Using Association Metrics to Help Users Navigate API Documentation

Daniel S. Eisenberg, Jeffrey Stylos, Andrew Faulring, Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
dsecmu@gmail.com, {jsstylos, faulring, bam}@cs.cmu.edu
http://www.cs.cmu.edu/~apatite

***Abstract* -- In the past decade there has been spectacular growth in the number and size of third-party libraries, frameworks, toolkits and other Application Programming Interfaces (APIs) available to modern software developers. However, the time-saving advantages of code re-use are commonly hampered by the difficulty in finding the correct methods for a given task among the thousands of irrelevant ones. We have developed a tool called Apatite that helps address this issue by letting programmers browse APIs by viewing associations between their components. Apatite indicates which items of an API are popular in different contexts and allows browsing by initially selecting verbs (methods and actions) in addition to classes and packages. The associations are calculated by leveraging existing search engine data and source code, and verbs are identified by parsing the documentation descriptions. Apatite is available on the web and is being used by developers worldwide on a regular basis.**

Keywords-API Documentation; Search tool; Browsing; Visualizations; Web applications.

I. INTRODUCTION

Software development increasingly relies on the use of Application Programming Interfaces (APIs). Therefore, learning how to use an unfamiliar API is a key step in writing code efficiently and effectively. However, this is not a trivial task. Modern APIs like the Java SDK contain thousands of classes and tens of thousands to hundreds of thousands of methods. Finding the correct class and method to use is often challenging, even for experienced developers.

Despite the explosive growth of API usage, the many advancements in programming language design, and the extensive research on locating example source code, API documentation itself has seen far fewer improvements. After twenty years of Java development, static Javadoc entries are still the dominant form of API references. Although modern programmers often use Google to search for API usage assistance, evidence indicates that this is an imperfect strategy [15] and is not an option for developers using proprietary or less-popular platforms [2].

There are several issues that prevent traditional, static API references from being effective:

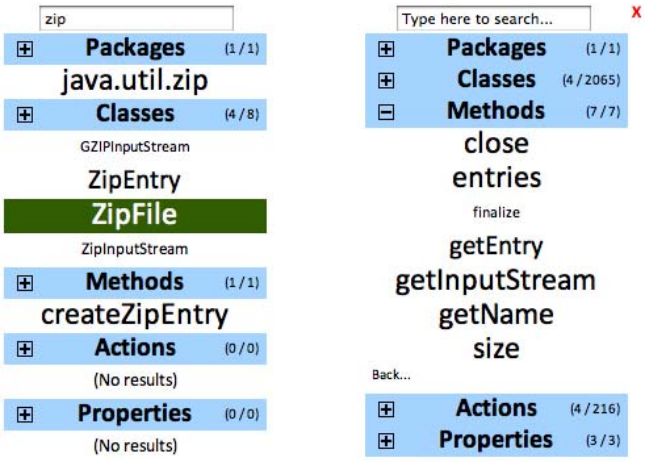

Figure 1. Apatite's novel multi-column, multi-section interface. The text boxes support keyword searching, and the "+" icons expand sections in an accordion-like interface.

1. Conventional documentation requires that users start browsing API documentation by choosing a package or class, whereas developers are sometimes searching for a particular action and do not know which class implements it.
2. Developers may lack knowledge of which classes and methods are most often used in practice.
3. Developers using an unfamiliar API may think about their problems using different terminology than the one used by the API (the "vocabulary problem" [7]).

Apatite (see Fig. 1), which stands for **A**ssociative **P**erusal of **API**s **T**hat **I**dentifies **T**argets **E**asily, takes a novel approach to addressing these issues. Instead of forcing users to begin by choosing a package or class, Apatite allows users to search across any level of an API's hierarchy by traversing associations between items. This is enabled by a unique interaction technique that displays iterative results in vertical columns, which contain the most relevant items from each level of the hierarchy, as we described in our short CHI 2010 note [6]. In the current paper, we describe the implementation of these features, including techniques for generalizing the system to new APIs by extracting association information from search engine data and existing source code.

978-0-7695-4206-5/10 $26.00 © 2010 IEEE
DOI 10.1109/VLHCC.2010.13

We also introduce a new feature that associates methods with verbs.

