Linking Usage Tutorials into API Client Code

Naihao Wu
Computer Science
Clarkson University
Potsdam, New York, USA 13699
wun@clarkson.edu

Daqing Hou, Qingkun Liu
Electrical and Computer Engineering
Clarkson University
Potsdam, New York, USA 13699
dhou, qliu@clarkson.edu

ABSTRACT

Traceability links between software artifacts have important applications in the development process. This paper concerns a special case of traceability recovery, i.e., the automated integration of API usage tutorials with the API client code. Our solution involves partitioning the client code into multiple semantic groups/snippets and linking each snippet to the best matching tutorials. Evaluation using benchmarks created for two popular APIs reveals that our solution can find the expected tutorial links at the average rank of 1.6 and 1.4 in the top ranked results, for the two API's, respectively, and with good average precision and recall. We also evaluate the impact of both method partitioning and JavaDoc query expansion on tutorial linking performance. Lastly, we conduct a formative user study to pinpoint the scenarios where our solution actually helps a software developer. We conclude that it is a promising approach to speeding up the maintenance of API client code.

CCS Concepts

•Software and its engineering → Integrated and visual development environments; Software libraries and repositories;

Keywords

API; Documentation; Traceability; Information Retrieval; Vector Space Model; Abstract Syntax Tree; Java.

1. INTRODUCTION

It is estimated that up to 60 percent of software maintenance is spent on program comprehension [17]. A substantial component of this comprehension cost can be attributed to using Application Programming Interfaces (APIs). While APIs provide powerful abstraction mechanisms that can be used to build client programs, the usage does not come for free: understanding how to use an API can be time-consuming [12].

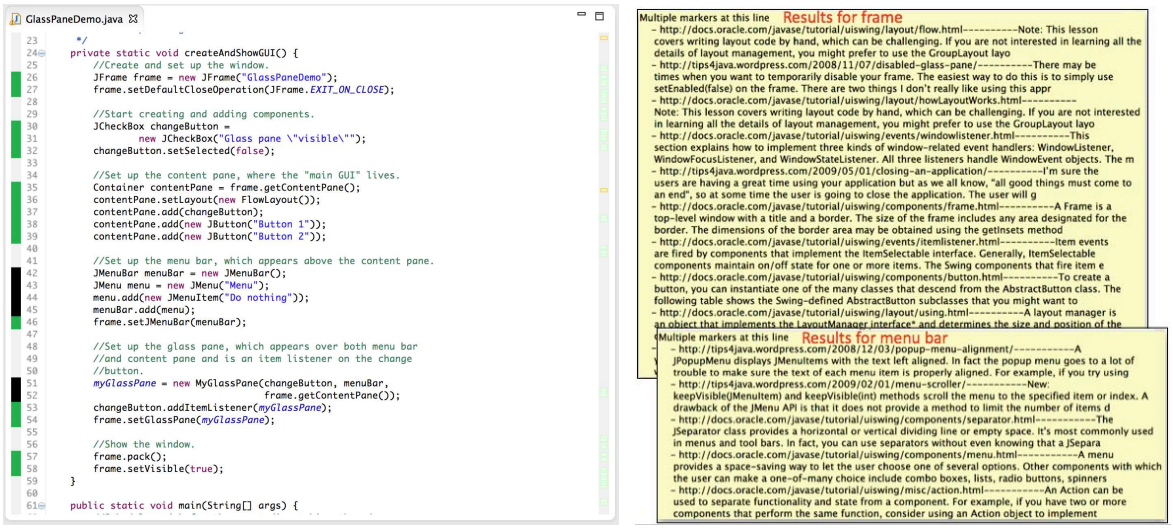
CSI-SE'16, May 16 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897659.2897665

When faced with information needs, software developers search the web for help, including learning unfamiliar programming concepts and solutions, reminding themselves of solution procedures and syntactic details, confirming hypotheses, and clarifying error messages [5]. Despite the prominence of web search and exploration in software developers' daily work, the tools of software development and web browsing remain disconnected from each other [11, 4]. Retrieval of relevant web pages requires a developer to select a set of keywords, enter them in the browser, evaluate multiple pages returned from a search engine, and rinse and repeat until appropriate resources are found. Clearly, this effortful process can be further optimized for efficiency. In particular, if web pages can be made readily available within the developer's current working context, without the need of forming a query, a developer can stay more focused on their task-at-hand and be more efficient [19, 18, 20].

