Live API Documentation

Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
s23subra,lminozem,rtholmes@uwaterloo.ca

ABSTRACT

Application Programming Interfaces (APIs) provide powerful abstraction mechanisms that enable complex functionality to be used by client programs. However, this abstraction does not come for free: understanding how to use an API can be difficult. While API documentation can help, it is often insufficient on its own. Online sites like Stack Overflow and Github Gists have grown to fill the gap between traditional API documentation and more example-based resources. Unfortunately, these two important classes of documentation are independent.

In this paper we describe an iterative, deductive method of linking source code examples to API documentation. We also present an implementation of this method, called Baker, that is highly precise (0.97) and supports both Java and JavaScript. Baker can be used to enhance traditional API documentation with up-to-date source code examples; it can also be used to incorporate links to the API documentation into the code snippets that use the API.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Languages, Experimentation

Keywords

Source code examples, source code search, documentation

1. INTRODUCTION

Using third-party libraries can greatly reduce the effort required to develop a new system. Unfortunately, understanding how to use these libraries correctly can be difficult. While the application programming interface (API) documentation can be a valuable means of understanding the library, it can be insufficient on its own. One of the main issues is that the documentation is often out of date. Recent work has confirmed the popular belief that writing documentation and keeping it up to date is very difficult [8, 9]; consequently, developers ignore the documentation that does exist and declare that "code is king" [17].

As a result of this situation, developers often turn to online resources such as Stack Overflow. Parnin et al. have previously studied online resources and have found that they do a good job of covering APIs [15]. In fact, they found that 87% of Android classes were referenced in Stack Overflow answers. Unfortunately, there are rarely links between online resources and official API documentation: the official documentation does not link to the examples that could help developers, and the examples rarely link to the documentation.

Previous work has tried to identify source code references within non-code resources (e.g., [2, 14, 9, 16]). Detecting these references is a first step toward linking them to the relevant API documentation. Unfortunately, these approaches have several limitations. Some systems explicitly ignored external references, meaning that the target system for an analyzed document must be specified [9]. Others only returned partially qualified names, which are insufficient for documentation linking [16]. Our initial investigation found that this approach could work for a subset of Java programs [18]. However, none of the previous approaches worked for dynamically-typed languages.

Our paper extends previous work in this area by using a constraint-based technique to uniquely identify fine-grained type references, method calls, and field references in source code snippets with high precision. We demonstrate the generality of the approach by providing implementations for both typed (Java) and dynamic (JavaScript) languages. We also evaluate the ability of the approach to correctly link code to documentation.

More specifically, the contributions of this paper are as follows:

- A constraint-based, iterative approach for determining the fully qualified names of code elements in source code snippets. This approach works with both statically and dynamically typed languages.

- A prototype tool that implements this approach and uses the results to automatically create bidirectional links between documentation and source code examples by marking up HTML using a web browser extension.

Section 2 presents a scenario that motivates our approach and demonstrates the kinds of links it can identify. The approach and our implementation of it are explained in detail in Sections 3 and 4. Section 5 then presents our evaluation of Baker. The documentation linking prototype is described in Section 6, followed by discussion in Section 7. Related work is described in Section 8; Section 9 concludes the paper.

2. SCENARIO

Consider the Java code snippet shown in Figure 1. This snippet (pertaining to a library called GWT) was posted to Stack Overflow to assist a developer who did not understand how to manipulate the state of `History` objects. The figure contains a number of bolded elements. These are the types and methods that our tool, Baker, can uniquely link to the API; i.e., the elements for which it can determine a fully-qualified name. With this information we can automatically augment the HTML version of the official API documentation for `History` by dynamically injecting the code example into the web page. We can also inject the links to the official API into the Stack Overflow post; these two additions to the documentation would make it easier for developers to learn how to use this class.

```
1   public FirstPanel() {
2     History.addHistoryListener(this);
3     String token = History.getToken();
4     if (token.length() == 0) {
5       History.newItem(INIT_STATE);
6     } else {
7       History.fireCurrentHistoryState();
8     }
9     .. rest of code
10  }
```

Figure 1: A Java code snippet representing a Java API usage. Baker can associate each of the bolded terms with a fully qualified name; this information can be used to include the code example in the API documentation.