II. Previous Work and Motivations

A. Tools

In the past several years there have been a large number of tools developed to help make using APIs easier. Most have focused on automatically locating example code in a number of different contexts, for example, Sourcerer [1] and Prospector [10]. This task was also addressed by XSnippet, a code assistant tool developed by Sahavechaphan and Claypool [14]. Some tools, such as Strathcona, have attempted to bypass the need for users to construct their own queries by attempting to infer them automatically from code currently under development [9]. However, these tools offer limited help to developers who do not yet know which methods they should be investigating, or who do not have enough high-level knowledge about an API to construct an effective context with which to generate examples.

Other projects have aimed to actively detect popular usage patterns from large corpora of source code. CodeWeb uses data mining techniques to extract library reuse patterns and displays usage information to the user [11]. SpotWeb is a similar tool that detects API "hotspots," patterns that are frequently re-used in open source frameworks [18]. The PopCon prototype calculates popularity statistics about each item in an API to help programmers and API developers answer high-level questions about usage [8].

Apatite builds on these tools by introducing new kinds of API associations and displaying them in a novel interface.

B. Studies of programmers

Many studies have looked at the issues around API usage. In one, programmers attempting to use an unfamiliar API were observed going through six distinct phases: initial design, high-level API understanding, architectural design, finding methods, finding examples, and integrating examples [15]. The tools described in the previous section are targeted towards these last two phases. With Apatite, we primarily target users who are still in the "finding methods" stage or earlier.

Recent research reinforces the notion that the problem of finding examples is often superseded by the issue of not knowing what exactly to look for [3]. That study of programmers examined how they used the web to assist in coding. Three different use intentions were identified: learning, clarifying, and reminding. The third category encompassed searching for frequently used code snippets with very exact and accurate queries. However, for other types of queries, subjects often searched using terminology from the wrong programming language, searching for an analogue in the target platform. This finding mirrors the observation from our previous study that identifying an API's standard terminology is an important early step in the process of learning an API [15].

C. Psychology of searching and retrieving

There is much evidence that people's memories are based on associations [5]. Our memories are highly linked, and we often remember things indirectly, as being related to other things. A recent study shows that people prefer to find information on a computer in a similar manner, using associations to make a series of small steps rather than one single leap to the destination as in a typical search [17].

Traditional tools for exploring API documentation either support the "teleport" style of search or support only a limited set of associations, for example the classes contained in a package or the methods contained in a class. Apatite adds many other associations to enable new ways of browsing.

D. Design Inspiration

In addition to addressing this previous research, Apatite's design draws inspiration from several of our group's previous projects.

Mica is a search-based website for exploring API documentation [15]. Because it allows users to type arbitrary text queries and bases its results on analysis of the Google result pages, it is limited to only analyzing the first 10 results so that it can be fast and responsive. Mica's user interface looks very similar to Google's results pages.

Another inspiration for Apatite was the associative browsing tool Feldspar [4]. Feldspar's interface allows users to browse personal information like email, contacts, and events by association rather than with keyword search. For example, using Feldspar you could find people mentioned in an email about an event last year, without needing to remember any of the specifics or enter any text queries.

The Jadeite Java documentation system gives programmers more cues about which classes and methods are commonly used [16]. Jadeite uses font sizes similar to tag clouds but in a single alphabetical list. In the user study of Jadeite, this feature seemed to be very effective in the context of Javadoc-like documentation, so we adopted this in Apatite.

III. Apatite Interface Overview

Apatite's interface was designed iteratively, making extensive use of user feedback. Many of the features described here were developed and refined by running a series of formal and informal user studies during prototype stages.

Fig. 2 demonstrates a sample search sequence of the standard Java 6 API using Apatite. The interface uses vertical columns to display each step of a search. When the tool is first loaded, a single column (on the far left) is displayed, showing the four most popular items in five different categories – *Packages*, *Classes*, *Methods*, *Actions*, and *Properties*. Font size indicates relative popularity, analogous to a tag cloud. The algorithm for determining popularity is explained

