A Learning Agent that Assists the Browsing of Software Libraries*

Chris Drummond†, Dan Ionescu† SM IEEE and Robert Holte††

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

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 [3, 19, 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.

A person's search through a software library is mediated by what we call a "browsing system" (in [10] it is called a software information system). Typically, such systems provide a variety of "browsing" operations, including various forms of content-based retrieval ("indexing") and ways of navigating through the library. The browsing system user evaluates alternatives, and chooses a browsing operation that is expected to bring the search closer to the user's implicit goal. The system plays the purely passive role of executing the chosen operations and presenting their results.

Research aimed at improving the speed and reliability of searching software (or other kinds of) libraries has considered a wide variety of techniques for improving browsing systems. A survey of techniques specific to software reuse may be found in [31] and [17]. Analogous techniques for information retrieval systems are summarized in [38].

In the most common approach to improving browsing, the browsing system remains purely passive. The speed and reliability of browsing are improved by increasing the effectiveness of the browsing operations, either by introducing new, more powerful operations or by organizing the library so that the existing operations are more effective. For example, [25, 32, 35, 42] provide expressive query languages and flexible matching, and [10] uses a rich knowledge-base and a powerful inference technique to answer its queries. In some approaches, the user may customize the browsing system [29, 14], or enter into a dialogue with the system in order to select better operators[5]. In other approaches, the user may refine the query, as in relevance feedback systems [22, 23], or query-reformulation systems [16, 13, 25, 34, 40]. Imposing structure on the library [35, 42] can also improve browsing speed and reliability.

Our approach to improving browsing systems is complementary to the preceding ones. We aim to add an active component to the browsing system, so that in addition to passively supporting user-directed search, the system actively assists or guides the user. An active assistant is an addition to, not a replacement for, a browsing system. Browsing proceeds as usual and may, in some circumstances, continue to completion without any interruption by the assistant. The usefulness of the assistant is that, often, it will recognize the user's "intent" and provide guidance or advice that accelerates the search.

There are two main types of active assistants, "daemons" and "learning agents" . Both types of assistant are background processes that watch the user's actions and, in certain circumstances, interrupt the user and offer unsolicited advice. The difference is that daemons are preprogrammed to recognize particular patterns of user behaviour, whereas learning agents learn from the user's actions when to interrupt and/or what advice to give.

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.

Research on learning agents is very recent and still highly exploratory. 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 agent research has addressed either of these issues.

This paper presents and evaluates a learning agent to assist the browsing of software libraries. The learning agent is completely unintrusive 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 particular question addressed is, how often can a user's search goal be inferred from normal browsing actions. Our experimental results are encouraging: roughly 40% of the time the system succeeded in identifying the user's search goal before the user reached 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 describes browsing in general, and a particular form of learning agent, called active browsing. These are illustrated with a concrete example. Section 4 describes strategies for inferring the search goal. Section 5 gives the architecture of the learning agent. This section highlights issues related to a specific implementation that is used to search libraries of object oriented code. Section 6 presents the experimental methodology used to test the feasibility of active browsing. Section 7 presents and discusses the experimental results. In section 8 the limitations of this approach are discussed and section 9 presents some general conclusions.

2 Related Research

The work presented in this paper is most closely related to research on active assistants. As mentioned above, an active assistant is a background process that monitors the user's actions and, in certain circumstances, interrupts the user and 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 behaviour, and, when a particular pattern of behaviour is detected, to issue the corresponding preprogrammed response. For example, the critics in Fischer's design environment [15] 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 [37]. Finin's active help system [12] 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 [27, 7], or "parse" the user's actions with a given set of plan schemas [21]. Most "programming by demonstration" systems that do inference employ daemons. For example, [4, 8, 41] all use a preprogrammed notion of "similar action" to detect repeated sequences of actions.

"Learning agents" do not have a preprogrammed set of situation–action rules, but instead they learn from the user's actions when to interrupt the user and/or what advice to give. For example, [36] 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 [28] 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.

Learning agents have also been developed to assist a user in filling in a form [9, 26, 33]. 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.

An important feature of all these learning agents, and of 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. If our learning agent's purpose was to learn in the long-term, feedback about its predictions after the search had ended would be useful. But its purposes is to speedup the search: feedback after the search has ended is of no use.

The information that is available to the learning agent during search is "noisy" and only very indirectly related to the correctness of the agent's predictions. It is noisy because the user is searching somewhat blindly. To some degree, the user will pursue deadends and circuitous routes, thus giving misleading feedback about which directions are "most promising". There are two reasons why the feedback is not directly related to the correctness of the agent's predictions. The first is simply that the user might choose to completely disregard the agent's suggestions. This would happen, for example, if the user is in the midst of pursuing his own search strategy. 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.

3 From Browsing To Active Browsing

The activities involved in normal browsing are shown on the left hand side of figure 1. The user starts with 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. It is true serendipity may result in the user discovering properties very different from those of the original requirements, but nonetheless useful to solve the broader problem. Although we made a conceptual division between the requirements and the search goal, from the active browser's viewpoint the two are indistinguishable. Thus such a change in direction of the search would be as if the user made a radical revision to the search goal. Such a situation is discussed in the limitations section (sect 8).

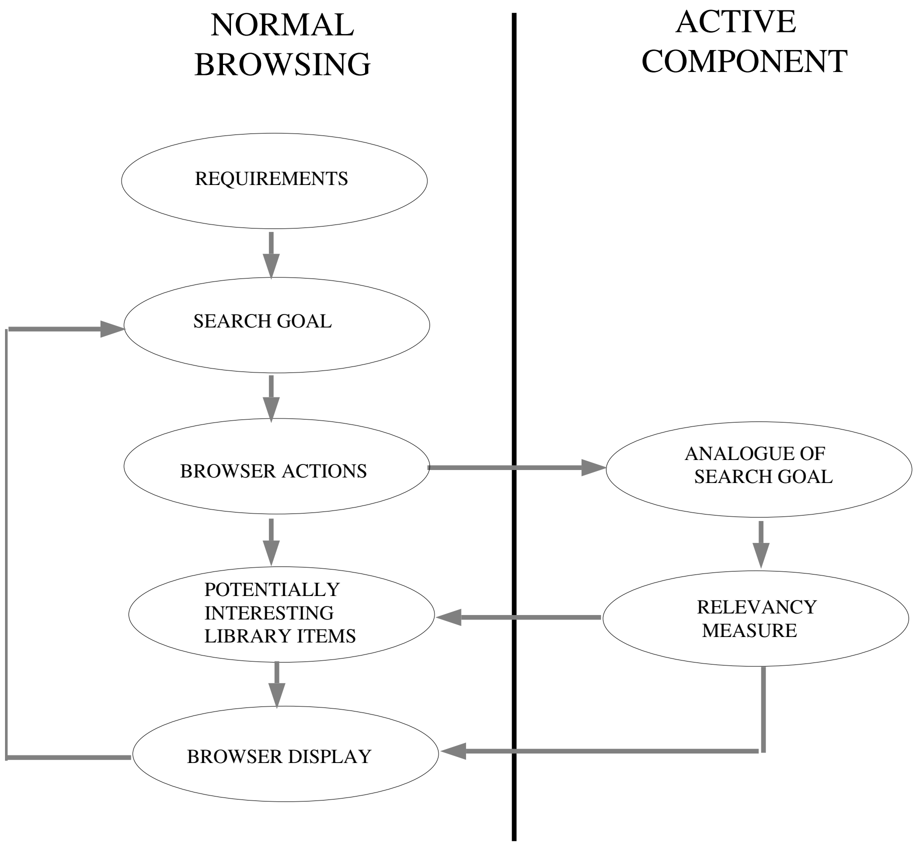

Figure 1: A Model Of Active Browsing.

The user converts the requirements into a search goal which describes 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 expectations about the form of the target item, even though the original requirements have not changed. Therefore, although the requirements should remain essentially constant, the search goal is likely to change during search. Transforming the requirements into a search goal represents the major source of uncertainty in browsing. Although the user may be sure of the requirements, the exact form of the item that best meets them is not clear. For instance even when the requirements characterize one specific item, known to be in the library, the user is unlikely to be able to describe it accurately.

Search begins with the user selecting an initial item in the library as a starting point, usually by indexing or directly selecting from a list of possible starting points. The user compares this item to the search goal and selects one or more browsing actions to move through the library to locate items that better match. Each action produces a set of potentially interesting items, which are displayed to the user. The user compares the items with the current search goal, perhaps revises the search goal, and then continues searching by selecting an item from any of the available lists and taking further actions. This process is repeated until the user is satisfied that an item meets the requirements or none can be found.

