A Learning Agent that Assists the Browsing of Software Libraries

Chris Drummond, Dan Ionescu SM IEEE and Robert Holte
School of Information Technology and Engineering,
University of Ottawa, Ontario,
Canada, K1N 6N5

Index Terms: Software Reuse, Searching Software Libraries, Learning Agents, Simulated Human Users, Active Browsing

Abstract

Locating software items is difficult, even for knowledgeable software designers, when searching in large, complex and continuously growing libraries. This paper describes a technique, we term active browsing. An active browser suggests to the designer items it estimates to be close to the target of the search. The novel aspect of active browsing is that it is entirely unobtrusive: it infers its similarity measure from the designer's normal browsing actions, without any special input. Experiments are presented in which the active browsing system succeeds 40% of the time in identifying the target before the designer has found it. An additional experiment indicates that this approach does, indeed, speed-up search.

1 Introduction

The key claim of this paper is that adding an "active" component to an existing browser speeds up a user's search for relevant software items. The novel aspect of this active component is that it is entirely unobtrusive, its only input is the user's normal browsing actions. To verify this claim, we compare the performance of the browser with and without the active component. We are not claiming that our approach is the only way to speed up a user's search. We expect that additional, more powerful, search operators could and should be added to the browsing system. Yet we believe our approach is complementary to such additions and thus any speed up should be largely independent of the operators available.

This research is motivated by the fact that searching a software library for a particular item (source code or design documentation) is a central activity in all stages of the software development life-cycle. In software reuse, for example, the locating of relevant software artifacts has long been recognized as an essential activity [5, 21, 20, 18]. At present, it is a slow knowledge-intensive process that is by no means guaranteed to succeed, and almost all aspects of software development are hampered by the cost and low success rate of this search. As libraries grow so does the problem. Research aimed at improving the speed and reliability of searching software (or other kinds of) libraries has considered a wide variety of techniques. A survey of techniques specific to software reuse may be found in [34] and [19]. Analogous techniques for information retrieval systems are summarized in [43].

The speed and reliability of search are improved by increasing the effectiveness of the search operators, either by introducing new, more powerful operators or by organizing the library so that the existing operators are more effective. Our approach to improving search is complementary to this. Because of the ubiquity of browsers in software design, we take as a starting point an existing browser that is prototypical of browsers used to explore software libraries. The current implementation is an enhancement of a commercial browser for libraries of object-oriented software written in Objective-C. An active component is added to the browser, in the form of a learning agent. In addition to passively supporting user-directed search, the system actively assists or guides the user. The learning agent is completely unobtrusive in the sense that it attempts to learn what item the user is searching for simply by watching the user's normal browsing actions. The user's search proceeds as usual and may, in some circumstances, continue to completion without any interruption by the agent. The usefulness of the agent is that, often, it will recognize the user's "intent" and provide guidance or advice that accelerates the search.

This learning agent's task can be divided into two stages: (1) inferring the user's search goal and (2) using this to offer advice to the user. It is the goal inference task that is the primary focus of this paper, in particular the experiments of section 5. Our experimental results are encouraging: roughly 40% of the time the system succeeds in identifying the user's search goal before the user reaches the goal. Further experimentation suggests that when the user utilizes the advice of the system the average number of browsing actions needed to reach the search goal is reduced.

The rest of this paper is organized as follows. Section 2 discusses related research. Section 3 gives the architecture of the active browsing system and discusses the implementation used to search libraries of object oriented code. Section 4 presents the experimental methodology used to test the feasibility of active browsing. Section 5 presents and discusses the experimental results. In section 6 the limitations of this approach are discussed and section 7 presents some general conclusions.

2 Related Research

This section discusses research closely related to ours. The connection to information retrieval comes firstly from the fact that the locating of software items is an information retrieval task. Of perhaps equal importance is that the learning agent produces essentially a query whose matching process is similar to those used in information retrieval. As our system uses a learning agent, other learning agent research is also of direct relevance.

2.1 Information Retrieval

This discussion of information retrieval uses as its primary source the review paper of Belkin and Croft [4]. They point out that large scale use of information retrieval systems is mostly confined to simple string matching often extended through the use of Boolean connectives. An important alternative approach is the use of partial matching schemes. The most common of these is the "Vector Space Model". A feature vector is constructed with positions corresponding to words, phrases or concepts. A query and a document are vectors in the feature space and are matched using a distance measure, typically the cosine formula defining the angle between the two vectors. This method overcomes problems associated with boolean queries of returning too few or too many items.

In any of these approaches, for a query to match with a document they must be in the "same language". Yet users, particularly novice ones, are unlikely to know the "language". In software libraries this problem is exacerbated by the relatively small amount of text associated with a software item. Some success has been demonstrated using the automatic extraction of indices from text, such as "man" pages [35]. But when such text is limited or non-existent, it is necessary to hand craft the library item representation [40, 47] through careful domain analysis. But even with such analysis the problem may remain, because as [22] demonstrates there is a large amount of inconsistency that occurs when people assign names to items. Even when familiar with the language, it may be difficult to formulate a query accurately representing the user's requirements. Any match on the query is therefore unlikely to be exactly what is wanted.

One way to address these problems is to turn the search into an iterative process. Two popular versions of this approach are query reformulation and relevance feedback. In the former, [17, 15, 27, 38, 45] 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. In relevance feedback the user critiques the items returned by the query [24, 25]. The system can use this information to adjust the weighting of individual terms to find more relevant items.

To further increase support for iterative search the system can allow the user to explore closely related items. The items may be grouped through some sort of hierarchical clustering procedure and the items explored in a top down fashion. Alternatively links can be added based on features or relations between the items. The user may directly explore these links or the system may use them to find interesting items or new query terms. In Henninger's research [27] the matching process uses spreading activation through a graph of keywords and library items in which the arcs represent mutual association. When a given set of nodes are "activated", this activation is passed through the links to the neighboring nodes, but reduced according to the weight on the link. A node's activation is the sum of the activations it receives from all its neighbors. 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.

Frakes and Pole [18] carried out an extensive experimental comparison of different iterative search methods for locating reusable software items. The methods they investigated were hierarchical search, query reformulation using automatically extracted keywords and search using facets or attributes based on a domain analysis. They found that there was no statistical difference in the effectiveness of search using the different methods. They recommended that as there was no clear advantage to using methods requiring domain analysis, considerations of cost suggest that querying using automatically extracted keywords is the best approach. To produce object-oriented software libraries domain analysis is done not to support search but as part of the software design process. The results of this analysis can be exploited to locate software items by browsing the structure thus produced.

Browsing is a form of iterative search, where the query process is largely replaced by the user exploring links until the correct item is found. The cognitive attractiveness of browsing has been observed and analyzed by several authors. Studies [7] have shown that when given the choice, browsing is preferred by many users over "analytic strategies which require formulation of specific well structured queries". This is further supported by [17] which states "This (human remembering) theory postulates that people naturally think about categories of things not in terms of formal attributes but in terms of examples". Overall, browsing is seen as a more natural and effective process when the user is uncertain of the target description. In summarizing the advantages of browsing, [43] cites [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." But simply exploring links is not enough, [43] discusses browsers with an added query mechanism. It is such enhanced browsers that form the basis of our research.

2.2 Learning Agents

An active assistant is a background process that monitors the user's actions and, in certain circumstances, offers unsolicited advice. We categorize active assistants based on the nature of their internal inference mechanism.

"Daemons" are preprogrammed to recognize particular patterns of user behavior, and, when a particular pattern of behavior is detected, to issue the corresponding preprogrammed response. For example, the critics in Fischer's design environment [16] are daemons: each critic recognizes certain types of flaws in the user's current design (e.g. violations of design constraints) and draws these to the user's attention (possibly also suggesting corrections). A good review of critics is given in [42]. Finin's active help system [14] is similar, consisting of a collection of rules each of which defines a particular situation, specified by a (generalized) sequence of actions, and the advice to give should that situation arise. Likewise, plan recognition systems are daemons because they simply match the user's action sequence against a given library of plans [29, 9], or "parse" the user's actions with a given set of plan schemas [23]. Most "programming by demonstration" systems that do inference employ daemons. For example, [6, 10, 46] all use a preprogrammed notion of "similar action" to detect repeated sequences of actions.

Daemons are well understood and provide an extremely useful form of assistance for many tasks. It is evident, however, that daemons can offer only limited assistance in the browsing task, because, by definition, the users that need assistance are the ones whose actions sequences are not advancing them steadily and directly towards their goal (and certainly not following any pattern that could be preprogrammed). What is needed is a learning agent, i.e., a system that can infer a user's browsing goal without relying on a fixed library of action-patterns.

"Learning agents" do not have a preprogrammed set of situation–action rules, but instead they learn from the user's actions when to give advice the user and/or what advice to give. For example, [41] describes a "personalized information filtering" agent, which assists a user by suggesting USENET news articles that might be of interest. The user directly states his actual interest in the articles and this feedback drives a form of artificial evolution that improves the agent's performance. The news filtering agent in [30] serves a similar purpose but uses different techniques for learning. Unlike classical relevance feedback systems, which "adapt" to the user's immediate concerns, these systems learn a user model over an extended period. This idea of a user model is important in many fields. As Kobsa discusses [33] research in this area has traditionally focused on stereotype identification and plan recognition but has more recently addressed more dynamic methods of model acquisition. Learning agents are one way to acquire user models [39].

Learning agents have also been developed to assist a user in filling in a form [11, 28, 36]. As the user fills in the various fields of the form, the agent suggests how the remaining fields should be completed. If the agent's predictions are correct, the number of keystrokes needed to complete the form will have been reduced, thereby speeding up the process. Each completed form is added to the set of training examples on which the learning agent's subsequent predictions will be based.

2.3 Distinctive Features of Our System

The system presented in this paper is built on top of an existing browser. This is motivated by two factors. Firstly, the browsing of libraries to support software design, particularly for object-oriented languages, has become increasingly prevalent in recent times. Thus users are already familiar with the general approach. Secondly, when the user is somewhat uncertain about the search goal and the usefulness of individual actions, browsing is an appropriate form of search. Uncertainty is reduced through iterative search, with evaluation and refinement of the goal at each step.

But browsing is not the complete solution. Thompson and Croft [43] discuss one important improvement to browsing: it should "provide recommendations of what appears to be a promising direction to head given the user's current state". Our aim, although similar, differs in two important ways. Firstly our recommendations are not the next best step but what library items are likely to be interesting in terms of the whole search. Secondly we obtain the information on which to base these recommendations without interfering with the user's normal browsing operations. Thus there is little or no additional load placed on the user and any benefits need not be traded off against these additional costs.