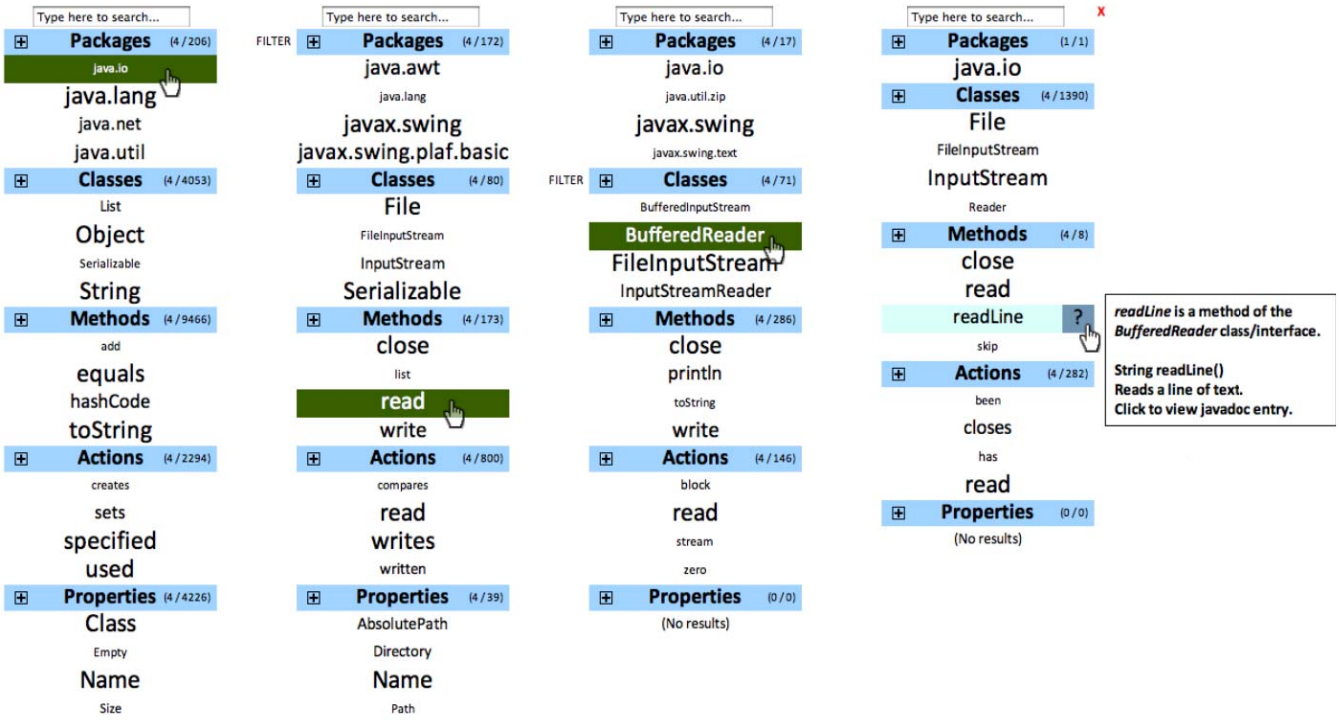

Figure 2. A screenshot of the Apatite user interface after several steps. Each column shows one step in the search process.

in the next section. The first three categories list components of the API. *Actions* are verbs that are carried out by the API's methods (this is explained in more detail in Section V). *Properties* are anything that can be accessed by "get," "set," or "is" methods (such as `getName()` or `isEmpty()`). Clicking on a category name expands that section and collapses everything else, revealing more entries (see Fig. 1, right column). Additionally, there is a search box at the top of the column that instantly filters all entries based on one or more keywords (Fig. 1 left column).

The user begins browsing by clicking on an item (`java.io` in Fig. 2), which generates a new column to the right of the current one. The new column contains items that are related to the selected entry in the previous column, and font sizes indicate the strength of association. For example, the second column in Fig. 2 indicates that the `File` class is highly associated with the `java.io` package. The kind of relationship depends on the particular categories involved; explanation text describes the nature of the relationship when the cursor hovers over an item. A "Filter" option, demonstrated in Fig. 3, brings up a list of the kinds of relationships that are being displayed in a particular category (for example, subpackages of a selected package or implementations of a selected method) and allows the user to exclude any of them. Relationships between items in adjacent columns are described in more detail in the next section.