As illustrated in Figure 1, we focus on integrating into API client code a particular kind of information resource, API usage tutorials. Like related recent research [19, 18, 20], our proposed integration is seamless, without requiring a user to issue queries. Unlike theirs, our work focuses on API tutorials, rather than recommending StackOverflow posts [19, 18] or searching the web for exceptions and error handling messages [20]. API usage tutorials contain high-quality information, such as examples, solution procedures, explanations, and noteworthy API design rationales, that helps a developer understand and maintain API client code. Software developers need to consult API tutorials for learning and understanding, finding new solutions, or debugging. Automatically integrating tutorials with API client code will make it easier for a developer to access the useful programming information contained therein.

Using the Vector Space Model (VSM), we have implemented an API usage tutorial linkage tool called *CUTIe* (Code and Usage Tutorials Integration) in Eclipse. CUTIe automatically converts API client code into queries, freeing the developer from manually forming them. Given that API client code in a method often uses multiple API features, CUTIe partitions a method into multiple snippets and separately links each to their corresponding tutorials. In this way, each snippet will contain fewer features and be more cohesive, leading to higher precision and recall when matching the tutorials. Using a benchmark that is created objectively and with minimal involvement of a researcher's subjective judgment, we evaluate the performance of CUTIe for linking API tutorials. We have also conducted a formative user study to explore how CUTIe may help a developer.



(a) CUTIe partitions Java Swing code into two code groups for frame creation and menubar setup, respectively (highlighted).

(b) Top-5 tutorial links recommended for frame creation (top) and the menu bar logic (bottom).

Figure 1: Java Swing sample code that creates a frame and a menubar and the tutorial links recommended by CUTIe

This work has made the following contributions:

- The implementation of the proposed tutorial linking solution in a tool called CUTIe,
- A code-tutorial traceability link benchmark for two popular APIs, Java Swing and Servlet, and an evaluation of CUTIe's key components using this benchmark [1], and
- A formative user study to explore how CUTIe may actually help a developer.

The remainder of this paper is organized as follows: Section 2 describes the design of CUTIe. Section 3 presents an evaluation of the performance of CUTIe and its components from an IR perspective, and Section 4 presents a formative user study to explore how CUTIe may help a developer. Finally, Section 5 surveys related work, and Section 6 concludes the paper.

2. DESIGN OF CUTIE

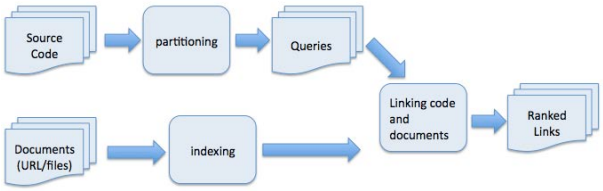


Figure 2: CUTIe's Workflow

As shown in Figure 2, the design of CUTIe forms a pipeline structure. It converts API client code into queries (Section 2.3), computes the similarities between each query and the tutorial documents in a corpus (Section 2.1), and returns to the programmer a ranked list of candidate tutorial documents that may be relevant to the API client code. To improve the query performance and user experience, we propose to partition a method into multiple smaller code groups (Section 2.2). We use the vector space-based models (VSM) [3] to link code units and tutorial documents.

2.1 Corpus and Vector Space Model (VSM)

As shown in Table 1, CUTIe currently supports five Java APIs, Swing, Servlets, JDBC, Java IO, and Java Collections. Our corpus includes high-quality tutorials contributed by both the original vendors of these APIs and other reputable sources. We currently treat each individual HTML tutorial page in our corpus as a document. Given page URLs, CUTIe automatically downloads these pages from the Internet, removes tags using the jsoup HTML parser [2], and saves the resulting text as corpus documents. As the servlet tutorials are stored as presentation slides organized by topics, we manually copy the text for each topic to form a document. The presentation slides are clearly grouped by topics, so it is relatively straightforward to decide the set of slides that should belong to the same document.