This issue of intrusiveness is, we think, an important one. There is a continuum of possible "degrees" of intrusiveness (called the "relevance feedback continuum" in [31]). Most research on learning agents is concentrated in the "more intrusive" end of the spectrum, where the user's goals or preferences are conveyed explicitly by the user through direct interaction with the agent, either before problem-solving begins, or during the course of problem-solving. At the other end of the spectrum lie completely unintrusive learning agents, which infer the user's requirements, goals, or plans automatically from the user's interaction with the passive system. Systems that use relevance feedback might be placed somewhere in the middle of this continuum, requiring the user to rank documents. Our approach can be viewed as a special form of relevance feedback. Each time a new item is explored, terms in which the user shows interest have their weights increased, the others decreased. But the user is required to do no more than the usual browsing actions.

An important feature of the learning agents and relevance feedback systems is that they receive immediate feedback from the user directly indicating the correctness of their predictions. In the news reading and relevance feedback applications, the user indicates immediately whether the articles retrieved are or are not relevant. In the form-filling applications the correct entry for a field is immediately provided. This is crucial to the operation of these systems because this feedback provides new, highly informative training data that can be used to improve the agent's subsequent predictions. In the browsing task that our learning agent assists, the correctness of the agent's predictions cannot be determined until the search has ended. Only then does the user know the library item that satisfies his requirements. As the learning agent's purpose is to speedup the search, feedback at the end of the search is of no use.

The information that is available to the learning agent during search is “noisy”, because the user is searching somewhat blindly, pursuing dead-ends and taking circuitous routes. The information is also only very indirectly related to the correctness of the agent’s predictions. There are two reasons for this. The first is simply that the user might choose to completely disregard the agent’s suggestions. We expect the user to consult the learning agent only occasionally, when the need for assistance is felt. A more subtle reason is that, unlike the learning agents described above, a learning agent for browsing is not attempting to predict or suggest the next action (or sequence of actions). It is trying to predict the final stopping point of the user’s search, and this is only remotely related to the user’s judgement about which action is leading in the most promising direction.

Our system learns the user’s search goal and then turns this into a query to retrieve potentially interesting items. The terms of the query are matched with items in the library in much the same way as in the vector space model. The terms themselves are typically a concatenation of individual words forming essentially a phrase. The system infers the importance of the term to the user and this is reflected in the matching process. This inferencing is uncertain, so, as with the logic based systems discussed in Belkin and Croft’s paper the weighting of terms also reflects the “confidence” the system has in the inference.

Our system is also similar to the structure sensitive systems discussed by Belkin and Croft. The classes in the library have a structure, being composed of methods and instance variables, which stand in certain relationships to one another. The relationships investigated by the user are used in the matching process. But the system does not adhere rigidly to them, allowing partial matches when the relationship is different and when a term is not present but is closely associated with a term that is. The inference process may also add additional relationships to the query. Suppose the user is looking at the item “Scroll Bar”, potentially of interest are other similar items or ones that exploit scroll bars like scroll-able windows. The query is adjusted to include relationships appropriate to both sorts of item.

Overall our system aims to exploit the natural advantages of browsing. But by unintrusively learning a user’s search goal it can locate potentially interesting items independently. The use of a very flexible form of partial matching, enhances the ability of the system to find items relevant to the user’s search goal.

The work presented here is part of new rapidly growing research area, the application of Machine Learning techniques to user modeling [39, 3]. The browsing task is particularly challenging for two reasons: (1) the information available to the learning system is highly ambiguous and noisy; and (2) learning must take place in "real-time" (i.e. it must succeed before the user's search has finished). No previous learning research has addressed either of these issues. Another important distinction of our system, is the degree to which it has been evaluated. As Kobsa [33] points out "The question of whether user models and adaptive interactive computer systems offer sufficient enough advantages .... can only be empirically determined. Further empirical work to this end is thus urgently needed." This is a critical concern and the latter part of this paper details extensive experiments aimed at quantifying the advantages of our approach.

3 The Active Browsing System

In normal browsing, the user has a set of requirements and the purpose of the search is to find an item that best satisfies them. Although the requirements could be as formal as a specification in our view they are typically a loose set of properties the target class should have. The user mentally converts the requirements into a search goal describing the expected form of the target item in the library. During search, new information is acquired about the content, organization, and descriptive language of the library. This may alter the user's search goal, even though the original requirements have not changed. Transforming the requirements into a search goal represents the major source of uncertainty in browsing. Even when the requirements characterize one specific item, known to be in the library, the user often will be unable to describe it accurately.

The idea of active browsing is to enhance the effectiveness of the browsing system by having the system draw attention to items that may be of interest. The user's actions implicitly carry information about the search goal because they have been deliberately chosen, to the best of the user's ability, to serve the user's interests. The active browser infers from these actions an analogue of the user's search goal. 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 then converted into a form that can be readily used to measure the relevance of an individual item to the user. The measure is used to evaluate library items and the result of the evaluation is used to influence the user's search.

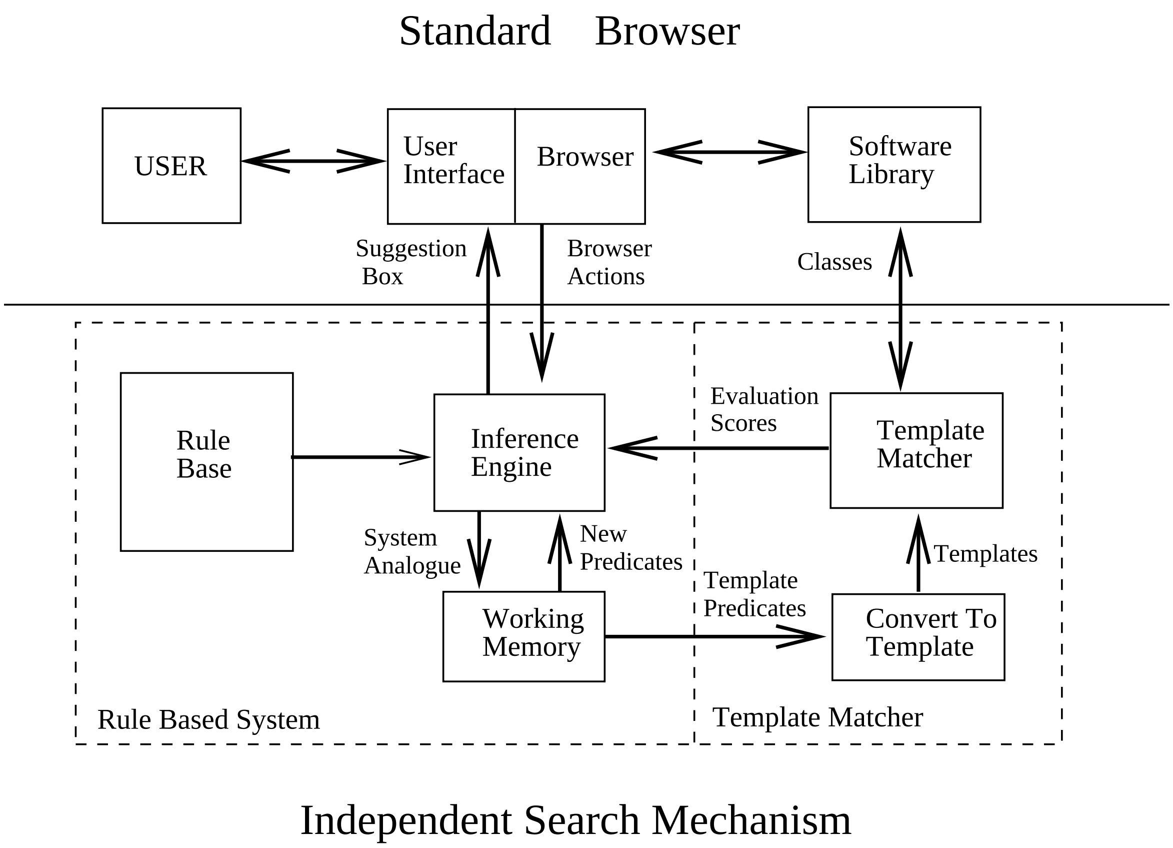

Figure 1: System Architecture.

There are a wide variety of ways to use the evaluation of library items produced by the active browsing system. The simplest method, the one used in the current implementation, is to have a single, small, extra window (the Suggestion Box) in which library items are displayed ordered according to the active browsing system's estimate of their relevance to the user. Information in the Suggestion Box is displayed and used in an identical manner to the information in ordinary browsing windows. This way of using the active browser's output is capable of exerting sufficient influence on the user to accelerate search and yet, at the same time, is not so distracting or disruptive that it frustrates or impedes users who do not desire assistance. The user is not required to look at the Suggestion Box at all, but may easily consult it when the need arises.

The system architecture is shown in figure 1. The top section is the standard browser. The user selects browsing operators which return information on the classes in the software library. The user might select a general class, such as "Collection", and request a list of its subclasses (which, for "Collection" might include "Set", "Array", and "Bag"). Subsequent user actions might move further down the hierarchy and investigate the methods that define the behavior of particular classes.

Each browsing action is passed to the lower section of the system architecture, the active component of the browser, called the independent search mechanism. This is divided into two parts: an inference system and a template matcher. The inference system takes the actions and converts them into predicate form so that the rules in the rule base can be applied to them. The rules are used to infer the system analogue, which comprises the properties of a class that are (believed by the system to be) of interest to the user. These properties range from the names of the particular classes and methods the user has visited to the common properties of the classes being explored. In addition to making inferences from the actions, the inference system also converts its conclusions into a template that can be readily matched against items in the library. For example, if the user has been exploring the descendants of the general "Collection" class, the inference system should produce a template whose terms represent the sort of attributes that are typical of a class implementing a dynamic collection of elements. The template matcher computes the degree of match between the template and individual classes in the library. The result of this process is a ranked list of classes which is shown to the user in the suggestion box. Although some of these classes will be ones the user's own actions have produced (e.g. the descendants of "Collection" the user has not yet explored) others will be classes with similar names or methods from more remote parts of the hierarchy. The following sections describe the inference system and template matcher in detail.

3.1 The Inference System

The inference system is a standard rule-based system with

- a forward chaining inference engine supporting conjunction, disjunction, negation and Mycin-like confidence factors [1].
- a working memory containing facts representing the user's browsing actions and the current analogue and template.
- a rule base containing rules. The rules represent the system's knowledge of how to infer the user's intent from his actions.