Clicking on an item in the second column generates a third column, this time ranking items based on the strength of association with both of the previously selected items. The user can continue to browse the API in this fashion, column-by-column, until the desired item is located. The browser automatically scrolls horizontally as the search progresses. All previous columns remain clickable, so users can inspect their current search and backtrack if desired. Each entry has a "?" area that, when hovered over, displays additional documentation information (see Fig. 2, far right); clicking on the question mark pops up a new browser window pointed directly at that item's Javadoc entry. If the entry is ambiguous, such as a method implemented by multiple classes, an intermediate page prompts the user to select a particular instance.

Our intention with this design is to allow the user to explore an API in either a top-down *or* a bottom-up fashion. If the user is seeking information about an unfamiliar method or action, it can be selected in the first column to discover which class has the most commonly used implementation. Alternatively, if the containing package or class is already known, the user can select it in the first column to guide the rest of the search.

Fig. 2 demonstrates a typical use scenario. The user is looking for a method in java.io that reads data. The java.io package is selected first, which reveals the top classes, methods, actions, and properties associated with that package. The user discovers the popular read method, which sounds promising. After clicking it, the *Class* category in the next column reveals which classes implement read and are used frequently. The *Methods* category shows us other associated methods, like `write` and `close`. Finally, the user can

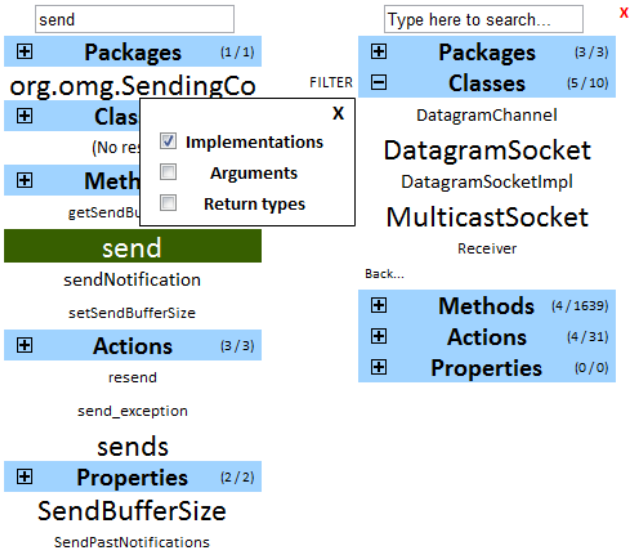

Figure 3. Demonstration of filtering a list of items according to the relationship with the previously selected item.

choose the `BufferedReader` entry and look in the next column to find additional relevant methods specific to that class, like `readLine` and `skip`.

IV. Calculating Components' Associations

Apatite is implemented as a web-accessible application using MooTools,[1] an open-source framework that extends JavaScript with object-oriented functionality and cross-browser visual effect methods. All of the necessary data is stored on the server in a MySQL database and served on-demand via AJAX-style requests. Our primary motivation in using web-based technology was to create a tool that the general public could easily use without needing to install any additional software.

Apatite's core functionality depends on the associations that it visualizes between an API's components. We have designed and implemented two different techniques for mining API usage data and extracting reasonable popularity and association metrics. The first method, designed for APIs with high public exposure, leverages search engine data to infer widespread usage patterns. The second method collects similar information from a corpus of source code, to be used in cases where an API is not heavily documented or discussed on the public web. These are explained next.

A. Collecting Popularity and Association Data from Search Engine Results

1) Populating the initial column. The first column that Apatite displays is based on how frequently each API item is used by programmers. Popularity information for each item is computed with the same technique used in the Jadeite documentation tool [16]: The popularity for each package, class and method is based on the number of hits returned by a Google search for each item. For example, the weight for the `InputStream` class is computed based on the number of results returned for the query: "java.io" +InputStream. These results are computed ahead of time as a batch process and are cached in Apatite's database.

Popularity data for API items tends to follow a power law [16], so within each category a logarithmic function is applied to each item's Google hits to derive a corresponding font size:

$$Scale = 75\% + (100 \times \frac{\log w - \log w_{\min}}{\log w_{\max} - \log w_{\min}})\%$$

where w is the number of hits for each item, $w_{\max}$ is the number of hits for the most popular item in the result set, and $w_{\min}$ is the number of hits for the least popular item the result set. The scale value is multiplied by a baseline font size to determine the final size (up to a maximum of 150%). When a category is expanded to show more items, they are once again sorted alphabetically, and their font sizes are recomputed using the new $w_{\max}$ and $w_{\min}$ values.