When the user is somewhat uncertain about the search goal and the usefulness of individual actions, browsing is an appropriate form search. Uncertainty is reduced through iterative search, with evaluation and refinement of the goal at each step. The cognitive attractiveness of browsing as a means for locating artifacts in a library has been observed and analyzed by several authors. Studies [6] 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 in [16] 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, [38] 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."

3.1 Active Browsing

As just described, a browsing system is a completely passive tool. The idea of active browsing is to enhance the effectiveness of the browsing system by having the system guide the user towards 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 first step in the active browsing process (the right hand side of figure 1) is to infer 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. This relevancy measure returns a numerical value representing the degree to which any particular library item matches the analogue. The measure is used to evaluate library items and the result of the evaluation is used to influence the user's search.

There are a wide variety of ways to use the evaluation of library items produced by the active browsing system. Perhaps the least disruptive option is to use this evaluation to allocate computational resources, such as the contents of a cache or the CPU time available for precomputing information the user is expected to request. This use of the evaluation improves the response time of the browsing system but does not reduce the amount of searching done by the user. The general strategy for reducing the user's search is to have the system somehow draw the user's attention to the items that are judged most relevant, and to draw his attention away from items of very low relevance. The most direct, but also most disruptive, way to do this is to alter the list returned in response to a query. Classes that are judged highly relevant, but would not normally have been returned by the query, could be added to the list or the ordering of classes in the list changed to reflect their relevance.

The simplest method, the one used in the current implementation, is to have a single, preferably small, extra window (the Suggestion Box) in which library items are displayed in order 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.

3.2 An Example Of Active Browsing

The current implementation of active browsing is an enhancement of a commercial browser for libraries of object-oriented software written in Objective-C. An item in this library is a "class", in the object-oriented sense. In this section, active browsing is illustrated on a combination of two commercially available sets of classes, the "ICpak101 Foundation Classes" and the "ICpak201 User Interface Classes" produced by the Stepstone Corporation.

3.2.1 Human Interface

The human interface to the system is shown in figure 2. At the top is a series of windows, which normally contain lists of classes. The user can select any class on any of these lists and apply a browsing operator to the selected class. Whenever a class (or method) is selected to be operated upon, it is highlighted in the display.

As in all object-oriented browsers, there is an operator that produces the list of the selected class's subclasses, and an operator that produces the list of its ancestors in the object hierarchy (its parent, its parent's parent, etc.). A third operator returns the list of classes whose names are "similar" to the name of the selected class (the similarity measure for class names is described below). The class list that these operators produce is placed in the rightmost window at the top of the interface.

The final operator that can be applied to a class produces a list of the names of the methods the class implements. This method list is shown in the lower right portion of the interface. This operation is called "expanding" a class. In figure 2 two classes have been expanded in this manner.

There are also operators that can be applied to methods. Individual methods in a class can be expanded in several stages to give more and more information about the method. This information is presented in the lower left portion of the interface. In this window the user can select a set of methods. Several operators can be applied to the set of currently selected methods. One operator returns a list of classes that implement

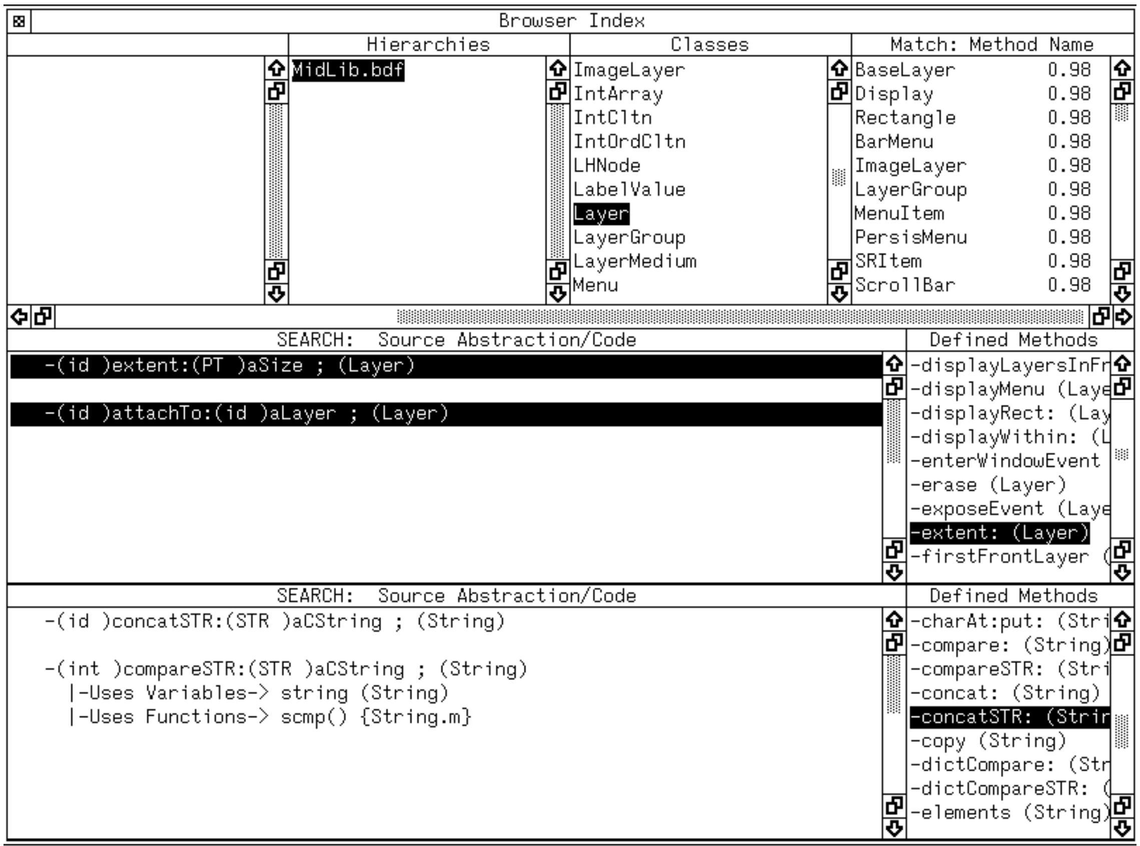

Figure 2: Exploring Two Different Classes

the marked methods. Another returns the list of classes that use the marked methods. An important feature of these operators is that matching is a matter of degree: a class can partially implement (or use) the set of selected methods. Partial matches occur, for instance, when the class implements a method with a similar but not identical name, or if a class implements a method that sends a message to a method with a similar or identical name. The partial matching scheme is discussed in detail in section 5.2. The class list resulting from one of these operators is sorted in order of match strength and displayed in the rightmost window at the top of the interface.

The output of the active component of the browser, a "Suggestion Box", shows the list of classes that the active browser considers potentially interesting to the user. This would normally be seen positioned to the left of the standard browser in the full display. The format of the suggestion box is identical to the class windows and allows the application of the same set of operators.

3.2.2 Locating an Item to Display Text

In the following specific example of active browsing, we suppose the user requires a means of managing and displaying text: this is the "requirements" in figure 1. 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 body of classes used for general display purposes. The example demonstrates how information collected earlier in the search can be used to break the deadlock produced by the latest query and thus help a user find relevant classes faster.

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. As we see this initial goal is not correct and will change as search progresses. 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 applies the operator that expands this class, producing an alphabetically ordered list of the names of methods it implements. The user then investigates two interesting methods in the class. 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. The results of these actions are shown in figure 2 in the bottom half of the lower left window.