Each time the user makes a browsing action, a description of the action is added as a fact to working memory and the inference process initiated.

3.1.1 Working Memory

Facts in working memory represent the current browsing action and the analogue and template inferred from previous actions. Associated with each fact is a confidence factor between 0 and 1 reflecting the degree to which the system "believes" that the fact holds. Facts representing browsing actions always have perfect confidence (1.0). The confidence of an inferred fact depends on which rule was used to infer the fact (each rule has a hard coded confidence factor) and the confidence factors of the facts that caused the rule to "fire". The confidence of an inferred fact also depends on how frequently it was inferred. For example, if a fact having confidence "Old Confidence" is inferred again, with confidence "New Confidence", its confidence is adjusted by the following formula:

Confidence = Old Confidence + ((1 - Old Confidence) * New Confidence)

This has the intuitive effect that confidence is increased proportionally to both the New Confidence and the current "doubt" reflected in the existing fact (1.0 - Old Confidence).

Browsing actions are represented in such a way that the system can reason about the entire chain of actions that directly led to the current one. For example, the system could detect particular patterns in the user's search, even when these are interrupted by spurious exploratory actions. The analogue of the search goal is represented by a collection of facts each of which summarizes the user's interest in some particular feature, such as the name of a class or method the user has examined.

3.1.2 Rule Syntax and Meaning

The rules the general form:

IF (LogicalConnective Literal$_1$....Literal$_n$)
THEN (AND Literal$_1$....Literal$_m$)) Confidence

LogicalConnective is either "AND" or "OR". Each literal is a predicate and its arguments. The confidence factor is a hard coded number between 0 and 1 indicating the strength with which the consequent may be concluded from the antecedents. The system only uses positive confidences but literals can be prefixed by "NOT". For the antecedent this condition is met if the fact does not exist in working memory. In the consequent a "NOT" causes the fact to be removed regardless of its confidence factor.

A rule is "fired" by binding each of its antecedent literals to a fact in working memory. Firing a rule causes the unnegated literals in its consequent to be added as a facts to working memory, with a confidence determined by the confidences of the rule itself and the facts bound to the rule's antecedent literals. If the antecedent is a conjunction (disjunction) then the minimum (maximum) confidence value from the antecedent facts is selected. This is multiplied by the rule's confidence and assigned to the consequent.

3.2 Template Construction and Matching

The template is a list of terms to be matched against items in the library. A term consists of the element to be matched, a matching procedure and a weight, formed from the product of the confidence value and a scale factor. The scale factor is hard coded domain knowledge about the importance of this type of term relative to other types in the template. For example, an instance variable name may be considered of lower importance than a method name as only the latter is visible outside the object. The library designers are likely to have been more careful in the choice of method names, making them better discriminants. The template is much like a "canned query" used in commercial database systems. Such a query is designed by an expert to be effective and efficient but allows a user to change the individual terms. In the template the element and the confidence value are changeable but the matching procedure and scale factor are fixed. Of course, in our system the user does not directly change the terms. Rather the inference system generates the terms by watching the user's browsing actions.

The template is matched to a class term by term. The individual scores are summed and divided by the maximum possible weight. This results in a normalized total so that a perfect match will have a score of one. The template matching process returns scores for all classes in the library. The scores are used to produce the list in the Suggestion Box.

3.2.1 Template Matching

The element in each term is a name. The names are normally a concatenation of individual words, a standard practice in most object oriented libraries. So each name can be broken down into an ordered list of words, each of which can be individually used in matching. The matching score is computed by finding the common words of the template term name and the equivalent name in a library item. For example, methods are compared to methods and instance variables to instance variables. A perfect match occurs when all words match and are in the same position. Lower scores are assigned if words are in different positions or are missing entirely.

When matching names of unequal numbers of subterms, the difference in position depends on whether matching starts from the first or last word. This allows two simple grammatical forms of matching based on noun and verb phrases. With class and instance variable names, the match starts with the last word; a compound name is assumed to be a noun preceded by adjectives. With method names it starts with the first word; a compound name is assumed to be a verb followed by adverbs or other parts of speech.

Matching a template to a class is illustrated in Figure 2. On the left is an abstract representation of a template. It contains terms for two class names "SmallThing" and "BigThing", and terms for three method names "emptyStore", "printIt" and "saveItAll". It also contains terms relating methods "printIt" and "saveItAll" to the instance variable "dataStore". On the right is shown a class that would return a high matching score. The class name "VeryBigThing" matches with both template class names due to the common subterm "Thing" and it being the "noun of the noun phrase". The match on "adjective Big" further increases the score. The method "saveItAll" matches well, sharing the "verb" and "pronoun" with the method "saveIt" implemented in the new class . It also has the

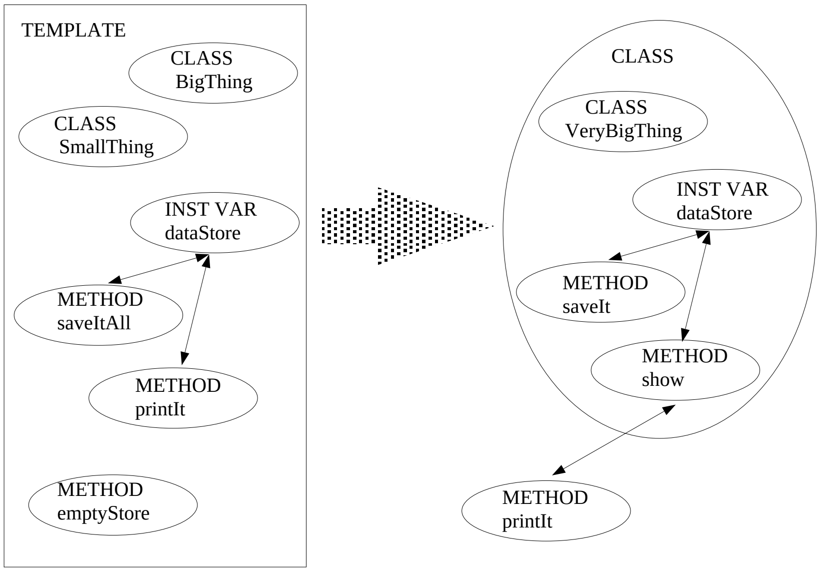

Figure 2: Matching Templates To Classes

same relationship to an instance variable with the same name "dataStore".

Sometimes there is no match, or a weak match, in the class itself but a related class has a feature that strongly matches. Here a simple form of graphical matching is used. The match score is inversely related to graphical distance. For example, typically one class will use the methods of another class. A distance of one is when the method, in the class being evaluated, uses a matching method in another class. A distance of two is when a method it uses, itself uses a matching method. In this example although "printIt" is not implemented in the class, the method "show" uses a method called "printIt". The matching score is half what it would have been had the class actually implemented this method.

3.3 Strategies for Inferring the User's Goal

The fundamental inference strategy is based on the assumption that the user will usually examine library items whose visible properties are akin to the properties of the current search goal. Each time the user applies an operator to an item the system records the properties of the item that are currently being displayed. For example, if the user decides to expand a method knowing only its name, then the system infers that the name is related, in some way, to the name of some method in the user's search goal. We do not expect that all inferences will be accurate. The user will examine things that have no connection with the target item and will fail to examine others that do. We expect, however, that in general enough good properties will be examined to make the locating of interesting items effective.

We expect as search proceeds, those properties closely related to the search goal will be continually re-asserted, whereas incidental properties of the items visited will not. By weighting properties according to the frequency of appearance a progressively more accurate estimate of the search goal will emerge. For this basic inference strategy to be successful the visible properties must indeed explain why the user examined an item. We achieve this by introducing useful browsing actions (see section 3.3.2). The system is made less sensitive to "misleading" actions by exploiting flexible matching and by decreasing the importance of old information (see section 3.3.3).

3.3.1 Rules Implementing the Inference Strategies

The design of rules and the choice of confidence factors was determined using domain knowledge and some initial experiments. The rules produced largely rely on very general properties - reasonably consistent naming of similar items and consistent structure of the library. Thus analogous rules to those given below should be effective in other libraries where these two properties hold. The confidence factors depend in a fairly general way on how users search the library, which is largely dictated by the browser's human interface. Our initial experiments allowed us to set these factors. It was apparent that, in general, the most effective strategy for updating the confidence values is to increase them slowly. But experimentation would be needed in other domains to establish the confidence factors. The domain knowledge for this implementation came from one of the author's experience with this and other similar browsers for object oriented code. The very similarity of the browsers and library structure leads one to expect that the rules and their confidence factors will be directly applicable to many object oriented libraries.

The rules were designed by considering the types of information the browser displays and the types of browsing operator. In this implementation the main types of information the browser displays are class and method names. The browsing operators are essentially of two types: exploratory and querying. The exploratory operator traverses the structure of the library. In object oriented languages this structure produces two hierarchies: horizontal and vertical. The former is the class hierarchy where one class is the subclass of another, similar to an "IS-A" relation. The latter is a "PART-OF" hierarchy where one class uses another as part of its function, by its methods referencing the other's methods. Each time the user views a "node" (class or method) in these hierarchies, the information associated with the node is extracted as interesting to the user: this is typically a class or method name. But, in addition, if the subtree of a node is extensively explored by the user the properties of this node can inferred as interesting. The querying operators allow the user to find classes with a particular property, or group of properties. Such a direct interest in properties obviously increases the systems confidence in its importance to the user. But the confidence is not immediately set to one as this interest may be transient and the system needs more evidence before being confident it is likely to be part of the search goal.

```
1)  IF User Opens Class            THEN InterestedIn Class Name 0.1

2)  IF User Opens Method in Class  THEN InterestedIn Class Name 0.05

3)  IF User Opens Method           THEN InterestedIn Method Name 0.1

4)  IF User Expands Method         THEN InterestedIn Method Name 0.1

5)  IF User Matches on Method      THEN InterestedIn Method Name 0.2
```

Figure 3: Some of the Rules Implementing the Goal-Inference Strategy

Figure 3 shows some of the rules dealing with methods and classes. On receipt of a browser action, the analogue and template are updated, by adding new terms or revising the confidences in existing ones. The first rule is fired when a user opens a class for the first time. All the user has seen is the class name, so the system infers this is of interest to the user. If the user continues exploring the class by opening its methods, the second rule fires increasing the confidence in the interest in this class and thereby its name. If a method is opened, interest is also inferred in the method name (rule 3). Confidence in this inference is increased (rule 4) if the method is expanded to show further detail. It is further increased (rule 5) if the user shows a more direct interest by initiating searching for methods with similar names.

