Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers

Raphael Hoffmann, James Fogarty, Daniel S. Weld
Computer Science & Engineering
University of Washington
Seattle, WA 98195
{raphaelh, jfogarty, weld}@cs.washington.edu

ABSTRACT
Programmers regularly use search as part of the development process, attempting to identify an appropriate API for a problem, seeking more information about an API, and seeking samples that show how to use an API. However, neither general-purpose search engines nor existing code search engines currently fit their needs, in large part because the information programmers need is distributed across many pages. We present *Assieme*, a Web search interface that effectively supports common programming search tasks by combining information from Web-accessible Java Archive (JAR) files, API documentation, and pages that include explanatory text and sample code. Assieme uses a novel approach to finding and resolving implicit references to Java packages, types, and members within sample code on the Web. In a study of programmers performing searches related to common programming tasks, we show that programmers obtain better solutions, using fewer queries, in the same amount of time spent using a general Web search interface.

ACM Classification
H5.2. Information interfaces and presentation: User Interfaces.

Keywords: Web search interfaces, implicit references

INTRODUCTION AND MOTIVATION
The explosion of information available on the Web and on personal computers has made search a fundamental component of modern user interface software. This has led not only to new approaches to visualizing the results of keyword-based Web search [24], but also applications for quickly finding personal information [6, 9], augmenting highly-structured sites with browser-based search [14], tools to help people collect and summarize information from search sessions [8], and keyword-based approaches to invoking commands in desktop applications [18].

Because a vast number of code libraries and related information are now available on the Web, programmers increasingly use search as a part of their development process. Our analysis of logs from a major search engine show many examples of queries related to Application Programming Interfaces (APIs), including queries attempting to identify an appropriate API for a problem, queries seeking more information about a particular API, and queries seeking samples that use an API. Interviews with developers confirm that Web search engines are the single most important source for this information.

Unfortunately, current Web search interfaces have important shortcomings when used for this purpose. General search engines, such as Google, traditionally generate a flat listing of ranked pages, but developers seeking an API require information dispersed on many pages: tutorials, documentation pages, the API itself (in source or binary format), and pages with code samples which demonstrate usage. It is currently time-consuming to locate the required pieces of information and difficult to get an overview of alternatives. Unless a programmer already has a significant understanding of an API, it is almost impossible to judge the relevance and quality of results or to understand dependencies contained in sample code. Numerous queries and visits to many pages are therefore required. Code-specific search engines have recently been introduced, but these are also unsatisfactory, largely because they ignore documentation, tutorials, and pages containing a mix of code samples with explanatory text. Pages that contain both explanatory text and sample code have generally been intentionally created to illustrate the use of an API, but the raw code returned by existing code-specific search engines lacks context and is frequently incomprehensible.

This paper presents Assieme, a Web search interface for programmers based on a novel approach to combining information currently distributed across many pages. Assieme analyses Web-accessible Java Archive (JAR) files, API documentation, and pages that mix explanatory text with sample code. By finding and resolving implicit references from code samples to Java packages, types, and members, Assieme can combine relevant information from different Web-accessible resources. Assieme therefore provides a coherent search-based interface that allows programmers to quickly examine different APIs that might be appropriate for a problem, obtain more information about a particular API, and see samples of how to use an API. In a study of programmers performing common programming-related search tasks, we show that programmers obtain better solutions, using fewer queries, in the same amount of time spent using a general Web search interface, Google.

UIST'07, October 7–10, 2007, Newport, Rhode Island, USA.

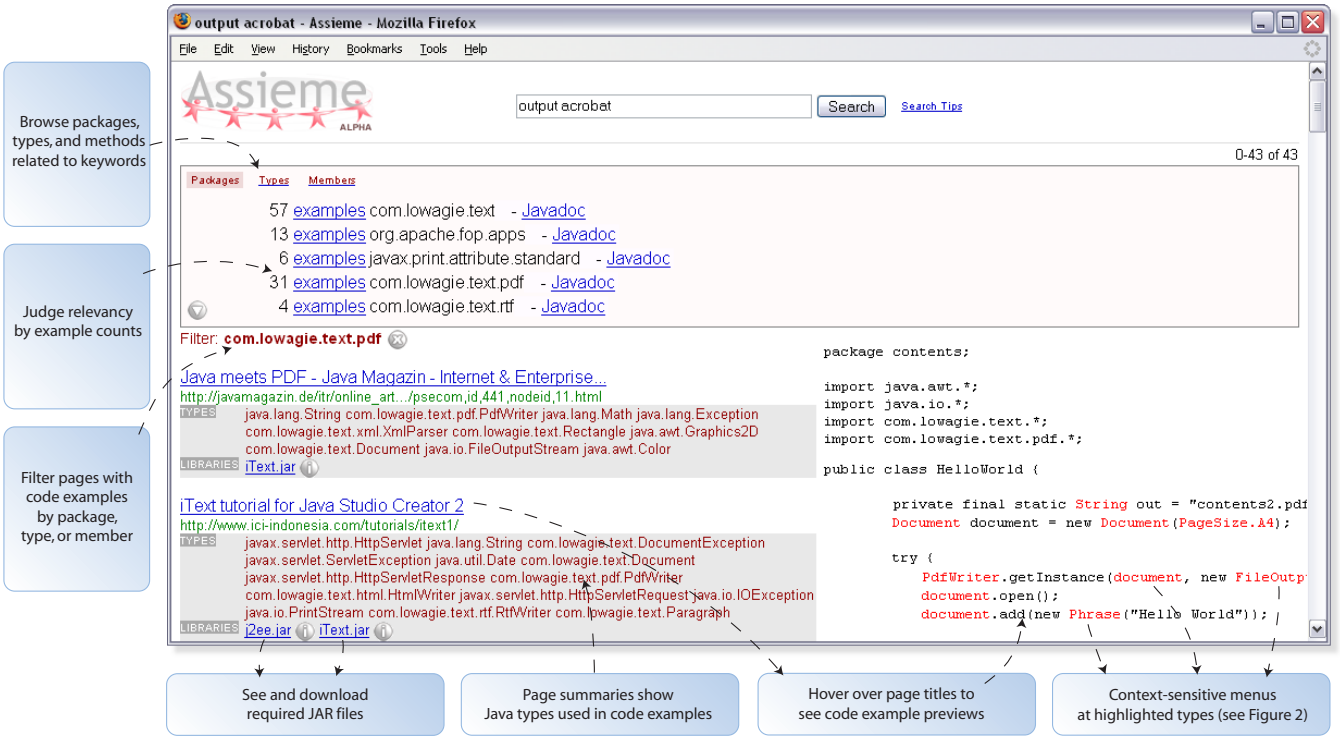


Figure 1: Assieme provides a search-based interface that allows programmers to quickly examine different APIs that might be appropriate for a problem, obtain more information about a particular API, and see samples of how to use an API.

