo];
          totalScore  += matchScore * weight;}
        [seq free];}
```

Figure 31 Code To Cycle Through Template Class Names

The template then directs “Database Index” to match the template and passes a minimum matching score to be achieved before a class should be added to the list (message f). This score or the score of matching the template to the most general class “Object”, which ever is the larger must be exceeded to be included in the list. “Database Index” passes the template

to the root of the inheritance tree (message g) where the first match occurs. The root passes the template to its children, who do the same in turn until all the classes have been matched.

```
// This method compares the two class names passed as
// arguments. There are bias terms for position and length
// The final score is reduced by applying a square function
// to reduce the difference between a partial and
// a perfect match

   -(float)matchClassName:m1 and:m2 {
      char *m1Parts[20], *m2Parts[20], buff1[128], buff2[128];
      int i1 = 0, i2 = 0, j1,j2;
      float tempScore = 0, score = 0;
      strcpy(buff1,[m1 splitName]);
      strcpy(buff2,[m2 splitName]);
      m1Parts[i1] = strtok(buff1,",");
      while ((m1Parts[++i1] = strtok(NULL,",")) != NULL);
      m2Parts[i2] = strtok(buff2,",");
      while ((m2Parts[++i2] = strtok(NULL,",")) != NULL);
      for (j2 = 1; j2 <= i2 ; j2++){
        for (j1 = 1; j1 <= i1 ; j1++){
         if (!strcmp(m1Parts[i1 - j1],m2Parts[i2 - j2])){
           tempScore =
             1.0 / ( 1.0 + (0.2 * (float)abs(j2 - j1)));
                 break;}}
       score += tempScore;
       tempScore = 0;}
       score /= i2 + (0.2 * (float)abs(i2 - i1));
       score = sqrt(score);
       return score;}
```

Figure 32 Code To Calculate Match Score On Class Names

The matching process is based on dividing the names down into their constituent words. Figure 31 shows the code that cycles through the template terms for class names. Each term

is extracted and its weight retrieved and added to the total weight for the whole template. This is used to normalize the score for the class. If there is already a score for the term this is used as the current score. If not the code shown in figure 32 is used to calculate one. The returned score is compared to that of the parent reduced by a factor (0.4). If it is smaller than this the reduced score of the parent becomes the current score. The score chosen by the routine is then multiplied by the new weight and added to the total score of the class.

The matching code (Figure 32) determines the score assigned to a class name term. This is the same matching as used for the user's operators, discussed in the section 4.1.2. The class name is already split into its constituent words, e.g. "BaseLayer" becomes "base layer". The name of the present class and the name in the term are compared word by word. In this case the comparison starts at the last word. When the score is computed by comparing each term in one name with all words in the second the score is reduced by a factor related to the difference in length of the names. The final square root function lessens the numerical difference between a total and partial match.

4.3 Conclusions

This chapter has discussed an implementation of the generalized architecture given in chapter 3. In particular the preceding sections demonstrated how the browsing actions have been extracted and how a forward chaining inference engine is used to form an analogue representing the user search goal. It was further demonstrated how this analogue is converted to a template and used to evaluate terms in a template and thus assign a score to a class.

The final stage of the process is using the results to indicate interesting classes to the user. At the end of the matching process described in the preceding section a score has been assigned to every class in the library. The classes are assembled into a list. To be included

the class must have a score greater than a fixed value or the score of the most general class "Object" which ever is the larger. The list is sorted by the evaluation score, with the highest score at the top. The result is a window showing a ranked list of class names each with the matching score (Figure 33) shown to the user as potentially interesting classes.

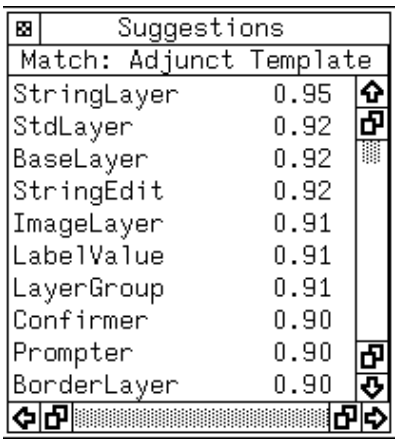


Figure 33 Suggestion Box

Chapter 5 Experimental Results

This thesis has proposed the method of active browsing to address the problem of locating items in extensive libraries. The main purpose of this chapter is to validate the fundamental principle of active browsing by demonstrating that the user's search goal can be inferred solely from normal browsing actions. Section 5.2 also serves a second purpose namely to illustrate the mechanics of the inference process. The following section describes the overall rationale of the experimental methodology. The last two sections discuss the particular experiments and their results.

5.1 Overview Of Experimental Rationale

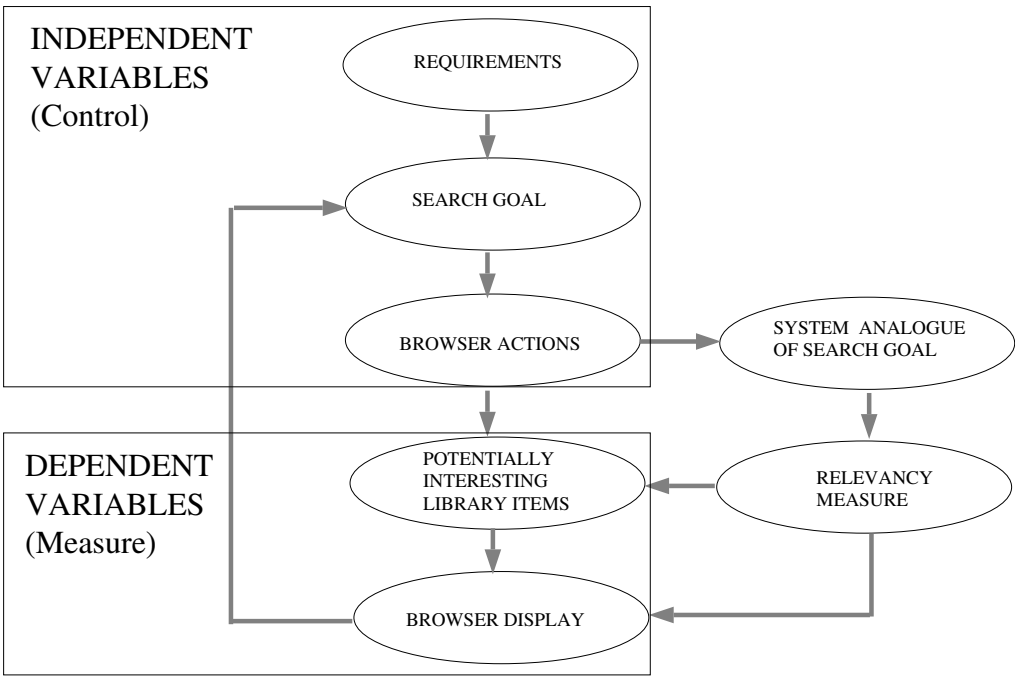


Figure 34 The Experimental Methodology

The methodology (Figure 34) adopted for the following experiments reflects back to the view of active browsing presented in section 3.2 . The independent variables — the

inputs to the experiments that are directly controlled — are the requirements, the way they are translated into a search goal and then progressively mapped to a series of browsing actions. The dependent variables — the outputs from the experiments that are measured — are the items displayed and their respective ranks. The following sections describe how this methodology is used and the results achieved.

Validation of the technique can be considered to consist of three stages. The first stage, plausibility, addresses the question "will it ever succeed ?". To answer this it must be demonstrated that the approach produces a definite benefit in realistic circumstances. This is essentially an existence proof. To meet this requirement, section 5.2 of this chapter outlines examples of the active browser's successful operation in situations which are likely to occur during search.

The second stage, generality, requires that the benefit obtained occurs with sufficient frequency, in a wide range of circumstances, to be of value. The third stage, effectiveness, requires that a sufficient gain is obtained to justify the cost. Both these factors could be measured by carrying out tests with a large number of users with many different requirements. An alternative approach is to use a system that simulates user browsing actions. Section 5.3 describes the heuristic based simulator, how this is used to generate a large variety of different searches, and the results obtained.

5.2 Plausibility

This section describes two different examples of system operation. They use a small library (circa 80 classes) consisting of a combination of two commercially available sets of classes. These are the "ICpak101 Foundation Classes" and the "ICpak201 User Interface Classes" produced by the Stepstone Corporation.

With reference to the experimental methodology, both these examples start with a vague set of requirements and by implication an ill-defined search goal. This is translated into a plausible set of browser actions. The ranking of relevant classes displayed in the suggestion box is a measure of the success of the inference.

In both cases the active browsing system returns different classes to the ones shown as a result of the user application of operators. The examples both validate the different methods of inference and present practical situations where benefit is achieved. The first demonstrates the notion of directing the user's interest to a class that has particular characteristics. The second demonstrates expanding the search area to highlight alternatives to the classes being explored. These examples of operation are not meant to be a complete searches, but merely stages in a longer search.

5.2.1 Focusing The Search

This example demonstrates how the system focuses the search by using information collected over time. It illustrates the inference method termed "Incremental additions" discussed in section 3.5.1.1. The user opens two quite different classes exploring properties in each. In this example the user requires a means of displaying text and therefore shows an interest in "string-like" and "layer-like" items. Thus a plausible inference is that the user might be particularly interested in items with both properties. The system returns as the best match a class which contains a conjunction of these properties.

Figure 35 shows the state of the standard browser after application of the series of browsing actions described below. Figures 36 and 37 show the browser actions in predicate form, on the left, and the template predicates produced, on the right, at two stages in the process. Suppose that the user starts with a list of all classes in the library. This is an alphabetically ordered list of class names. The user scrolls until the class "String" is visible

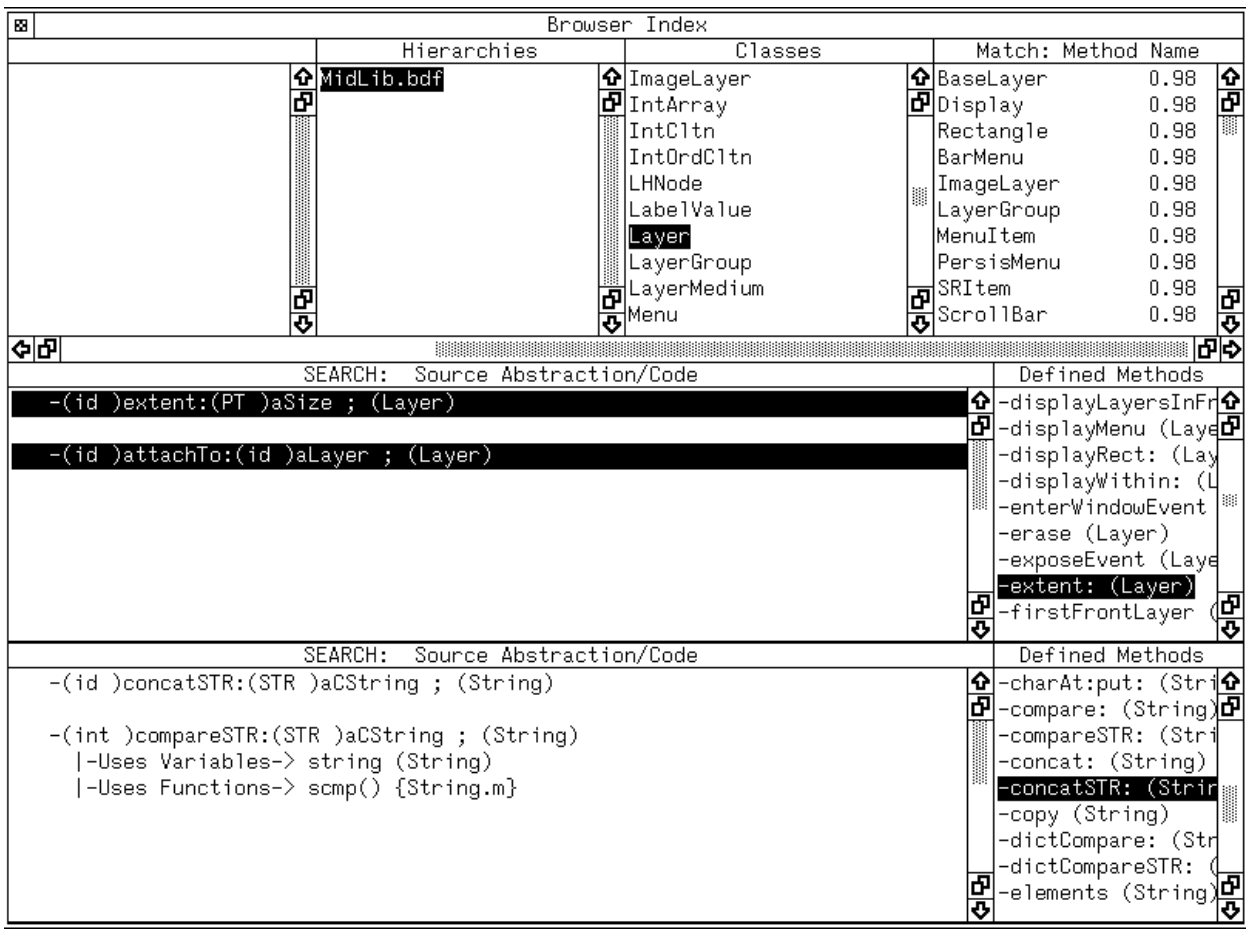


Figure 35 Exploring Two Different Classes

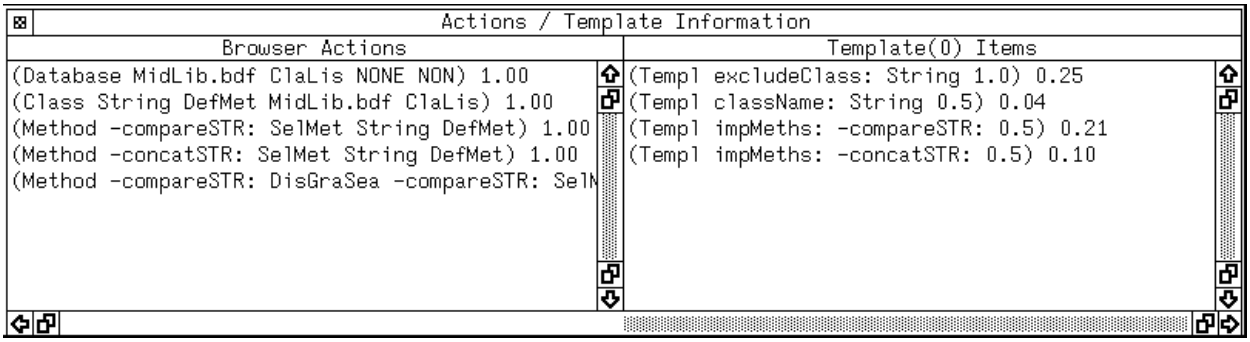


Figure 36 Predicates After Exploring Class “String”

and then applies the operator “Defined Methods” to produce a list of method names. This action produced the top two template terms “excludeClass:” and “className:” (Figure 36) which were added to the template. The method "concatSTR:" is then selected from the

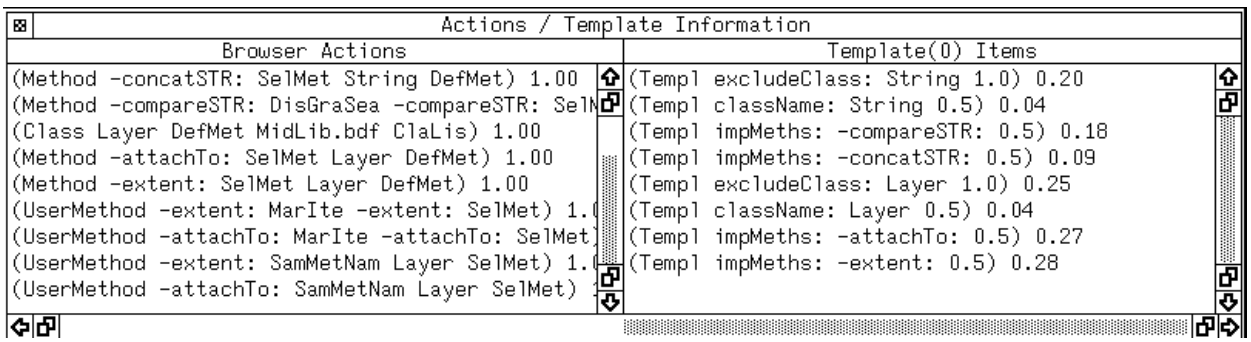


Figure 37 Predicates After Exploring Class “Layer”

list and shown in the “Source Abstraction/Code” window along with the input/output types associated with this method. Next the method "compareSTR:" is selected and expanded to show the instance variable and function used by the method, as well as the input and output types. This produces the next two template terms, the latter with a higher confidence due to this expansion. Figure 36 shows the browsing action and template predicates produced up to this point.