We apply the VSM (Vector Space Model) IR method to link API client code and the tutorial documents in our corpus. A typical tutorial page contains both API elements and natural language description. We remove both the standard English stop words and Java keywords from corpus documents. We also perform stemming on the corpus using Porter Stemmer [3]. After these preprocessing steps, the terms extracted from the documents are stored in a $m \times n$ matrix (called *term-by-document matrix* [3]), where m is the number of all unique terms that occur within the documents, and n is the number of documents in the corpus. Entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the i^{th} term in the j^{th} document [3]. Thus in the vector space model, each document as well as the query is represented as a vector. Their similarity is calculated as

[1]Our benchmark and source code are publicly available at http://www.clarkson.edu/~dhou/projects/CUTIe2014.zip. All URLs last verified on 12/16/2015.

[2]http://jsoup.org

[3] http://tartarus.org/martin/PorterStemmer

Table 1: CUTIe Corpus

API Supported	Sources of Tutorials	#Tutorial Doc's (319)
Java Swing (javax.swing, java.awt)	official tutorials (http://docs.oracle.com/.../uiswing) third-party tutorials (http://tips4java.wordpress.com)	97 105
Servlets (javax.servlet)	third-party tutorials (http://courses.coreservlets.com/.../csajsp2.html)	54
JDBC (java.sql, javax.sql)	official tutorials (http://docs.oracle.com/javase/tutorial/jdbc)	25
Java IO (java.io)	official tutorials (http://docs.oracle.com/javase/tutorial/essential/io)	24
Java Collections (java.util)	official tutorials (http://docs.oracle.com/javase/tutorial/collections)	14

the cosine value of the angle between these two vectors, as follows:

$$sim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}| \times |\vec{q}|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} {w_{i,j}}^2} \times \sqrt{\sum_{i=1}^{t} {w_{i,q}}^2}}$$

The weight $w_{i,j}$ for each word in a corpus document is calculated as the multiplication of Term Frequency (tf) and Inverse Document Frequency (idf), where n_t is the number of documents that contain term t, and D the total number of documents in the corpus.

$$w_{i,j} = tf_{i,j} \times idf_{i,j}$$

$$tf_{i,j} = \frac{freq_{i,j}}{max_l freq_{l,j}}$$

$$idf_t = \log \frac{D}{n_t}$$

The weight $w_{i,q}$ for each query word is calculated differently, using the following formula instead [3]:

$$w_{i,q} = \left(0.5 + \frac{0.5 \times freq_{i,q}}{max_l freq_{l,q}}\right) \times \log \frac{N}{n_i}$$

When comparing a query against the corpus, CUTIe does not need to know which APIs the query is related to or uses.

2.2 Partitioning API Client Code

As shown in Figure 2, CUTIe converts source code into a query in order to retrieve relevant documents. Because a method in general uses multiple API features, we believe that instead of treating each method as a single document for querying, it will be more beneficial if we partition each method into multiple smaller groups, where each of them corresponds to ideally one, or at least a smaller number of API features. In this way, it will be easier for a developer to see the structure of a long method. It will also be more likely for documents that are relevant to the used API features to appear in the top region of the ranked candidate list.

CUTIe's partitioning algorithm is based on receiver objects as well as their co-occurrence in the same method call. The main idea is that API calls with distinct receiver objects are first put into different code groups. But if the two receiver objects from two code groups co-occur in a third method call, CUTIe considers that the receiver objects are *related*, and thus merges the two code groups to form a new one. Figure 1a depicts an example for method partitioning, where CUTIe partitions the `createAndShowGUI()` method into two groups, one mainly about a frame, and the other about a menu bar. Notice that `contentPane` is considered *related* to both `frame` and `changeButton` via lines 35 and 37, respectively. Also notice that the code for the frame group is not continuous but gapped. Sections 3.3 evaluates the impact of method partitioning.

CUTIe also sets a limit on the size of the merged group. Two groups will not be merged unless their total size is no larger than the limit. The motivation behind the limit is to avoid too large a group; a group should contain one or few key points such that the user needs to inspect fewer documents and more useful documents appear as top ranked candidates. Currently, we empirically set the limit to ten.

CUTIe uses Eclipse's JDT infrastructure for parsing and code analysis. Specifically, for each new method that it finds, CUTIe traverses all of its method calls in a top-down fashion. It will only further process a method call if the method being called is from one of the packages shown in Table 1. For such a method call, if there already exists a code group that is indexed by its receiver object, the method call will be added to the group. Otherwise, a new group will be created and indexed by the receiver object, and the method call will be added to the new group. Lastly, CUTIe merges all the code groups that are indexed by variables used in the arguments of the method call according to the the merging rule discussed above.