This paper makes four contributions. First, we analyze query logs of a major Web search engine and show that many searches by programmers are related to finding an appropriate API for a problem, finding more information about a particular API, or seeking samples of the use of an API. Second, we present Assieme, a Web search interface for programmers that uses implicit references in sample code to enable a novel interface that better addresses the information needs of programmers. Third, we analyze the key algorithms powering Assieme, reporting the reliability of automatic sample code extraction and reference disambiguation. Finally, we report on a study of programmers performing searches related to common programming tasks, showing that programmers obtain better solutions, using fewer queries, in the same amount of time spent using a general Web search interface.

AN OVERVIEW OF ASSIEME

Figure 1 presents Assieme, with a screen capture taken while a developer seeking to programmatically generate a file in Adobe's PDF format is exploring the results of a query for "output acrobat." The interface contains three major areas.

The shaded bar across the top shows Java packages, types, and members corresponding to the search query. In this example, the programmer has indicated an interest in seeing packages. The fully-qualified names of appropriate packages are shown, ranked by their relevance to the query, as well as how many code samples Assieme has found that demonstrate the use of types from that package. The number of samples that use a package, type, or member can be helpful for determining relevance, as frequently used items may be more robust and better supported than less known options.

Figure 2: A context-sensitive menu reveals the fully qualified names of elements appearing in Java code and provides links to additional information.

Choosing a package, type, or member in this area filters the results in the next area, similar to other interfaces that use faceted search to examine large datasets [31].

In this case, the programmer has filtered the results to focus on the `com.lowagie.text.pdf` package, so the bottom-left portion of the interface shows pages that contain code samples that use classes from that package. The blue links and green URLs here provide the traditional functionality, allowing a programmer to navigate to a page. But instead of the generic text snippet preview provided by general search engines, Assieme presents information that is more likely to match the information needs of programmers. Specifically, Assieme shows what Java types are used in code samples on the page and which libraries contain those types. Assieme also provides a link to download those libraries. When a programmer mouses over a link on the left side, sample code snippets from that page are shown on the right.

Previewing code samples allows programmers to quickly get more information about the relevance of a page. The sample itself may provide the information a programmer

needs, or it might give the programmer more information about whether navigating to the full page is likely to be helpful. Assieme adds further information to the sample code preview by providing context-sensitive menus within the code. These reveal the fully-qualified names of packages, types, and members used in code and provide direct links to additional samples, Javadoc documentation, and JAR files containing implementations.

We defer extensive discussion of Assieme's implementation until the body of this paper, but some aspects warrant attention at this point. Specifically, Assieme searches the Web and combines information from unstructured Web pages with more structured information sources, such as automatically-generated Javadoc pages and compiled Java libraries. This introduces important challenges, most notably the need to find and resolve *implicit references*. In code samples like those in Figures 1 and 2, nothing in the text of the page explicitly indicates that the token `Phrase` is a reference or that it corresponds to the type `com.lowagie.text.Phrase`. In fact, many code samples are stripped of import statements (for the sake of brevity within a Web page where the package in use is obvious to a reader) and will not compile. Assieme infers both the presence of implicit references (identifying code samples in Web pages) and their referents (identifying the fully-qualified types associated with an implicit reference). This inference enables major components of Assieme's interface, including the ability to filter pages according to what types are used in code samples, browsing from sample code to related Javadoc pages, and improved search. For example, our query for "output acrobat" identifies the `com.lowagie.text.pdf` package, though neither keyword is contained in the name of the package, because Assieme uses the text on pages that implicitly reference this package.

The next section discusses related work, positioning Assieme with regard to research on understanding programmer needs, previous systems supporting programmers, and general issues of search in modern user interface software. We then examine a set of programming-related queries submitted to an existing general Web search engine, showing that current search patterns indicate a need for the functionality provided by Assieme. We next present Assieme's architecture and implementation, including discussion of the crawl that currently provides the basis of Assieme, the identification of sample code in Web pages, the resolution of implicit references in sample code, and Assieme's approach to indexing and scoring. This is followed by our evaluation, examining both the reliability of Assieme's internal inference components and conducting a study of programmers completing common programming-related search tasks using Assieme and existing general and code search engines.

RELATED WORK

Ko *et al.* present six learning barriers faced by programmers [17]. *Design* barriers occur when a programmer is unsure what he wants to do, as when a programmer cannot conceive of an appropriate algorithm. *Selection* barriers occur when a programmer knows what he wants to do, but not what to use to do it, as when attempting to identify an appropriate API. *Coordination* barriers occur when a programmer knows what set of things to use to achieve a goal, but not how those things should be combined. *Use* barriers are related, occurring when a programmer knows what to use, but not how to use it. *Understanding* barriers occur after a potential solution has been implemented, when a program does not perform as expected and a programmer is unsure why. Finally, *information* barriers occur when a programmer believes they know why a program did not behave as expected, but do not know how to check that belief.

In the terminology of Ko *et al.*, Assieme is a novel Web search interface intended primarily to address selection, coordination, and use barriers. In addressing selection barriers, Web search provides important advantages because it allows Assieme to find APIs that would not be found if searching only in a local code repository. Beyond this, Assieme's use of implicit references provides additional power for addressing selection barriers because the explanatory text on pages that reference an API can be used in indexing that API. Assieme's discovery and previewing of code examples on the Web are similarly powerful, as code samples on the Web have often been constructed to explicitly illustrate the type of information needed to address selection and coordination barriers.

Cutrell and Guan [5] present an eye-tracking study to examine contextual snippets in search results. Motivated by the idea that typical search result pages may not provide enough information for people to make informed decisions about what pages to visit, they examined the effect of using shorter or longer contextual snippets. For informational queries like those supported by Assieme, they found that performance improved as more information was made available in the contextual snippets. This work is consistent with the results of our study, as the additional information available with Assieme allows programmers to more effectively perform common search tasks.