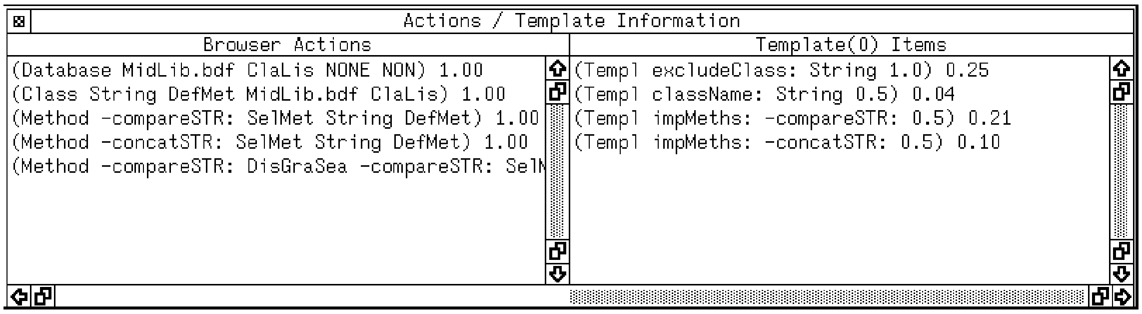

Figure 3: Inferences After Exploring Class “String”

The right window in figure 3 shows the four inferences the active browsing system has made from the sequence of actions so far. This window is used for debugging and demonstration purposes and would not be seen by the user in normal operation. The exact meaning of the “templates predicates” in this window is discussed in section 5, but it can be summarized briefly as follows. 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”. The number at the end of each line is the weight, or importance, associated with the property described on that line. 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 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 is will be similar to "Layer" but with methods specific to strings.

Returning to the original list of class names, the user selects Layer and expands it to show its methods. This list is in the upper half of the lower right portion of figure 2, just above the corresponding list for class String. 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 it to displays of other information. The user selects and expands these methods then marks them, and requests a list of all classes that implement them, or similar methods. This list, the potentially interesting classes of figure 1, is shown in the upper right hand corner of figure 2. The score associated with each class indicates the degree to which it implements both of the methods.

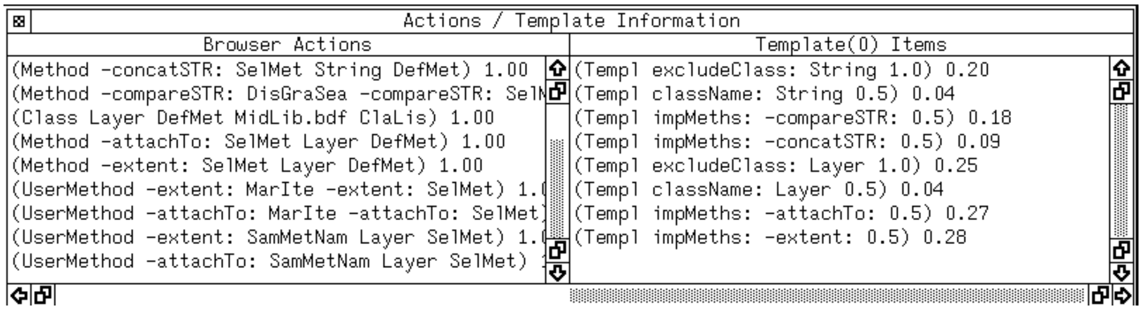

Figure 4: Inferences After Exploring Classes "String" and "Layer"

The right window in figure 4 shows all the inferences the active browser has made at this point. The first four lines are the same as in figure 3. except that the weights of these properties have been reduced to reflect the fact that 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" 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 applying the operator “Implemented In Classes” (top right of figure 2) have identical scores. The methods selected, or similar ones, are extensively used in Layer-type classes, so the user’s operation is a poor discriminator.

The active browser discriminates better. To match maximally with the template shown in figure 4, 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.

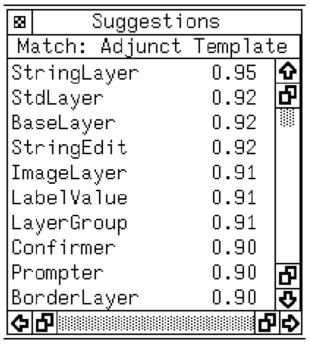

Figure 5: Suggestions

The suggestion box (Figure 5) shows “StringLayer” as a clear favourite, 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 focused the search on layers with various ways of displaying strings.

In summary, the user first looked at the class “String”, hoping it would also include ways of displaying itself as text in a window. Unfortunately it had no such methods. The user then looked at the "layer" classes which were known to be used to display things. There are many different types of layer so the browsing operations selected just brought up a long arbitrarily ordered list of classes. The active browsing system, by using the information collected when the user looked at class "String", was able to order that list in a more effective fashion.

This example illustrates the extraction of a good approximation to the user's search goal automatically from the user's actions. With this, the system is able to search the library independently (i.e. without any input from the user) and identify the classes of interest to the user. Each of the inferences made in the example is based on a general inference strategy discussed in the following section.

4 Strategies for Inferring the User's Goal

The fundamental inference strategy is based on the assumption that the user will only 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. If further information about the method were also visible to the user, such as the types of its inputs and outputs, then these properties would be also recorded. 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 when the user does find something of interest it will be used as a search index. We encourage the user to express this interest directly by making available useful browsing actions, as discussed in the next section. The system is also less sensitive to "misleading" actions by exploiting flexible matching (Sec 5.2.1) and by decreasing the importance of old information (Sec 4.2).

We expect as search proceeds, those properties that are closely related to the search goal will be continually re-asserted, whereas accidental properties of the items visited will not. Therefore, by weighting properties according to the frequency of appearance a progressively more accurate estimate of the search goal will emerge as search proceeds.

For this basic inference strategy to be successful, two conditions must be guaranteed:

1. the visible properties must indeed explain why the user examined an item, and
2. the user's search goal must not change too quickly

These conditions and techniques for establishing them are discussed in the following sections.

4.1 Ensuring That Actions have Unambiguous Explanations

There are two difficulties under this general heading. An action may be ambiguous, in the sense that the system can formulate a great many equally plausible explanations for why the action was taken, or an action may be inexplicable, in the sense that the system is unable to formulate any plausible explanation for the action.

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 stages, where 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 need to be a tradeoff between the benefits of goal inference against the losses due to more complex browsing. Furthermore, new actions must be highly useful in normal browsing so that they will be preferred by the user over the original, more ambiguous actions. 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 all the method's code and documentation, it also displayed information about whatever methods happened to be adjacent to the selected method in the file. This action thus produced a large amount of information, much of which was irrelevant or more detailed than was needed for most browsing purposes. With this manner of displaying 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.

Inexplicable actions arise when the browsing system permits the user to freely "jump" to an arbitrary item in the library. For example, in the present browsing system, the user can expand any item by selecting it from the list of all class names in the library. Such an action does not relate the selected one to any item previously selected, nor does it provide any property about the selected item other than its name. Although names by themselves do sometimes explain why an action was taken (the user investigated the class named "String" because he expected it to be about strings), most often the true reason for "jumping" to a class are properties of the class other than its name that are known or inferred by the user. To reduce the use of these inexplicable operations, the browsing system must provide as an alternative an equally powerful suite of operations that are individually explicable, unambiguous, and convenient to use. In this implementation the repertoire of operators applicable to classes has been extended. One new operator returns a list of classes with similar names, another returns classes with a similar set of methods. Such operators encourage the user to move between classes in an explicable manner, which facilitates the inference process.

4.2 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 relevancy measure 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 relevancy measure. 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.

5 The Active Browser Implementation

This section discusses the specific implementation of an active browser that produced the example given in the preceding section. Figure 6 shows the system architecture. It consists of two main parts. The top section represents the standard browser. This allows the user to display items related in specific ways to a selected item and thus navigate through the library. The lower section is the active component of the browser, called the independent search mechanism. Its input is the sequence of user browsing actions and its output is the Suggestion Box, that is, a list of classes in descending order of their expected relevance to the user's search goal. Classes whose relevance score is below that of "Object", the root of the inheritance tree and therefore the most general class, are not included in the Suggestion Box.

The independent search mechanism can be divided into two parts, an inference system and a template matcher. The inference system infers an analogue of the user's search goal from the user's browsing actions and from this analogue it infers a relevancy measure, in the form of a template. The template matcher computes the degree of match between a given template and an individual class in the library. The following sections describe the inference system and template matcher in detail.

5.1 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].