Next, consider the JavaScript snippet in Figure 2, where a developer is trying to make a web app that can take a photo and inject it into an element in an HTML document. This example interacts with the JavaScript DOM (getElementById), takes a photo using the Cordova project (getPicture), and uses JQuery to detect when the the photo should be taken ($ and on). For each of these method references Baker can can identify the API that it is from.

The code snippets in Figures 1 and 2 were both submitted as the correct solution to problems developers posted on Stack Overflow. Since Stack Overflow posts are ranked, and accepted answers are known to have solved a real problem, Stack Overflow is a good source of high quality code snippets that demonstrate the correct usage of many APIs. Increasing the integration between these examples and the official API documentation will make documentation maintenance easier and increase the visibility and accessibility of the official API documentation within source code examples.

```
1   $("#addphoto").on('click',
2     function() { useGetPicture();}
3   );
4   function useGetPicture() {
5     var cameraOptions = { ... };
6     navigator.camera.getPicture(onCameraSuccess,
7       onCameraError, cameraOptions);
8   }
9   function onCameraSuccess(imageData) {
10    var image = document.getElementById("..");
11    image.src = "data:image/jpeg" + imageData;
12  }
13  function onCameraError(message) {
14    alert("Failed: " + message);
15  }
```

Figure 2: A JavaScript code snippet containing Cordova, JQuery and JavaScript DOM API usage. Each of the bolded elements can be linked back to the relevant API documentation.

3. APPROACH

Identifying API elements in code snippets requires the ability to parse these snippets. This is more difficult than parsing full files because code snippets can be ambiguous. Dagenais and Robillard highlighted four kinds of ambiguity that can hamper the identification of elements [9]; two of these were specific to the plain-text analysis they were performing, while the other two were more generally relevant. These two were *declaration ambiguity* and *external reference ambiguity*.

Declaration Ambiguity. Snippets are, by definition, incomplete fragments of code. That is, snippets might not be embedded in methods or classes, they may reference fields whose declaration is not included, and their identifiers are largely unqualified. In source code examples this is often exacerbated by authors ending lines with '...' or using code comments to describe parts of the functionality that are elided.

External Reference Ambiguity. Source code examples frequently refer to external identifiers; for example, Java snippets frequently reference types from the JDK. While a previous study [9] dealt with external references by eliding everything that was not from a pre-specified library, we designed Baker to handle these kinds of ambiguities. We accomplished this by using an oracle: a large database containing information about the code elements in popular APIs. When Baker encounters an ambiguous code element, such as the `History` class in Figure 1, it uses the oracle to identify the possible types of the code element. In this case, there are 58 `History` classes in the oracle, but by using information from other parts of the code snippet, we can identify which of the 58 is the correct one. Section 4 will present more information about how the oracle is constructed, what it contains, and how much of a problem ambiguity really is.

3.1 Deductive Linking

Baker handles declaration ambiguity and external reference ambiguity through a process we call deductive linking. At a high level, it generates an incomplete abstract syntax tree (AST) for the code snippet being analyzed, then uses information from the oracle to deduce facts about the AST. We perform this deduction step iteratively since each phase can reveal new facts that can be used in subsequent phases.

More specifically, in each phase, Baker performs a depth-first traversal of the AST and examines all nodes involved in declarations, invocations, and assignments. When an AST node of interest is encountered, Baker builds a list that represents the potential matches for that element from the oracle. As the traversal takes place, we track the scope of all data being used to ensure that nodes are only combined if allowed by the scoping rules of the language.