There has been extensive work on searching within code [11, 21, 23, 25], and more recent work has applied modifications of techniques developed for Web search to code search [1, 15, 22, 26, 27]. Relevant to this work, we use text on Web pages to score code samples and the libraries that are referenced in that code. This idea has some resemblance to anchor text scoring, another Information Retrieval technique that works well for general Web search [10]. The text on a page serves as a description of code samples on that page as well as descriptions of the objects in libraries implicitly referenced in those code samples.

Other work has focused on automatically creating code snippets by mining databases of code or API specifications [13, 20, 29, 30]. Such systems generally provide little assistance with selection barriers, and their output can be more difficult to interpret than code samples that Assieme finds on the Web, which have generally been carefully constructed by a person with the intent of concisely and effectively illustrating the use of an API. Further, the page containing a code sample can also provide important explanations of the sample code.

Prior work has also examined support for browsing among different types of documents within a project. For example, Cubranic *et al.* [4] present a recommender system that links information from CVS repositories, bug-tracking systems, communication channels, and online documentation. Given a keyword query or an artifact such as a Java class, they find related information within the project. While such tools can help programmers find information within a project, including non-code artifacts like CVS comments, they do not provide Assieme's ability to find code samples on the Web.

Assieme is unique in that we have identified shortcomings of current interfaces for programmer Web search, designed a new Web search interface based on providing more appropriate previews and combining the information that programmers currently must visit many pages in order to obtain, and then developed the inference needed to detect the implicit references critical to enabling such an interface. While search and search-related technologies are increasingly fundamental to modern user interface software [6, 8, 9, 14, 18, 24, 31], and while the machine learning community continues to make important advances in extracting information and identifying relationships from unstructured Web content [2, 7, 28], there are few examples of leveraging these new machine learning advances in appropriate interfaces. Assieme provides such an example, while also illustrating the potential of future work to address the limitations of current interfaces by automatically identifying relationships on the Web.

EXAMINING PROGRAMMER WEB QUERIES

To better understand what developers are searching for on the Web, we analyzed the query logs and click-through data from 15 million queries submitted to the MSN search engine, now called Windows Live Search, from May 2006. The data represents a uniformly random sample of all queries in that month and includes timestamp, session, query, as well as click-through data with result ranks and URLs. To mitigate privacy concerns, email addresses and numbers longer than 8 digits were replaced by placeholders.

We extracted all sessions containing at least one query which included the term "java", yielding 2,529 sessions. Manually examining these sessions and formulating regular expression filters, we removed all sessions relating to coffee or the island of Indonesia (356), job qualifications (24), games and mobile phones (218), the Java/byteverify security issue (115), browser settings (49), Java runtime installation (526), and Javascript (185). Another 661 sessions contained only overly general and therefore ambiguous queries (mostly "java" and "java.com") and in 56 sessions the information the searcher was interested in could neither be inferred from the queries nor from click-through data. This left 339 unambiguous sessions on Java programming, which were manually classified into an informal taxonomy of eleven categories. Categorization was determined using query reformulation as well as click-through data. For example, searchers often added terms like "example", "download", or "tutorial" to a query. In other cases, click-through data revealed that users only clicked on Javadoc pages, indicating that they were probably interested in API reference documentation.

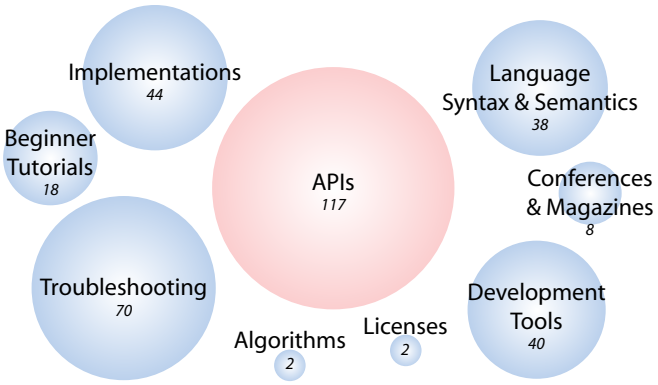


Figure 3: Classification of types of query sessions initiated by developers on the Web. The sizes of the circles correspond to the relative number of searches.

Figure 3 depicts our results, with circle size corresponding to the number of sessions in each category. In this analysis, the APIs category is clearly the largest, containing 117 sessions (34.2%) versus 70 sessions (20.6%) in the next largest category, Troubleshooting.

Looking more closely at these API-related sessions, we found that 64.1% of the sessions contained queries that were merely descriptive but did not contain actual names of APIs, packages, types, or members. These sessions seem to be focused on attempting to identify an API that might be appropriate for a problem, corresponding to Ko *et al.*'s notion of a *selection* barrier. As noted in our discussion of related work, the use of Web search is particularly appropriate for addressing such questions because it provides more information than approaches that search only local files.

The remaining API-related sessions contained API or package names (12.8%), type names (17.9%) or even method names (5.1%). These sessions seem to be focused on obtaining more information about a particular API that a programmer was already aware of, corresponding to Ko *et al.*'s notion of *use* barriers. Many of these queries also contained descriptive keywords and some included a term that indicated exactly what type of document a programmer was looking for, e.g. "javadoc", "tutorial" or "download".

Among all of these API-related sessions, 17.9% contained terms like "example", "using", or "sample code" that suggest a programmer was interested in seeing samples showing how an API is used. These types of queries correspond to Ko *et al.*'s notion of a *coordination* barrier, as the specially-constructed sample code that can be found on many Web pages is usually very effective at showing how different types or methods should be used together.

Interestingly, many sessions contain queries that fall into several categories. For example, consider this four-query session from our data:

```
java JSP current date
java JSP currentdate
java SimpleDateFormat
using currentdate in jsp
```

The first two queries are fairly general and descriptive, as the programmer is not yet targeting information about a particular package, type, or member. By the third query, the programmer has decided that the type `java.text.SimpleDateFormat` is relevant and specifically targets it in a new search. The final search then appears to target code samples or explanations of usage. While current interfaces require this type of query reformulation in order to address programmer information needs, Assieme explicitly supports the transition from determining what API might be relevant for a problem, to learning more about a particular API, to obtaining samples of how to use that API.