2) Populating additional columns. When the user clicks on an item, a new column appears consisting of items that are associated with the user's selection. These associations have been pre-computed using various methods (described below). Each item in this associated column also has a numeric score that represents how strong the association is; this is an analogue to the number of Google hits used in the first column. With this data, Apatite displays the new column using the same process that it uses on the initial column: the strongest associations in each category are retrieved, sorted alphabetically, and then sized using the logarithm of their score, as described above.

The process for computing these associations and their weights depends on the nature of the association.

For associating a *method* with other *methods*, we collected a new set of popularity data that indicates how often methods are used in conjunction with one another. To determine the metric for associating *Method A* with *Method B*, we counted the number of times the text "`.methodB(`" appears in the contents of the first 100 Yahoo! search results for "`methodA`". We switch to Yahoo! search to compute the method-to-method associations because the Yahoo! search API allows more queries per day than the Google API allows. The result approximates how often the methods are used together in actual code.

For associating a *method* with *classes*, we include three different types of relationships: classes that *implement* the selected method, classes that are *returned* by the selected method, and classes that are *arguments* to the selected method. For each of these, the association weight is the sum of the number of Google hits over all classes and interfaces that meet the association's criteria. For example, the weight for

[1] http://mootools.net

associating the `read` method with the `ByteBuffer` class is the total number of Google hits for all `read` methods which take a `ByteBuffer` object as an argument. If a certain class is associated with a method in more than one way, we use the maximum weight of these associations.

For associating a *class* with other *classes*, we take the sum of the association weights between `Class A`'s methods and `Class B`'s methods, multiplied by the logarithm of `Class B`'s overall popularity. Using the logarithm of the popularity in this case avoids overemphasizing extremely popular classes like `Object`.

The remaining associations (package to class, class to package, class to method, etc.) can all be characterized as having an encapsulation relationship (for example, `BufferedReader` is *contained* in the `java.io` package). For these, we assign weights by summing the Google popularity data over all implementations that satisfy the encapsulation condition. For example, to generate associations between the `read` method and various packages, we first group all of the classes and interfaces that have a `read` method by their containing packages, and then take the sum of the popularities over each group.

B. Collecting Popularity and Association Data from Source Code

Unlike the standard Java SDK API, many other APIs are not sufficiently indexed by search engines like Google and Yahoo! because they receive limited use or are proprietary. In fact, developers using these kinds of APIs must rely even more heavily on traditional Javadoc-style documentation due to a lack of effective tutorials and discussion forums. To support such APIs, we implemented an application that mines Java API association data using only existing source code.

Calculations made in this implementation are relatively straightforward and are based around counting method calls. The program takes as inputs the implementation of the API (with Javadoc-style commenting) and a corpus of existing source code that uses that API. The Javadoc tool[2] is used to inspect the API source and extract a list of all its classes and interfaces. Next, we use the MAPO tool developed by Xie and Pie [21] to extract all of the method call sequences in the provided source code, inlining any private method calls to better judge the frequency with which each API method is called. The number of occurrences in the source code of each method from the target API is counted, and this is used to calculate the absolute popularity of each method, class, and package.

For the association strength between two methods, we count the number of times they are called within seven method calls of each other. To choose this number, we analyzed how it affected the top associations for each method. Fig. 4 shows how stable the top 4 and top 8 results were for each method compared to the window of examined code. For example, for each given method in the API, we calculated its association to every other method by counting the number of times each other method was called within six calls of the target method. After repeating this process and examining seven calls instead of six, the top four strongest associations for each method remained the same nearly 90% of the time. After conducting this analysis, we found that change among the top 4 and top 8 results for each method were relatively stable (derivative less than or equal to 1%) with a window size of seven method calls.