Table 2: CUTIe generates a query for each group of API client code by producing different content for AST node.

Type of AST Node	Contribution to Query
MethodInvocation	type name of receiver expression, if there exists one; JavaDoc for the method
ClassInstanceCreation	type name; names of all super types except Object; JavaDoc for constructor
SuperMethodInvocation	JavaDoc for the method
QualifiedName	JavaDoc for the qualified name from the API
StringLiteral	content words
Comment	content words
Assignment	type of the left-hand side variable; contribution to query by the right-hand side
VariableDeclaration	type name of variable(s)

2.3 Generating Query for API Code Group

Once a method is partitioned into multiple code groups as described in Section 2.2, CUTIe parses each code group to identify AST nodes that are API elements. It then generates a query based on the kinds of AST nodes, according to the rules outlined in Table 2.

The query produced for a code group consists of names and structures of each API element contained therein (types,

method prototypes, and constants). A comment will belong to a code group that contains it or is right below it. Since others have shown that corpus expansion can improve the performance of traceability link recovery [7], we also add into the query the JavaDoc for each involved API element. Because API usage tutorials tend to be general or orthogonal to application domains, but variables tend to be domain-specific, we choose to exclude variable names from the query.

CUTIe uses VSM to compute the similarity between the generated query and corpus documents and returns the top ranked links to the user.

3. EVALUATION

To evaluate CUTIe, we have created two code-tutorial link benchmarks for Swing and Servlet, respectively (Section 3.1). Based on the two benchmarks, we evaluate CUTIe by quantitatively answering the following research questions:

RQ1: To find tutorial links, is it beneficial to partition a method that uses APIs into multiple code groups? What impact does method partitioning have on precision and recall? (Section 3.3)

RQ2: In addition to using the API elements contained by API client code to form a query, is it beneficial to expand the query with the JavaDoc of the API elements? What impact does JavaDoc expansion have on precision and recall? (Section 3.4)

Overall, the results of our evaluation reveal that both method partitioning and JavaDoc expansion are beneficial for improving tutorial linking.

3.1 Benchmarks

To evaluate CUTIe, we have decided to create a benchmark for two of the five APIs that it currently supports, namely Swing and Servlet. We download 84 example projects for Swing from its official tutorial site [4], and 13 example projects for Servlet from the website for a professional training course [5]. We recover code-tutorial links for both APIs based on the information available on these sites.

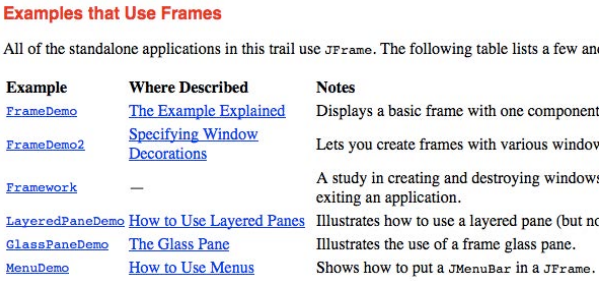
The main reason we choose to use these example projects is that information about code-tutorial links is also available on these sites. While plenty of example projects may be found elsewhere for these two APIs, they are not necessarily accompanied with such worked-out code-tutorial link information. As a result, it can be challenging and time-consuming to establish accurate, objective code-tutorial links for a benchmark. The information on these two sites, on the other hand, makes it much easier for us to objectively prepare a set of code-tutorial links for a benchmark while minimizing subjectivity and inaccuracy.