THE ASSIEME SEARCH ENGINE

In order to power its interface, Assieme needs to identify two types of implicit references: uses of packages, types, and members in JAR files on pages with code samples, and matches of these objects to corresponding Javadoc documentation pages. In this section, we explain how our system infers implicit references and discuss how these references are used both to provide navigational capabilities within Assieme's interface and to improve search relevance.

Crawling for Code

Instead of attempting to crawl the entire Web, Assieme currently uses existing search engines to find potentially relevant content. Different strategies are used to obtain Web pages likely to contain sample code, to obtain likely documentation pages, and to obtain compiled libraries.

To collect pages that are likely to contain code samples, Assieme uses a general Web search engine to find pages with keywords which frequently appear in Java code. Specifically, Assieme automatically calls Google for the 2^{16} queries described by the pattern `java ±import ±class ±interface ±public ±protected ±private ±abstract ±final ±static ±if ±while ±for ±void ±int ±long ±double`, retrieving the top 1000 results for each query.

Documentation pages are similarly obtained by calling Google for queries including the term `overview-tree.html`. The term is the name of the automatically generated page that summarizes a Javadoc site, and it contains links to all other Javadoc pages for the same API. Assieme downloads the summary as well as pages referenced by outgoing links.

In order to locate libraries on the Web, we manually searched for code repositories which were referenced on a random subset of sample pages. The most popular Java library sites were Sun.com, Apache.org, Java.net, and SourceForge.net, so Assieme downloads library files for all projects hosted on these sites. We found that a library often contains other required libraries, and Assieme extracts these as well. Since Assieme retrieves many duplicates, it computes a hash of each library and keeps only one copy.

This approach yielded 2,361,331 Web pages, 481,220 documentation pages, and 79,302 unique JAR files.

Extracting Code Samples

The ability to automatically extract and analyze code from HTML pages is crucial to Assieme's success. We note that most code samples have several structural properties that

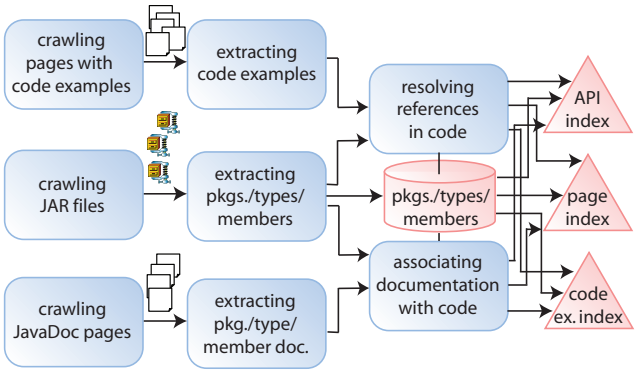


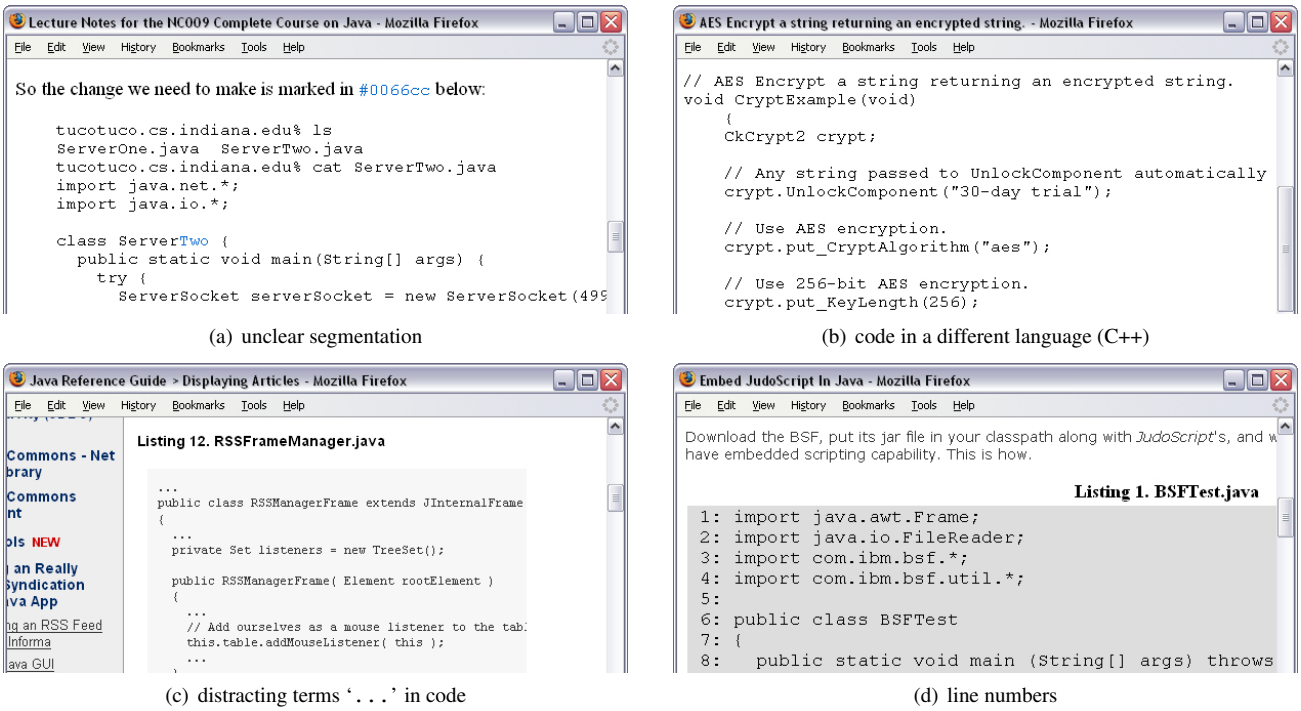
Figure 4: Assieme's system architecture.

distinguish them from text and which might be exploited for classification. For example, code samples are often formatted differently from text. Also, the distribution of terms and special characters like "{" or ";" is generally very different from that of other text on a page. A final obvious difference stems from the fact that code mostly obeys the language's syntax, so one might use compliance with a parser to differentiate code from text.