Suppose that the user then returns to the original list, selects the class “Layer” and displays a list of its methods. An operator is applied to split the lower screen so that the two classes could be viewed together. The two methods “extent:” and “attachTo:” are expanded. They are highlighted and the operator “Implemented In Classes” selected returning the list in the upper right hand corner (Figure 35). The complete browser action and template predicates are shown in figure 37. Of particular note is that the confidence factors of the template predicates associated with the two methods from class “Layer” are higher than those produced in class “String”. This stems from the browsing action predicates of the type “UserMethod”. They are a result of the user specifically directing the browser to find other classes where the methods are implemented. This gives a much stronger confidence in the user’s interest in these as properties of the search goal.

The classes shown in the list, returned as a result of applying the operator “Implemented

In Classes” (top right of figure 35), have almost identical scores (the display shows only two digits of precision). The methods selected, or similar ones, are extensively used in layer type classes, so the user’s operation is a poor discriminator.

The active browser discriminates better. To match maximally with the template, and thus be high in the suggestion box, a class should meet various criteria. It should have a name that matches with "String" and "Layer". It should match with methods “extent:” and “ attachTo:" and more importantly with "concatSTR:" and "compareSTR:". It should be a class the user has not shown a lot of interest in previously. The suggestion box (Figure 38) shows “StringLayer” as a clear favourite, a class used for the display of strings of text. This is due to the user’s previous interest shown in properties of strings. A second layer type potentially of interest is fourth in the list. “StringEdit” is similar to “StringLayer” but in addition it allows editing of text. Thus active browsing has focused the search on string type layers.

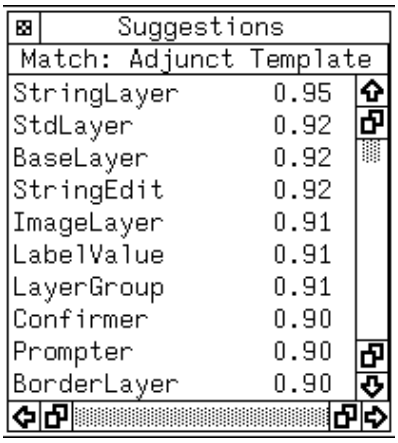


Figure 38 Suggestions

5.2.2 Expanding The Search Area

This example demonstrates that active browsing extends the area of search by suggesting classes that are remote in the class hierarchy. Here use is made of all three inference methods discussed in section 3.5.1. In this example the user requires collection type classes. Thus

a plausible inference is that all classes with such properties are potentially interesting. The example shows that the active browser retrieves other classes with similar properties not included in the user's requested list.

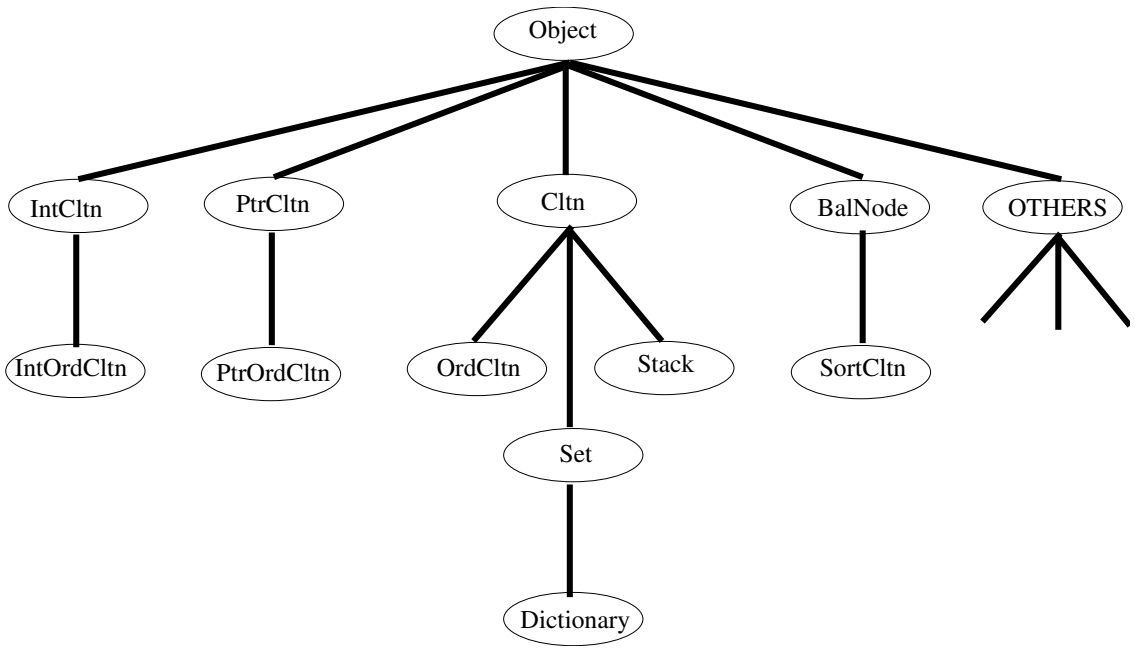
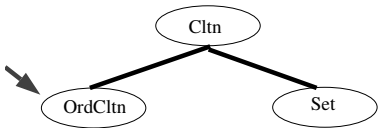


Figure 39 Class Hierarchy

Figure 39 gives part of the hierarchy for the library. The main points of interest in this example are four groups of classes, all types of collection. The central cluster, "Cltn" and its children, are collections whose elements are added and extracted sequentially or by indexing using the list position. There are also two special types: those with methods specific to integers "IntCltn" and "IntOrdCltn" and those with methods specific to pointers "PtrCltn" and "PtrOrdCltn". The fourth type "SortCltn" has elements that are added and automatically positioned alphabetically. This type is based on a balanced tree for rapid update and retrieval.

For the purposes of browsing the inheritance structure is just one of many types of relationship between classes. Still in object oriented programming this is the principal structure for any library and thus a likely one for the user to explore. We suppose in

this example the user starts by studying some methods in one class and then decides to review its siblings as alternatives that have the same general properties. The active browser suggests other classes with similar properties not included in the list of siblings.



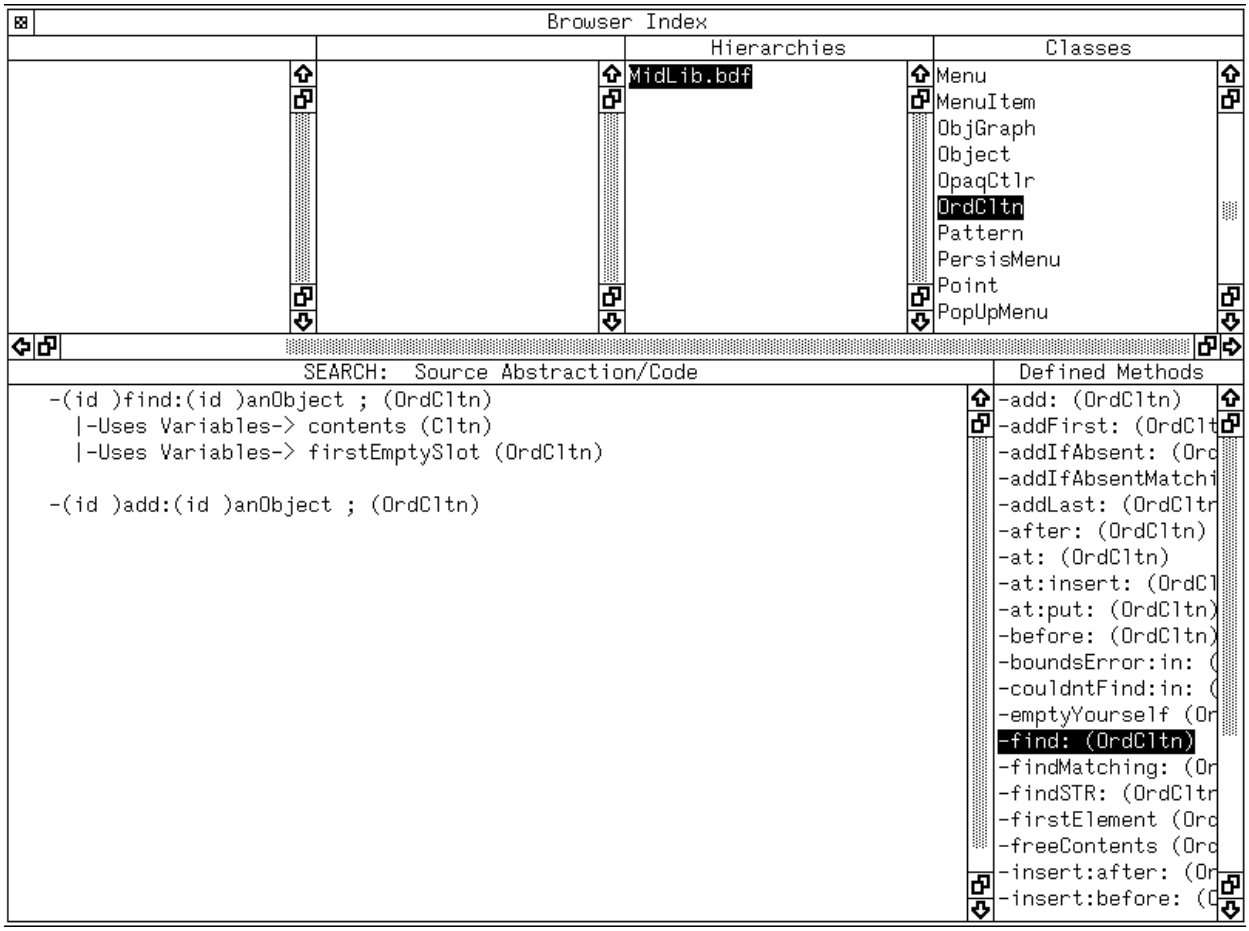


Figure 40 Exploring The First Sibling

The example begins with the user selecting the class “OrdCltn” , from the general list of classes, and expanding it to show the locally defined methods. The result is the list on the lower right hand side of figure 40 giving the methods locally defined in the class. The

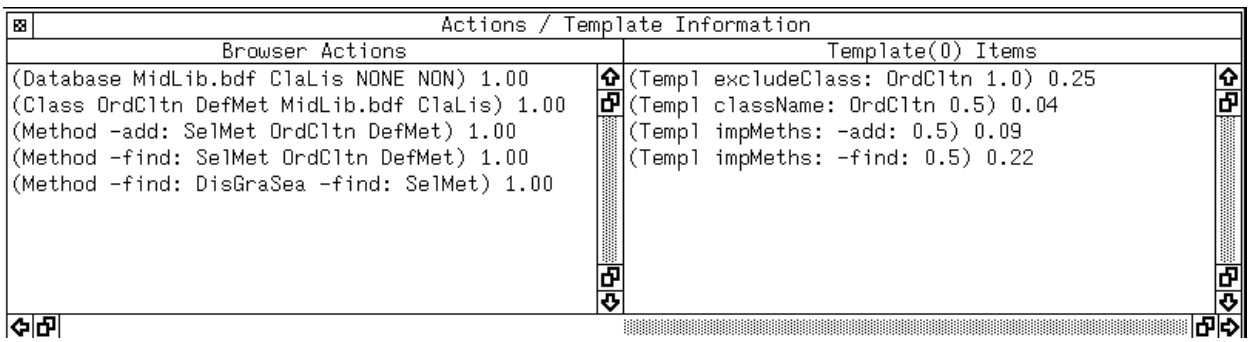


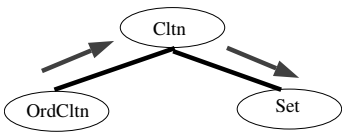
Figure 41 Predicates After Visiting The First Sibling

methods "-add:" and "-find:" are selected from the list and the latter expanded to its graphical form. Figure 41 shows these actions in predicate form and the inferred template predicates.

The operator ”Superclasses” is then applied to “OrdCltn” which returns the ancestor list containing “Cltn” (Figure 42). By applying the operator “Subclasses” to “Cltn” a list of its children are displayed. The result is a list of sibling classes including “OrdCltn”. The class “Set” is selected, its methods viewed and a method, with the same name, "-find:" is inspected. On this occasion the user expands it one more level to show the actual code of this variant of the method. Figure 43 shows these actions and the template predicates.