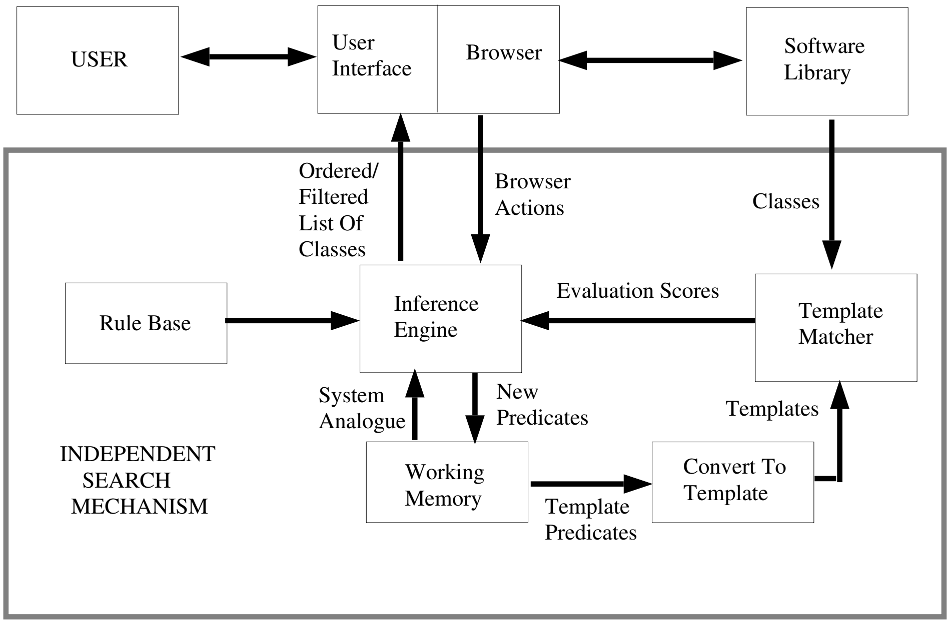

Figure 6: System Architecture.

- a working memory containing facts representing the user's browsing actions and the current analogue and template.
- a rule base containing rules and meta-rules. The rules represent the system's knowledge of how to infer the user's intent from his actions. The meta-rules control when template matching and other tasks are executed.

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.

5.1.1 Working Memory

The facts in working memory represent the current browsing action and the current state of the analogue and template inferred from previous actions. All facts are of the form

(Predicate Arg$_1$....Arg$_n$) Confidence

The confidence is a number 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 by facts of the form:

(Type, Element, Operator, Prior Element, Prior Operator) Confidence

An action involves selecting a particular element from a list and applying an operator to the element. **Type** is a generic description of the element e.g. a class or a method. The list of elements from which the selection was made is the result of some previous browsing action (not necessarily the most recent one). This action is represented by **Prior Element** and **Prior Operator**. By including these in the fact representing the current action, it is possible for the system to 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.

A simple case occurs when the user opens a class and inspects a number of its methods. The **Prior Element** is a class name, **Type** is method and each **Element** is the name of an inspected method. In the example given earlier (figure 3) there are the actions

(Method -compareSTR: SelMet String DefMet) 1.00

(Method -concatSTR: SelMet String DefMet) 1.00

These are both expansions of methods in class "String". This is indicated by the prior element being "String" which was expanded using the "Defined Methods" operator (DefMet). The "Select Method" operator was then used to show the input/output argument types of the two methods. Based on these actions the system increases its confidence that the user is interested in classes with similar names to "String".

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. The predicates corresponding in the analogue are represented by facts of the form

(interestedIn, Type, Element) Confidence.

Finally, the terms of the relevancy measure are represented in working memory by the collection of facts involving the predicate "Templ" (for "template"). These are usually of the form:

(Templ, Matching Procedure, Element, Scale Factor) Confidence

The arguments in this fact specify: what features should be matched against those of library items (Element), how matching should be performed (Matching Procedure) and the importance of this term relative to others in the template (Scale Factor).

A second form of template predicate has an extra argument. This defines the relationship between the original element and a second one. Thus, for instance, the fact that one method uses another can be specified. The relationship as well as the original element will be used in the matching process. Matching the template to library items will be discussed in greater detail in section 5.2.

5.1.2 Rule Syntax and Meaning

The rules and meta-rules have the same 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, possibly prefixed by "NOT". The confidence factor is an number between 0 and 1 indicating the strength with which the consequent may be concluded from the antecedents. The confidence factor is hard coded with the rule and represents the authors' belief in the value of the rule.

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.

The system only uses positive certainties but literals do allow for the provision of "NOT". For the antecedent this condition is met if the fact does not exist in working memory and is assigned a certainty of one. In the consequent a "NOT" causes the fact to be removed independent of its certainty.

5.1.3 Rules Implementing the Inference Strategies

Figure 7 shows some of the rules that deal 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. These inferences are not part of the same rule to allow different confidence factors to be used.

```
1)   IF ((Class ?name ?op ?pnode ?pop))
     THEN ((interestedIn Class ?name)
           (Templ excludeClass: ?name 1.0)) 0.1
2)   IF ((Class ?name ?op ?pnode ?pop))
     THEN ((Templ className: ?name 0.5)) 0.02
3)   IF ((Method ?name ?op ?pnode ?pop)
         (NOT interestedIn Method ?pnode))
     THEN ((interestedIn Class ?pnode)) 0.05
4)   IF ((Method ?name ?op ?pnode ?pop)
         (NOT interestedIn Method ?pnode))
     THEN ((Templ className: ?pnode 0.5)) 0.01
5)   IF ((Method ?name ?op ?pnode ?pop))
     THEN ((interestedIn Method ?name)) 0.1
6)   IF ((Method ?name ?op ?pnode ?pop))
     THEN ((Templ impMeths: ?name 0.5)) 0.1
7)   IF ((UserMethod ?name SamMetNam ?pnode ?pop))
     THEN ((tried impMeths: ?name ?pnode)
           (Templ impMeths: ?name 0.5)) 0.2
```

Figure 7: Some of the Rules Implementing the Goal-Inference Strategy

The first literal in the consequent of Rule 1 (Figure 7) denotes that the user is "interested in" the class to which the operator was applied. The second literal means that as the user has already seen this class, there is little value in the system suggesting it. As the confidence is not one, whether or not it is actually shown to the user will depend on the matching score for the rest of the template. The matching score is reduced by a factor inversely related to the confidence value. Thus if a class strongly matches the template and the user had only visited it briefly it will appear in the suggestion box.

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

Rules 5,6 are similar to the class rules but for methods. They add terms to the analogue to reflect the user's interest and to the template for matching. Rule 7 is fired when the action is of a particular type indicated by the predicate "UserMethod". Such an action occurs when the user applies an operator that returns a list of classes that match on this and possibly other methods. The confidence that the template should include a term containing the method name is greater than in the previous rule because the user has shown very specific interest in classes implementing this method. Of course this may only be a transient interest so the confidence is not increased too strongly.

5.2 Template Construction and Matching

All template predicates are translated into terms which collectively form the template. Each term consists of the element to be matched and the procedure for carrying out the match. A weight is also associated with each term, formed from the product of the confidence value and the scale factor. The former, as described above, represents how strongly the system believes that this is a desirable feature in any retrieved class. The latter, hard coded in a rule, represents domain knowledge about the importance of this type of term relative to other types in the template. For example the name of an instance variable may be considered of lower importance than the name of a method as only the latter is visible outside the object. In other words the library designers will have been more careful in the choice of method names than those of instance variables and therefore the former are better discriminants. Variability in scale factors has not, to date, been used extensively.

The template is matched to a class term by term. The results for all terms are summed and then divided by the total 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.

The following describes two parts of the matching process: firstly how two individual terms are compared to produce a numerical degree of match, secondly how the match score is adjusted for "graphical distance".

5.2.1 Basic Matching

Each element has essentially two parts: a name and associated types. The names are normally constructed from the 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 types are the base types of C, associated with, for example, methods' arguments and instance variables. This information is limited, however, as Objective-C is not a strongly typed language and most objects are of the generic type "id". Still, it can be used as part of the matching process.

The matching score is computed by counting the number of 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. When matching names of unequal numbers of subterms, the match can start from either the first or last word. This allows two simple grammatical forms of matching based on noun and verb phrases. A match on the noun or verb, itself, contributes the largest proportion to a term's score. 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 the first word; a compound name is assumed to be a verb followed by adverbs.

The type information is also taken into account when appropriate, but its importance is dependent on the element being matched. With instance variable, for example, more emphasis is given to the type than for the arguments of method names.

5.2.2 Adjusting Score for Graphical Distance

There are cases where there is no match, or a weak match, in the class itself but a related class has a feature that strongly matches. In this situation a simple form of graphical matching is used.

For instance, due to the inheritance structure of the library the feature may not be defined locally in a class but in one of its ancestors. The matching score of an ancestor, measured as described in the previous subsection, is used but reduced by a small factor. Another example occurs when one class uses the methods of another. Here the match value is degraded with graphical distance. 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. To produce a score for each term, the maximum match value is used and multiplied by the weight associated with the term. In general, the matching algorithm returns a score inversely related to graphical distance.