Unfortunately, none of these differences is foolproof; each fails to discriminate correctly on a significant percentage of Web pages (see Figure 5). A natural solution is the construction of a classifier that identifies code using features based on formatting, term and symbol distributions, and syntax parse errors. In our first approach to code extraction, we trained a Support Vector Machine with Gaussian Kernel on 181 such features. Although this enabled us to achieve high precision and recall, the approach has important drawbacks. First, the cost of computing these features was difficult to scale to millions or even billions of Web pages. Second, note that what we care about most are not actually the code samples themselves, but rather the external references contained in those samples. As we will discuss in the next section, however, external references can only be reliably detected by parsing code. Instead of optimizing for the number of terms classified correctly as either text or code, it may thus be better to ensure that our system can parse as many fragments of a code sample as possible.

Assieme currently extracts samples by first removing formatting commands from HTML. While doing so, it tries to preserve line breaks by taking into account semantics of HTML commands such as `<br>`. The text is then preprocessed using a number of simple heuristics that help to remove distractions. Most importantly, line numbers are detected and removed. Furthermore, Assieme deletes non-code character sequences that frequently appear in code samples such as "`...`" or "`etc.`" as well as characters of unconventional encodings. Finally, Assieme launches an error-tolerant Java parser [1] at every occurrence of a line break in the preprocessed text or wherever a HTML command appeared in the page source. If a large

[1] Error-tolerant parsers can recover from many parse errors and resolve references within syntactically incorrect code. They are frequently used for providing code assistance during editing in an IDE. We use Eclipse's JDT compiler.



(a) unclear segmentation

(b) code in a different language (C++)

(c) distracting terms '. . .' in code

(d) line numbers

Figure 5: Difficulties in extracting code snippets.

block of text parses with few reported syntax errors (such as unexpected characters), Assieme performs corrections suggested by the parser. The end of a code fragment is determined by tracking the state of the parser. Since few code samples contain complete Java compilation units, Assieme separately attempts to parse for types, methods, and sequences of statements. When extracted fragments are overlapping, shorter fragments are ignored. Fragments are then re-assembled, and Assieme creates compilation units by heuristically adding placeholder method and type signatures around fragments not otherwise within a method or type.

When dealing with code samples that do not fully parse, Assieme does not need to rely on its heuristic pre-preprocessing steps alone. By separately parsing types, methods, and statements, Assieme can parse code fragments around distractors and later re-assemble samples.

Resolving External Code References

Once a code sample has been extracted from a Web page, Assieme searches it for references to code in external libraries, using the library index created earlier to match package, type, method, and field names. The just-discussed code fragment re-assembly is critical to this because it allows Assieme to use the Java compiler to obtain fully-qualified names. A naive approach might search for pure term matches between samples and libraries. Unfortunately, this does not work well because names are frequently not stated contiguously, as the following example shows.

```
import java.util.*;
class c {
   HashMap m = new HashMap();
   void f() { m.clear(); }
}
```

Here, the method `java.util.HashMap.clear()` is referenced in line 4, but this can only be detected by combining information from different lines of the program.

When samples contain external references, the compiler identifies a set of unresolved names. Assieme searches the library index to find candidate libraries that might be required by the samples. After determining a set of libraries that might be referenced, Assieme places these onto the Java class path and attempts a new compilation. While compilation can verify dependencies and eliminate false solutions, it is computationally expensive and therefore only feasible in moderation. Furthermore, multiple libraries will often satisfy the compiler, necessitating further disambiguation.

Even worse, several libraries are often necessary to resolve all references, and so there is a need to select the *best subset* of libraries that supplies all references. Since the number of potential subsets is huge, an exact solution is impractical.

To confront these problems, Assieme employs a recursive greedy search algorithm with a utility function that uses the number of references covered as its highest-order factor; ties are broken by taking into account the library's popularity, as measured by counting its number of duplicates on the Web. Assieme limits the number of attempted compilations for the code samples of a page, marking a reference as unresolved if this threshold is reached.

Using Implicit References to Improve Scoring

Most Web search engines model queries and documents as vectors of weighted term frequencies and use a variant of cosine distance, weighted by inverse document frequency, for the similarity metric. Often this model is augmented by including URL terms, document titles, font information,

and anchor text into the weighting. Hypertext link analysis algorithms, such as PageRank [3], may also be used to differentially weight important documents.

None of these techniques work well for APIs. JAR files with code in binary format do not provide additional context that can be used for finding relations to free-text keywords. Yet such context is important for supporting the queries we found in our query logs. Even indexing API source code does not help much, since code contains few relevant keywords [12, 19, 27]. Finally, since Web search engines ignore structure in code, they are unable to recognize how frequently objects in APIs are referenced and thus how relevant they are.

Assieme overcomes these limitations by using implicit references to simultaneously exploit structure in code and information on Web pages. The following sections describe how Assieme separately scores APIs and Web pages.

Scoring APIs When scoring an API, Assieme utilizes text on Javadoc documentation pages as well as text on Web pages with code samples that reference an API.

Many pages with code samples are tutorials and articles containing not just code but also explanations. Not surprisingly, we found that the text around code often provides high-quality context which is also useful for describing the purpose of APIs referenced in that code. Assieme's strategy is therefore to explicitly use text around code samples for scoring APIs. As a result, our system can recognize that the `com.lowagie.text.pdf` package may be relevant to a programmer searching for "output acrobat." Assieme's method resembles the technique of anchor text indexing [10], which uses text of all incoming hyperlinks to index the target document. Anchor text often serves as a short summary of a page and therefore contains very relevant terms. In contrast to anchor text indexing, however, Assieme does not rely on hyperlink structure, instead using computed implicit code references.

In addition to using text on pages with code samples, Assieme also considers documentation on associated Javadoc pages. Although Javadoc pages are automatically generated from comments appearing in source code, our approach enables Assieme to access that information even when a library is not open-source.

To differentially weight important packages, types, and members in the index, Assieme assigns a static score based on a logarithm of the number of times each is referenced in the sample code Assieme has discovered. In some ways our approach resembles adaptations of PageRank to the graph of code references [1, 15].