To explain how we recover the code-tutorial links to form our benchmarks, consider the example shown in Figure 3 for Swing. Figure 3 shows a section from the bottom of a tutorial page for the JFrame component [6]. The left column (**Example**) contains a list of URLs for example projects that contain code that uses the component described by the current tutorial page, the middle column (**Where Described**) provides a tutorial link that describes portion of

[4]http://docs.oracle.com/javase/tutorial/uiswing/examples/components

[5]http://courses.coreservlets.com/Course-Materials/csajsp2.html

[6]http://docs.oracle.com/javase/tutorial/uiswing/components/frame.html



Examples that Use Frames

All of the standalone applications in this trail use `JFrame`. The following table lists a few an

Example	Where Described	Notes
`FrameDemo`	The Example Explained	Displays a basic frame with one component
`FrameDemo2`	Specifying Window Decorations	Lets you create frames with various windov
`Framework`	—	A study in creating and destroying windows exiting an application.
`LayeredPaneDemo`	How to Use Layered Panes	Illustrates how to use a layered pane (but nc
`GlassPaneDemo`	The Glass Pane	Illustrates the use of a frame glass pane.
`MenuDemo`	How to Use Menus	Shows how to put a `JMenuBar` in a `JFrame`.

Figure 3: Information about code-tutorial links can be recovered from the Swing tutorials.

the example project shown on the left, and the right column (**Notes**) contains specific information on exactly which aspect of the example projects is described in the tutorial link provided by the middle column. Therefore, from this section of the tutorial, we were able to collect two code-tutorial pairs for each row in Figure 3. The first pair is between the code fragment in the project that uses the current component, and the link for the current page. To find the second pair, we first used the information shown in Figure 3 to locate the relevant region of code from the project in the first column (Example), which is supposed to implement what is described in the third column (Notes). We recorded the code and the corresponding tutorial link (the middle column Where Described) in a spreadsheet as part of our benchmark. The servlet benchmark was similarly prepared.

As shown in Table 3, the benchmark for Swing contains 347 tutorial links for 251 of the 1,016 Swing code groups. In general, there are two to three links worked out for each Swing code group. The benchmark for Servlet is much smaller, containing 50 links, one for each of 50 of the 186 Servlet code groups. The benchmarks were prepared by the first author independently by inspecting all tutorial pages such as the one shown in Figure 3. They were then checked by the second and third authors. Agreements were initially reached on 95% of these links, with a few cases of disagreements in our treatments. All the disagreements were resolved collectively. The high percentage of initial agreements (95%) reflects the objective nature of our benchmark links.

3.2 Evaluation Metrics

Table 3 depicts our evaluation results. We calculate two measures for evaluating CUTIe's performance, MAP (Mean Average Precision) and AR (Average Recall). MAP is a favored IR metric for ranked retrieval [15]. Furthermore, it will take longer for a user to find a relevant link if it is present further down in the ranked link list. Therefore, as a crude estimate for the effort a developer would spend before finding the needed API tutorial links, for those code groups for which CUTIe returns the expected links, we also collect the ranks for the expected links in the ranked link list and calculate an average rank and standard deviation.

We calculate the MAP for the top n ranked results, by setting n to 5 and 10.

$$MAP(Q)@n = \frac{\sum_{q \in Q} AP(Relevant(q,n))}{|Q|}$$

where Q is the set of queries evaluated against the benchmark. $Relevant(q,n)$ is the set of relevant links in the top n ranked results for query q. $AP(R)$ is the average precision for a set of relevant links R, which is in turn defined as follows:

Table 3: Benchmarks and evaluation results

Statistics	Java Swing	Java Servlet
#Evaluation projects	84	13
#Methods	893	192
#Methods using APIs	538	100
#Code groups	1,016	186
#Groups with known expected links / #expected links	251 / 347	50 / 50
#Methods with known expected links / #expected links	155 / 347	50 / 50
MAP@5 (**+partitioning +JavaDoc**)	0.784	0.852
Recall@5	0.872	0.98
Average rank / stddev for expected links	1.68 / 1.0	1.42 / 0.9
MAP@5 (**+ partitioning -JavaDoc**)	0.836	0.848
Recall@5	0.861	0.98
Average rank / stddev for expected links	1.6 / 1.1	1.42 / 0.9
MAP@5 (**-partitioning +JavaDoc**)	0.785	0.888
Recall@5	0.841	0.98
Average rank / stddev for expected links	1.8 / 1.1	1.2 / 0.52
MAP@10	0.768	0.852
Recall@10	0.905	0.98
Average rank / stddev for expected links	2.39 / 2.1	1.2 / 0.52

$$AP(R) = \frac{\sum_{r \in R} Precision(Rank(r))}{|R|}$$

where $Precision(k)$ represents the proportion of correct links contained by the top k results.