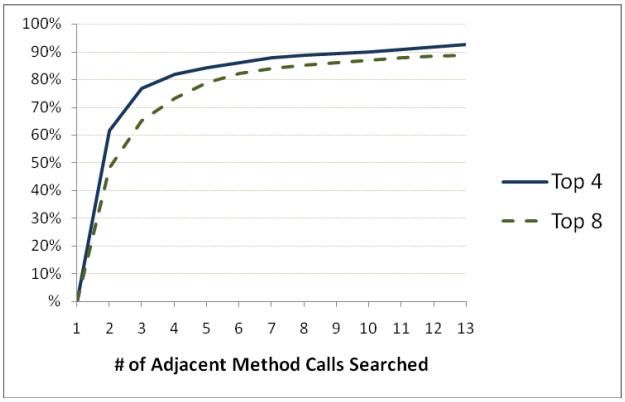

Figure 4. Percentage of non-changing associated method sets after incrementing the number of adjacent method calls searched.

Associations between all remaining categories are calculated in the same manner as discussed in section IV.A, aggregating the method pair association metrics to infer relationships between all other types of items.

V. Calculating Verb-Method Associations

A fundamental aspect of Apatite's interface is that it allows users to view and browse different categories of search results (classes, methods, etc.) in a single session. This feature allows us to experiment with integrating higher-level concepts and abstractions into the API browsing experience.

As described in Sections I and II, a previous study [2] indicates that programmers often have in mind a particular operation or action but do not know which classes or packages contain these methods. However, methods that achieve similar tasks in two different APIs are likely to have different names, and this vocabulary problem is a significant barrier to switching between APIs efficiently. Although Apatite allows users to begin queries with method names, users might not even know what method to look for. To accommodate for this use case, we added a new category "Actions" for associating methods with verbs. Tables I and II show some sample method-verb associations generated by our algorithm.

A. Algorithm

We first attempted implementing the verb-method associations by simply splitting up method names by camel case

[2] http://java.sun.com/j2se/javadoc/

and identifying whether the first "word" was a verb. Although this did do a somewhat effective job of clustering methods that represent similar actions, it did not alleviate issues arising from terminology mismatches.

To generate a better association graph between methods and actions, we anticipated two properties that a solution would need to have in order to substantially address the vocabulary problem. First, it would need to associate a single, common action with several different variations on its meaning. For example, the act of "removing" things from a `List` can be accomplished with several distinctly different methods, including `remove` (to remove the first occurrence of an item), `retainAll` (to remove everything *except* certain items), and `clear` (to remove all items). Second, it would need to associate a single method with a variety of different actions – essentially, be able to accommodate synonyms.

Our solution to this problem is a new technique that leverages the text of the existing documentation to determine likely verb-method relationships. Our hypothesis is that, since method names are often similar across different classes of the same API, the frequencies of particular verbs used by API authors in method documentation text provide good approximations for how strongly each method name is associated with particular verbs. For example, if 75% of methods in the standard Java API called `valueOf` have the verb "represents" in their descriptions and 50% of them have the verb "converts" in their descriptions, then methods in this API named `valueOf` *in general* are strongly associated with the act of representing and slightly less associated with the act of converting.

B. Implementation

The process of calculating the association graph between verbs and methods consists of three stages: tagging, stemming, and aggregating.

After initially extracting each method's Javadoc description (accomplished with the *Javadoc* tool mentioned in Section IV.B), we need to identify which words are actually verbs. We identify verbs within the documentation using the *Stanford Log-linear Part-Of-Speech Tagger*[3] developed by Toutanova *et al.* [19], which was configured to use the standard, pre-trained English model. One caveat we encountered was that the first sentences of method descriptions are often technically sentence fragments, with the method as an implied subject (for example, `File.delete` has the description "Deletes the file or directory…") The tagger has difficulty recognizing the leading verbs in these situations, presumably because it fails to locate any obvious subject. To work around this problem, we add an assumed "This method" prefix in front of every method description (changing the above example into "This method deletes the file or directory"). This correction fixes the half-formed sentences but does not seem to interfere with the tagger's ability to recognize verbs in other, fully-formed leading sentences.

TABLES I AND II. SAMPLE ASSOCIATIONS BETWEEN CLASSES, ACTIONS, AND METHOD NAMES.