This two-step matching process – basic matching of a class followed by an adjustment based on the class's neighbours' basic matching scores – is very similar to spreading activation [25, 28]. In spreading activation, the library is represented by nodes connected together by weighted links. Some of the nodes are the library items, themselves, others might represent, for instance, common terms. When a given set of nodes are "activated", this activation is passed through the links to the neighbouring 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 neighbours. If this sum exceeds a threshold, the node passes its activation to its neighbours; thus the activation spreads through the library. Ideally, the spreading process would continue until the activation levels of all nodes had stabilized. However, convergence is not guaranteed, so in practice the process is simply stopped after a fixed, small number of iterations. Upon stopping, the activation levels are interpreted as the relevance of the individual library items.

The basic matching step is the exact equivalent of the initial activation of the term-nodes and the spreading of this activation to those nodes' neighbours. The adjustment step is the counterpart of a few cycles of spreading the activation along the links in the library. This flexible matching goes some way towards addressing the vocabulary problem.

In [20] the authors demonstrate the large amount of inconsistency that occurs when people assign names to items. By basing our system around a browser which encourages the user to express queries directly in the language of the library and by using a flexible matching scheme to find items that have related but not identical indices we hope to reduce this problem. At this time we use only the links that are already present in the library. To include additional information, such as a thesaurus, it would only be necessary to convert it to nodes and links, the same form as the library.

5.2.3 An Example Of Matching

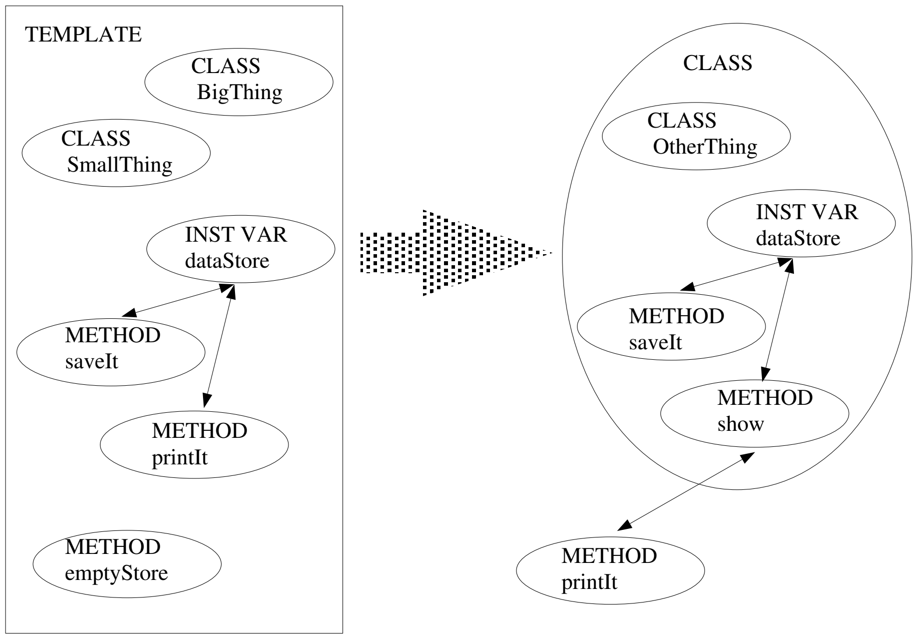

Figure 8: Matching Templates To Classes

The matching of a class to a template is illustrated in Figure 8. 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 "saveIt". It also contains a term relating method "printIt" and "saveIt" to the instance variable "dataStore". On the right is shown a class that would return a high matching score. Firstly the class name "OtherThing" matches with both the template class names due to the common subterm "Thing". Secondly the method "saveIt" is implemented in the new class and has the same relationship to an instance variable with the same name "dataStore". Lastly although "printIt" is not implemented in the class, the method "show" uses a method called "printIt" as part of its processing. In this case, the matching score is half what it would have been had the class actually implmented this method.

6 Design of Experiments to Evaluate the Active Browser

The experiments reported in this paper focus on the inference capabilities of the active browsing system. The main question addressed is, how often is the user's search goal correctly inferred from his normal browsing actions? Some additional experimentation also investigated the effect on search time when the suggestion box is used.

Two experiments are presented. The first is a large-scale experiment in which every one of the 189 classes in the library is used as a search goal. As it was infeasible to conduct so large an experiment with human subjects, an automated heuristic browsing agent was developed and used. A similar experimental method is used in[22] to compare relevance feedback systems. The advantage of using an automated browsing agent is the generally accepted idea [30] that by using "artificial data" (which is what the heuristic browsing agent provides), one can run experiments that are larger-scale, much more carefully controlled, and repeatable, than experiments using "real" data (human browsing agents, in this case). To validate the results of this experiment, a second experiment using human subjects was undertaken.

6.1 The Heuristic Browsing Agent

The heuristic browsing agent has not been designed to simulate the rich behavioural characteristics of a human searcher. Rather it encompasses some general heuristics that a human might be expected to follow. In fact the search strategy it follows is very similar to that demonstrated in the example discussed earlier in section 2.2. The agent consists of two parts (Figure 9): a "fuzzy oracle" that represents the search goal, and a heuristic search strategy that consults the oracle and selects browsing actions. The fuzzy oracle contains a target class selected by the experimenter from amongst the classes in the library. The oracle gives YES/NO answers to questions about whether a given library item matches the target class in certain ways. The oracle is "fuzzy" because its answers are not always correct; for each type of question, the experimenter can set the probability that the oracle will give an incorrect response. This noisiness represents the browsing agent's uncertainty in evaluating the degree of match between a library item and the requirements. Each question reflects what a human user would see in the latest window of the browser. For instance, seeing a new list of class names the user must asses whether or not the name, itself, is sufficiently interesting to warrant opening the class for deeper inspection.

6.1.1 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 wholly or 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 answer to each question is affected by a noise factor. The noise factor is a uniform random variable whose mean and variance is selectable independently for each question. If the degree of match, determined in response to any of these questions, exceeds the value of the random variable, a yes is returned by the oracle; otherwise a no is returned.

The answers to these questions have some intuitive relationship to the knowledge of a user. A positive response for the matching of a class name is biased towards names that have common subterms and in a weaker sense to the children of classes that match. The matching on method names is biased towards identical names and in a weaker sense to names with common subterms and to inherited methods. The answers to these questions could be construed as the semantic knowledge the user has about the meaning of the names and in a weaker sense to knowledge about the library structure. The assessment that a new class is better than a previous one could be related to the user's general knowledge about the items in the library and the role(s) they can play.