3.3.2 Ensuring That Actions have Unambiguous Explanations

An action may be ambiguous, in the sense that there are many equally plausible explanations for why the action was taken. Ambiguous actions most frequently arise because the browsing system displays many properties of an item all at once. If the user further investigates the item, the system cannot immediately infer which of the item's properties are of interest and which are not. Actions can be made unambiguous by presenting information about an item in a series of progressively less abstract stages, at each stage the user must specify what additional information should be displayed. The principal example of this in the present system is the expansion of methods to show progressively more detail. The method name can be expanded to show its argument types, expanded further to show the variables, methods etc. it uses and further still to show its code.

It is important that any changes to the browser should not make normal browsing less effective. Otherwise there will be a tradeoff between the benefits of goal inference and the losses due to more complex browsing. Compared to the original all-at-once presentation of information, our sequential presentation of information about methods is undoubtedly an improvement for normal browsing as well as being more informative for active browsing. The original action loaded into a window the entire file containing all the information about all the methods and scrolled the window until the selected method was centered in it. Not only did this display a lot of irrelevant information, it was also impossible to view the information about several selected methods at once. With the sequential presentation, the aim is to first present the information that is most likely to be useful for understanding the method's function and its interaction with other items in the library. Further details are easily available by applying additional operators.

3.3.3 Goal Tracking

The initial goal of the user's search 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 template up to date. 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. The simplest way to exploit this fact is to decay with time the confidence in the terms of the analogue and template. Thus newer information is considered more reliable and important than older information. Ideally terms should never completely disappear: even very old data might be relevant in breaking a deadlock based on more current information. By decaying confidences by a fixed percentage of its current value this information is never lost.

3.4 An Example Of Active Browsing

This section demonstrates how information collected over the entire search can help the user in finding relevant classes. The browser's user interface is shown in figure 4. The bold capital letters have been added and will be used as references in the following discussion. Across the top are windows, normally displaying lists of classes (Windows A and B). The user can select a class in any of these lists and apply a browsing operator to it. Whenever a class (or method) is selected, it is highlighted in the display. As in all object-oriented browsers, there is an operator that lists the selected class's subclasses, and one that lists its ancestors in the object hierarchy (its parent, its parent's parent, etc.). A third operator lists classes whose names are "similar" to the name of the selected class. The class list produced is placed in the rightmost window at the top of the interface. Another operator opens a class producing a list of the names of the methods the class implements. Two such method lists for two different classes are shown in the lower right portion of the interface (Windows C and D).

Operators can also be applied to methods. Individual methods can be expanded to show progressively more information, shown in the lower left portion of the interface. In figure 4 two methods "extent" and "attachTo" are selected from the method list in window

C and shown in window E. Several operators can be applied to them. One lists classes that implement the methods (Window B). Another lists of classes that use the methods. The matching is a matter of degree and is identical to that used on template terms. The resulting class list is sorted in order of match strength and displayed in the rightmost window at the top of the interface.

Figure 4 shows the browsing interface after a series of operators have been applied. In the following example of active browsing, we discuss the application of operators that brought the browser to this state and how active browsing exploits them. Let us suppose that the user requires a means of managing and displaying text. The user first focuses on a class that manages text. On discovering that this does not have methods for displaying the text, the user then surveys a large group of classes used for general display purposes.

The user knows there is a class called "String" in the library and expects this class to implement methods for comparing, combining and displaying strings. This expectation defines the user's initial search goal. At the beginning of the search, the only class list available to the user is a list of the names of all classes in the library. The user scrolls until the class "String" is visible and then opens this class, showing the methods it implements (Window D). The user investigates two of these. The method "concatSTR:" is expanded to show its argument types. The method "compareSTR:" is expanded further to examine the instance variables and functions it uses (Window F).

The left window in figure 5 shows the users browsing actions, the right window the template terms inferred from these actions. These windows are used for debugging and demonstration purposes and would not be seen by the user in normal operation. Looking at the window on the right, the first line indicates that the class "String" has already been visited and therefore ought not to be suggested to the user. Each of the other three lines describes a property of a class that interests the user: the class's name should match or contain the word "string" and the class should implement methods "compareSTR" and "concatSTR". Note that the weight associated with method "compareSTR" (0.21) is higher than the weight associated with method "concatSTR" (0.10). This is because the user showed more interest in the former by studying it in greater detail.

Suppose now that after scanning other method names the user realizes that the expectation that class "String" contains a method for displaying strings is incorrect. To find a

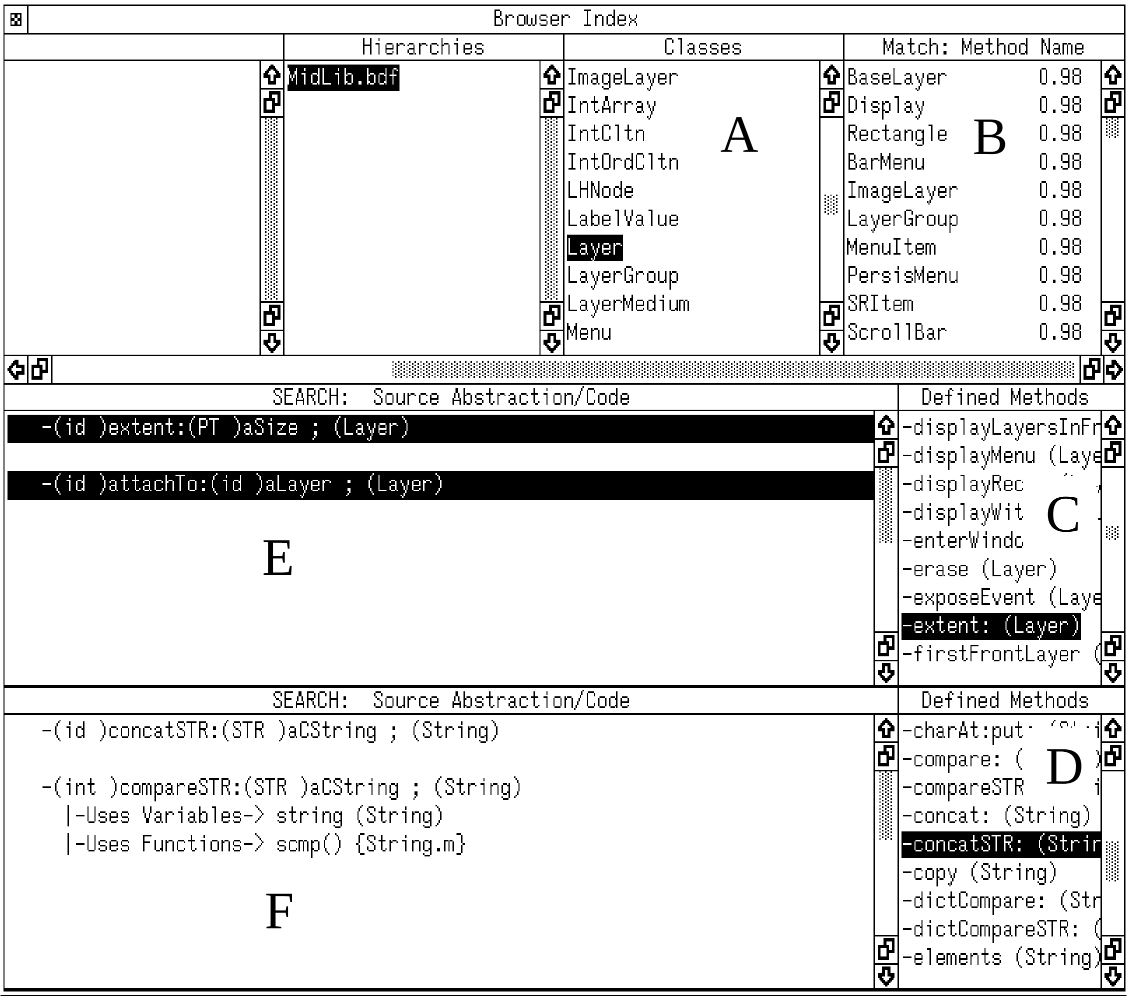

Figure 4: Exploring Two Different Classes

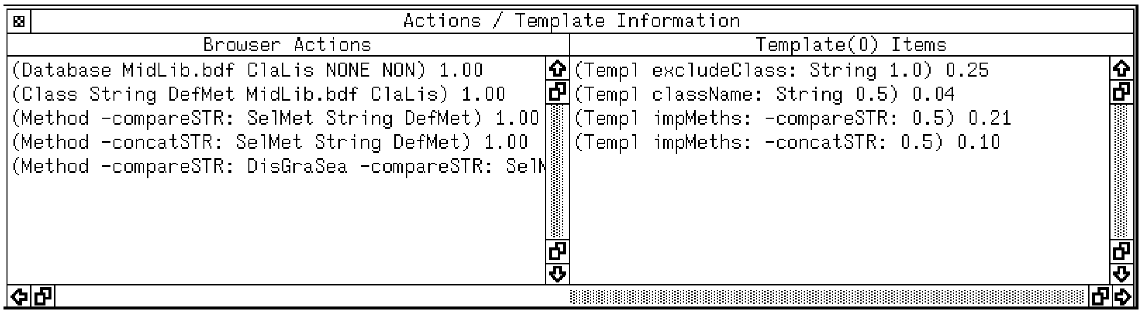

Figure 5: Inferences After Exploring Class "String"

class satisfying all his requirements the user must search elsewhere. Knowing that there is a general class called "Layer" for displaying many kinds of objects the user might at this point adopt an alternative strategy. Rather than focusing on classes that are likely to have methods to manage strings, the user focuses instead on those likely to have display methods. The user's search goal has changed somewhat. The expectation, now, is that the class will be similar to "Layer" but with methods specific to strings.

Returning to the original list of class names, the user selects Layer (Window A) and opens it to show its methods. This list is in the upper half of the lower right portion of figure 4 (Window C), just above the corresponding list for class String (Window D). The user sees that there is no method for displaying strings and so decides to broaden the search. The user refines the search goal to include two particular methods that affect how the final display of text will appear. The first, "extent:", sets the boundary size of the display area; the second, "attachTo:", relates to attaching text to displays of other information. The user expands these methods and requests a list of all classes that implement them, or similar methods. This list is shown in the upper right hand corner of figure 4 (Window B). The score associated with each class indicates the degree to which it implements both of the methods.