Lastly, we calculate the average recall over the set of all queries Q:

$$Recall@n = \frac{\sum_{q \in Q} Recall(q)}{|Q|}$$

where $Recall(q)$ is the recall for query q, defined as the ratio of relevant links over the set of all relevant links for q.

3.3 Impact of Method Partitioning

In this section, we evaluate the impact of method partitioning on tutorial linking performance (precision, recall, and the average rank of matching links).

Using our benchmarks described in Section 3.1, we have evaluated the impact of method partitioning versus non-partitioning (using a whole method as a query) on tutorial linking performance. As shown in Table 3, the performance metrics for method partitioning and non-partitioning are labeled as **+partitioning+JavaDoc** and **-partitioning +JavaDoc**, respectively. Consider the evaluation results for Java Swing's top-5 ranked links first. Although the two approaches have produced almost the same value of Mean Average Precision for the top 5 links (0.784 and 0.785), the recall for method partitioning is higher than non-partitioning (0.872 and 0.841, respectively). The method partitioning approach also produces a lower average rank (1.68) with smaller variance (standard deviation 1.0) than the non-partitioning strategy (1.8 / 1.1).

When considering only the top-5 ranked links, why does method partitioning result in higher recall than non-pa- rtitioning? This is probably because when the non-partitioning approach is used, the larger query produced for a method contains more relevant search words than the ones produced by method partitioning. As a result, in the results of the non-partitioning approach, there will be more matching links competing for the top five positions in the ranked list, and it becomes more likely for otherwise relevant links being pushed away from the top five positions. To shed more light on this observation, we calculate MAP@10, $Recall$@10, and average rank for the Swing benchmark. As shown by the last three rows in Table 3, as expected, the top-10 evaluation results indeed yield a lower mean average precision and a higher average recall than the top-5.

In general, to save time spent in inspecting the ranked results, a developer will probably focus on checking only a small number of results. Therefore, we prefer a strategy that performs better for a shorter list of ranked results.

Therefore, we conclude that method partitioning for CUTIe is an effective strategy for improving linking performance.

3.4 Impact of JavaDoc Expansion

We have also evaluated the impact that query expansion with JavaDoc may have on linking performance. As shown in Table 3, the performance metrics for method partitioning with and without JavaDoc expansion are labeled as **+partitioning+JavaDoc** and **+partitioning-JavaDoc**, respectively. As expected, document expansion results in a lower mean average precision (0.784 versus 0.836), a slightly lower average rank (1.68 versus 1.6), but a higher recall (0.872 versus 0.861) than without expansion. Our result is consistent with that of Dasgupta et al. [7]. Since in our particular use case of linking tutorials, we have decided to focus on only the first five results to reduce a developer's inspection effort, increasing recall is more valuable in our case.

Therefore, we conclude that JavaDoc expansion is an effective strategy for improving tutorial linking performance.

4. A FORMATIVE USER STUDY

CUTIe is motivated by the following observations on a developer's online information seeking behavior. When seeking information online to complete a task, a developer generally follows the following process: (1) Formulate and issue a query, (2) Inspect returned links to assess their relevance, (3) If not relevant, reformulate the query and repeat the process. For an atomic task, the developer normally needs to carry out only one iteration of search. But for a compound task that consists of multiple subtasks, she needs to first find out what are the subtasks and then carry out one search process for each subtask. Finding out the subtasks can be a major challenge by itself. Given that many software engineering tasks are compound and that a developer may have to reformulate queries and search repeatedly, it is not too hard to see why online information seeking can be time-consuming.

We hypothesize that CUTIe would help by automatically returning links that contain information about: (1) how a major task is divided into multiple subtasks (that is, a solution procedure), and (2) links for each subtask. Furthermore, CUTIe presents all of these information in one popup

pane (Figure 1b), freeing the developer from the manual task of collection and organization, so she can focus on analyzing the information. We also hypothesize that CUTIe may return links that are hard to find when searched manually.

To further explore these hypotheses, we have conducted a small, formative user study for CUTIe in a laboratory. Note that with such a formative study, our goal is to seek initial evidence that potentially support our hypotheses, rather than concluding with statistical significance.