The template includes the names of the three classes visited and the methods inspected. The method "-find:" has a higher confidence factor than "-add:" as it was selected in both classes and examined more closely. There are template terms to exclude three classes. Those for classes “OrdCltn” and “Set” have higher confidences than that for “Cltn” because the user explored them in detail. The second from last item shown in figure 43 is added as a result of a rule that implements the sibling search notion discussed in section 4.2.1.2.1. The resulting template item directs the template matcher to compare classes with all methods implemented in, or inherited by, the parent class “Cltn”.

In this case to match maximally with the template a class should have an interface similar to class "Cltn" and have a name that matches with "OrdCltn", "Cltn" and "Set". It should



Browser Index

Hierarchies	Classes	Superclasses	Subclasses
MidLib.bdf	Menu	Cltn	OrdCltn
	MenuItem	SharObject <Object>	Set
	ObjGraph	DepObject <Object>	Stack
	Object	Object	
	OpaqCtlr		
	OrdCltn		
	Pattern		
	PersisMenu		
	Point		
	PopUpMenu		

SEARCH: Source Abstraction/Code

```
SOURCE FOR -find: (Set)

//
//  If any element in the receiver |isEqual:| anObject, returns that
//  element, else returns nil.
//
- find: anObject
{
    return anObject ? *[ self findElementOrNil:anObject ] : (nil);
}
```

Defined Methods

+new (Set)
+new: (Set)
-add: (Set)
-addContentsTo: (S
-addNTest: (Set)
-contains: (Set)
-difference: (Set)
-emptyYourself (Se
-expand (Set)
-filter: (Set)
-find: (Set)
-findElementOrNil:
-freeContents (Set
-intersection: (Se
-occurrencesOf: (S
-remove: (Set)
-replace: (Set)
-shrink (Set)
-size (Set)
-union: (Set)

Figure 42 Exploring The Second Sibling

Actions / Template Information

Browser Actions	Template(0) Items
(Method -add: SelMet OrdCltn DefMet) 1.00	(Templ excludeClass: OrdCltn 1.0) 0.24
(Method -find: SelMet OrdCltn DefMet) 1.00	(Templ className: OrdCltn 0.5) 0.06
(Method -find: DisGraSea -find: SelMet) 1.00	(Templ impMeths: -add: 0.5) 0.09
(Class OrdCltn Sup MidLib.bdf ClaLis) 1.00	(Templ impMeths: -find: 0.5) 0.46
(Class Cltn Sub OrdCltn Sup) 1.00	(Templ excludeClass: Cltn 1.0) 0.09
(Class Set DefMet Cltn Sub) 1.00	(Templ className: Cltn 0.5) 0.02
(Method -find: SelMet Set DefMet) 1.00	(Templ excludeClass: Set 1.0) 0.16
(Method -find: DisGraSea -find: SelMet) 1.00	(Templ className: Set 0.5) 0.03
(Method -find: DisCodSea -find: SelMet) 1.00	(Templ classInh: Cltn 2.0) 0.21

Figure 43 Predicates After Visiting The Second Sibling

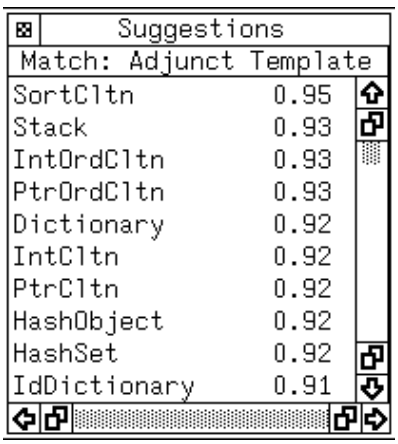


Figure 44 Suggested Classes

implement methods "add" and more importantly "find" and be a class the user has not shown a lot of interest in previously.

The result of this template being matched is shown in figure 44. The third sibling “Stack” , as yet unexplored by the user, is not the next best match. There is a higher scoring class, “SortCltn” (Sorted Collection). In addition classes “IntOrdCltn” and “PtrOrdCltn” score above “Dictionary”, a child of “Set”, and several others tie. These and the remaining classes in the list are situated in unrelated parts of the library. The only common ancestor is “Object” the root of the inheritance tree. The high score is due to the extensive polymorphism deliberately designed into the library. These classes are intended to fulfill similar roles to the children of “Cltn” but because of their implementation differences are remote in the hierarchy.

5.3 Generality and Effectiveness

The traditional method used to assess the performance of information retrieval systems, is to measure recall and precision. “Recall” relates to the systems ability to retrieve all relevant items, “precision” to retrieve only relevant items. W. Croft [16] summarizes this approach in the statement ”Retrieved documents can be assessed as to their relevance to the request, and these relevance judgements form the basis of evaluation of the effectiveness of

retrieval". This is not really applicable to the system described in this thesis. To measure these characteristics one has to assume a good mapping from requirements to a query. The argument presented earlier in support of browsing was that this mapping is far from clear. In fact the user is not consciously making a query at all. A better approach would be to give a series of development tasks to a variety of users requiring them to search the library. Results of speed and accuracy could then be compared with and without the support of the active browser. This requires a large number of experimental subjects and could not be justified with the research at this level of maturity.

In lieu of such experiments how can generality and effectiveness be measured? The experimental approach described in the rest of this chapter is to use a heuristic based simulator to mimic the actions of a human user. This has a number of advantages. It produces a controlled, and thereby repeatable, experiment. It allows the experimenter to monitor the effect of different parameters on system performance. This is very similar to the idea of artificial domains, or data sets, used in machine learning to verify and compare different learning algorithms. This approach is described by D. Kibler and P. Langley [36] in the statement "Studies with multiple domains ..... provide little aid in understanding the effects of domain characteristics on learning, since they do not let one independently vary different aspects of the environment. Artificial domains provide a way out of this dilemma by letting one control domain characteristics as independent variables." The main disadvantage of this approach is the uncertainty of whether or not it represents real data (in this case how well it represents normal human search patterns).

Referring back to the experimental methodology, the requirements and search goal are represented by a noisy view of the contents of a class selected by the experimenter. Thus the noise level determines how well the search goal is defined. The simulator then maps

this to a series of browsing actions. The output is evaluated by comparing the ranking of the target class in the simulator's lists (for brevity referred to as the simulator's rank) and in the active browser's lists (the active browser's rank). The simulator's rank is for the most recent list produced by the standard browser as a result of the actions. The active browser's rank is for the most current list in the suggestion box.

Before presenting the results of the experiments the simulator will be described in detail.

5.3.1 The Simulator Of Human Browsing Patterns

The simulator consists of two parts (Figure 45). A black box representing the requirements and search goal and the heuristic agent that maps this to browser actions.

5.3.1.1 The Black Box

The black box contains information about a selected class and will answer specific questions. It will answer yes if a class name partially matches with the name of the selected class. It will answer yes if a method name wholly or partially matches with a name of a method in the selected class. It will answer yes if the group of matched methods are more similar to those of the selected class than the last matched group. The answer to each question is affected by a noise factor. The noise factor is a uniform random variable whose mean and variance can be set within the rules. If the match value is above a threshold determined by the random variable a yes is returned otherwise a no.

As the mean and variance is set in a rule, the noise value can be assigned independently for the different questions. The answers to these questions have some intuitive relationship to the knowledge of a user. A positive response for the matching of a class name is biased towards names that have common subterms and in a weaker sense to the children of classes that match. The matching on method names is biased towards identical names and in a

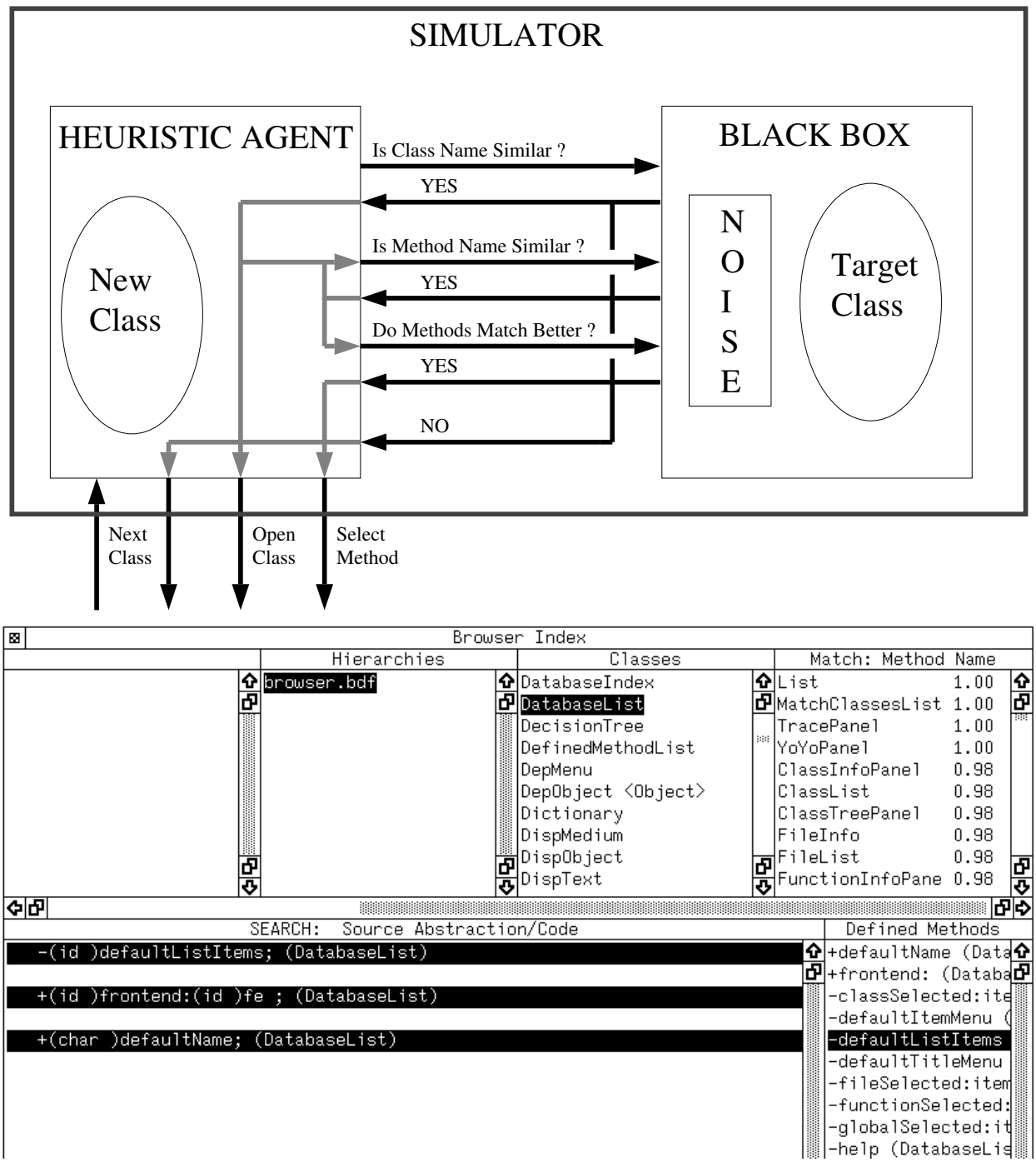


Figure 45 The Simulator Of Human Search Patterns

weaker sense to names with common subterms and to inherited methods. The answers to these questions could be construed as the semantic knowledge the user has about the meaning

of the names and in a weaker sense to knowledge about the library structure. The assessment that a new class is better than a previous one could be related to the user's general knowledge about the items in the library and what role they play.

5.3.1.2 The Heuristic Agent

The heuristic agent is rule based and uses the same inference engine as the active browser. The advantage of defining it around a collection of rules is that in future different rule bases can be used to simulate different human search patterns. At present there is a single search strategy using only a subset of the available standard browser operators. The top of figure 45 shows the current heuristic model, the bottom shows a typical series of windows that result from the actions generated.

The heuristic agent retrieves the name of a class from the most recent browser list. It then asks the black box if this name is similar to that of the target class. If the answer is yes the operator is applied that expands the class to show the implemented methods. In figure 45 the methods of "DatabaseList", an expanded class, are shown on the lower right hand side of the screen.

The heuristic agent then asks the black box if a randomly chosen method name matches with the name of a method in the target class. If the answer is yes this method is saved and the question repeated. This continues until a maximum number of methods have been saved (a random number from three to five) or there are no more methods in the class. The heuristic agent then asks if this group of methods is a better match, to those of the target, than the ones selected in the previous class. If the answer is yes these methods are selected and highlighted in the window on the lower left (Figure 45).

The heuristic agent applies the operator "Implemented In Classes" which generates a list of classes — that best match the selected methods — in the upper right hand corner of the

```
1) Select next class name in list
    IF class name same as target STOP
    IF class name not in top ten of list
       THEN backtrack to last list
            repeat from 1
    IF class name does not match with target
       THEN repeat from 1
2) Expand class
   assign maximum number of methods from 3 to 5
3) Select method at random
    IF method does not match with target
       THEN repeat from 3
4) Save method
    IF number of saved methods less than maximum
       THEN repeat from 3
    IF matching score less than previous class
       THEN repeat from 1
5) Expand saved methods
6) Apply operator to retrieve matching classes
8) Select first class
9) Repeat from 2
```

Figure 46 Heuristic Search Algorithm

browser's interface (Figure 45). The heuristic agent continues searching in the next list of classes until one of the following conditions arises. If the target class is found the search is stopped. If a better class is opened, a new list is produced and the cycle is repeated. If the search reaches the tenth class in the list, the last one visible on the screen the simulator backtracks to the previous list and continues the search from the next class. The heuristic search is summarized in figure 46.

Three effects encourage the heuristic agent's search to converge. Firstly, the top classes in the new list based on matching selected methods should, on average, be closer to the target than those of the previous list. Thus, on opening a new class, the probability of finding useful methods should be greater. Secondly, a noisy form of hill climbing is used by preferring a class with a score higher than the previous one expanded. Thirdly, the noise in the black box is progressively reduced each time a new class is opened. The heuristic agent gets progressively more accurate answers to its questions. This is analogous to the refinement of the user's knowledge about what is required, and thus the refinement of the search goal, as more classes are explored.

5.3.1.3 The Experiments

The first step in a single experimental run, is to select a target class which is placed in the black box. Next a random starting point is chosen in the alphabetically ordered list of all classes. The simulator then begins search from this point, as described in the preceding section. An experimental run is completed for each class in the library in turn. The complete cycle of classes is, itself, repeated with different degrees of noise selected for the black box's reply to method matches (the other noise values were kept constant). This is summarized in figure 47. The library used contains 189 classes. It is the same library used for the examples described in section 5.2 except the classes developed for the standard browser and the independent search mechanism were also added.