The right window in figure 6 shows all the inferences the active browser has made at this point. The first four lines are the same as in figure 5, except the weights have been reduced as this information is older and therefore less reliable than the more recent information. The bottom four lines represent inferences made from the actions involving the "Layer"

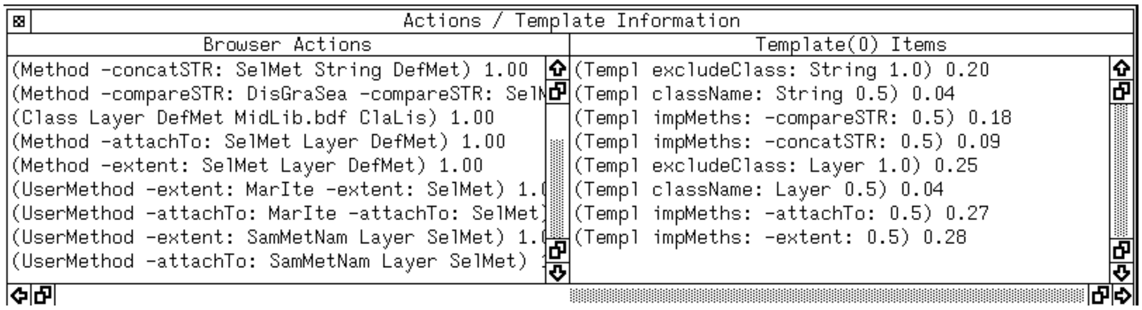

Figure 6: Inferences After Exploring Classes “String” and “Layer”

class and its methods. Note that the weights associated with the two methods examined in class “Layer” (0.27 and 0.28) are higher than those associated with the methods examined in class “String” (0.18 and 0.09). This is because the user not only examined the “Layer” methods but also specifically directed the browser to find other classes implementing the “Layer” methods. This gives the active browsing system much greater confidence that these methods are important elements of the user’s search goal.

The classes shown in the list returned as a result of the user’s most recent action (top right of figure 4) have identical scores. The methods selected, or similar ones, are extensively used in Layer-type classes, so the user’s asking for classes that implement them is a poor discriminator. The active browser discriminates better. To match maximally with the template shown in figure 6, and thus be high in the suggestion box at the present moment, a class should meet various criteria. It should have a name that matches with “String” and “Layer”. It should match with methods “concatSTR:” and “compareSTR:”, and, more importantly, with the heavily weighted methods “extent:” and “attachTo:”. It should be a class in which the user has not previously shown a lot of interest.

The suggestion box (Figure 7) shows “StringLayer” as a clear favorite, a class used specifically for the display of strings of text. “StdLayer” and “BaseLayer” are the two most general layer classes that allow a string to be displayed as window’s title. “StringEdit” is similar to “StringLayer” but in addition it allows editing of text. “LabelValue”, “Confirmer” and “Prompter” are specific types of “StringLayer”. Thus active browsing has succeeded in focusing the search on “Layers” with various ways of displaying strings.

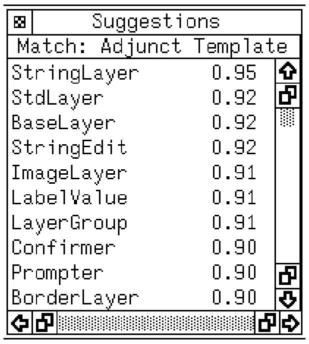

Figure 7: Suggestions

4 Design of Experiments to Evaluate the Active Browser

The remainder of the paper is an experimental evaluation of the system just described. The main question addressed is, how often is the user's search goal correctly inferred from his normal browsing actions? Some additional experimentation also investigates the effect on search time when the suggestion box is used.

Two experiments are presented. The first is a large-scale experiment in which each of the 189 classes in the library is used as a search goal. The library contains two commercially available sets of classes, the "ICpak101 Foundation Classes" and the "ICpak201 User Interface Classes" produced by the Stepstone Corporation. As it was infeasible to conduct so large an experiment with human subjects, an automated heuristic searcher was developed and used. The advantage of using a automated searcher is the generally accepted idea [32] that by using "artificial data" (which is what the automated searcher provides), one can run experiments that are larger-scale, much more carefully controlled, and repeatable, than experiments using "real" data (human searchers, in this case).

The use of an automated searcher is a variation of the standard method for evaluating relevance feedback systems in Information Retrieval research. In the latter, standard document collections are accompanied by a predefined set of queries and expert-supplied relevance judgements relating each document to each query. Given this data, experiments can be run without human subjects. "User judgements are simulated" [24] by assuming that users pass judgement on the n top-ranked documents returned by the initial query and by further assuming that users' judgements are exactly the same as experts'. Our main innovations are to add a parameter ("noise") representing the fallibility in the user's judgement and to adapt the 1-step relevance feedback studies to multiple steps involving a variety of user decisions by adding a search control strategy. To validate the results of the large-scale experiment with an automated searcher, a second experiment using human subjects was undertaken.

4.1 The Automated Searcher

The automated searcher has not been designed to simulate the rich and varied behavioral characteristics of human searchers. Rather it encompasses some general heuristics that a systematic searcher might be expected to follow. The search strategy it follows is very similar to that demonstrated in the example just discussed. The automated searcher consists of two parts (Figure 8): a "fuzzy oracle" that represents the search goal, and a heuristic search strategy that consults the oracle and selects browsing actions.

4.1.1 The Fuzzy Oracle

The fuzzy oracle contains information about a selected class and will give yes/no answers to specific questions. It can be asked if a class name partially matches with the name of the selected class. It can be asked if a method name partially matches with a name of a method in the selected class. It can be asked if the group of matched methods are more similar to those of the selected class than the last matched group. The oracle is "fuzzy" because its answers are not always correct; for each type of question, there is a probability that the oracle will give an incorrect response. This probability represents the searcher's uncertainty in the degree of match between a library item and the requirements.

The answers to these questions have some intuitive relationship to the knowledge of a human user. A positive response for a class name is biased towards names that have common sub-terms and in a weaker sense to the children of classes that match. A positive response for a method name is biased towards identical names and in a weaker sense to names with common sub-terms and to inherited methods. The answers to these questions

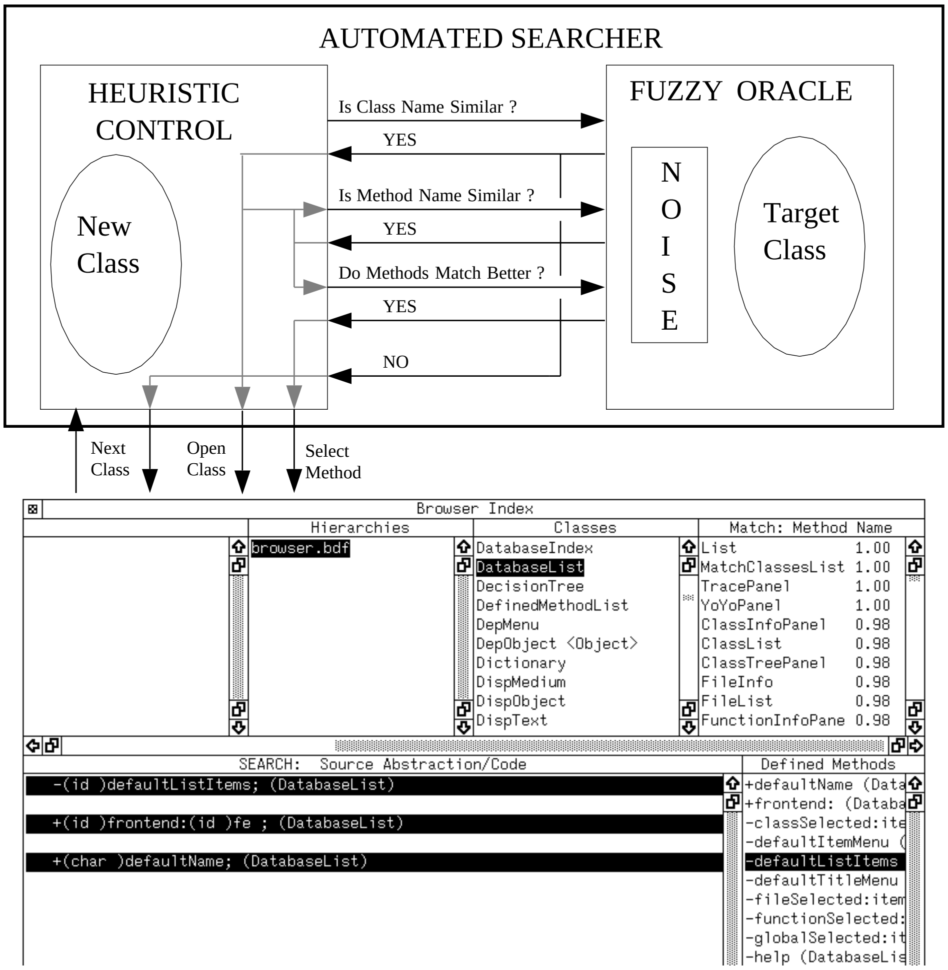

Figure 8: The Automated Searcher

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 the role(s) they can play.

4.1.2 The Heuristic Search Strategy

The heuristic search strategy is a combination of depth-first search and hill-climbing. The automated searcher iterates through the most recently generated list of class names from top to bottom. For each name it asks the fuzzy oracle if the name is similar to that of the target class. If the answer is no, the searcher proceeds to the next class in the list; if there are no more classes on the current list it backtracks to the previous list. In figure 8, the searcher began on the list entitled "Classes" and asked the fuzzy oracle if the first name on this list, "DatabaseIndex", is similar to that of the target class. The answer was no, so the searcher proceeded to the next name on the list, "DatabaseList" and asked the fuzzy oracle if this name is similar to that of the target class. This time the fuzzy oracle answered yes.

When the fuzzy oracle says that a class's name is similar to the target's, the class is opened. In figure 8, this resulted in the names of the methods implemented by "DatabaseList" being displayed on the lower right hand side of the screen. The automated searcher now picks method names from this list at random and asks if each matches with the name of a method in the target class. Each method name for which the oracle answers yes is selected, causing information about it to be displayed in the lower left window. This process continues until a maximum number of methods have been selected (a random number from three to five) or there are no more methods in the class. In figure 8, three methods have been selected: "defaultListItems:", "frontend:" and "defaultName:".

The automated searcher now asks if the selected methods are a better match to the methods in the target class than those selected in the previous class. If the answer is yes, as it is in the example, the searcher applies an operator that lists the classes that best match the selected methods. This list appears in the upper right hand corner of the browser's interface (Figure 8). The searcher then continues its search using this newly-generated list of classes. The search terminates when the automated searcher find the target class.