Once the entire AST has been traversed, the process starts again; information uncovered in the previous iteration can now be used to further restrict the lists of candidate elements. In theory, this iteration continues until either all elements are associated with a single fully qualified name (FQN), or an iteration fails to improve the results for any element. In practice we find that very few iterations are typically needed.

While the goal of the approach is to identify the sole fully-qualified element that a given identifier can represent, sometimes there is not enough information to choose from a set of candidates. In this case, Baker returns a match with *cardinality* (c) greater than 1. When this happens, we can either return all candidate elements or simply report that a unique match cannot be found. Sometimes a specific FQN cannot be identified for an element, but examining the set of candidates reveals that they are all related – for example, if one of the elements is a supertype for all other elements in the set. In this case, we report the supertype as the match and elide the concrete subtypes from the results.

While the core functionality of Baker is the same for both Java and JavaScript, some language-specific functionality is needed. Baker is thus implemented as two different linking engines. One engine computes links between Java snippets and API documentation while the other computes links between JavaScript snippets and API documentation. JavaBaker leverages the static structural relationships present in Java code to deduce the correct links for code elements. Though JavaScript code lacks static type information, JSBaker takes advantage of the fact that JavaScript library developers are wary of naming conflicts; they therefore usually use an object literal as an implied namespace and make functions and variables properties of the object literal. This allows us to link JavaScript objects and functions back to the specific API they reference.

3.1.1 JavaBaker Example

To create ASTs from Java snippets we built a parser on a headless version of Eclipse. Since the Eclipse parser is robust to badly formed input, it was able to manage many of the problems associated with source code snippets. However, before any snippet could be parsed, we had to determine if the snippet was surrounded by valid class and method declarations. If it was not, we added dummy class and method wrappers to allow parsing. We also built a web service for the parser that enables us to use HTTP POST to send snippets of code to JavaBaker and get a JSON response with the results.

To describe the JavaBaker engine more concretely, we will revisit the Java code fragment from Figure 1 and describe how it would be analyzed. Since the fragment in this case does not contain a class declaration, it is wrapped in a synthetic class before the parsing process begins.

1a. `History.addHistoryListener(this)` on line 2 is the first expression we encounter that requires analysis. From the oracle we retrieve all elements called `History`, corresponding to the left-hand side of this expression. These 58 candidate types are recorded for `History`, along with the scope of the method call.

1b. Next, Baker considers addHistoryListener(this). Since this method is being invoked on `History`, we examine its 58 candidate types to see which ones contain a method called `addHistoryListener(...)` that take a single object parameter. This results in 4 candidate methods. Since Baker is still evaluating the expression, the left-hand side (`History`) is updated to reflect the number of candidates (reduced from 58 to 4).

1c. For the assignment on line 3, Baker considers the right-hand side first. Here, Baker assumes that the name `History` refers to the same `History` class as the reference on line 2 and subsequently starts using its 4 candidates; this is because in Java conflicting names in the same class must be fully qualified. Evaluating the `getToken()` method, `History` is further reduced to 2 candidates; `getToken()` also has cardinality 2. Since we have not uniquely identified `getToken()` we cannot yet use its return type to determine the type of `token`.

1d. The same procedure continues for lines 4 through 10. On lines 5 and 7 the scope being assigned to the `History` nodes is updated to reflect the inner block being analyzed. After the whole snippet has been analyzed, Baker iterates again.

2a. Once Baker returns to line 2 `History` can be identified as `com.google.gwt.user.client.History` because the method call constraints from lines 3, 5, and 7 leave only one possible candidate. All other `History` references are updated to the same FQN, as are the method calls being made on it.

2b. The return type of the now-resolved `getToken()` can be used to confirm that token is of type `java.lang.-String`.

2c. Since all elements have been fully qualified, Baker does not need to do another pass.

Baker uses a number of relationships to help identify elements when those relationships are available. They include import statements (rarely present in example code), cast expressions, field declarations, return statements, `super` invocations, `extends/implements` relationships, and parameter types.