During search the suggestion box is being updated whenever the template changes. When a new class list is generated by the simulator applying the operator "Implemented In Classes" the rank of the target class in this list and the suggestion box are displayed on the screen and written to a file. The search is terminated under two conditions. Firstly when the class is visible in the suggestion box, in the top ten of the 189 classes, for five steps. A step occurs

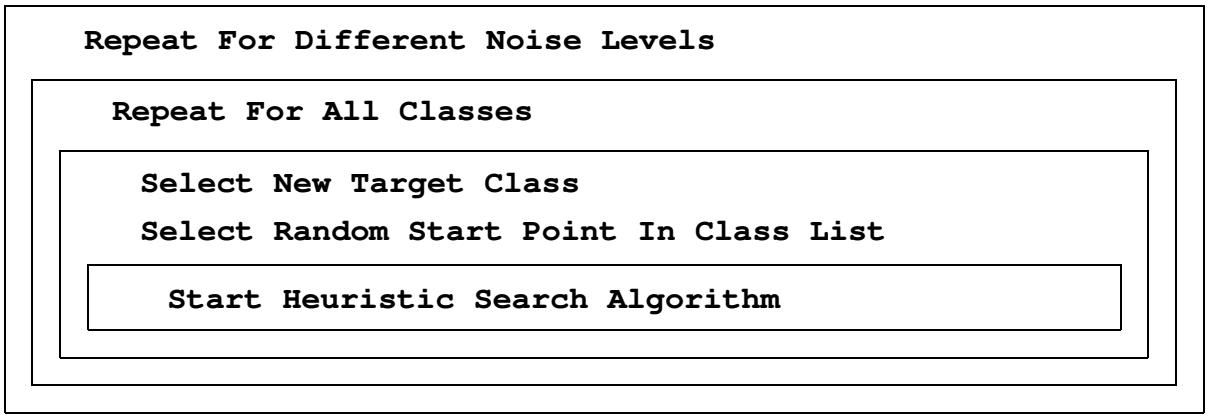


Figure 47 Experimental Procedure

Target Class ProbableMethodsList

STEP	Active Browser	Simulator	Difference
1	16	20	4
2	13	20	7
3	9	33	24
4	6	20	14
5	6	28	22
6	3	28	25
7	3	28	25
8	4	39	35

Figure 48 Record Of Ranking During Search

each time the simulator brings up a new list of classes or backtracks to an old one. Secondly the search terminates when the simulator would have opened the target class, determined by a name match.

An example of the recorded information is shown in Figure 48. It begins with the name of the target class in this case ”ProbableMethodsList”. The first column is the search step. The second column is the active browser’s rank for the target class. The third column is the simulator’s rank. The rank is a number representing how far the target class is from the top

of the list with zero being the highest rank possible. The fourth column is the difference between the two ranks. The steps are repeated until the search terminates. In the case shown, termination is due to the active browser's rank being higher than ten for more than five steps.

5.3.2 Experimental Results

The results of these experiments are analysed and presented in several different ways. The first represents a simple and intuitive measure of performance by defining criteria in which the active browser can be said to "win" against the simulator. The sensitivity of these results to the definitions underlying these criteria is then analysed. The second and third presentations of results are alternative methods of measuring system performance. The second assesses performance on a step-by-step basis measuring the percentage of classes, where the active browser's rank is higher than the simulator's rank for any given step. In figure 48 this is true for every step of the search. The third relates the difference in rank between the active browser and the simulator, averaged over a search, to the total length of the search. In figure 48 the sum of the differences divided by the length of search would give a value of nineteen for a search length of eight. In all the experiments the performance is assessed with respect to the target class alone. If either the active browser or the simulator were to find a close (very similar) class this would not be recorded in the results.

5.3.2.1 Assigning Wins

The first presentation of results is given in Tables 1 and 2. One of three possible outcomes is assigned to each individual search, namely a win, loss or draw. A win represents the case when the active browser has correctly inferred the target class before it was found by the simulator. There are two criteria for making this assessment which can be considered effectively lower and upper bounds on the number of wins. In tables 1 and 2, the columns are for five different noise levels for the method selection starting at zero. The numbers

shown in the column headings are for the variance of the noise factor in the black box, associated with questions about whether or not a method name matches. The rows represent the number wins, losses and draws for the active browser.

NOISE	VARIANCE 0.00	VARIANCE 2.50	VARIANCE 2.75	VARIANCE 3.00	VARIANCE 3.50
WINS	87	84	78	78	61
LOSSES	44	55	64	66	76
DRAWS	58	50	47	45	52

Table 1 Upper Bound On Wins

Table 1 shows the results for the upper bound. A win is awarded if the target class is in the suggestion box for five steps or if the active browser's rank is higher than the simulator's rank when the latter finds the target class. A loss is recorded if the simulator finds the target class and it does not appear in the suggestion box. All other conditions are draws.

NOISE	VARIANCE 0.00	VARIANCE 2.50	VARIANCE 2.75	VARIANCE 3.00	VARIANCE 3.50
WINS	34	44	46	40	29
LOSSES	68	79	92	90	99
DRAWS	87	66	51	59	61

Table 2 Lower Bound On Wins

Table 2 shows the results for the lower bound. A win is awarded only if the target class is in the suggestion box for five steps. The second condition has been removed making wins less likely. A loss is recorded if the simulator finds the target class and the simulator's rank is higher than the active browser's rank. Here the additional condition makes the requirement less restrictive and therefore losses more likely. All other conditions are draws.

The results in table 1 show that for zero noise the upper bound on the active browser wins is very close to half the classes. This number is twice the number of losses (the clear wins for the simulator). As the noise increases the number of wins declines. The lowest value for the upper bound on wins represents about a third of the total classes. There is however a large gap between the two bounds, particularly at zero noise. This indicates that a large part of the wins come from the case where the simulator finds the class but it is also in the suggestion box, though for less than five steps. Note that the conditions for a win in the lower bound preclude the active browser from winning any searches less than five steps in length. There are seventy-five searches length five or less. The lower bound for zero noise, the second lowest, represents about a third of the searches longer than length five.

Two factors are critical in the assessment of wins. Firstly the number of times the target class has to appear in the suggestion box before a win is recorded and secondly the size of the suggestion box itself. To examine the sensitivity of the results in tables 1 and 2 to the choice of "five consecutive appearances in a box of size ten", the wins are compared with the two factors (Figures 49 and 50). The same basic criteria are used to determine wins — the numbers underlined correspond to values in tables 1 and 2. The number of appearances forms the rows. The suggestion box length forms the columns. On the far right, in parenthesizes, is the total number of searches of a length greater or equal to the number of appearances for a win. The two figures 49 and 50 show three noise values ranging from zero through the medium to the highest.

Figure 49 gives the results for the zero noise factor. The first point of note is that the results are not highly sensitive to the suggestions box length. There is a sensitivity, not surprisingly, for very low numbers but this quickly levels out after a size of greater than eight. So the value of ten used in these experiments is a good one.

VARIANCE = 0 # ACTIVE BROWSER WINS (LOWER BOUND)

SB=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(N)
1)	54	72	89	104	115	122	126	128	134	*137*	140	142	142	145	148	(184)
2)	11	24	34	47	55	63	68	78	83	*85*	88	91	92	95	96	(166)
3)	6	10	18	23	30	36	43	51	56	*57*	60	62	64	67	68	(150)
4)	5	6	8	14	17	22	28	32	39	*42*	43	48	49	50	52	(131)
5)	*3*	*4*	*5*	*8*	*9*	*12*	*19*	*27*	*33*	*34*	*36*	*38*	*39*	*40*	*42*	(114)

ACTIVE BROWSER WINS (UPPER BOUND)

SB=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(N)
1)	85	94	100	112	118	123	126	128	134	*137*	140	142	142	145	148	(184)
2)	66	70	75	85	89	93	94	101	102	*103*	105	108	109	110	111	(166)
3)	65	68	70	73	78	82	88	92	93	*93*	94	94	95	97	97	(150)
4)	65	66	68	70	71	76	81	84	87	*89*	90	92	92	92	93	(131)
5)	*63*	*64*	*65*	*68*	*69*	*71*	*76*	*83*	*86*	*87*	*88*	*89*	*89*	*89*	*90*	(114)

Figure 49 Wins For Zero Noise

There is a higher degree of sensitivity to the number of appearances before a win is assigned. This is somewhat more pronounced for the lower bound as this is the only criterion for a win. Thus, as this number is reduced the two bounds become progressively closer.

5.3.2.2 Alternative Performance Measures

The results in this section show two alternative measurements of performance. The purpose of these results is threefold. To demonstrate that the active browser is performing well even in situations where a win is not assigned. To determine how the performance depends on the step number in the search. To determine how the performance depends on the total length of the search. This distinction will be emphasized in the following discussion.

MEDIUM NOISE VAR= .275 # ACTIVE BROWSER WINS (LOWER BOUND)

SB=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(N)
1)	37	59	74	82	99	107	115	117	121	*122*	125	129	130	132	133	(183)
2)	10	23	31	39	47	56	68	73	80	*81*	84	86	93	94	95	(169)
3)	5	13	18	25	30	38	48	57	63	*65*	68	70	75	77	79	(156)
4)	4	9	12	15	19	26	35	43	51	*53*	55	57	60	64	66	(140)
5)	*1*	*6*	*7*	*12*	*14*	*18*	*24*	*36*	*43*	*46*	*49*	*50*	*53*	*56*	*58*	(127)

 # ACTIVE BROWSER WINS (UPPER BOUND)

SB=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(N)
1)	61	73	80	87	100	107	115	117	121	*122*	125	129	130	132	133	(183)
2)	51	56	59	63	69	77	85	89	94	*95*	97	99	104	105	106	(169)
3)	49	52	53	58	61	66	74	80	85	*86*	87	89	91	93	95	(156)
4)	48	52	53	55	57	62	69	74	81	*83*	83	84	85	88	90	(140)
5)	*45*	*50*	*50*	*53*	*54*	*57*	*61*	*70*	*76*	*78*	*79*	*80*	*82*	*84*	*85*	(127)

HIGH NOISE VAR= .350 # ACTIVE BROWSER WINS (LOWER BOUND)

SB=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(N)
1)	30	48	60	73	80	85	86	95	101	*103*	107	113	116	117	117	(182)
2)	6	14	20	28	34	39	43	48	52	*56*	63	67	70	73	74	(165)
3)	4	10	17	20	22	27	32	38	42	*44*	50	51	55	57	59	(152)
4)	2	3	10	14	16	18	21	29	34	*36*	39	41	44	46	47	(135)
5)	*2*	*2*	*2*	*7*	*8*	*9*	*12*	*15*	*25*	*29*	*30*	*32*	*32*	*33*	*34*	(119)

 # ACTIVE BROWSER WINS (UPPER BOUND)

SB=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	(N)
1)	49	59	68	77	82	86	87	95	101	*103*	107	113	116	117	117	(182)
2)	38	43	48	55	59	62	64	67	69	*72*	77	80	82	85	85	(165)
3)	37	41	47	49	50	54	57	63	66	*67*	70	71	74	76	77	(152)
4)	36	37	42	45	46	48	51	58	63	*64*	66	66	69	70	70	(135)
5)	*36*	*36*	*36*	*41*	*41*	*42*	*45*	*47*	*57*	*61*	*62*	*63*	*63*	*64*	*65*	(119)

Figure 50 Wins For Medium And Highest Noise

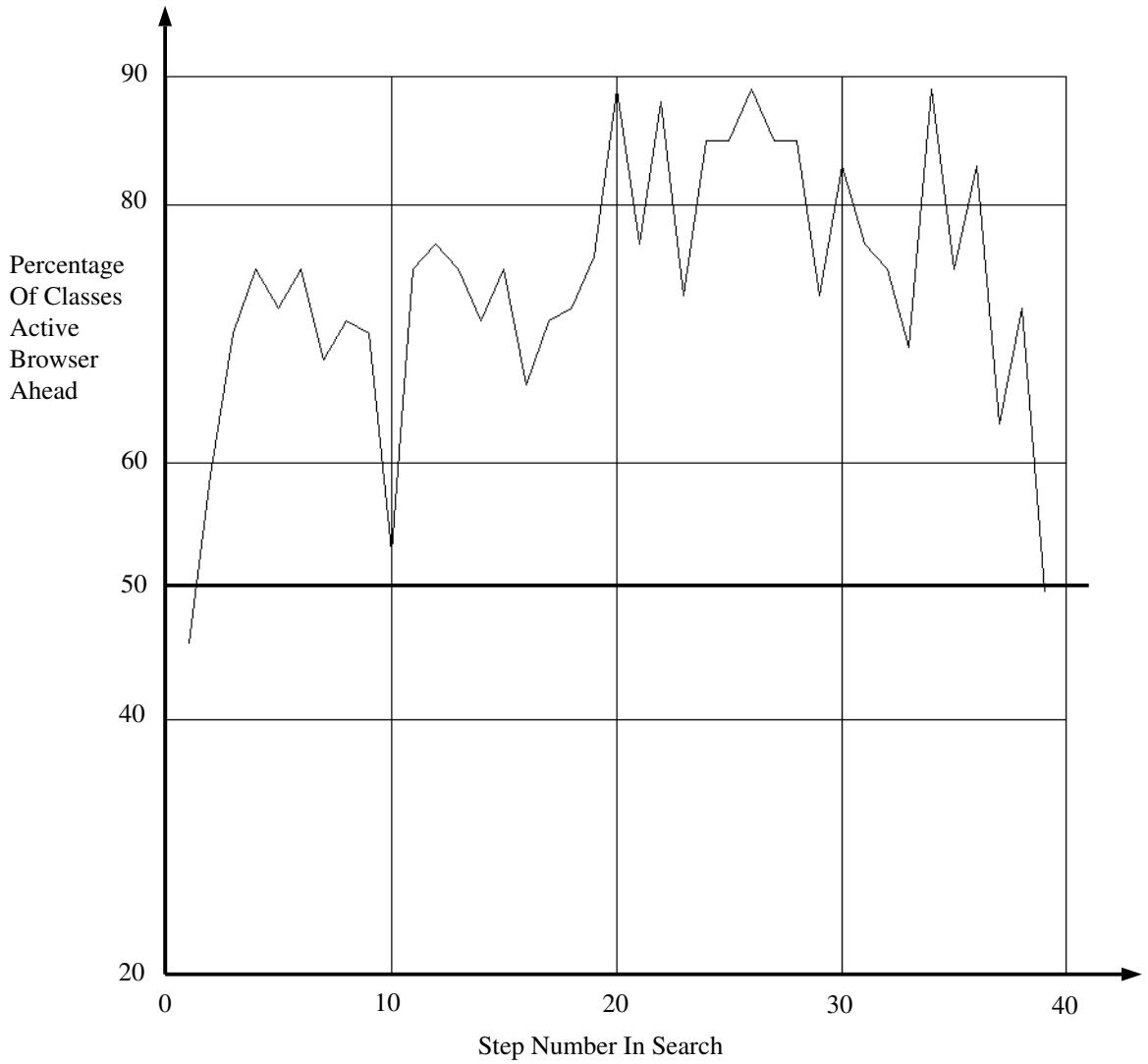


Figure 51 Zero Noise: Steps Into Search

5.3.2.2.1 Performance Against Steps Into Search

In figures 51, 52 and 53, the abscissa denotes the number of steps taken so far into the search. The ordinate denotes the percentage of classes where the active browser's rank is strictly higher than the simulator's rank. For example in figure 51 at step twenty in the search the active browser's rank is higher for close to ninety percent of the classes.