Class	Action	Top Methods
java.util.List	removes	remove, clear, add, removeAll
java.lang.String	converts	toString, hashCode, valueOf, format
java.io.InputStream	read	read, available, close, skip
javax.swing.JButton	resets	updateUI, reset, close, clear

Method Name	Top Actions
retainAll	removes, contains, retains, modified
setEnabled	enabled, disabled, prevent, sets
Format	format, appends, writes, result
Restart	restart, cancels, pending, fired

Once the verbs in each method description are identified, each one is mapped to its underlying verb stem. Verb stems are generated using an implementation of the Porter Stemming Algorithm [12][4]. The verb-stem mappings are examined to identify the most popular concrete form of each stem, which is chosen as the stem's "representative." For example, if *remove*, *removes*, and *removing* are all used throughout the documentation and *removes* is used the most, all three are represented by *removes*.

Finally, the verb instances are grouped by method name and counted. To avoid universally common verbs from dominating the results, we apply the term frequency-inverse document frequency (*tf-idf*) formula to determine the strength of the relationship between a verb and a method name. Specifically,

$$A_{mv} = n_{mv} * \log \frac{|D|}{1 + |\{d \mid v \in d\}|}$$

where A_{mv} is the association metric between method m and verb v, n_{mv} is the number of times verb v appears in descriptions of methods named m, $|D|$ is the total number of descriptions, and $\{d \mid v \in d\}$ is the number of descriptions that have at least one instance of the verb v [13]. The 1 in the denominator prevents divide-by-zero errors.

In the final dataset, we manually prune out extremely common and generic verbs like "is," "has," "returns," "called," and "specifies."

VI. USAGE AND LIMITATIONS

During the final stages of iteratively evaluating Apatite's design, a majority of our subjects had positive reactions to Apatite. Most requested to be notified when it was made public and felt it would become significantly more useful after they used it over a longer period of time. Several commented that Apatite would have been useful during introductory programming courses.

[3] http://nlp.stanford.edu/software/tagger.shtml

[4] http://tartarus.org/~martin/PorterStemmer/

Various versions of the Apatite interface have been publicly available as a web application since June 2009. Usage statistics suggest that Apatite's unique features are being discovered and utilized. 64% of all visits to the application have been searches involving at least three columns. 83% have used at least two columns, and 11% of the remaining visits have used the rapid text search.

Apatite's non-linear search style does carry some disadvantages. Displaying entity names without their fully-qualified identities (for example, `read` instead of `BufferedReader.read`) allows us to display aggregate information, but sometimes this makes it difficult to identify which particular instantiation of a method should be used. Name collisions also preclude Apatite from intuitively displaying results from more than one API in a single interface.

Our approach of computing popularity and associations between different API items relies on the existence of a corpus that is representative of how people actually use an API. For existing APIs that have seen significant usage, we think this works well, but it does not work for APIs that are completely new. One possibility is to allow an API designer to manually designate the expected popularity of each item, but for reasonably large APIs, this would probably be infeasible.

VII. Future Work

There are several aspects of Apatite that we believe hold potential for promising future work.

A commonly requested feature that we plan to explore in a future version of Apatite is to display the example code from which associations and popularity measures are computed. However, since there are multiple examples for each measure, it is not obvious which one or ones should be displayed.

Embedding Apatite inside of a programming development environment like Eclipse could help programmers explore APIs without having to leave the code view. This would also allow the tool to use contextual information about the code that has already been written to help decide which items to display.

Apatite was initially envisioned for users who already know how to program. However, the interface may be applicable to new programmers as well. A version for that audience might use nouns in addition to verbs as the primary sections and help learners to access example code and tutorials in addition to the standard documentation pages.

We think that Apatite could be a useful tool in helping API designers create new APIs, by helping them see the commonly used parts of current APIs and by helping them anticipate which classes and methods will likely be used together.

We also envision other new research directions inspired by our experiences. The approach of letting users browse information by association could be a useful technique in domains outside of programming as well. This approach could allow users to explore many different kinds of large heterogeneous data sets starting from different directions.

VIII. Conclusion

Apatite demonstrates new interaction techniques for browsing APIs by association. It allows programmers to browse *verb first* when needed, letting them discover the relevant items rather than forcing them to guess which class to start from. We have developed techniques for extracting popularity and association statistics from web data and existing source code, in addition to identifying associations between methods and verbs from API documentation.

Apatite can be accessed at http://www.cs.cmu.edu/~apatite, where it is already being used by users worldwide on a regular basis.