Two factors combine to cause the top classes in each new list to be more similar to the target, on average, than those of previous lists. The first factor is the hill-climbing. The second is that the noise in the fuzzy oracle is reduced each time a new class is opened, so that as search proceeds the automated searcher 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 Experimental Results

The main experiment measures how successful the active browsing system is in inferring the target class from the searcher's actions. The measure of "success" used is the frequency with which the active browsing system identifies the target class prior to its being found by the searcher. As described in section 4, two versions of this experiment were done. The first is a large-scale study using the automated heuristic searcher. The second is a small-scale study using human searchers. The results of these experiments are reported in sections 5.1 and 5.2 respectively. In neither version of this experiment is the "Suggestion Box" used by (or even shown to) the searcher. Section 5.3 discusses these experiments and describes a further experiment that measures the effect on the search of using the suggestion box.

In all experiments, each time the searcher (whether human or automated) creates a new class list or backtracks to a previous list, a record is made of the rank of the target class in the searcher's current list (henceforth called the searcher's ranking) and the rank, at that moment, of the target class in the suggestion box produced by the active browsing system (henceforth called the system's ranking). For example, the information shown in figure 9, is recorded for the target class "ProbableMethodsList".

In the experiment with the human subjects, the search for each target class continued until it was found by the subject. In the experiments with the automated searchers, the search was terminated when either (a) the searcher found the target class, or (b) the target class was ranked in the top ten by the active browsing system for five consecutive steps. Criterion (b) applies in the example of figure 9.

Target Class ProbableMethodsList

Step Number	System's Ranking	Searcher's Ranking	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 9: Record Of Ranking During Search

5.1 Results Using the Automated Heuristic Searcher

The experiment consists of successively using each class in the library as the target class. By varying the noisiness of the oracle's answers to questions posed by the automated searcher the effect of a human searcher's uncertainty was investigated. The experimental results presented below are for three different noise levels.

Intuitively, a target class is a "win" for the active browsing system if the system correctly infers it before it is found by the searcher. Likewise, a target class is a "loss" if the searcher finds the target class before the active browsing system infers it. There are a variety of different ways of defining a "win" and a "loss"; here two criteria are considered.

Criterion 1: A target class is a win for the active browsing system if it is ranked in the top ten by the active browsing system for five consecutive steps OR if the system's ranking is higher (better) than the searcher's ranking when the searcher finds the target class. A target class is a loss if the searcher finds the target class and, in that final step, the target class is not in the system's top ten. All other target classes are draws.

Criterion 2: A target class is a win only if it is ranked in the top ten by active browsing system for five consecutive steps. A target class is a loss if the searcher finds the target

Crtn. No.	Low Noise			Moderate Noise			High Noise		
	Wins	Losses	Draws	Wins	Losses	Draws	Wins	Losses	Draws
1	84	55	50	78	66	45	61	76	52
2	44	79	66	40	90	59	29	99	61

Table 1: Wins-Losses-Draws

class and, in that final step, the searcher's ranking is higher than the system's ranking. All other target classes are draws.

The second criterion is less generous towards the active browsing system, awarding it fewer wins and more losses. There are other realistic definitions of "win" that are more generous than criterion 1: for example, one could award a win if the active browsing system inferred a class that was extremely similar to the target class. Table 1 shows wins-losses-draws for the two criteria and three representative values of the one noise parameter that was varied. The number of wins, according to criterion 2, is much smaller. But with this criterion a win is impossible for target classes that are found by the searcher in fewer than five steps. There are 75 such target classes; if these are disregarded, active browsing succeeds on almost exactly the same percentage of classes with either criterion. For example, with low noise, the 84 wins according to criterion 1 is 44% of all the searches and the 44 wins according to criterion 2 is 40% of the searches with five or more steps. The results in table 1 show that for low noise using criterion 1 the number of active browser wins is very close to half the classes.

Two factors are critical in the definition 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 table 1 to the choice of "five consecutive appearances in a box of size ten", the number of wins were recorded while varying the two factors. Details of this experiment can be found in [13]. The results were found not to be highly sensitive to the suggestion box length. There is a sensitivity, not surprisingly, for very small suggestion boxes. However, this quickly levels out for sizes greater than eight. The value of ten used in these experiments is therefore a representative one. There is a higher degree of sensitivity to the number of appearances in the suggestion box required for a win to be assigned. This is somewhat more pronounced for the the second criterion, where this the sole definition of a win. As this number is reduced wins for the two criteria become progressively closer.

The measurement of wins-losses-draws directly indicates how often the target class is highly ranked by the active browsing system at the termination of the search, but it gives no indication of how the ranking evolved as the search progressed. If a searcher is to benefit from an active browsing system in practice, it must be true that throughout the search the system's ranking of the target class is consistently significantly higher than the searcher's own ranking of the target. Experimental results given in [13] show that for low noise the target class is consistently ranked much higher by the active browsing system than by the searcher on all steps except the first few. A similar pattern occurs for other noise values, but as the noise value increases the difference diminishes and becomes more erratic. As the noise level is further increased the rank difference gradually returns to being consistent and significant, but with the searcher having a better ranking than the active browsing system.

5.2 Experiments with Human Subjects

This experiment is identical to the preceding one except that human subjects were used, not an automated searcher, and only a small number (5) of library classes were used as targets. The purpose of this experiment is to verify that the results obtained using an automated searcher carry over to human searchers. The experimental design was the same. The human subjects were required to find a specific target class using the normal browser. The active browsing system was disabled during search. The user's actions were recorded and replayed later with the active browser enabled.

In preparation for this experiment, some initial experimentation was done by the authors, and some modifications were made to the rule base and the values used for template matching. Of these, the most noteworthy is the addition of rules associated with browsing operators applied to a selected group of methods. This initial experimentation showed that a good search strategy was to investigate classes that might be used by the target before "homing in" on the target class itself. Rules were added that embodied the inference that if a user showed interest in certain methods, classes that sent messages to these methods, as well as those that actually implemented them, might be of interest to the user.

Five example classes were chosen as search targets. These classes are principally concerned with user-interfacing. Using classes which could be displayed on the screen and manipulated by the user minimizes any experimental bias in the specification of the target class. The only changes made were to remove names in titles and menus which where strong pointers to the target class. Four of the five examples are shown in figure 10.

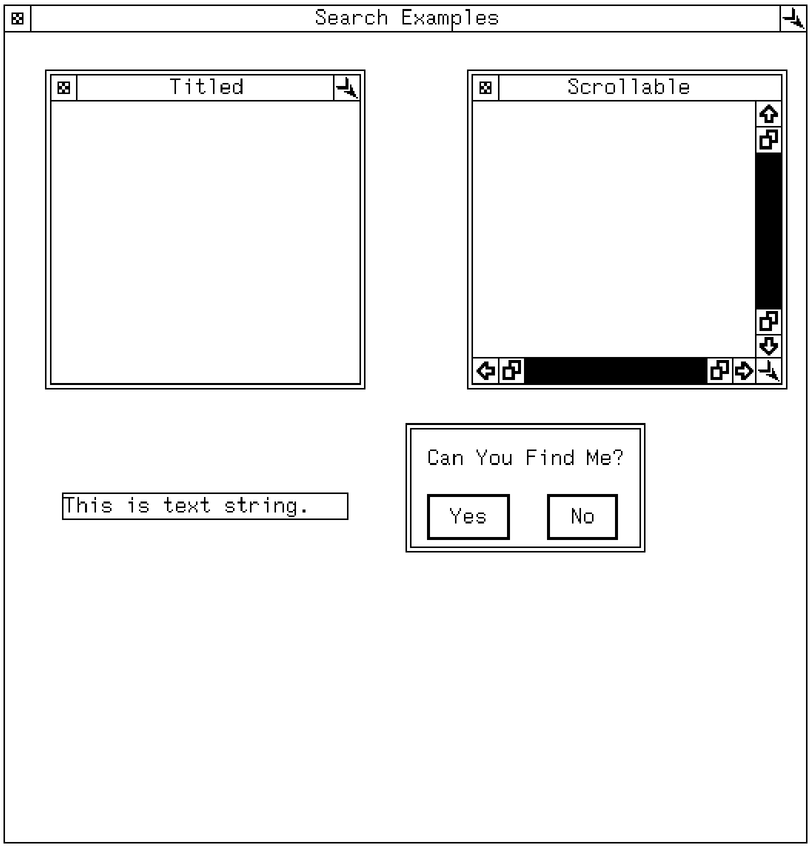

Figure 10: Four of the examples used in the experiments with human subjects

The order with which each class was presented to the the user is as follows:

- StdLayer: A "window" with just a title displayed.

- StdSysLayer: Similar to StdLayer but with scrollbars.

- Confirmer: Asks yes/no style questions of the user.

- SpecTLayer: Displays a text string and can be "attached" to other layers. (Not in the library used in the experiment with the automated searcher.)

- EchoBox: A "rubber rectangle" whose size is set by the user moving the mouse.

Two groups of experimental subjects were selected from a population of 550 undergraduate and 150 graduate students in two departments, Computer Science, and Electrical and Computer Engineering. The first group comprised 5 graduate students with some experience of object-oriented languages but no more than a cursory knowledge of Objective-C. The second group comprised 10 undergraduate students, all of whom were less knowledgeable than the subjects in the first group. Each subject was give some initial instruction on how to use the browser. The experiment was automated so that each target was presented in turn to the subject. When the current target class was opened to explore its methods, the search ended and the next target was presented. Due to some minor experimental problems, there is not a complete record for every subject on every target. Out of the total of 75 runs of all subjects on all targets, data for only 65 is available for analysis.