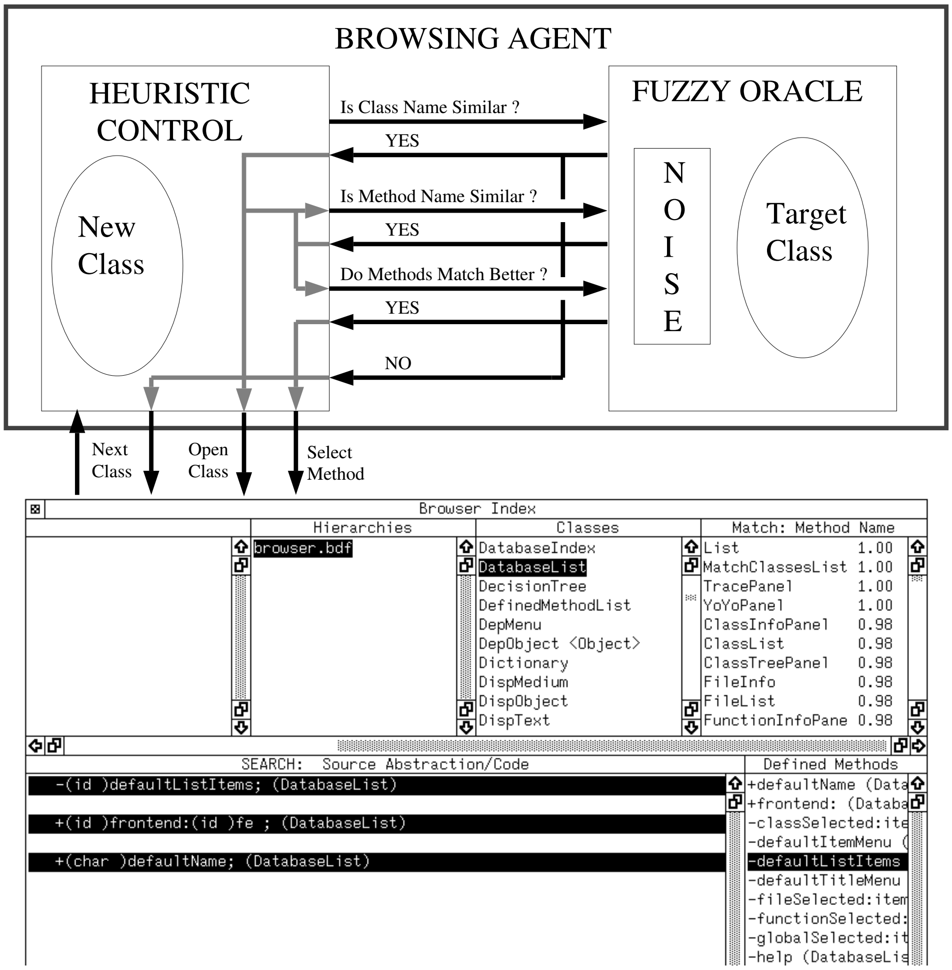

Figure 9: The Heuristic Browsing Agent

6.1.2 The Heuristic Search Strategy

The heuristic search strategy is a combination of depth-first search and hill-climbing. The agent 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 agent 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 the example shown in the bottom half of Figure 9, the agent 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 agent 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 selected and expanded. In the example this results, in figure 9, in the names of the methods implemented by 'DatabaseList" being displayed on the lower right hand side of the screen. The heuristic agent 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 marked, causing some information about it to be displayed in the lower left window. This random examination process continues until a maximum number of methods have been marked (a random number from three to five) or there are no more methods in the class. In the example, three methods have been marked: "defaultListItems:", "frontend:" and "defaultName:".

The heuristic agent now asks if the set of marked methods is a better match to the methods in the target class than the set of methods marked in the previous class. If the answer is yes, as it is in the example, these methods are selected and highlighted in the window on the lower left (Figure 9). The heuristic agent applies an operator that generates a list of the classes that best match the selected methods. This list appears in the upper right hand corner of the browser's interface (Figure 9). The heuristic agent then continues its search using this newly-generated list of classes. The search terminates when the agent 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 the fact that the noise in the fuzzy oracle is reduced each time a new class is opened, so that as search proceeds the heuristic agent gets progressively more accurate answers to its questions. This is analogous to the refinement of the user's knowledge about what is required, and thus the refinement of the search goal, as more classes are explored.

7 Experimental Results

The main experiment measures how successful the active browsing system is in inferring the target class from the agent's actions. Two measures of "success" are used: the frequency with which the active browsing system identifies the target class prior to its being found by the browsing agent, and a comparison of how the ranking of the target class by the agent and by the system varies across the whole search. As described in section 6, two versions of this experiment were done. The first is a large-scale study using the automated heuristic browsing agent. The second is a small-scale study using human subjects. The results of these experiments are reported in sections 7.1 and 7.2 respectively. In neither version of this experiment is the "Suggestion Box" used by (or even shown to) the browsing agent. Section 7.3 describes an experiment that measures the effect on the agent's search of using the suggestion box. The number of steps required to find the target class is compared to that required without the use of the suggestions.

In all experiments, each time the browsing agent (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 agent's current list (henceforth called the agent'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 10, is recorded for the target class "ProbableMethodsList".

The first row gives the rankings after the browsing agent has made its first step, i.e., created its first new class list. The target class is 20th in this list. The target class is ranked slightly higher (16th) by the active browsing system. As the agent proceeded in its search its ranking of the target class actually got worse, dropping at step 8 to 39th

Target Class ProbableMethodsList

Step Number	System's Ranking	Agent'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 10: Record Of Ranking During Search

place. This is not a surprising effect. The agent may well evaluate a number of different variants on a theme. For instance, different collections of methods might be investigated or methods with similar but not identical names. Thus the search will, at least temporarily, move in the "wrong" direction. Despite this fact, the history of the actions carries enough information for the active browsing system to identify the characteristic features of the target class. This is demonstrated by the fact that from the third step onwards, the target class is in the system's top ten.

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 browsing agents, the search was terminated when either (a) the browsing agent found the target class, or (b) the target class was ranked in the top ten by the active browsing system for six consecutive steps. Criterion (b) applies in the example above.

7.1 Results Using the Automated Heuristic Browsing Agent

The experiment consists of successively using each class in the library as the target class. The library is the combination of two commercially available libraries plus the classes developed as part of this implementation. It contains 189 classes in total. By varying the noisiness of the oracle's answers to questions posed by the heuristic agent 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 active browsing system correctly infers it before it is found by the browsing agent. Likewise, a target class is a "loss" if the browsing agent 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 agent's ranking when the agent finds the target class. A target class is a loss if the agent 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 agent finds the target class and, in that final step, the agent'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 browsing agent 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. This number is over 50% more than the number of losses (the clear wins for the agent). As the noise increases the number of wins

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

declines. The lowest number of wins represents about a third of the total classes.

Two factors are critical in the assessment of wins. Firstly the number of times the target class has to appear in the suggestion box before a win is recorded and secondly the size of the suggestion box itself. To examine the sensitivity of the results in 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 [11]. The results were found not to be highly sensitive to the suggestion box length. There is a sensitivity, not surprisingly, for suggestion boxes containing only a very small number of classes. 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.

7.1.1 Step-by-Step Comparison of the Rankings

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 browsing agent 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 agent's own ranking of the target.

The difference between the agent's ranking and the system's ranking is plotted in Figure 11 for two different values of the noise parameter. The results for low noise are plotted with the solid line and the results for moderate noise are plotted with the dotted line. Each step in the agent's search for the target class corresponds to a different point on the

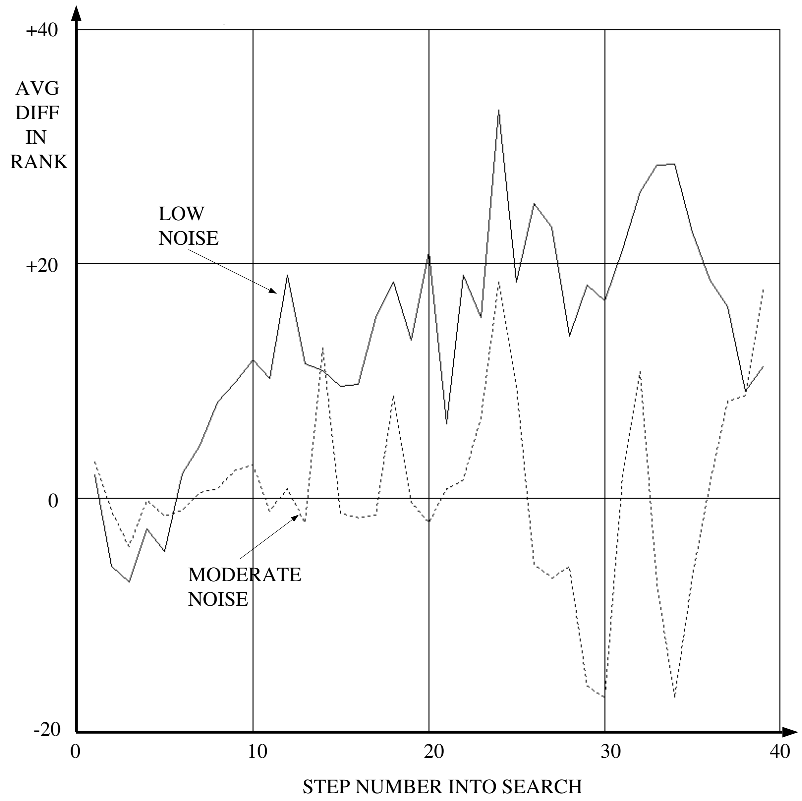

Figure 11: Difference In Ranking During Search

X-axis. The Y-axis indicates the difference between the agent's and the system's ranking on a given step, averaged over all the target classes. A positive Y-value indicates that the target class is ranked higher by the active browsing system than by the agent.

Note that the search for some target classes involves fewer steps than the search for others. For example, 75 target classes have a search involving 4 or fewer steps. Thus, as one moves right along the X-axis, the number of classes contributing to the Y-value decreases rapidly; the data for large values of X are based on the small number of target classes that require many steps to find.