3.1.2 JSBaker Example

JavaScript snippets are parsed by the ESPRIMA[1] parser. ESPRIMA is very tolerant of malformed input because it is frequently found in JavaScript code. Before the analysis takes place, the code snippet is wrapped in a function declaration if one is not already present. As with JavaBaker, we implemented JSBaker as a web service so code snippets could easily be parsed from a variety of applications.

---

[1] `http://esprima.org/`

We revisit the code snippet in Figure 2 to describe the deductive linking process in detail.

1a. Line 1 contains two function expressions, `$` and `on`. Checking the oracle, we find only one instance of `$` from jQuery. Because we can uniquely identify `$`, we use this fact while examining any other methods in the call chain. In this case, there is only one jQuery method called `on` so it matches correctly on the first try (even though there are three `on` methods in the oracle). This library preference is only used for chained calls.

1b. The next function expression encountered is a call to `useGetPicture()`. The oracle does not contain a result for this identifier.

1c. When it encounters the local function definition for `use-GetPicture()`, Baker records that this function is locally defined, rather than being an external function.

1d. On line 5, the scope of `cameraOptions` is recorded to ensure that any constraints applied to it do not 'leak' outside its scope. When Baker reaches the function expression `getPicture`, the oracle is queried for methods with the same name taking at least three variables; this returns only one possible match. This match is called `navigator.camera.getPicture` in the oracle so the full expression ends up matching.

1e. The next function expression is on line 10; `getElement-ById` matches 3 functions. Next, Baker checks to see if any of these are defined as `document.getElement-ById`; this results in a single match.

1f. Some JavaScript libraries are augmented with return type information. In this case, the oracle knows that `document.getElementById` returns an `Element`; as such, `image` is annotated with 68 possible types. On line 11, the reference to the property `image.src` further reduces the number of possible types to three. It is important to note that even if the returned object did not have a `src` property, this would be valid JavaScript code – a new property would be added to the object. Baker assumes that library code will not be dynamically augmented in this way.

1g. The function call to `alert` matches two elements, `win-dow.alert` and `notification.alert`. Since `window` is the default namespace for JavaScript executed in the browser, we link `alert` to `window.alert`.

2a. In the second iteration we link the call to `useGetPic-ture` in line 1 to the function declaration on line 4.

2b. Next, we update the link between `onCameraSuccess` and `onCameraError` on lines 5 and 7 to match the function declarations on line 9 and 13.

2c. No new information has been learned about the exact type of `image` on lines 11 and 12; as such, this element is left with a cardinality of 3. That said, if the developer were interested in this element they could be given the option to choose between `HTMLInputElement`, `HTMLImageElement`, and `HTMLScriptElement`. Given the `data:image/jpg` string on line 11, the developer could likely make the right choice.

4. ORACLE GENERATION

As demonstrated by the two detailed examples in the previous section, Baker's oracle is key to its success. In this section, we explain why the use of an oracle is necessary, then describe how we created our Java and JavaScript oracles.

4.1 Why Use an Oracle?

For some traceability tasks, an oracle is not necessary. For example, Rigby and Robillard developed a tool that can extract the code elements contained in various documents with high precision without an oracle [16]. However, without an oracle, it is generally impossible to identify the fully qualified names of the code elements in a snippet. These fully qualified names are essential to documentation linking tasks; thus, an oracle is required. As we will see in the remainder of this section, this is not a difficult requirement to satisfy: the initial oracle can be generated fairly quickly and subsequent updates can be done dynamically.

4.2 Oracle Generation

We built oracles for both Java and JavaScript. The oracles are implemented as web services, allowing them to be updated dynamically by any user or program. New development resources can be `POST`ed to the service and are automatically analyzed and incorporated into the oracle. Similarly, any program or linker can query the service to determine what elements are present that meet a certain set of constraints.