4.1 User Study Setup

Our study required a subject to solve three programming tasks in order. The first two tasks asked the subject to enhance two Swing programs to properly close an application (**T1**), and to pop up a new window (**T2**), respectively. The third task (**T3**) was a compression question about a paint program, asking the subject to explain how scroll bars are automatically created when the mouse-based drawing is extended beyond the canvas window. The paint program had been used by prior user studies in software maintenance [14] and clone management [13].

Our tasks were motivated by our hypotheses discussed above. Specifically, **T1** and **T2** consist of four and three subtasks (not shown), respectively. **T3**'s solution is a single piece of fact that is hard to find. Before we started to invite subjects into our laboratory, we worked out solutions for the three tasks and created a grading rubric, which included a breakdown of subtasks and a solution link for each subtask.

We invited three senior undergraduate students to participate in our study. Although all three of them have previously taken lessons in Swing, we found that subject **S2** was more experienced than **S1** and **S3**. We followed the task assignment shown in Table 4. When using CUTIe, subjects were asked not to search online. We introduced the task requirements before the subject started, both orally and in writing. Subjects took between 1.5 to 2 hours to complete all three tasks. To help us understand a subject's problem solving process, we asked them to speak aloud. We recorded a subject's computer screen and audio.

Table 4: Task assignment. Y / N: Using CUTIe or not.

Subject	T1	T2	T3
S1	Y	N	Y
S2	N	Y	Y
S3	Y	Y	N

4.2 Summary of Findings

We reviewed the recorded video to identify the amount of time a subject took to find a useful link, which is defined as the time elapsed between the announcement of a goal and the moment when a useful link was found. We also counted the number of queries created when not using CUTIe, as well as the number of useful links inspected when using CUTIe.

Table 5 shows the average time taken to find a useful link. While our subjects found similar numbers of useful links with or without using CUTIe, using CUTIe helped to cut the time for finding a useful link by at least half. Multiplied by a developer's high number of online search, this difference can result in significant saving of time.

Without using CUTIe, a subject spent more time in formulating/refining queries. But with CUTIe, the time spent on query formulation was saved. Consider task **T1** as an example. **S2** (not using CUTIe) issued a total of seven queries only to find three useful links, whereas **S1** (using CUTIe) directly inspected the three useful links from CUTIe.

Table 5: Average time for finding useful link.

Task	CUTIe	no CUTIe
T1	1 m 8.5 sec	2 m 20 sec
T2	1 m	2 m
T3	1 m 45 sec	>10 m (S3 gave up)

On the other hand, when solving task **T2** without using CUTIe, **S1** issued a total of nine queries but was not able to find enough useful links to finish. This would indicate that CUTIe's links were more critical for a novice such as **S1** who would have challenge in formulating an effective search plan.

Lastly, as shown in the bottom right in Table 5, when a link is hard to find manually, without CUTIe, subject **S3** actually failed to find the answer for task **T3**. In contrast, both **S1** and **S2** took less than two minutes to find and explain to us the answer link for task **T3**.

During a brief exit interview, all three subjects expressed that they highly liked CUTIe. They found CUTIe easy-to-use, unobtrusive, and would like to use it regularly.

5. RELATED WORK

Information retrieval techniques have been widely used to link source code units (e.g., classes, functions, or files) and various forms of documentation (traceability link recovery) [9]. They are also used in feature location [10]. For example, Antoniol et al. [1] evaluate the performance of both a probabilistic model and Vector Space Model (VSM) for linking classes to sections of manuals generated from the code, while Marcus and Maletic [16] use latent semantic indexing (LSI). The units of code for queries in these work are either classes or files, whereas CUTIe proposes to partition a method into multiple smaller code groups in order to improve the quality of matching API usage tutorials.

In the context of linking between requirements and code, Dasgupta et al. [7] evaluate the performance of combinations of several IR methods and corpus treatment methods. In particular, they show supporting evidence that both API elements and corpus expansion with API documentation can act as *semantic anchors* for improving the performance of traceability recovery. A difference is that their work focuses on the general problem of tracing code and requirements while we instead on connecting API client code and usage tutorials. Unlike tracing requirements, CUTIe's retrieval performance is impacted more by API terms than terms specific to the application domains to which the APIs are applied.

Traceability link recovery has also found applications in other software development tools. For example, De Lucia, Di Penta, and Oliveto [8] propose COCONUT, an IR-based system that helps developers select the most meaningful identifier names, which are consistent with the domain terms found in the high-level artifacts, e.g., requirements.