The graph for low noise shows that the target class is consistently ranked much higher by the active browsing system than by the browsing agent 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 browsing agent having a better ranking than the active browsing system.

Instead of measuring the numerical difference in rankings, one could simply measure the sign of the difference in rankings at each step [11]. This indicates, for each step, how frequently the target class is ranked higher by the active browsing system than by the browsing system. A similar pattern emerges. For low to moderate noise levels, after the first few steps of search, roughly 70% of the target classes are ranked higher by the active browsing system. Unlike the numerical rank difference, this measure declines quite slowly as the noise level increases; for the high noise level, the system's ranking is higher than the agent's on 45% of the target classes.

7.2 Experiments with Human Subjects

This experiment is identical to the preceding one except that human subjects were used, not an automated browsing agent, 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 agent carry over to human agents.

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 minimises 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 12. The order with which each class was presented to the the user is as follows:

StdLayer: the "window" with just a title displayed

StdSysLayer: similar to StdLayer but with scrollbars.

Confirmer: which asks yes/no style questions of the user.

SpecTLayer: which displays a text string and can be "attached" to other layers. This target was not in the library used in the experiment with the automated browsing agent.

EchoBox: produces a "rubber rectangle" whose size is determined by the user moving the mouse.

The experimental subjects were all graduate students, 3 from the Electric Engineering Department, two from the Computer science Department. All had some experience with Object-Oriented languages but none had more than a cursory knowledge of Objective-C. Each subject was give some initial instruction on how to use the browser. The experiment was automated such that each example was presented in turn to the subject. The search was stopped when the particular target class was opened to explore its methods and a new example presented.

Table 2 shows the lengths of the searches required by each subject (H1–H5) on each target. The most striking feature of this table is its diversity. Each target was found quickly

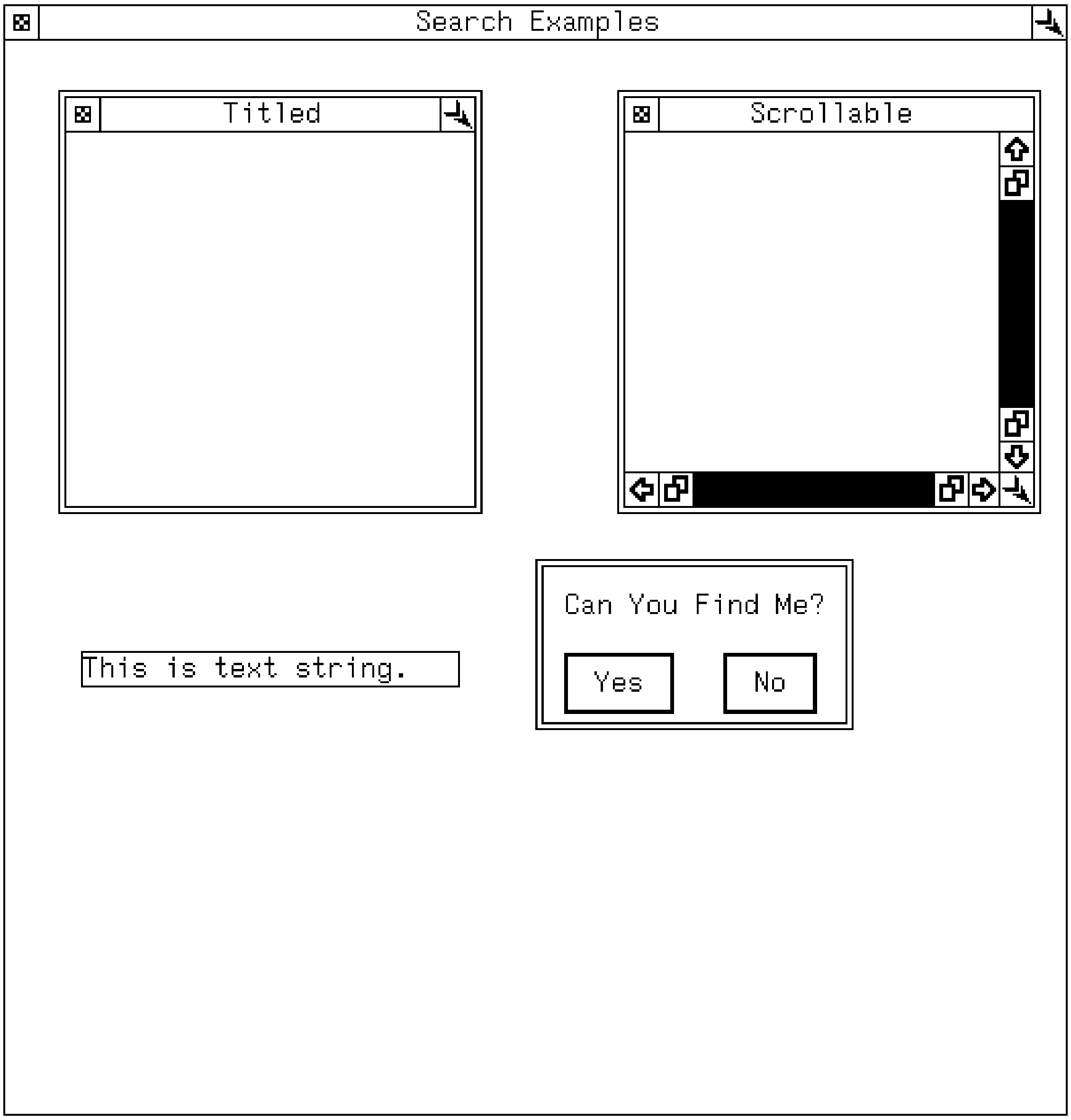

Figure 12: Four of the examples used in the experiments with human subjects

Target	H1	H2	H3	H4	H5	Automated Agent
StdLayer	66	43	5	10	6	6
StdSysLayer	1	12	12	2	3	17
Confirmer	30	3	1	2	21	8
SpecTLayer	5	3	n/a	6	14	n/a
EchoBox	39	5	n/a	4	7	7

Table 2: Lengths of searches for all Browsing Agents

(5 steps or fewer) by at least one subject, but not by other subjects. For some targets the range of variation is very high. There is no apparent correlation between the results for different subjects. Every subject was the fastest on some target and the slowest on some target (except H4 was never slowest and H5 was never fastest). This diversity and lack of correlation confirms our observations during the experiment that the subjects were using very different search techniques.

The results for the automated browsing agent are shown in the rightmost column; they are indistinguishable from those of the human subjects except for the absence of any extremely short search. Note, however, that on other targets the automated agent did have extremely short searches: recall that 40% (75 out of 189) of its searches were length four or less. This percentage is same as for the human subjects (8 of the 23 searches are length four or less).

Crtn. No.	Result		
	Wins	Losses	Draws
1	13	2	8
2	6	7	10

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

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 40% of the searches five steps or longer. Criterion (1) credits active browsing with wins in 13 (56%) of the searches. However upon closer examination it became apparent that criterion (1) is being overly generous; in our subjective opinion three of these wins should be considered losses and one a draw. By this reckoning, active browsing wins on 9 of the 23 searches (40%). The results are very similar to the low noise results for the automated browsing agent, where 40-44% of the searches were wins for active browsing.

It is important to note that the clear wins for active browsing – as defined by criterion (2) – were evenly distributed across the users, the targets (except that Confirmer was never a clear win), and the lengths of the searches. StdLayer was a win with subjects H1 and H2; StdSysLayer was a win with H3; SpecTLayer was a win with H4 and H5; and EchoBox was a win with H5. The fact that active browsing succeeded equally for all the users indicates that it is able to cope with a wide variety of search techniques.

7.3 Selecting from the Suggestion Box

One extension to the experiment, more akin to how the tool would be used in practice, is to have the agent make selections from the suggestion box. Here, the agent 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 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 agent 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.

7.4 Discussion And Analysis Of Results

The results show that the active browsing system frequently (40% of the time) infers the target goal before it is found by the browsing agent. This success rate was consistently

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

Table 4: Using the suggestion box

achieved across all the agents studied, five humans and the automated heuristic borwsing agent. It was also shown that during search the active browsing system consistently ranks the target class significantly higher than the heuristic browsing agent. There are two conditions under which active browsing did not outperform the heuristic browsing agent: during the early steps of search, and when the browsing agent is highly uncertain about the search goal (simulated in the experiments by a high noise level).