Scoring Web Pages Assieme also makes use of implicit references when ranking Web pages. In addition to including Web-specific properties such as URL terms or document title into the weighting of terms of a page, Assieme considers qualified names to external referenced objects such as `java.util.HashMap`. This has three purposes: First, notice that a type like this might be referenced multiple times in a code sample, but with the term `HashMap` appearing only once. By taking into account the number of actual references, we can assign more accurate term weights than by merely counting term occurrences. Sindhgatta [27] discusses this problem in more detail. Second, as shown earlier, the complete character sequence `java.util.HashMap` might not contiguously appear in the code sample at all. Having the fully qualified name in the index enables Assieme to retrieve relevant pages for queries containing such references. Third, it allows Assieme to efficiently filter a query result set to only pages with code samples using particular APIs. This is an essential feature of Assieme's user interface, as it helps users transition smoothly from API selection to coordination or use without a need for entering new queries.

Assieme also uses information about extracted code for differentially weighting important pages in the index. Not all pages with code samples are helpful for programmers. For example, we noticed that a significant number of Web pages with code are generated by software version tools such as WebCVS. Code on these pages tends to be long and complex, and there is usually no accompanying text with explanations. In contrast, high-quality tutorial pages generally contain short and simple code samples together with helpful documentation. Assieme therefore attempts to favor the latter by taking into account the length of surrounding text in proportion to code samples. Specifically, Assieme sets the static score of a page to a weighted sum of its PageRank and a logarithm of this proportion.

EVALUATION

Assieme includes significant inference components in support of a new approach to searching for code on the Web, so we take two approaches to evaluation. We first analyze the reliability of Assieme's inference components, specifically our code extraction and reference resolution algorithms. We then present a user study examining programmers completing common search tasks using Assieme, a general Web search engine, and a code search engine.

Code Extraction and Reference Resolution

To understand the reliability of Assieme's code extraction and reference resolution, we examined precision and recall. As ground truth, we hand-labeled code samples in 350 randomly selected pages from Assieme's data. Precision and recall values were then computed by counting the number of terms correctly or incorrectly classified as code or text.

Our extraction algorithm identified code on 117 pages, while 54 pages contained hand-labeled code samples. Counting terms, our system reached a recall of 96.9% at a precision of 50.1%. Precision was affected by a large number of Web pages with C, C#, JavaScript, and PHP code samples, as well as by a number of pages generated by tools like Fisheye and diff that showed changes in Java codebases. We consider such code fragments to be false positives because we are interested only in high quality Java code samples.

While the recall of this initial code extraction is important, precision is of less concern because the reference resolution step filters many false positives when it cannot find a Java package, type, or member that corresponds to the extracted code. In this evaluation, 47 of the 63 pages containing false code samples did not contain any references that resolved.

Because terms on these pages are then ignored, the effective precision is increased to 76.7%. The majority of remaining false positives are from pages that show changes in Java codebases. Informed by this result, we intend future work to develop additional inference to identify this type of page so these tokens can be ignored.

We further analyzed Assieme's reference resolution in the code samples identified by our extraction algorithm. Assieme resolved 3011 references in our 350 selected pages, the majority of which pointed to packages, types, and members in Java's J2SE API. Of these, 2606 were in actual code samples, while 405 were false positives (a precision of 86.5%). Of these false positives, 267 are the result of Java code change sites, so the references resolved correctly but were not contained within code samples.

Another 301 actual references were not resolved (a recall of 89.6%). Some of these could not be resolved because code samples were incomplete. 104 references on 13 pages failed to resolve because necessary definitions were missing. Another 77 references on 4 pages failed to resolve because import statements were missing. Incomplete identification of code samples resulted in failure to resolve 45 references on 5 pages. Finally, 75 references on 6 pages failed to resolve because the referenced library was not in Assieme's index.

Since Assieme indexes more than 2.3 million Web pages, the runtime efficiency of our inference components is also important. Using a single thread on a single 3.2 GHz Intel Pentium D, sample code extraction took 38.3 milliseconds per page ($\sigma = 84.9$) and reference resolution 289.22 milliseconds per page ($\sigma = 655$). Our actual system is multi-threaded and runs on a cluster of 16 machines.

User Study

In addition to our analyses of Assieme's inference components, we conducted a user study comparing Assieme to a general Web search engine (Google) and a code search engine (Google Code Search). We planned to examine how quickly programmers complete common tasks, the quality of solutions obtained, and the number of queries issued.

Design Our study is based on a set of 40 search tasks designed to represent the types of searches that programmers perform as a part of their everyday work. We developed these tasks by examining the query log discussed earlier. For example, we used the query "socket java" as motivation for the task "Write a basic server that communicates using Sockets." Other tasks include loading an image in JPEG format, finding several libraries that could be used to parse an XML file, computing an MD5 hash of a String, and arranging four buttons in a 2x2 layout. Our complete task list is available upon request.

Although many of our tasks were specific programming assignments, we asked our participants not to write actual code, but to instead find code samples that best matched the task description, such that one could solve the actual task by making only minor modifications. Tasks were presented in a separate browser, and participants entered their solution for each task into a form in that browser.

Task Completion Time	Assieme	107s	**vs. Assieme**
	Google	117s	$p \approx .17$
	GCS	125s	$p \approx .017$
Solution Quality	Assieme	.861	**vs. Assieme**
	Google	.722	$p \approx .013$
	GCS	.467	$p < .0001$
Queries Issued	Assieme	1.42	**vs. Assieme**
	GCS	1.82	$p \approx .002$
	Google	1.90	$p \approx .001$

Figure 6: Summary of participant performance data. Programmers using Assieme produce higher-quality responses, using fewer queries, in the same amount of time they took when using Google.

Each session started with an overview and training period. During this training period, participants were shown 10 tasks, drawn randomly without replacement from our set of 40. The search interfaces were explained, and participants were encouraged practice with the 10 tasks until they felt comfortable with the interfaces. The primary portion of the experiment then began, and participants completed three blocks of 10 tasks, using a different search interface for each block and again drawing tasks randomly without replacement. Every participant was therefore exposed to each of our 40 tasks exactly once, and there was no systematic relationship between tasks and interfaces. A Latin Square design was used to control for the order in which search interfaces were presented to participants during the primary portion of the experiment.

Participants We recruited nine participants. Four were undergraduate students and five were graduate students, all majoring in Computer Science. The four undergraduate students reported between 3 and 5 years of programming experience, while the graduate students reported between 7 and 19 years. One participant had no Java experience, five had 2 to 3 years Java experience, and three had 8 to 11 years Java experience. One participant reported 6 years of industry experience, while the eight others reported less than a year.