An important point to note on this graph is that a significant number of searches are short

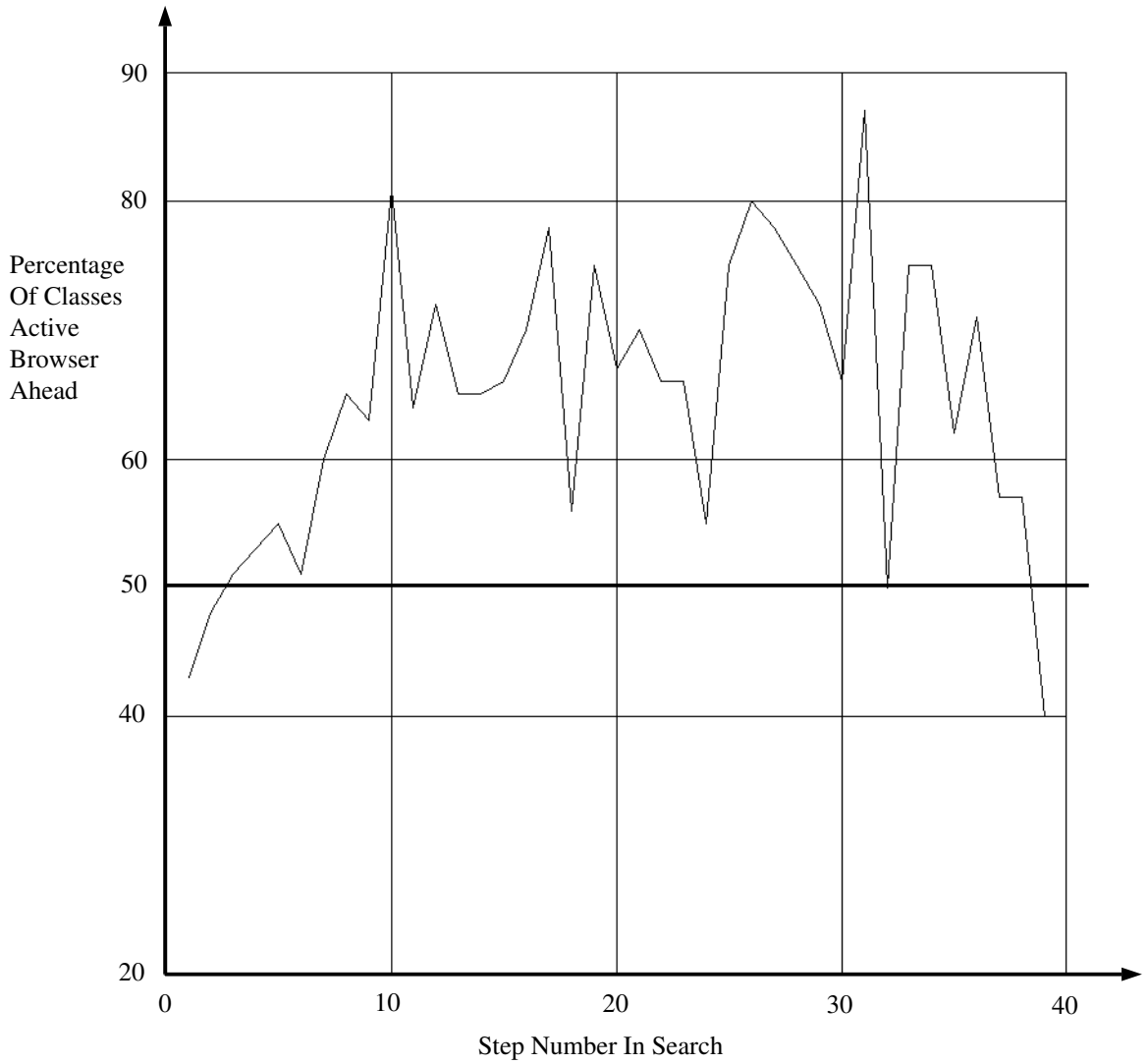


Figure 52 Medium Noise: Steps Into Search

(say less than six steps). Thus moving from the origin along the x axis, the total number of classes decreases rapidly. The maximum number of steps was chosen to be forty as at this point the number of classes (between six and ten) begins to make statistical analysis unreliable. In figures 51, 52 and 53, the comparison is presented for the three values of noise: zero, the medium, and the highest.

At the very first step in a search the active browser's rank is higher than the simulators's rank for only forty-five percent of the classes. But this percentage increases rapidly to achieve

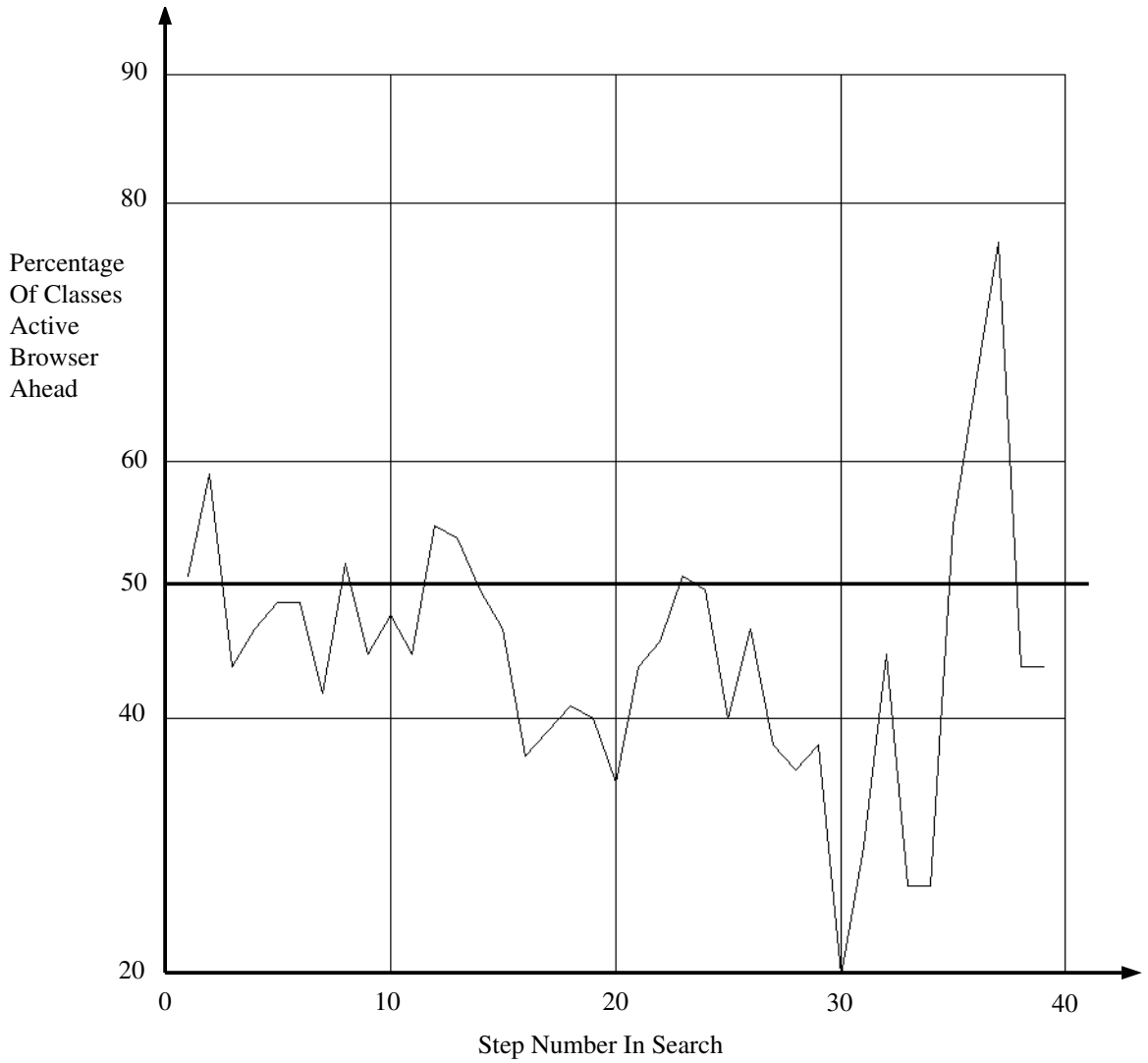


Figure 53 Highest Noise: Steps Into Search

a value of between 70 and 80 percent. This general trend is repeated in the medium noise case with an slightly lower mid point of approximately seventy percent. In the third case, the highest noise, the value in fact declines and the simulator can be seen to be ahead of the active browser for the majority of the time.

5.3.2.2.2 Performance Against Length Of search This performance measure differs from that presented in section 5.3.2.2.1 in both the abscissa and the ordinate. The ordinate, instead

of being based on the binary criterion of which system is ahead, is based on a numerical difference in ranking. The difference is calculated by subtracting the active browser's rank from the simulator's rank. Thus the value is positive if the former is higher than the latter. This value is then divided by the length to determine the mean. The abscissa is the total length of the search. For example in figure 54 a search of length eight has a mean difference in rank of plus ten. In other words, the active browser's rank is on average ten higher than the simulator's.

The search lengths typically range from one to thirty steps though some are in the sixties, with the majority less than fifteen. To make this analysis more statistically relevant, similar length searches are grouped and the results within a subgroup averaged. The subgroups were selected so that each contained a sufficient number of elements, between eight and fifteen, to make the calculation of a mean significant. Under a length of ten the subgroups are for single step differences, this increases to two steps and later to four. The value for length thirty-eight includes all longer searches. A mean of the difference over all steps of one search and over all searches is calculated to give a single value for each subgroup. The mean value of the number of steps for each subgroup is calculated. Then the mean of the difference is plotted against the mean number of steps.

Figure 54 shows the result for the medium noise value. For a search length of one, when the simulator uses only one selection of methods to find the target class, the difference is close to zero. At this stage the active browser has almost identical information to the simulator. As the number of steps increases the difference in rank becomes progressively more negative. This indicates that the simulator is generally ahead of the active browser in very short searches (up to length seven). At approximately a length of five steps the trend reverses until after eight or nine steps the active browser tends to dominate. This

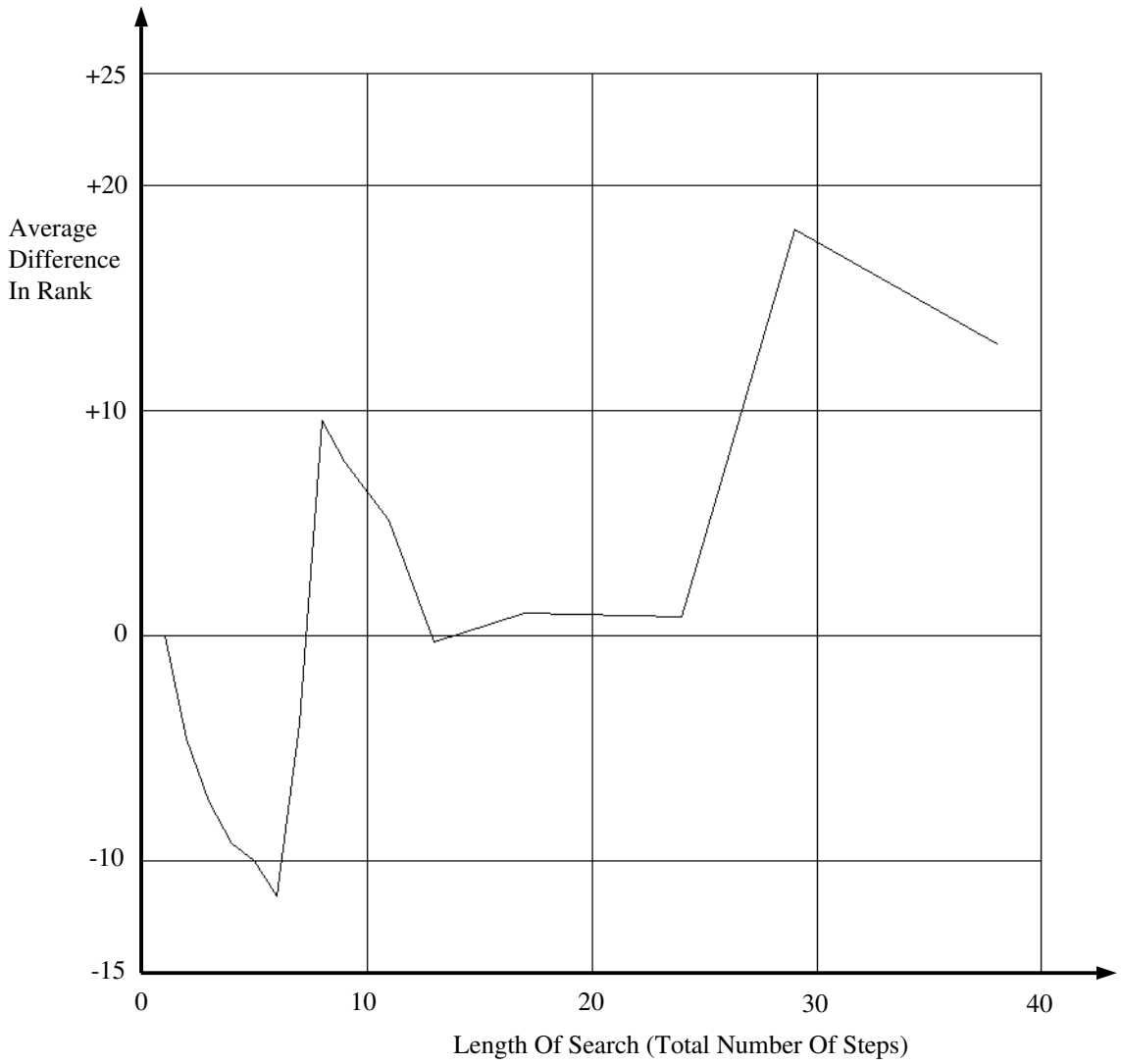


Figure 54 Difference In Rank: Medium Noise

continues until about fifteen when again the trend reverses and there is little difference in average rank. Looking at the raw data there are several classes, where the active browser does very badly, which seem to be associated with this part of the curve. These classes have relatively few methods with unique names. This suggests that the situation is sensitive to the library itself. Certainly as can be seen after just over twenty steps the active browser progressively improves.