Table 2 shows the lengths of the searches required by each subject on each target, together with the number of runs and the median value for the group. The '*' in Table 2 indicates the search length of the automated searcher on each target (it was not run on SpecTLayer, so there is no '*' in that row). The most striking feature of this table is its diversity. Each target was found quickly (5 steps or fewer) by at least one subject, and, except for SpecTLayer, very slowly by at least one subject. In fact, except for the first and last targets, almost exactly half the subjects found the target quickly, while the other half required 10 steps or more. The small number of steps taken by the less knowledgeable group to find the last target in the experiment, EchoBox, is suspicious, and there is evidence that several of these subjects resorted to scanning for the target by exhaustively scrolling down the alphabetical list of classes, opening them one by one. That this worked at all is a property of the experiment, if the target class is opened the current search is terminated and a search for the next target class started. Such scrolling is not an effective browsing technique in reality, the subjects were not thinking about the choice of

target	runs	med.	search lengths													
StdLayer																
group 1	5	10	5	6		10			43				66			
group 2	9	46			9		19	40		45	46	47		73	76	77
combined	14	44		*												
StdSysLayer																
group 1	5	3	1		2		3				12	12				
group 2	9	12		2		2		3	3	12			12	24	27	149
combined	14	8								*						
Confirmer																
group 1	5	3	1	2	3					21			30			
group 2	8	17				3	3	4	11		23	24		51	52	
combined	13	11						*								
SpecTLayer																
group 1	4	6	3		5		6				14					
group 2	9	14		3		6		6	6	14		14	17	17	17	
combined	13	6														
EchoBox																
group 1	4	6				4			5		7		39			
group 2	7	4	2	4	4		4	4		6		7				
combined	11	4									*					

Table 2: Lengths of searches for all Searchers

browsing action. Such searches seem short in the table because scrolling does not constitute a "step" according to our definition. On the other four targets the superior performance of the group-1 subjects is evident. The diversity of search lengths for each target confirms our observations during the experiment that the subjects were using very different search techniques.

Table 3 shows the number of wins, losses and according to the criteria defined in the preceding section. Criterion 2 credits active browsing with wins in 35% of the searches, which is 52% of the searches five steps or longer. Criterion 1 credits active browsing with wins in 69% of the searches. These numbers might be somewhat inflated by the performance on StdLayer, the first target in the experiment. This target had unusually

Criterion No.	Result		
	Wins	Losses	Draws
1	45 (69%)	8 (12%)	12 (19%)
2	23 (35%)	17 (26%)	25 (39%)

Table 3: Wins-Losses-Draws results for human subjects

long searches and active browsing was awarded a win by both criteria on this target for all but 3 subjects. Whether these effects are due to its being first or are intrinsic to the target could only be determined by an experiment in which the order in which the targets were presented was varied from subject to subject. For the other four targets there is no clear evidence that the fixed order of presentation affected the number of wins observed. But further experimentation would be needed to confirm this using a factorial design as in the experiments of Frakes and Pole [18].

If the first target is ignored, criterion 1 credits active browsing with wins in 66% of the remaining searches, criterion 2 with 24%, which is 40% of the searches five steps or longer. These results are similar to the low noise results for the automated searcher, where 40-44% of the searches were wins for active browsing.

The wins for active browsing – as defined by either criterion – were approximately evenly distributed across targets StdSysLayer, SpecTLayer, and EchoBox. As just noted, StdLayer was virtually always a win for the automated searcher. Conversely, for Confirmer the automated searcher was awarded just one win by criterion 1 and none by criterion 2.

The potential usefulness of active browsing is highlighted in our experiments by the following observation. About half the time when the target appeared in the top ten of a human subject's class list the subject did not select it until many steps later. Yet in almost all of these cases the target appeared in the suggestion box at about the same time as in the user's top ten and remained there for at least 5 steps (sometimes many more). Thus active browsing clearly is useful in drawing attention to the target when the user has overlooked it. A second important observation is that active browsing can identify the target even before it has ever appeared in the top ten of any of the user's class lists. This happened on about 20% of the searches that are five steps or longer.

5.3 Discussion And Analysis Of Results

The results show that the active browsing system frequently (more than 40% of the time) infers the target goal before it is found by the searcher. This success rate was consistently achieved across all the searchers studied, fifteen humans and the automated heuristic searcher. This section will address three outstanding questions: are the results "statistically significant", will the approach actually speed-up search and why is inference poor under certain conditions?

5.3.1 Statistical Significance

To test statistical significance of the number of wins achieved by the active browsing system (criterion 2, Table 1), we use a Monte Carlo method [8, pp.147-152] to test the null hypothesis that the observed number of wins is the same as the number that would be achieved by a system that randomly generates suggestion boxes. The random system that was implemented for this purpose initializes the box randomly and then on each step retains X of the current box members (randomly selected) and draws $boxsize - X$ from among the classes not in the box. $X = 8$ was found experimentally to optimize the results for $boxsize = 10$ It proceeds until it wins according to criterion 2 or the maximum number of steps for this target (see below) is exceeded. Then it moves on to the next target. A trial consists of running the random system on each of the targets that required 5 or more steps to be found. The number of wins in each trial is recorded, and the significance level with which the null hypothesis can be rejected is estimated as the percentage of trials on which the random system had the same or more wins as the active browsing system. To get an accurate estimate of this significance level 10,000 trials were executed.

The maximum number of steps for target T, maxsteps(T), in this Monte Carlo test should be equal to the number of steps required by the user to find the target. Unfortunately, this number is not known for the targets that were wins for the active browsing system because the user's search was terminated before those targets were found. For the Monte Carlo test it was therefore necessary to set maxsteps(T) to an estimate of the number of steps the user would have taken to find the targets that were wins. Here we report two different methods of estimating maxsteps(T): the first method is perhaps slightly optimistic, the second is almost certainly pessimistic.

In the first method, maxsteps(T), is set as follows:

```
let steps(T) = number of steps for target T in section 5.1's experiment
if T was not a win for the active browser in section 5.1's experiment
   then  maxsteps(T) := steps(T)
if T was a win for the active browser in section 5.1's experiment
   then  maxsteps(T) := average of maxsteps(T') for all targets T'
                        such that steps(T') > steps(T).
```

For example, the target $T1$ that was the longest win in section 5.1's experiment was 63 steps. There were two runs that were longer, 68 and 81 steps thus $maxsteps(T1) = (68 + 81)/2$, which rounds to 75. The second longest win, $T2$ was 43 steps. The runs that were longer were nonwins of lengths 47, 47, 68 and 81, and the T1 run, whose maxsteps we have just computed as 75. Thus $maxsteps(T2) = (47 + 47 + 75 + 68 + 81)/5$, which rounds to 64.

Using this method of setting maxsteps(T), none of the Monte Carlo trials equaled or surpassed the number of wins achieved by our active browsing system in the low noise experiment. The most wins on a Monte Carlo trial was 25, whereas the active browser won on 44 of the 113 targets. Since there were 10,000 Monte Carlo trials, the performance of the active browser is significant at the 0.01% level. The same is true with the high noise results, where the active browser had 29 wins out of 122 targets and the most wins on a Monte Carlo trial was 27.

The second method of estimating maxsteps(T) for targets that are wins in section 5.1's experiment is based on the assumption that the wins would have been the longest of the user's searches. Maxsteps(T) is thus set to be the longest recorded user search, which is 81 in the low noise case and 94 in the high noise case. As this method gives the random system more time to guess the target by chance, the confidence with which the null hypothesis can be rejected will be smaller. Using this method, the active browser's performance in the low

Low Noise			Moderate Noise		
Wins	Losses	Draws	Wins	Losses	Draws
52	32	105	51	44	94

Table 4: Using the suggestion box

noise case is still significant at the 0.01% level, but its performance in the high noise case is significant at the 0.24% level. The latter is still a very strong result, since it is obtained under two very pessimistic assumptions (large maxsteps(T) and an optimized value of X) and it is superior to 0.5%, which is generally regarded as highly significant,

5.3.2 Speeding-up Search

One extension to the experiment using the automated searcher, more akin to how the tool would be used in practice, is to have the automated searcher make selections from the suggestion box. Here, the searcher makes a selection from either the suggestion box or the previous class list with equal probability every time it backtracks. This follows the intuitive notion that the suggestions are most likely to be used when a human user is unsure of how to proceed. At this juncture the user may equally well look at the suggestion box or unexplored classes in earlier lists.

The results in table 4 are for the low and moderate noise values. For each value the experiment is run twice, with and without the use of the suggestion box respectively. The search is terminated when the searcher finds the target class. The number of steps required to find each class in the library is compared. In this case only a single criterion is used. A win is assigned to the active browser if it takes fewer steps when turned on, a loss if it takes more and a draw if they are equal. The results show a significant benefit at the lower noise values, some of which is lost at the higher values. The ratio of wins to losses is comparable to that given by criterion 1 of the simple comparison experiments.

5.3.3 Inference Failures

There are two conditions under which active browsing did not outperform the automated searcher: during the early steps of search, and when the searcher is highly uncertain about the search goal (simulated in the experiments by a high noise level). That active browsing does not outperform the searcher in the early steps of search can be explained by the fact that it has insufficient information upon which to base its rankings.

The results clearly show that the performance of active browsing system is degraded by increased noise levels more than the searcher's performance. This was unexpected: the evidence-accumulating mechanism in the active browsing system ought to act as a noise filter. There are numerous factors that might have produced this effect. The most likely explanation is that the searcher's behavior consisted of a large number of uninformative actions followed by a relatively small number of highly informative actions that led the searcher to the target. If the searcher behaves in this way, the active browsing system will lag behind the searcher in moving towards the target because the final few informative actions will not immediately outweigh the mass of previous uninformative actions. This kind of "misleading" initial behavior by the searcher is promoted by the fact that the oracle's noise level decreases as the searcher's search progresses. It is further promoted by the fact that in the test library, as in most software libraries, there are some fairly large groups of highly similar items. The searcher may explore one of these groups for a long time before "realizing" that such classes do not truly meet the requirements. One approach to address this problem is the use of machine learning methods to cope with "drifting" concepts [44].

6 Limitations

This section discusses limitations to the whole approach of active browsing and presents various ideas as to how these might be overcome.

The system design is based on the assumption that the library consists principally of one type of item and the target of the user's search is a single item. Of course a single item may not satisfy the user's requirements, but finding all combinations of items that satisfy the search goal would be too computationally expensive. One possible solution requires additional layers of granularity in the library. For instance, the same entity could simultaneously be an individual library item and also a component in a larger item. In this case, relevance would be evaluated of both the isolated entity, and the collection that contains it. The score, for both types of item, would still form the basis for ranking, perhaps biased in favor of the smaller ones. This situation arises within the object-oriented paradigm, where each class is an individual item and may also be a part of a "framework".

An important consideration that has not been addressed in this paper is the computational complexity of the active browsing system. Our long-term aim is to have the active browsing system transparent, by having it use only the computer time that is available between user actions. At the present stage of development each time a new template term is added, it is evaluated against every class in the library. Each update, therefore, takes in order of the product of the number of items in the library and the average number of terms in each class to be matched. As this is the critical operation the complexity scales linearly with library size. With the existing scheme and a library of 189 classes the system is able to operate in real time, with a barely noticeable delay between user actions. For a much larger library, the time available between actions is insufficient for our current active browsing system to operate without introducing appreciable delays. We are currently investigating a variety of methods that reduce the time required for active browsing without significantly degrading its success rate [37]. The general idea underlying these techniques is to be highly selective about which classes and template terms are evaluated.