Task Completion Time We first report on task completion time, measured from when our browser presented a task to the time when the participant submitted a response (by entering it into the form in our browser and clicking a submit button). We performed a Mixed Model analysis, modeling $ParticipantID$ as a random effect, finding significant effects for UI (which interface was being used) and $Order$ (which task, numbered 1 to 30, was being completed).

That $Order$ is significant ($F(1, 258) = 6.22, p \approx .013$) indicates that participants completed tasks more quickly as the experiment proceeded. We tested for an interaction between $Order$ and UI, finding no effect ($F(2, 138) = 0.35, p > .70$). This validates our experimental design, as completion time did improve, but the Latin Square design ensured that this improvement was not disproportionate to any one search interface.

UI is marginally significant ($F(2, 258) = 2.89, p \approx .057$), leading us to investigate pairwise differences. Assieme is

significantly better than Google Code Search ($F(1, 258) = 5.74, p \approx .017$) but not different from Google ($F(1, 258) = 1.91, p \approx .17$). Google and Google Code Search are not significantly different ($F(1, 258) = 1.03, p > .31$).

Solution Quality We coded the quality of each solution on a three-point scale, where 0 indicated a seriously flawed solution, .5 indicated a generally good solution that fell short in some critical regard, and 1 indicated a fairly complete solution. This coding was done blind to what search interface was used to obtain the solution. We again performed a Mixed Model analysis, modeling $ParticipantID$ as a random effect, finding significant effects for UI and $ExternalLib$ (whether a task required the participant find and download a Java Archive File) [2].

That $ExternalLib$ is significant ($F(1, 258) = 8.97, p \approx .004$) indicates that tasks with this property are significantly more difficult. Testing for an interaction between $ExternalLib$ and UI again validates our experimental design, as there is no effect ($F(2, 259) = 1.42, p > .24$). So this category of task was more difficult, but the random assignment of tasks to interfaces ensured that this property of the tasks did not impact the overall quality of responses associated with each interface.

UI is highly significant ($F(2, 258) = 28.7, p < .0001$), leading us to investigate pairwise differences. Assieme is significantly better than both Google Code Search ($F(1, 258) = 55.5, p < .0001$) and Google ($F(1, 258) = 6.29, p \approx .013$). Google is significantly better than Google Code Search ($F(1, 258) = 24.5, p < .0001$).

Queries Issued We logged the number of queries issued during each task. We performed a Mixed Model analysis, modeling $ParticipantID$ as a random effect, finding a significant effect only for UI ($F(2, 255) = 51.1, p < .0001$). Examining pairwise differences, Assieme is significantly better than both Google ($F(1, 259) = 9.77, p \approx .002$) and Google Code Search ($F(1, 259) = 6.85, p \approx .001$). Google Code Search is better but not significantly different than Google ($F(1, 259) = 0.259, p > .61$).

Subjective Ratings After completing the experiment, participants completed a brief questionnaire, the results of which are shown in Figure 7. Using paired t-tests to compare interface ratings, there is no significant difference between Assieme and Google when participants were asked about their overall impression of the interfaces ($t(8) = 0.32, p > .75$). Assieme rates significantly higher than both Google and Google Code Search on every other question, including how relevant the results are, how clear the presentation is, whether the presentation helps to get an overview of solutions, whether the presentation helps to judge the relevance of results, whether the presentation makes it easy to see the dependencies of code, and whether the presentation facilitates the comprehension of complex results.

[2] We model solution quality as continuous, but an argument could be made to model it as nominal. We conducted such an analysis and obtained the same results as we present here.

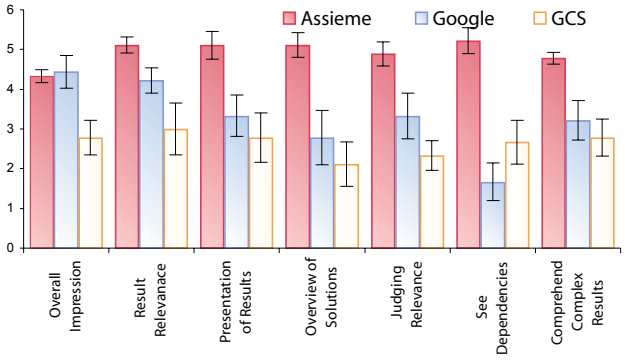


Figure 7: Mean and standard error of participant preference on seven-value Likert scales.

DISCUSSION AND CONCLUSION

We have presented Assieme, a novel Web search interface that helps programmers to quickly examine different APIs that might be appropriate for a problem, obtain more information about particular APIs, and see samples that show how to use an API. Assieme is implemented using a novel approach to finding and resolving implicit references in code samples on the Web.

Our evaluation shows that programmers using Assieme for common programming-related search tasks obtain better solutions, using fewer queries, in the same amount of time they spend using a general Web search interface (in our study, Google). We expected the result that programmers would issues fewer queries, as Assieme brings together information that is otherwise accessible only by formulating new queries. But the result that Assieme leads to better solutions in the same amount of time (versus, for example, more quickly leading to solutions of the same quality) is more complex.

Examining the click-through data from our study, participants using Google visited an average of 3.31 pages per task ($\sigma = 2.38$). In contrast, participants using Assieme visited only 0.27 pages per task ($\sigma = 0.68$), completing 73 of 90 tasks without visiting any external page. But Assieme allows the previewing of sample code without actually visiting a page, and considering such previews shows that participants saw content from 4.30 pages per task ($\sigma = 3.31$). The extent to which this result can be interpreted is limited by the fact that some of these previews are the result of incidentally mousing over a link, but a Mixed Model analysis treating $ParticipantID$ as a random effect shows that participants saw content from significantly more pages when using Assieme than Google ($F(1, 259) = 5.77, p \approx .017$).

Although additional studies are needed, we anecdotally observed that participants may have viewed content from more pages using Assieme because the ability to quickly preview code samples seems to have changed participant strategies. When using Google, participants seemed to issue queries, navigate to pages, and prepare a solution for the task based on the first page they encountered that seemed to provide the necessary information. When using Assieme, participants seemed more likely to continue exploring even after encountering an initial potentially useful code sample. Even after participants encountered an initial potentially

useful code sample, they would view several additional code samples to see if a better sample was available.