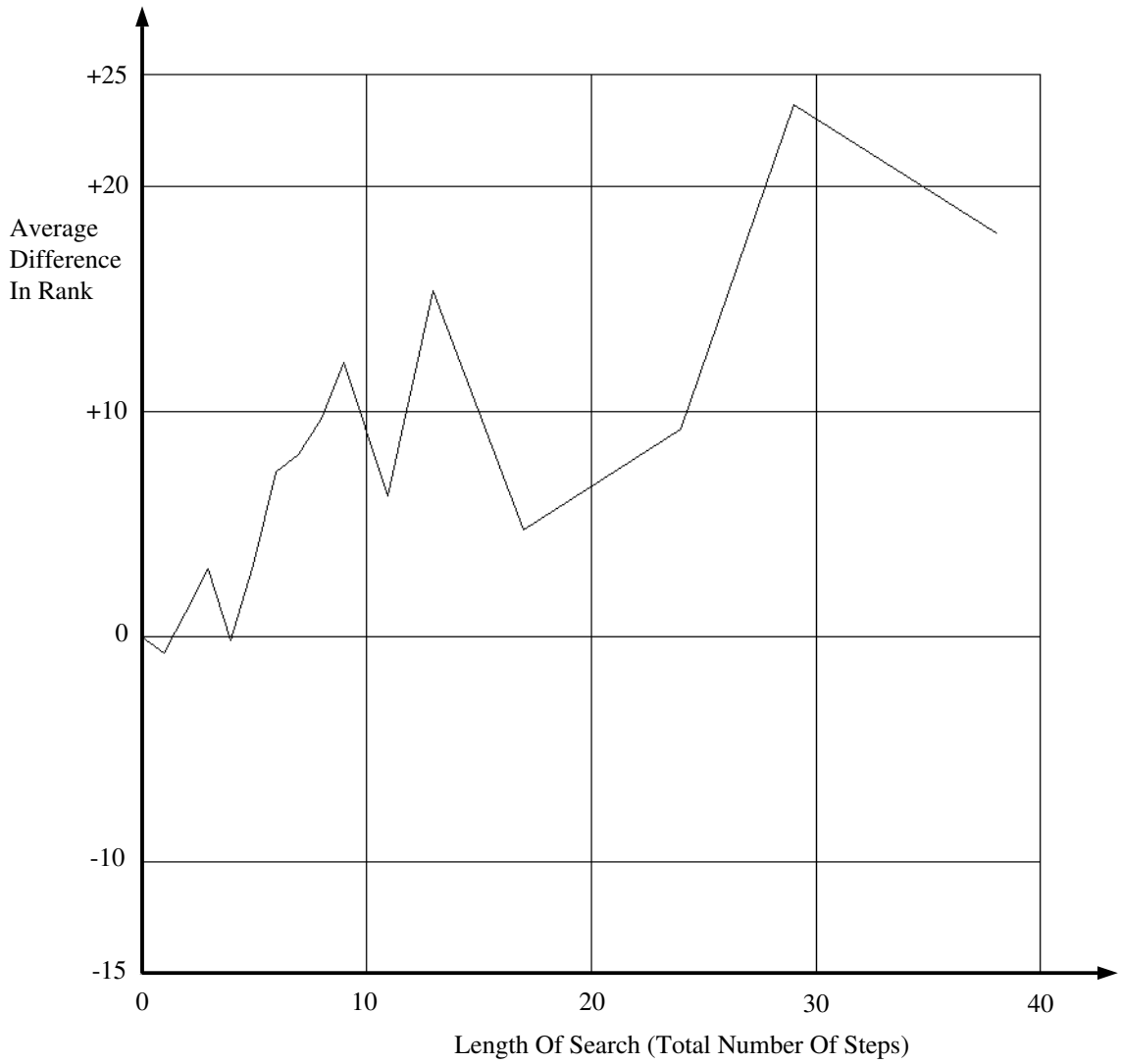


Figure 55 Difference In Rank: Zero Noise

In figure 55 the same information is shown for zero noise. The trends closely match those shown for the previous case except the curve is shifted upwards so that the active browser is on average ahead for all lengths.

In the third example with highest noise, shown in Figure 56, the pattern shown in the previous cases is not so apparent. The general impression though is that it still occurs but now shifted negatively. So on average the simulator is ahead of the active browser.

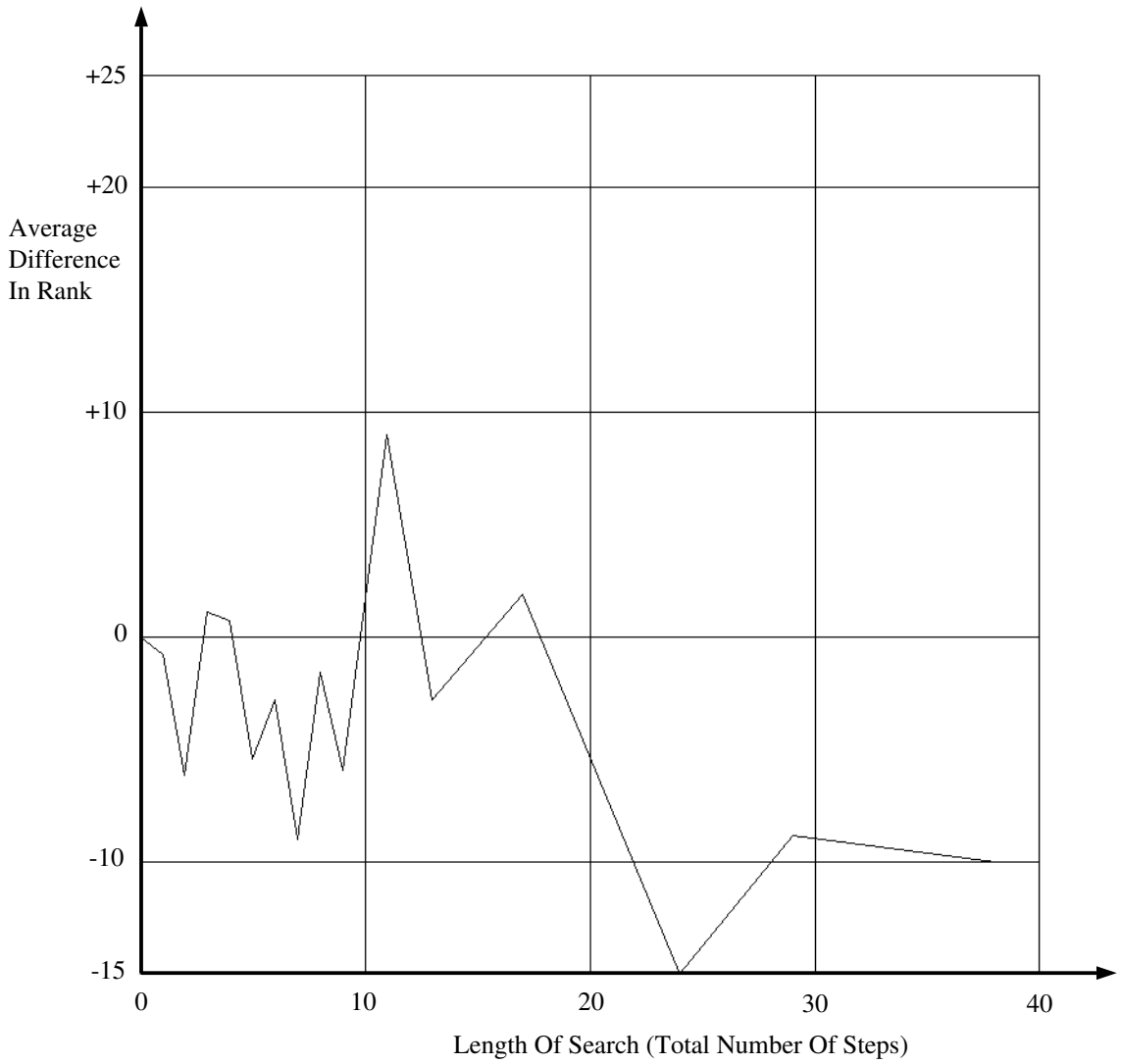


Figure 56 Difference In Rank: Highest Noise

5.3.3 Discussion And Analysis Of Results

The results shows that in general as the noise increases the performance of the active browser declines. A high noise means an increase in the number of irrelevant methods selected. Yet both systems are dependent on the relevancy of methods to find the target class. One advantage of the simulator is the high probability that occasionally it will select only relevant methods and thus bring the class high on its list. If it becomes visible on the screen

it will be selected either in this pass through the class list or later when backtracking. For the active browser the relevant methods may be outweighed by previously collected irrelevant ones. So the active browser‘s ability to extract information degrades with high noise.

The results show that the simulator performs better on short searches. Medium to long searches tended to be won by the active browser. This has an intuitive appeal to the potential use of the suggestion box. A user might not be expected to look at it early in the search. When the user has failed to find something useful after a number of steps then it may be used. The graphs suggest that this would be an effective strategy. That the active browser did not improve the early stages of browsing is not surprising because the analogue and template need sufficient terms for useful discrimination. If the simulator takes short series of rapid steps towards the target the active browser tends to lag behind.

One factor that may be relevant to both problems discussed in the previous paragraphs is the effect of selecting many methods from a group of strongly related classes. One example in the library is the many different types of “Layer” classes. As these are intended to be largely interchangeable they have many methods in common. The user may well select methods, in this group of classes, that only partially match the target class if it is not a “Layer” type. For the inference process to be successful, ideally the user would see variants of these methods. The common subterms would then establish the points of interest. When the user spends a lot of time in one of these groups of classes there is not sufficient variability. Thus the analogue becomes swamped with properties from this group. Thus even when other properties are selected it requires a number of steps to balance this effect. It was noticeable in some searches that the target class was above most of the other classes except for those of the layer type and thus not displayed in the suggestion box.

There are a number of answers to this problem. The first relates to the goal maintenance

issue discussed earlier in section 3.7.2. What is the status of properties the user showed interest in previously? One idea presented was to degrade these properties in relation to the abruptness of the user's change in direction. For instance suppose the group of layers is explored for a long time. If the user switches to another quite different group of classes say lists the properties might be degraded more strongly. This situation may be readily detected by a change in the ability of the analogue to reasonably predict the users selections. Another possible solution would affect the inference process itself. The only basic assumption is that the properties selected are likely to be similar to the desired ones. It is not known if they are exactly the same, although when first selected this might be preferred. Thus when properties are continually selected within these groups of highly similar classes, although the confidence that they contain useful features is increased, the confidence that the exact property is useful should stay the same. Ideally the analogue should be able to represent this distinction.

5.4 Conclusions

This chapter discussed an experimental methodology and proposed three stages in validating the active browsing approach. Plausibility was demonstrated by giving two realistic examples of system operation. In the last section generality of the approach was addressed. The results show that the search goal can be extracted from the user actions generally for many classes in a library. The alternative performance measures demonstrate that the active browser is performing well even in situations where a win is not assigned by the intuitive measure. This measure does, however, reflect a desired goal and the ideas presented in the discussion should serve to increase the number of "wins".

Effectiveness was not measured as the search was terminated when the target class was in the suggestion list for five steps. This was done to limit the time required to collect the data. Future tests can allow the target class to be reached and the difference in the number

of steps recorded. Although effectiveness was not proven the system's main advantage is that the information was extracted without cost in user input.

The simulator is a useful way of running controlled experiments on the active browser. Nevertheless the heuristic used is too simple to accurately reflect a human user's search patterns. In the future different and more complex heuristics will be developed and new test results generated. The space of user searches could be characterized and more heuristics produced to represent its different sectors. These could be used to decide how well the space is covered by this approach. If certain areas (types of human search) were well covered but others poorly, it would be useful to be able to infer which part of the space the current user occupies. In D. Canter et al. [9] measures are proposed which classify searches into broad groups such as "Scanning" "Exploring" etc. These classifications could form the basis of the inference and thus the selection of the most effective rules.

Chapter 6 Conclusions

This thesis has presented the notion of active browsing as a way of improving the speed and accuracy of normal browsing. This should encourage reuse by reducing the problem of locating software in extensive libraries. Section 6.1 presents the conclusions drawn from this research. Section 6.2 discusses the limitations of the general approach and the specific implementation. It also proposes ideas of how these limitations might be overcome and how generally such systems might be enhanced. Active browsing represents a particular form of machine learning. In section 6.3 it will be discussed how established techniques from this field of research might be applied to improve the performance of this approach. Finally section 6.4 will outline the principal directions for continuing this research in the future.

6.1 In Summary

Although this research has centered on software reuse, a general model of active browsing was developed in this thesis, applicable to many browseable libraries. This was motivated by both the general problem of locating items in large information bases and the wide applicability of browsing. This thesis has presented the view that browsing is a critical search methodology where the target is not well defined. Further it proposes, that this lack of definition is a very common problem in many situations. The searcher is often not looking for something particular but rather anything that best matches some vague, highly informal, set of requirements. This research has built on the foundation of browsing so that active browsing by improving an already useful approach has its general usefulness assured.

This thesis has demonstrated that the user's search goal can be inferred from normal browsing actions. It has shown that a representation of this goal allows independent search for items of relevance to the user's requirements. The user is able to make full use of the

facilities of the standard browser and is required to take no additional actions. By adding no additional cost in user input or in time to become familiar with a new tool the systems effectiveness is more easily enhanced.

The general claim of the model's applicability to different browsing tasks is not directly supported by the experimental validation as it was carried out for a particular object oriented library and browser. The results presented in this thesis do, however, give experimental evidence that the implementation and thus the model have some validity. The broad applicability is further supported by the examples in chapter 3, given for retrieval in both document and object oriented software libraries. They show a plausible mapping between features in the two domains and thus give some credence to the claim.