7 Conclusion

This paper has described a system for active browsing and experimentally demonstrated that a searcher's goal can often be inferred from normal browsing actions. The inferred goal provides a reliable way of estimating the "relevance" of a library item to the searcher's goal. Active browsing increases the effectiveness of browsing without imposing any restrictions or "cost" on the user. The searcher is able to make full use of the facilities of the standard browser, and is not required to take special actions or learn any new tools.

Future experiments should use a rich variety of realistic automated searchers. Results of such experiments should be verified by human studies in which targets are presented in random order to eliminate any possible learning effects. Further work on automated and human searchers using the system's suggestions would permit measurement of the relative effectiveness of different ways of presenting the inferred goals to the searcher.

8 Acknowledgments

The work was supported in part by a Strategic grant and operating grants from the Natural Sciences and Engineering Research Council of Canada. Our colleagues on the "Machine Learning Applied to Software Reuse" project provided helpful comments at all stages of this research.

References

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

[2] M. J. Bates.(1986) *Terminological Assistance For The Online Subject Searcher.* Proceedings of the Second Conference on Computer Interfaces for Information Retrieval. p 7

[3] M. Bauer. (1996) *Machine Learning for User Modeling and Plan Recognition.* Proceedings of the Machine Learning Meets Human Computer Interaction Workshop: Thirteenth International Conference on Machine Learning pp 5–16

[4] N.J. Belkin and W.B. Croft (1987) *Retrieval Techniques.* Annual Review of Information Science and Technology Vol. 22 pp 109–145

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

[6] E. Bos. (1992) *Some Virtues and Limitations Of Action Inferring Interfaces.* Fifth Annual Symposium on User Interface Software and Technology pp 79–88

[7] F.R. Campagnoni and K. Ehrlich.(1989) *Information Retrieval Using A Hypertext-based Help System.* Proceedings of the Twelfth International Conference on Research and Development in Information Retrieval pp 212–220

[8] P. R. Cohen. (1995) *Empirical Methods for Artificial Intelligence.* MIT Press

[9] R. Cohen and B. Spencer. (1993) *Specifying and Updating Plan Libraries for Plan Recognition Tasks.* Proceedings of the Ninth Conference on Artificial Intelligence Applications pp 27–33

[10] A. Cypher.(1991) *EAGER: Programming Repetitive Tasks by Example.* Proceedings of CHI-91: Human Factors in Computing Systems pp 33–39

[11] L. Dent, J. Boicario, J. McDermott, T. Mitchell and D. Zabowski. (1992) *A Personal Learning Apprentice.* Proceedings of the Tenth National Conference on Artificial Intelligence pp 96–103

[12] P. Devanbu, R. Brachman and P. Selfridge. (1991) *LaSSIE: A Knowledge-Based Software Information System.* Communications of the ACM Vol. 34(5) pp 34–49

[13] C. Drummond. (1992) *Automatic Goal Extraction from User Actions to Accelerate the Browsing of Software Libraries.* M.A.Sc. Thesis, University of Ottawa

[14] T. W. Finin. (1983) *Providing Help and Advice in Task Oriented Systems.* Proceedings of the Eighth International Joint Conference on Artificial Intelligence pp 176–178

[15] G. Fischer, S. Henninger and D. Redmiles. (1991) *Cognitive Tools For Locating And Comprehending Software Objects For Reuse.* Proceedings of CHI-91: Human Factors in Computing Systems pp 318–328

[16] G. Fischer, A. C. Lemke, T. Mastaglio and A. I. Morch.(1990) *Using Critics to Empower Users.* Proceedings of CHI-90: Human Factors in Computing Systems pp 337–347

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

[18] W. B. Frakes and T. Pole. (1994) *An Empirical Study of Representation Methods for Reusable Software Components.* IEEE Transactions on Software Engineering Vol. 20(8) pp 617–630

[19] W. B. Frakes and P. Gandel.(1990) *Representing Reusable Software.* Information and Software Technology Vol. 32(10) 1990 pp 47–54

[20] W. B. Frakes and B. A. Nejmeh.(1987) *Software Reuse Through Information Retrieval.* Proceedings of the Twelfth Annual Hawaii International Conference on System Sciences pp 530–535.

[21] P. Freeman. (1983) *Reusable Software Engineering: Concepts And Research Directions.* Proceedings of the Workshop on Reusability in Programming pp 2–16.

[22] G.W. Furnas, T.K. Landauer, L.M. Gomez and S.T. Dumais. (1987) *The Vocabulary Problem In Human-System Communication.* Communications of the ACM. Nov Vol. 30(11) pp 964–971

[23] B. A. Goodman and D. J. Litman. (1990) *Plan Recognition for Intelligent Interfaces.* Proceedings of the Sixth Conference on Artificial Intelligence Applications pp 297–303

[24] D. Haines and W. B. Croft.(1993) *Relevance Feedback and Inference Networks.* Proceedings of the Sixteenth International Conference on Research and Development in Information Retrieval pp 2–11

[25] D. Harman.(1992) *Relevance Feedback Revisited.* Proceedings of the Fifteenth International Conference on Research and Development in Information Retrieval pp 1–10

[26] R. Helm and Y. S. Maarek.(1991) *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

[27] S. Henninger.(1991) *CodeFinder: A Tool For Locating Software Objects For Reuse.* Automating Software Design: Interactive Design Workshop Notes Ninth National Conference on Artificial Intelligence pp 40–47

[28] L.A. Hermens and J.C. Schlimmer.(1993) *A Machine-Learning Apprentice for the Completion of Repetitive Forms.* Proceedings of the Ninth Conference on Artificial Intelligence for Applications 1993 pp 164–170

[29] K. Hook, J. Karlgren and A. Woern.(1993) *Inferring Complex Plans.* Intelligent User Interfaces-93 pp 231–234

[30] A. Jennings and H. Higuchi.(1993) *A User Model Neural Network for a Personal News Service.* User Modeling and User-Adapted Interaction Vol. 3 1993 pp 1–25

[31] C. Kaplan, J. Fenwick and J. Chen.(1993) *Adaptive Hypertext Navigation Based on User Goals and Context.* User Modeling and User-Adapted Interaction Vol. 3 1993 pp 193–220

[32] D. Kibler and P. Langley. (1989) *Machine Learning As An Experimental Science.* Proceedings of the Third Working Session on Learning pp 1–12

[33] A. Kobsa. (1993) *User Modeling: Recent Work, Prospects and Hazards.* Adaptive User Interfaces: Principles and Practice pp 111–128

[34] C. W. Krueger.(1992) *Software Reuse.* ACM Computing Surveys Vol. 24(2) pp 131–184

[35] Y. S. Maarek, D. M. Berry and G. E. Kaiser. (1991) *An Information Retrieval Approach For Automatically Constructing Software Libraries.* IEEE Transactions on Software Engineering Vol. 17(8) pp 800–813

[36] P. Maes and R. Kozierok. (1993) *Learning Interface Agents* Proceedings of the Eleventh National Conference on Artificial Intelligence pp 459-465

[37] J. D. Ng Yuen Yan (1997) *Improving Active Browsing with the Negative Inference and Selective Search Methods.* M.C.S. Thesis, University of Ottawa

[38] P. F. Patel-Schneider, R. J. Brachman and H. J. Levesque.(1984) *ARGON: Knowledge Representation meets Information Retrieval.* Proceedings of the First Conference on Artificial Intelligence Applications pp 280–286

[39] W. Pohl. (1996) *Learning About the User - User Modeling and Machine Learning* Proceedings of the Machine Learning Meets Human Computer Interaction Workshop: Thirteenth International Conference on Machine Learning pp 29–40

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

[41] B. Sheth and P. Maes. (1993) *Evolving Agents For Personalized Information Filtering* Proceeding of the Ninth Conference on Artificial Intelligence For Applications pp 345–352

[42] B. G. Silverman and T.M. Mazher. (1992) *Expert Critics in Engineering Design: Lessons Learned and Research Needs.* AI Magazine Vol. 13(1) pp 45–62

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

[44] G. Widmer and M. Kubat.(1993) *Effective Learning in Dynamic Environments by Explicit Context Tracking.* Proceedings of the European Conference on Machine Learning (ECML-93) pp 227–243

[45] M. D. Williams. (1984) *What Makes RABBIT Run ?* International Journal of Man-Machine Studies Vol. 21 1984 pp 333–352

[46] I. H. Witten and D. Mo.(1993) *TELS: Learning Text Editing Tasks from Examples.* Watch What I Do. MIT Press 1993 pp 183–204

[47] M. Wood and I. Sommerville. (1988) *An Information Retrieval System For Software Components.* Tech Report No. RR-88-14 University of Strathclyde Scotland

List of Figures

List of Tables

Biographies

CHRISTOPHER G. DRUMMOND received a B.Tech. in Applied Physics from Brunel University (England) and a M.A.Sc. in Electrical Engineering from the University of Ottawa (Canada). He is at present completing his Ph.D. in Computer Science at the University of Ottawa. His research interests center on Machine Learning, with primary focus on learning agents and on hybrid systems that combine symbolic and subsymbolic learning algorithms.

Dr. DAN IONESCU is a Full Professor with the University of Ottawa, Canada, the School of Information Technology and Engineering and the Director of the Machine Intelligence Research Laboratory. He graduated from the Polytechnic Institute of Bucharest in Computers and Control and from the University of Timisoara in Mathematics. He obtained his doctoral degree from the Polytechnic Institute of Bucharest. His current research interests are in formal frameworks for real-time and distributed systems, applications of real-time systems in communications, multimedia and in distributed programming and applications such as developing distributed applications in CORBA. He has published more than 95 papers in selected journal and conferences. His publication record counts also two textbooks and a chapter in another book. He submitted recently two patents applications. He is currently a senior member of IEEE, a member of SIAM and ACM.

ROBERT C. HOLTE received his B.Sc. and M.Sc. in Computer Science from the University of Manitoba, and his Ph.D. from Brunel University (England) in 1988. He was a visiting research scholar in the Computer Sciences Department of the University of Texas at Austin during 1988-89. He joined the University of Ottawa in 1989 and is currently a tenured associate professor in its School of Information Technology and Engineering (SITE). His research is in two areas: Machine Learning, and the use of "abstraction" techniques to speed up graph searching and heuristic search. On these topics he has published some 40 refereed scientific publications. He is currently the Editor-in-Chief of "Machine Learning", the leading international journal in that research field, and on the editorial board of the electronic Journal of Artificial Intelligence Research (JAIR).