As search becomes increasingly fundamental to modern user interface software and as the machine learning community continues to make important advances in extracting information and identifying relationships from unstructured Web content, the approaches demonstrated by Assieme are likely to be applied to many other problems. Beyond being a novel Web search interface for programmers, Assieme points towards a future of understanding human information needs, designing search interfaces to support those needs, and then developing the Web inference needed for those interfaces.

ACKNOWLEDGMENTS

This work was supported by NSF grant IIS-0307906, ONR grant N00014-06-1-0147, SRI CALO grant 03-000225 and the WRF / TJ Cable Professorship. We thank Microsoft for providing us access to the query logs used in our analysis. We would also like to thank Eytan Adar, Saleema Amershi, Patrick Baudisch, Jiun-Hung Chen, Xin Dong, Krzysztof Gajos, Mausam, Miryung Kim, Desney Tan, Michael Toomim, Jacob Wobbrock, and Fei Wu, as well as the anonymous reviewers, all of whom have provided valuable insights on this work.

REFERENCES

1. Bajracharya, S., Ngo, T., Linstead, E., Rigor, P., Dou, Y., Baldi, P. and Lopes, C. Sourcerer: A Search Engine for Open Source Code. *International Conference on Software Engineering (ICSE 2007)*.
2. Banko, M., Cafarella, M., Soderland, S., Broadhead, M. and Etzioni, O. Open Information Extraction from the Web. *International Join Conferences on Artificial Intelligence (IJCAI 2007)*, 2670–2676.
3. Brin, S. and Page, L. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks 30*, 1–7 (1998).
4. Cubranic, D. and Murphy, G.C. Hipikat: Recommending Pertinent Software Development Artifacts. *International Conference on Software Engineering (ICSE 2003)*, 408–418.
5. Cutrell, E. and Guan, Z. What are you looking for? an eye-tracking study of information usage in Web search. *ACM Conference on Human Factors in Computing Systems (CHI 2007)*.
6. Cutrell, E., Robbins, D.C., Dumais, S.T. and Sarin, R. Fast, Flexible Filtering with Phlat. *ACM Conference on Human Factors in Computing Systems (CHI 2006)*, 261–270.
7. Dong, X., Halevy, A.Y. and Madhavan, J. Reference Reconciliation in Complex Information Spaces. *ACM Special Interest Group on Management of Data (SIGMOD 2005)*, 85–96.
8. Dontcheva, M., Drucker, S.M., Wade, G., Salesin, D. and Cohen, M.F. Summarizing Personal Web Browsing Sessions. *ACM Symposium on User Interface Software and Technology (UIST 2006)*, 115–124.
9. Dumais, S.T., Cutrell, E., Cadiz, J.J., Jancke, G., Sarin, R. and Robbins D.C. Stuff I've Seen: A System for Personal Information Retrieval and Re-Use. *ACM Special Interest Group on Information Retrieval (SIGIR 2003)*, 72–79.
10. Eiron, N. and McCurley K.S. Analysis of Anchor Text for Web Search. *ACM Special Interest Group on Information Retrieval (SIGIR 2003)*, 459–460.
11. Frakes, W.B. and Nejmeh, B.A. Software Reuse Through Information Retrieval. *SIGIR Forum 21*, 1–2 (1987), 30–36.
12. Henninger, S. Using Iterative Refinement to Find Reusable Software. *IEEE Software 11*, 5 (1994), 48–59.
13. Holmes, R. and Murphy, G.C. Using Structural Context to Recommend Source Code Examples. *International Conference on Software Engineering (ICSE 2005)*, 117–125.
14. Huynh, D.F., Miller, R.C. and Karger, D.R. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. *ACM Symposium on User Interface Software and Technology (UIST 2006)*, 125–134.
15. Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M. and Kusumoto, S. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Engineering 31*, 3 (2005), 213–225.
16. Kleinberg, J.M. Authoritative Sources in a Hyperlinked Environment. *ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, 668–677.
17. Ko, A.J., Myers, B.A. and Aung, H.H. Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2004)*, 199–206.
18. Little, G. and Miller, R.C. Translating Keyword Commands into Executable Code. *ACM Symposium on User Interface Software and Technology (UIST 2006)*, 135–144.
19. Maarek, Y.S., Berry, D.M. and Kaiser, G.E. An Information Retrieval Approach For Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering 17*, 8 (1991), 800–813.
20. Mandelin, D., Xu, L., Bodík, R. and Kimelman, D. Jungloid Mining: Helping to Navigate the API Jungle. *ACM Conference on Programming Language Design and Implementation (PLDI 2005)*, 48–61.
21. Mili, A., Mili, R. and Mittermeir, R. A Survey of Software Reuse Libraries. *Annals of Software Engineering 5*, (1998), 349–414.
22. Neate, B., Irwin, W. and Churcher, N. CodeRank: A New Family of Software Metrics. *Australian Conference on Software Engineering (ASWEC 2006)*, 369–378.
23. Ostertag, E., Hendler, J.A., Prieto-Díaz, R. and Braun, C. Computing Similarity in a Reuse Library System: An AI-Based Approach. *ACM Transactions on Software Engineering and Methodology 1*, 3 (1992), 205–228.
24. Paek, T., Dumais S.T. and Logan, R. WaveLens: A New View onto Internet Search Results. *ACM Conference on Human Factors in Computing Systems (CHI 2004)*, 727–734.
25. Paul, S. and Prakash, A. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering 20*, 6 (1994), 463–475.
26. Puppin, D. and Silvestri, F. The Social Network of Java Classes. *ACM Symposium on Applied Computing (SAC 2006)*, 1409–1413.
27. Sindhgatta, R. Using an Information Retrieval System to Retrieve Source Code Samples. *International Conference on Software Engineering (ICSE 2006)*, 905–908.
28. Singla, P. and Domingos, P. Entity Resolution with Markov Logic. *IEEE International Conference on Data Mining (ICDM 2006)*, 572–582.
29. Tansalarak, N. and Claypool, K. XSnippet: Mining for Sample Code. *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2006)*.
30. Xie, T. and Pei, J. MAPO: Mining API Usages from Open Source Repositories. *International Workshop on Mining Software Repositories (MSR 2006)*, 54–57.
31. Yee, K.-P., Swearingen, K., Li, K. and Hearst, M.A. Faceted Metadata for Image Search and Browsing. *ACM Conference on Human Factors in Computing Systems (CHI 2003)*, 401-408.