Within the general model of active browsing three methods of inferring a representation of the user's search goal have been proposed. These are incremental additions, generalizing examples and pattern specific inferences. It has also been shown how the inference process can be improved by replacing ambiguous browsing actions by unambiguous ones that are functionally equivalent . With careful design, these actions do not diminish, and may in fact enhance, the ease of search in a standard browser. The general model led to the development of a generalized architecture based around a standard inference engine. An implementation of the architecture was produced and demonstrated how a standard browser may be enhanced by active browsing.

Although human simulators and more generally artificial data sets have been used to evaluate systems in other fields, the experimental method described in this thesis is a novel means of evaluating browsing systems. Such a simulator could also be used to generate queries to validate indexing systems, in much the same way as it has been used in this research. A black box model of the user's requirements which gives noisy information about

the target item could be used with heuristics designed to translate this into the particular query language.

This model referred to browsing. Still any system with a user interface could borrow from the idea of minimally disrupting normal usage. The addition of learning to such systems should only subtly change the users view of the interface. This is achieved by first using the user normal actions as the sole input. Secondly by altering the response of the interface in minimal ways. In the implementation this was achieved by having a one extra window with information displayed in the same form as in existing windows. The user is not required to make use of it but when it is used the information displayed is readily understandable. One simple example of this idea would be spelling checkers. A system might learn the most common misspellings of words when the user corrects them. The system could add these to the top of the list in future case of misspelling matches.

6.2 Limitations

This section discusses limitations to the whole approach of active browsing. Some were assumed at the outset and have been described in the chapter 3. Others have become apparent during experimentation. Various ideas will be presented as to how these might be overcome.

An important assumption, made in the general model described in chapter 3, was that the user is on a single search. In section 3.7.2 ways of dealing with changes in the user's search direction were discussed. These methods would not however readily cover the situation where the user is looking for multiple unrelated objects at the same time. This might be seen as a pathological case, nevertheless allowing the user to search for multiple related objects is more reasonable. Here the user might be seen as having a single search goal but at some point, possibly right at the beginning of search, realizes that this is not achievable

in a single item. In this case the analogue may be considered to represent a single search goal. Multiple relevancy measures must, however, be produced and activated at different points of the user's search.

This situation highlights the distinction made, in chapter 3, between the analogue representing the user's search goal and a relevancy measure to evaluate items. In the actual implementation they both had similar forms, the difference mainly appearing in the confidence values. Nevertheless these two parts represent two distinct ideas: how to best represent the users interests and how to best evaluate items.

Some experiments were tried with the idea of multiple relevancy measures in the implementation. One operator allows the user to store the information about the class presently under review. When this operator was selected an additional template was produced by the active browser. This included all the terms that were not well represented by the selected class. This template could then be used to find items which in conjunction with the one selected could meet the user's requirements. These were displayed to the user as a second list. The storing of an item does not in itself imply that the user has decided to split the search. It may only be a record of the best class found so far and the user might continue with the original search. To determine this, the first template was compared to the subsequent class selected. If there was sufficient agreement it was assumed that the user was continuing search in the original direction. No experiments were carried out to validate this idea so no results have been presented in this thesis.

Another problem occurs if the user sidetracks to look at items that are of interest but not connected to the present task. The user may, while looking for papers on a particular subject, chance upon an author's name related to some previous interest. If some time is spent looking at this and even related papers the inference process will become dangerously

skewed. As part of goal maintenance, in the implementation terms that did not even partially match with the classes visited were degraded faster than those that did. This would help the situation. Even so if the amount of data collected is too large, this problem will still arise and it is hard to see how this might be overcome.

A second assumption was that the active browser knows in advance the type of the target item. The form of the analogue is reflected in this assumption. This is defined by the primary granularity of the library, for example a document base may be principally built around conference papers. If the library is heterogeneous the user could be looking for one of many types of item. One approach, to remove this restriction, is to construct a number of analogues based on the different item types. These would represent hypotheses that the system has about the search target. The active browser would then need to be establish which one was relevant. The browser might have levels, or views determined by appropriate filters, associated with one specific type. Analogues could then be associated with the particular level or view. Results derived from them would only be shown when that level or view is active.

An alternative approach would be to have a less structured analogue that only represents the interest points that the user has shown during search. The matching items could be determined in a graphical manner, returning items that have most connectivity to these nodes. This would be very similar to spreading activation that has been used in software retrieval [33] and more generally in information retrieval [13]. The information in the analogue could be used to stimulate a graph appropriately to locate relevant items.

Two limitations were identified in section 5.3.3, the problems with high noise and short searches. A hypothesis was presented that this concerned groups of closely related items. New rules making use of various forms of statistics about library structure may help to alleviate this problem and thus improve the performance of the system.

The area of how to use information extracted to enable effective search by the active browser has not really been addressed in this thesis. In the implementation an exhaustive evaluation of all classes in the library was used. Certainly when dealing with very large libraries such as document bases this would be impractical and a more elaborate search strategy needs to be used. By using as much computer time as is available between user actions the system should explore as many close items as possible.

6.3 Useful Machine Learning Techniques

The task described in this thesis concerns learning over the short term, within the scope of a single user search. Existing methods of machine learning could be applied to this approach principally in two areas. Firstly by enhancing the inference process in producing a more accurate analogue. Secondly simplifying the relevancy measure and thus improving the speed of evaluating items. In addition information obtained from many searches could be extracted and used to improve the speed and accuracy of a particular search.

In the present form of active browsing, the learning is achieved by acquiring new terms (analogue predicates) and adjusting their confidence values. These are used to produce a weighted vector of terms which can be compared to the library items. The whole approach is similar to nearest neighbour matching as the template contains representations of all features in which the user has shown interest. This is often termed the "lazy" approach as generalization occurs at the time of comparison. If a sub-term is common to two terms it achieves a higher score by appearing in both terms.

There are many other learning strategies that may be useful. The "eager" approach would be to generalize the features when they are acquired. In this case common sub-terms would be extracted at the outset. To find appropriate generalizations ideally taxonomies would

be used. These could be entered as domain knowledge as in, for instance, the case of data types. For names, which in this paradigm often consist of concatenated words, certain simple grammatical rules could be used to produce a taxonomy. This would change the form of the analogue and implicitly the relevance measure. An alternative would be to maintain all the information in the analogue but generalize for the relevancy measure and thus reduce the matching cost.

Information obtained from many users over many searches could be used to form extra relationships between items determined by their correlation of use. It could also be used to adjust the confidence factors associated with the rules mapping browser actions to the analogue and template terms. This should produce the effect of improving the system search over time. A further use of data obtained over a long period would be in generalizing from examples. In this case the examples would be searches and could be used to generate new search heuristics or rules that could be used by an active browser.

From information obtained from single user search or certain types of user "User Models" could be created [37,48]. These might describe what the user's experience and knowledge is of items in the database. This would be of particular advantage in inferring exactly why a user moved between two library items. Further it might describe the sort of searches the user normally carries out and thus different sets of rules might be used for the inference.

One approach discussed in the preceding section as an alternative form of matching is the use of spreading activation. In this case the analogue describes the point of interest shown by the user. Such an approach might use a logic interface to determine the degree of stimulation given to a particular point in the program graph. The value assigned to a particular component would depend on the result of propagating values through weighted links. Lists might be generated in the same way based on these values.

6.4 Future Work

In the short term, the main work is the refinement and generation of new inference rules. Improvements in performance would be tested with the existing search heuristics for the user simulator. To prevent the possibility of the active browser becoming highly tuned to the simulator's particular search heuristics, new heuristics could be used to validate the performance. In general, more varied and different user search behaviours programmed into the simulator are required to better test this implementation of active browsing.

In the longer term, work will address how to extend the general model of active browsing to cover a wider area of applications. The model presented in this thesis should be applicable to the range of tasks that can be described by the browsing of a graph. The problem addressed can be abstracted readily to a higher level where it is seen as determining the goal of a heuristic agent moving through a space defined by a labelled directed graph. The model can be extended to cover such a definition. The problem can then be investigated at this more abstract level free of specific application details. Later the general validity of this approach to a broad range of applications can be determined.

7 Bibliography

[1] J.A. Alty and M.J. Coombs. *Expert Systems: Concepts And Examples* National Computing Centre Publications (1984)

[2] M. J. Bates. *Terminological Assistance For The Online Subject Searcher.* Proceedings Of The Second Conference On Computer Interfaces For Information Retrieval. (1986)

[3] D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell and J. Garza. *Construction Of File Management Systems From Software Components.* Technical Report TR-88-36, Dept. Of Computer Sciences University Of Texas (1988)

[4] James Bigelow and Victor Riley. *Manipulating Source Code In DynamicDesign.* Proceedings of ACM Hypertext'87 Conference pp 397–408 (1987)

[5] Ted J. Biggerstaff, Charles Richter. *Reusability Framework, Assessment And Directions.* Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis, ACM Press pp 1–17(1987)

[6] Brad Blumenthal. *Empirical Comparison Of Some Design Replay Algorithms.* Technical Report AI90–149 Artificial Intelligence Laboratory University Of Texas At Austin (1990)

[7] Guy A. Boy. *Indexing Hypertext Documents In Context.* Proceedings of 3rd ACM Conference on Hypertext (1991)

[8] F.R. Campagnoni and Kate Ehrlich. *Information Retrieval Using A Hypertext-based Help System.* SIGIR 89 Proceedings of the 12th International Conference on Research and Development in Information Retrieval (1989)

[9] David Canter, Rod Rivers and Graham Storrs. *Characterizing User Navigation Through Complex Data Structures.* Behaviour and Information Technology Vol. 4 No. 2 pp 93–102 (1985)

[10] Jamie G. Carbonnell. *Derivational analogy: A theory of reconstructive problem solving.* Machine Learning: An Artificial Intelligence Approach vol II Morgan Kaufman. (1986)

[11] Thomas E. Cheatham Jr. *Reusability Through Program Transformations.* Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis Pub. ACM Press pp 321–335 (1984)

[12] Thomas E. Cheatham Jr., Glenn E. Holloway, Judy A. Townley. *Program Refinement By Transformation.* Proceedings of 5th IEEE International Conference Software Engineering 1981 pp 430–437 (1981)

[13] Paul R. Cohen and Rick Kjeldsen. *Informational Retrieval By Constrained Spreading Activation In Semantic Networks.* Information Processing And Management Vol 23 No. 4 1987 pp 255–268. (1987)