That active browsing does not outperform the agent 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 agent'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 agent's behaviour consisted of a large number of uninformative actions followed by a relatively small number of highly informative actions that led the agent to the target. If the agent behaves in this way, the active browsing system will lag behind the agent 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 behaviour by the agent is promoted by the fact the oracle's noise level decreases as the agent'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 agent may explore one of these groups for a long time before "realising" that such classes do not truly meet the requirements. There are a number of possible answers to this problem.

The first relates to the goal tracking issue discussed earlier in section 4.2. One possible way to make the goal tracking more robust would be to have the system detect if the user abandons an originally interesting path. For example, this might be inferred if the user backtracks to the original start point and begins a search with items showing little commonality with those visited previously. More generally, it could be detected by a change in the ability of the analogue to reasonably predict the user's selections. The previous information collected would then be degraded in inverse proportion to the confidence in this inference. Techniques similar to this have been used in machine learning to cope with "drifting" concepts [39]. A second possible approach would be to add rules that make use of various forms of statistics about library structure. If the user is in an area of the library where all adjacent classes are very similar then the confidence in certain inferences should be weakened due to the high likelihood of particular selections.

Another possible solution would affect the inference process itself. The only basic assumption is that the properties selected are likely to be similar to the desired ones. It is not known if they are exactly the same, although when first selected this might be preferred. Thus when properties are continually selected within these groups of highly similar classes, although the confidence that they contain a useful features is increased, the confidence that the exact property is useful should stay the same. Ideally the analogue should be able to represent this distinction.

8 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 was 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. In the particular implementation discussed in this paper the target was a class. 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 solution depends on 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 with a bias in favour of the smaller ones. This situation arises in software within the object-oriented paradigm, where each class is an individual item and may also be a part of a "framework". It should also be relevant for modules consisting of a set of closely related procedures. If the library has many levels of granularity ranging say from classes to complete programs, it would be necessary to infer not only what properties of an items are important but also what level of item interests the user. One approach would be to change the browser to have layers, or views, associated with each level of granularity. Different analogues could then be associated with the particular level or view. Results derived from them would only be shown when that level or view is active.

An alternative is to allow the user to make the decision, presumably by finding one item that meets part of the requirements and then searching for others that meet the rest. In this case the analogue may be considered to represent a single search goal. Multiple relevancy measures must, however, be produced and activated at different points of the user's search. In the standard browser there is an operator that allows the user to save the information on a class to a separate window. One idea that was pursued briefly was to infer from the use of this operator that the user had met some the requirements. The system would then split the template into two; the part that matched with the stored class and the part that did not. The second part was used to scan the library for classes that might be used in conjunction with the one stored to meet the requirements. Of course this might not be the only reason the user stored information about a class. So an additional safeguard was added, checking how well the second template matched items the user visited from this point on. If it was poor, it was assumed the user was continuing with the original search and the two templates were merged.

Even if the user's target is a single item, the library might be heterogeneous and the user could be looking for one of many types of item. Ambiguity arises as to the form of the target item. One potential answer is to construct a number of analogues based on the different item types. These would represent hypotheses that the system has about the search target. The active browser would then need to be establish which one was relevant. The browser might have levels, or views determined by appropriate filters, associated with one specific type. Analogues could then be associated with the particular level or view. Results derived from them would only be shown when that level or view is active.

Another problem occurs if the user sidetracks to look at items that are of interest but not connected to the present task. The user may, while looking a particular class, chance upon another class related to some previous interest. If some time is spent looking at this and related classes, before returning to the original search, the inference process will become dangerously skewed. It might be possible to detect this situation if the intermediate search is orthogonal to the main one. This might become apparent when the match between the relevancy measure and the classes the user is exploring deteriorates abruptly. Information collected during this part of the search would be disregarded and information before the sidetrack would be used. Further study is needed to see if this is practical, or would just serve to weaken the inference process in other circumstances.

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. The results of each match are cached. This allows the more rapid reevaluation of previous terms when the associated weight is changed. 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. 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, on a Sparc 2.

Certainly 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. The general idea underlying these techniques is to be highly selective about which classes are evaluated and which template terms are (re-)evaluated.

9 Conclusion

This paper has described a system for active browsing and experimentally demonstrated that a browsing agent's search 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 agent's actual search goal. Active browsing increases the effectiveness of browsing without imposing any restrictions or "cost" on the user. The agent 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 browsing agents. Further work on agents using the systems' suggestions would permit measurement of the relative effectiveness of different ways of presenting the inferred goals to the browsing agent.

10 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. *Expert Systems: Concepts And Examples.* National Computing Centre Publications (1984)

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

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

[4] Edwin Bos. *Some Virtues and Limitations Of Action Inferring Interfaces.* 5th Annual Symposium on User Interface Software and Technology (1992)

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

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

[7] Robin Cohen, B. Spencer. *Specifying and Updating Plan Libraries for Plan Recognition Tasks.* Proceedings of the Conference on Artificial Intelligence Applications (CAIA-93) 1993 pp 27–33 (1993)

[8] Allen Cypher. *EAGER: Programming Repetitive Tasks by Example.* Proceedings of the SIGCHI Conference 1991 pp 33–39 (1991)

[9] Lisa Dent, Jesus Boicario,John McDermott,Tom Mitchell and David Zabowski *A Personal Learning Apprentice* Proceedings of AAAI 92 (1992)

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

[11] Chris Drummond, Automatic Goal Extraction from User Actions to Accelerate the Browsing of Software Libraries, M.A.Sc. Thesis, University of Ottawa, December 1992.

[12] T. W. Finin. *Providing Help and Advice in Task Oriented Systems.* Proceedings of IJCAI-83 1983 pp 176–178 (1983)

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

[14] Gerhard Fischer, A. Girgensohn. *End-User Modifiability in Design Environments.* Proceedings of CHI-90 ('Empowering People") 1990 pp 183–191 (1990)

[15] Gerhard Fischer, A. C. Lemke, T. Mastaglio, A. I. Morch. *Using Critics to Empower users.* Proceedings of CHI-90 ('Empowering People") 1990 pp 337–347 (1990)

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

[17] William B. Frakes, P. Gandel. *Representing Reusable Software.* Information and Software Technology vol. 32(10) 1990 pp 47–54 (1990)

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

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

[20] G.W. Furnas, T.K. Landauer, L.M. Gomez and S.T. Dumais. *The Vocabulary Problem In Human-System Communication.* Communications of The ACM. Nov 1987 Vol 30 No 11 pp 964–971 (1987)

[21] B. A. Goodman, Diane J. Litman. *Plan Recognition for Intelligent Interfaces.* Proceedings of the Sixth Conference on Artificial Intelligence Applications (CAIA-90) 1990 pp 297–303 (1990)

[22] David Hanies, W. Bruce Croft. *Relevance Feedback and Inference Networks.* Proceedings of the 16th International ACM SIGIR Conference on Research and Development in Information Retrieval 1993 pp 2–11 (1993)

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

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

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

[26] L.A. Hermens, J.C. Schlimmer. *A Machine-Learning Apprentice for the Completion of Repetitive Forms.* Proceedings of the Ninth Conference on Artificial Intelligence for Applications (CAIA-90) 1993 pp 164–170 (1993)

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

[28] Andrew Jennings, Hideyuki Higuchi. *A User Model Neural Network for a Personal News Service.* User Modeling and User-Adapted Interaction vol. 3 1993 pp 1–25 (1993)

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

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

[31] Charles W. Krueger. *Software Reuse.* ACM Computing Surveys vol. 24(2) 1992 pp 131–184 (1992)

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

[33] Pattie Maes and Robyn Kozierok *Learning Interface agents* Proceedings of AAAI (1993)

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

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

[36] Beerud Sheth and Pattie Maes *Evolving Agents For Personalized Information Filtering* Proceeding of the 9th Conference on Artificial Intelligence For Applications (1993)

[37] Barry G. Silverman and T.M. Mazher. *Expert Critics in Engineering Design: Lessons Learned and Research Needs.* AI Magazine, vol. 13, no. 1, pp 45–62 (1992)

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

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

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

[41] Ian H. Witten, Dan Mo. *TELS: Learning Text Editing Tasks from Examples.* Watch What I Do, edited by Allen Cypher, MIT Press 1993 pp 183–204 (1993)

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

11 List of Footnotes

Manuscript received

* This work was supported in part by the Natural Science and Engineering Research Council of Canada under Grant No. A6684.

† The authors are with Department of Electrical Engineering, University of Ottawa, Ottawa, Ontario, Canada K1N 6N5.

12 List of Figures