Sawadsky, Murphy, and Jiresal [22] present a study on how a developer revisits web pages to seek help for programming tasks and a tool called Reverb that takes advantage of page revisitation to improve the accuracy of recommendation. Reverb recommends web links based on the similarity between the code fragment that is currently visible in a developer's editor and a repository of web pages that the developer has visited so far. Unlike CUTIe, which fo-

cuses on APIs, Reverb does not distinguish between code elements that are part of the projectÕs internal code base and those that belong to external libraries. Moreover, instead of method partitioning as done in CUTIe, Reverb's query contains only portion of code that is visible in the code editor. Lastly, Reverb's query contains only the names of program elements but without JavaDoc expansion.

Recently, there have been several studies focusing on automatically identifying code elements from within natural language texts, such as developer emails [2] and forum discussions [6, 21, 23]. Bacchelli, Lanza, and Robbes [2] use regular expressions to match text terms that are actually class/function names. They find that the performance of the lightweight regular expressions for traceability link recovery is similar to two other IR methods (VSM and LSI). Another line of work relies on more sophisticated parsing techniques to obtain more precise and finer-grained information for code elements (fully qualified names) that appear in informal documents, e.g., PPA, Partial Program Analysis [6] and island parsers [21]. Subramanian, Inozemtseva, and Holmes' latest work [23] extends this line of techniques to support multiple APIs and to a dynamically typed language JavaScript. These proposed techniques can be used to integrate code examples in natural language documents and API documentation in both directions [23]. The challenge lies in accurately resolving the code elements embedded in the unstructured texts. We instead focus on establishing traceability links between API client code and API usage tutorials using an IR method. We use a region of API client code as a query, whereas the other work use the name of a revolved code element as a query.

6. CONCLUSION

We study an approach to automatically linking API client code with usage tutorials. We partition a method into multiple smaller code groups and converts the API elements in each group and their JavaDoc into a query. Given such a query, our approach uses VSM to rank and recommend the suitable tutorials for each group. Based on an evaluation with two benchmarks that we have established, we conclude that the proposed strategies of method partitioning and JavaDoc expansion are indeed beneficial for tutorial linking. A formative user study indicates that our approach not only is accurate, but also saves time in formulating queries. We find that our approach can be particularly helpful when novices maintain API client code. We believe that it can also be a useful assistant to the experts.

7. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *ICSE*, pages 375–384, 2010.

[3] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[4] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *CHI*, pages 513–522, 2010.

[5] 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. In *VL/HCC*, pages 1589–1598, 2009.

[6] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *ICSE*, pages 47–57, 2012.

[7] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *ICSM*, pages 320–329, 2013.

[8] A. De Lucia, M. Di Penta, and R. Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Software Eng.*, 37(2):205–227, 2011.

[9] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk. Information retrieval methods for automated traceability recovery. In *Software and Systems Traceability*, pages 71–98. 2012.

[10] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[11] M. Goldman and R. C. Miller. Codetrail: Connecting source code and web resources. *J. Vis. Lang. Comput.*, 20(4):223–235, 2009.

[12] D. Hou and L. Li. Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions. In *ICPC*, pages 91–100, 2011.

[13] P. Jablonski and D. Hou. Aiding software maintenance with copy-and-paste clone-awareness. In *ICPC*, pages 170–179, 2010.

[14] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE*, pages 126–135, 2005.

[15] D. Lawrie. Evaluation metrics. http://distat.unimol.it/TAinSM2012/slides/dawn.pdf.

[16] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–135, 2003.

[17] S. L. Pfleeger and J. M. Atlee. *Software engineering - theory and practice (4. ed.)*. Pearson Education, 2009.

[18] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *CSMR*, pages 57–66, 2013.

[19] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *MSR*, pages 102–111, 2014.

[20] M. Rahman, S. Yeasmin, and C. Roy. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *CSMR-WCRE*, pages 194–203, 2014.

[21] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *ICSE*, pages 832–841, 2013.

[22] N. Sawadsky, G. C. Murphy, and R. Jiresal. Reverb: Recommending code-related web pages. In *ICSE*, pages 812–821, 2013.

[23] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *ICSE*, pages 643–652, 2014.