[14] Panos Constantopoulus, Matthias Jarke, John Mylopolous and Yannis Vassilou. *The Software Information Base: A Server for Reuse* (Unpublished Report) (1991)

[15] Michael L. Creech, Dennis F. Freeze and Martin L. Griss. *Using Hypertext In Selecting Reusable Software Components.* Proceedings of 3rd ACM Conference on Hypertext (1991)

[16] W. Bruce Croft. *Approaches To Intelligent Information Retrieval.* Information Processing And Management Vol 23 No 4 pp 249–254. (1987)

[17] B. Curtis. *Cognitive Issues In Reusability.* Proceedings Of the Workshop on Reusability In Programming pp 192–197 (1983)

[18] L. Peter Deutsch. *Design Reuse And Frameworks In The Smalltalk-80 System.* Software Reusability Vol 2 Ed T.J. Biggerstaff A.J. Perlis, ACM Press pp 57–71 (1989)

[19] Premkumar Devanbu, Ronald J. Brachman, Peter B. Selfridge, Bruce W. Ballard. *LaSSIE: A Knowledge-based Software Information System* Communications Of The ACM May 1991 Vol. 34 No. 5 pp 35–49 (1991)

[20] Ed Dubinsky, Stefan Freudenberger, Edith Schonberg and J.D. Schwarttz. *Reusability of Design for Large Software Systems: An experiment with the SETL Optimizer.* Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis, ACM Press pp 275–293 (1989)

[21] Gerhard Fischer, Scott Henninger and David Redmiles. *Cognitive Tools For Locating And Comprehending Software Objects For Reuse.* Proceedings of CHI-91 Human Factors In Computing Systems pp 318–328 (1991)

[22] Gerhard Fischer and Helga Nieper-Lemke. *HELGON: Extending The Retrieval By Reformulation Paradigm.* Proceedings of CHI-89 Human Factors In Computing Systems pp 357–362 (1989)

[23] W. B. Frakes, B. A. Nejmeh. *Software Reuse Through Information Retrieval.* Proceedings of the 12th Annual Hawaii International Conference on system Sciences pp 530–535. (1987)

[24] Steven D. Fraser, Jose M. Duran and Raymond Aubin. *Software Indexing For Reuse.* Proceedings of 1989 IEEE International Conference On Systems, Man and Cybernetics pp 853–858 (1989)

[25] P. Freeman. *Reusable Software Engineering: Concepts And Research Directions.* Proceedings Of the Workshop on Reusability In Programming (1983)

[26] G.W. Furnas, T.K. Landauer, L.M. Gomez and S.T. Dumais. *The Vocabulary Problem In Human-System Communication.* Communications of The ACM. Nov 1987 Vol 30

No 11 pp 964–971 (1987)

[27] Pankaj K. Garg. *Abstraction Methods In Hypertext.* Proceedings of ACM Hypertext'87 Conference pp 375–395 (1987)

[28] John F. Gilmore, Hong Wing Pun and Frederick C. Hart. *Software Reusability In A Knowledge-based Environment.* SPIE Vol 1095 Applications of Artificial Intelligence VII pp 448–457 (1989)

[29] Robert Godin, Jan Gecsei and Claude Pichet. *Design Of A Browsing Interface For Information Retrieval.* SIGIR 89 Proceedings of the 12th International Conference on Research and Development in Information Retrieval (1989)

[30] Mehdi T. Harandi and Sanjay Bhansali. *Program Derivation Using Analogy* Proceedings Of The International Joint Conference On Artificial Intelligence pp 389–394 (1989)

[31] Donna Harman. *Relevance Feedback Revisited.* SIGIR 92 Proceedings of the 15th International Conference on Research and Development in Information Retrieval (1982)

[32] Richard Helm and Yoelle S. Maarek. *Integrating Information Retrieval And Domain Specific Approaches For Browsing And Retrieval In Object-Oriented Class Libraries.* Proceedings of OOPSLA-91: Object-Oriented Programming Systems, Languages, and Applications pp 47–60 (1991)

[33] Scott Henninger. *CodeFinder: A Tool For Locating Software Objects For Reuse.* Automating Software Design: Interactive Design Workshop Notes 9th National Conference on Artificial Intelligence AAAI–91 pp 40–47 (1991)

[34] E. Horowitz, J.B. Munson. *An Expansive View Of Reusable Software.* Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis ACM Press pp 19–41 (1984)

[35] W. Lewis Johnson. *Interactive Acquisition Of Requirements For Large Systems.* Au-

tomating Software Design: Interactive Design Workshop Notes 9th National Conference on Artificial Intelligence AAAI–91 pp 61–70 (1991)

[36] Dennis Kibler and Pat Langley. *Machine Learning As An Experimental Science.* Proceedings Of the Third Working Session On Learning (1989)

[37] Andre J. Kok. *A Formal Approach To User Models In Data Retrieval* Int. J. Man-Machine Studies 35 675–693 (1991)

[38] Yoelle S. Maarek, Daniel M. Berry and Gail E. Kaiser. *An Information Retrieval Approach For Automatically Constructing Software Libraries.* IEEE Transactions On Software Engineering Vol. 17 No. 8 Aug. 1991 pp 800–813 (1991)

[39] T. Patrick Martin, Hing-Kai Hung and Chris Walmsley. *Supporting Browsing Of Large Knowledge Bases.* (Unpublished report) (1991)

[40] Sigi Meggendorfer and Peter Manhart. *A Knowledge And Deduction Based Software Retrieval Tool.* Proceedings Of The 6th National Knowledge Base Software Engineering Conference pp 126–137 (1991)

[41] G. A. Miller. *The Magical Number Seven Plus Or Minus Two: Some Limits On Our Capacity To Process Information.* Psychological Review 1956 Vol 63 pp 81–97 (1956)

[42] Amihai Motro. *BAROQUE: A Browser For Relational Databases.* ACM Transactions on Office Information Systems Vol 4 No. 2 April 1986 pp 164–181. (1986)

[43] James M. Neighbours. *The DRACO Approach To Constructing Software From Reusable Components.* Proceedings Of the Workshop on Reusability In Programming 1983 pp 167–177 (1979)

[44] N.S. Prywes, A. Pnueli and S. Shastry. *Use Of A Nonprocedural Specification Language And Associated Program Generator In Software Development.* ACM Transactions on Programming Language Systems. 1979 pp196–217 (1979)

[45] Ruben Prieto-Diaz. *Implementing Faceted Classification For Software Reuse.* Communications of the ACM Vol 34 1991 pp 89–97 (1991)

[46] John Rice and Herb Schwetman. *Interface Issues In Software Parts Technology.* Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis Pub. ACM Press (1989)

[47] Richard H.C. Seabrook and Ben Shneiderman. *The User Interface In A Hypertext Multiwindow Browser.* Interacting with Computers Vol 1 No. 3 1989 pp 301–337 (1989)

[48] R. H. Thompson and W. B. Croft. *Support for Browsing In An Intelligent Text Retrieval System.* Int. J. Man-Machine Studies Vol. 30 pp 639–668 (1989)

[49] Randall H. Trigg. *Guided Tours and Tabletops: Tools For Communicating In A Hypertext Environment.* ACM Transactions on Office Information Systems Vol 6 No. 4 October 1988 pp 398–414 (1988)

[50] Dennis M. Volpano and Richard B. KieBurtz. *The Templates Approach To Software Reuse.* Software Reusability Vol 1 Ed T.J. Biggerstaff A.J. Perlis Pub. ACM Press pp 247–254 (1989)

[51] Douglas A. White. *The Knowledge-Based Software Assistant: A Program Summary.* Tutorial Overview. Proceedings Of The 6th National Knowledge Based Software Engineering Conference (1991)

[52] Rebecca J. Wirfs-Brock and Ralph E. Johnson. *Surveying Current Research In Object Oriented Design.* Communications Of The ACM Vol 33 No 9 Sept 1990 pp 105–124 (1990)

[53] Murray Wood and Ian Sommerville. *An Information Retrieval System For Software Components.* I.E.E. Software Engineering Journal (1987)

Appendix A Meta-Rules

```
Pre IF ((EndTask match ?x))
    THEN ((Task scores)) 1.0
Pre IF ((TempMatch ?op ?type ?name)
        (newCopyOf Class ?x))
    THEN ((Task rules: S)) 1.0
Pre IF ((TempMatch ?op ?type ?name ?pnode)
        (newCopyOf Class ?x))
    THEN ((Task rules: S)) 1.0
Pre IF ((TempMatch ?op ?type ?name)
        (NOT newCopyOf Class ?x))
    THEN ((Task rules: T)) 1.0
Pre IF ((TempMatch ?op ?type ?name ?pnode)
        (NOT newCopyOf Class ?x))
    THEN ((Task rules: T)) 1.0
Pre IF ((MatchScore ?x ?sc1 ?sc2)
        (newCopyOf Class ?x)
        (BOOL GT ?sc2 ?sc1)
        (BOOL GT 0.75 ?sc1))
    THEN ((Task newTemplate)) 1.0
Pre IF ((EndTask check ?file)
        (newCopyOf Class ?x))
    THEN ((Task secondTemplate)) 1.0
Pre IF ((EndTask check ?file))
    THEN ((Task readTemplateFrom:thr: ?file 0.45)) 1.0
Pre IF ((EndTask compClass ?file))
    THEN ((Task readClassGen:thr: ?file 0.50)) 1.0
Pre IF ((EndTask compMeth ?file))
    THEN ((Task readTemplateFrom:thr: ?file 0.45)) 1.0
Pre IF ((EndTask match ?file))
    THEN ((NOT matchOn)) 1.0
Pre IF ((EndTask classGen ?file))
    THEN ((Task compareList)) 1.0
```

```
Pre IF ((EndTask compList ?file))
    THEN ((Task readCompList:thr: ?file 0.25)) 1.0
Pre IF ((Class ?name ?op ?pnode ?pop)
        (NewClass ?x))
    THEN ((NOT NewClass ?x)) 1.0
Pre IF ((Class ?name SelRovCla ?pnode ?pop))
    THEN ((Task roverClass:decay: ?name 0.95)
          (Task wait: 2)) 1.0
Pre IF ((Class ?name ?op ?pnode ?pop)
        (BOOL NE ?op SelRovCla))
    THEN ((Task rules: C)
          (NewClass ?name)
          (NewClass)) 1.0
Pre IF ((Class ?name ?op ?pnode ?pop)
        (BOOL NE ?op SelRovCla)
        (interestedIn Class ?pnode))
    THEN ((ClassMove ?pnode ?name)) 1.0
Pre IF ((Class ?name WorCop ?pnode ?pop)
        (lastCopyIs Class ?lname))
    THEN ((NOT lastCopyIs Class ?lname)) 1.0
Pre IF ((Class ?name WorCop ?pnode ?pop))
    THEN ((newCopyOf Class ?name)
          (lastCopyIs Class ?name)
          (Task checkTemplate)) 1.0
Pre IF ((EndTask readTemp ?file)
        (newCopyOf Class ?name))
    THEN ((NOT newCopyOf Class ?name)
          (Task splitTemplate)) 1.0
Pre IF ((EndTask readTemp ?file)
        (NOT matchOn))
    THEN ((matchOn)) 1.0
Pre IF ((Method ?name ?op ?pnode ?pop)
        (NewClass))
    THEN ((NOT NewClass)
          (Task normalise)) 1.0
```

```
Pre IF (OR(Method ?name ?op ?pnode ?pop)
           (UserMethod ?name ?op ?pnode ?pop))
    THEN ((Task rules: M)) 1.0
Pre IF ((InstVar ?name ?op ?pnode ?pop))
    THEN ((Task rules: I)) 1.0
Pre IF ((Function ?name ?op ?pnode ?pop))
    THEN ((Task rules: F)) 1.0
Pre IF ((GlobalVar ?name ?op ?pnode ?pop))
    THEN ((Task rules: G)) 1.0
Pre IF ((newTemplItem)
        (NOT matchOn))
    THEN ((matchOn)
          (NOT newTemplItem)) 1.0
Post IF (OR (TempMatch ?op ?type ?name ?pnode)
            (TempMatch ?op ?type ?name))
     THEN ((NOT newTemplItem)) 1.0
```

Appendix B Search Rules

```
L IF ((List DatabaseList ?op ?pnode ?pop))
  THEN ((Task roverAction:on:from:
              classSelected:item: DatabaseList)) 1.0
L IF ((List ClassList ?op ?pnode ?pop))
  THEN ((Task roverAction:on:from:
              instanceMethods:item: ClassList)) 1.0
L IF ((List InstanceMethodList ?op ?pnode ?pop))
  THEN ((Task roverAction:on:from:
              select:item: InstanceMethodList)) 1.0
S IF ((TempMatch ?op ?type ?name)
       (BOOL NE ?op className:))
  THEN ((Templ ?op ?name 0.5)) 1.0
S IF ((TempMatch ?op ?type ?name ?pnode))
  THEN ((Templ ?pnode ?op ?name 0.5)) 1.0
S IF ((TempMatch className: ?type ?name))
  THEN ((Templ className: ?name 0.5)
         (Templ excludeClass: ?name 1.0)) 1.0
T IF ((TempMatch ?op ?type ?name)
       (BOOL NE ?op classMethods:)
       (BOOL NE ?op className:))
  THEN ((interestedIn ?type ?name)
         (Templ ?op ?name 0.5)) 0.02
T IF ((TempMatch className: ?type ?name))
  THEN ((interestedIn ?type ?name)
         (Templ className: ?name 0.5)) 0.02
T IF ((TempMatch classMethods: ?type ?name))
  THEN ((interestedIn ?type ?name)
         (Templ impMeths: ?name 0.5)) 0.02
C IF ((Class ?name WorCop ?pnode ?pop))
  THEN ((hasCopyOf Class ?name)) 1.0
C IF ((Class ?x ?op ?name Sub)
       (childrenOf Class ?name)
```

```
       (NOT interestedIn Class ?x))
  THEN ((childrenOf Class ?name)
         (Templ classInh: ?name 2.0)) 1.0
C IF ((Class ?name ?op ?pnode ?pop))
  THEN ((interestedIn Class ?name)
         (Templ excludeClass: ?name 1.0)) 0.1
C IF ((Class ?name ?op ?pnode ?pop))
  THEN ((Templ className: ?name 0.5)) 0.02
C IF ((Class ?name ?op ?pnode Sub))
  THEN ((childrenOf Class ?pnode)) 0.2
C IF ((Class ?name Sub ?pnode Sup)
       (interestedIn Class ?pnode))
  THEN ((siblingsOf Class ?pnode ?name)) 1.0
C IF ((Class ?y ?op  ?name Sub)
       (siblingsOf Class ?pnode ?name)
       (BOOL NE ?pnode ?y))
  THEN ((Templ classInh: ?name 2.0)) 1.0
C IF ((Class ?name SimClaInh  ?pnode ?pop))
  THEN ((tried classInh: ?name 0.5)) 1.0
C IF ((Class ?name SimClaDef ?pnode ?pop))
  THEN ((tried classInh: ?name 0.5)) 1.0
C IF ((Class ?name SimNamCla ?pnode ?pop))
  THEN ((tried className: ?name 0.5)) 1.0
C IF ((Class ?name ?op ?pnode ?pop)
       (tried ?sel ?name ?scale))
  THEN ((Templ ?sel ?name ?scale)) 0.2
C IF ((Class ?name ?op ?pnode ?pop)
       (tried ?sel ?pnode ?scale))
  THEN ((Templ ?sel ?pnode  ?scale)) 0.05
C IF ((Class ?x ?op ?pnode ?pop)
       (tried ?sel ?name ?pnode ?scale))
  THEN ((Templ ?sel ?name 0.5)) 0.05
M IF ((Method ?name ?op ?pnode ?pop)
       (NOT interestedIn Method ?pnode))
  THEN ((interestedIn Class ?pnode)) 0.05
```

```
M IF ((Method ?name ?op ?pnode ?pop)
      (NOT interestedIn Method ?pnode))
  THEN ((Templ className: ?pnode 0.5)) 0.01
M IF ((Method ?name ?op ?pnode ?pop)
      (NOT interestedIn Method ?pnode))
  THEN ((Templ excludeClass: ?pnode 1.0)) 0.1
M IF ((Method ?name ?op ?pnode ?pop))
  THEN ((interestedIn Method ?name)) 0.1
M IF ((UserMethod ?name SamMetNam ?pnode ?pop))
  THEN ((tried impMeths: ?name ?pnode)
        (Templ impMeths: ?name 0.5)) 0.2
M IF ((UserMethod ?name ClaUseMet ?pnode ?pop))
  THEN ((tried usesMeth: ?name ?pnode)
        (Templ ?pnode usesMeth: ?name 0.5)) 0.3
M IF ((UserMethod ?name MetUseCla ?pnode ?pop))
  THEN ((tried methUses: ?name ?pnode)
        (Templ methUses: ?name 0.5)) 0.3
M IF ((Method ?name ?op ?pnode ?pop))
  THEN ((Templ impMeths: ?name 0.5)) 0.1
M IF ((Method ?name ?op ?pnode ?pop)
      (interestedIn Method ?pnode))
  THEN ((Templ impMeths: ?pnode 0.5)) 0.2
I IF ((InstVar ?name InstName ?pnode ?pop))
  THEN ((Templ impInstVar: ?name 0.5)) 0.5
F IF ((Function ?name ClaUseFun ?pnode ?pop))
  THEN ((Templ ?pnode usesFunc: ?name 0.5)) 0.5
G IF ((GlobalVar ?name ClaUseGlo ?pnode ?pop))
  THEN ((Templ ?pnode usesGlob: ?name 0.5)) 0.5
```

Appendix C Program Understanding Rules

```
M IF ((Method ?pnode ?op ?pnode ?pop)
      (interestedIn Method ?pnode))
  THEN ((Templ impMeths: ?name 0.5)) 0.03
M IF ((Method ?name ?op ?pnode ?pop)
      (interestedIn Method ?pnode)
      (BOOL NE ?pnode ?name))
  THEN ((Templ ?pnode usesMeth: ?name 0.5)) 0.05
I IF ((InstVar ?name ?op ?pnode ?pop)
      (interestedIn Method ?pnode))
  THEN ((Templ impInstVar: ?name 0.5)) 0.05
F IF ((Function ?name ?op ?pnode ?pop)
      (interestedIn Method ?pnode))
  THEN ((Templ ?pnode usesFunc: ?name 0.5)) 0.O5
G IF ((GlobalVar ?name ?op ?pnode ?pop)
      (interestedIn Method ?pnode))
  THEN ((Templ ?pnode usesGlob: ?name 0.5)) 0.05
```