Acknowledgements

This work was funded in part by a grant from SAP, and in part under NSF grants CCF-0811610 and CCR-0324770. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect those of the NSF.

References

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. "Sourcerer: a search engine for open source code supporting structure-based search", *Companion To the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications.* Portland, Oregon, USA, October 22 - 26, 2006.

[2] J. Beaton, S. Y. Jeong, Y. Xie, J. Stylos and B. A. Myers. "Usability Challenges for Enterprise Service-Oriented Architecture APIs", *2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'08*, Herrsching am Ammersee, Germany, Sept 15-18, 2008. pp. 193-196.

[3] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S.R. Klemmer. "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code", *27th International Conference on Human Factors in Computing Systems,* Boston, MA, USA, April 04 - 09, 2009.

[4] D. H. Chau and B. Myers. "What to Do When Search Fails: Finding Information by Association", *Proceedings CHI'08: Human Factors in Computing Systems*, Florence, Italy, April 5-10, 2008. pp. 999-1008.

[5] G. Davies and D. Thomson. *Memory in Context: Context in Memory*. Wiley, England, 1988.

[6] D. S. Eisenberg, J. Stylos and B. A. Myers. "Apatite: A New Interface for Exploring APIs", *CHI'2010: Human Factors in Computing Systems*. Atlanta, GA, April 10-15, 2010.

[7] G. W. Furnas, T. K. Landauer, L. M. Gomez and S. T. Dumais. "The vocabulary problem in human-system communication", Commun. ACM. 1987. 30(11). pp. 964-971.

[8] R. Holmes and R. J. Walker. "A newbie's guide to eclipse APIs", 2008 *International Working Conference on Mining Software Repositories*, Leipzig, Germany, May 10 - 11, 2008.

[9] R. Holmes, R. J. Walker, and G. C. Murphy. "Strathcona example recommendation tool", *SIGSOFT Softw. Eng. Notes* 30, 5. Sep. 2005, 237-240.

[10] D. Mandelin, L. Xu, R. Bodík , D. Kimelman. "Jungloid mining: helping to navigate the API jungle", Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, June 12-15, 2005, Chicago, IL, USA.

[11] A. Michail. "Code web: data mining library reuse patterns", *23rd international Conference on Software Engineering,* Toronto, Ontario, Canada, May 12 - 19, 2001.

[12] M. F. Porter. "An algorithm for suffix stripping", *Readings in Information Retrieval*, Morgan Kaufmann Publishers, San Francisco, CA, 1997, pp. 313-316.

[13] G. Salton and C. Buckley. "Term-weighting approaches in automatic text retrieval", *Inf. Process. Manage.* 24, 5 (Aug. 1988), 513-523.

[14] N. Sahavechaphan and K. Claypool. "XSnippet: mining for sample code." *SIGPLAN Not.* Oct. 2006, pp. 413-430.

[15] J. Stylos and B.A. Myers. "Mica: A Programming Web-Search Aid", *2006 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'06*. Sept 4-8, 2006, Brighton, UK. pp. 195-202.

[16] J. Stylos, A. Faulring, Z. Yang and B. A. Myers. "Improving API Documentation Using API Usage Information", *2009 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC'09*, Corvallis, Oregon, Sept. 20-24, 2009.

[17] J. Teevan, C. Alvarado, M. Ackerman and D. Karger. "The Perfect Search Engine Is Not Enough: A Study of Orienteering Behavior in Directed Search", *Proceedings CHI'04: Human Factors in Computing Systems*. pp 415-422.

[18] S. Thummalapenta and T. Xie. "SpotWeb: detecting framework hotspots via mining open source repositories on the web", 2008 International Working Conference on Mining Software Repositories, Leipzig, Germany, May 10 - 11, 2008.

[19] K. Toutanova and C. D. Manning. "Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger," *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000)*, pp. 63-70.

[20] E. Tulving and D. Thomson. "Encoding specificity and retrieval processes in episodic memory", Psychological Review 80, 1973, pp 352-373.

[21] T. Xie and J. Pei. "MAPO: Mining API usages from open source repositories", *Proc. International Workshop on Mining Software Repositories (MSR)*, pages 54–57, 2006.