4.2.1 Java Oracle

The Java oracle is a database containing class, method and field signatures. We chose a graph database, Neo4j, for this purpose; the graph data structure makes it easier to represent the hierarchies between code elements that an object-oriented language like Java offers. Since Java is statically typed, we include full type information in the database including the types of classes, fields, return types, and parameters. The database also contains links for all inheritance relationships so we can more effectively handle polymorphism in example snippets.

The Java oracle can be dynamically updated by adding an appropriate JAR. We built a simple web service that allows a JAR file to be uploaded and automatically analyzed and added to the oracle by a tool called Dependency Finder[2]. This tool identifies the class, method and field signatures from the *.class* files contained in the JAR. These are then added to the existing Neo4j oracle graph. We chose to implement this web service to make the oracle update process as seamless as possible.

To bootstrap our oracle we used Daniel German's list of signatures extracted from over 1.5 million classes in the Maven repository. In total, it contains 14 million method signatures and 3 million field signatures [10].

4.2.2 JavaScript Oracle

The JavaScript oracle is built by statically analyzing the source files of the libraries to be included. We use ES-PRIMA[3] to parse the source code of each library. Since JavaScript libraries are frequently minified and obfuscated,

---

[2] `https://bitbucket.org/rtholmes/depfind_uw`
[3] `http://esprima.org/`

Table 1: Number of unique fully qualified names, partially qualified names, and unqualified names in the Java oracle for each kind of indexed element. 89% of method names are ambiguous if not fully qualified while 33% of partially qualified method names are still ambiguous.

	Types	Methods	Fields	Total	Average
Fully Qualified Names	1,646,650	14,206,944	3,149,206	19,002,800	/
Partially Qualified Names	/	9,455,644	2,571,384	12,027,028	/
Unqualified Names	1,121,887	1,600,053	1,115,099	3,837,039	/
% Ambiguous Partially Qualified Names	/	33%	37%	/	37%
% Ambiguous Unqualified Names	32%	89%	65%	/	80%

we used the 'source' version of each library, i.e., the version before these transformations were applied. The ESPRIMA parser returns a JSON representation of the AST. From this, we identify all of the 'FunctionExpression' and 'FunctionDeclaration' nodes. We traverse the path to each of these function nodes to identify the namespace hierarchy that would need to be used to access these functions. Since object assignments in JavaScript are pass-by-reference, an additional traversal of the AST is performed to map non-trivial and indirect Function Expression assignments.

As an example, consider the snippet of code from Backbone.js in Figure 3. A first pass is done to fetch all FunctionExpression variables, in this case, *extend*. An additional traversal of the AST is performed to identify transitive aliases like *History.extend* and *View.extend*, which inherit all properties of the *extend* object. In this second pass, all possible aliases are traced back and are subsequently entered into the oracle.

JavaScript libraries often make calls to external libraries in their source code. Function objects are passed as parameters to these external libraries to be modified and assigned to other objects. We follow these assignments one level deep, but since JavaScript does not have type information for objects returned from functions, our ability to reason about these assignments degrades after two assignments. For this reason, we stop after two passes.

```
1   _.extend(History.prototype, Events, {
2
3       getHash: function(window) {
4               ...
5           },
6   });
7   var extend = function(protoProps, staticProps){
8       ...
9   };
10  View.extend = History.extend = extend;
```

Figure 3: A snippet from the source code of Backbone.js

Generating the JavaScript oracle was somewhat harder than generating the Java oracle, since the dynamic nature of JavaScript makes it difficult to identify all method declarations by static analysis of source code. To overcome this, we take advantage of JSDoc[4] annotations (and other similar documentation tools) in the library source code whenever they are available.

Another challenge is that JavaScript is not annotated with visibility (e.g., `public` and `private`). This makes it difficult to differentiate between the public API and those internal methods that are not meant for public access. Including the internal methods in the oracle may make Baker slightly less accurate. Source code snippets are unlikely to use the internal methods, so these elements will rarely be matched, but their existence in the oracle may increase ambiguity, making it harder to match public API elements with the same name.

In this study, we populated the oracle with source code from seven different libraries, including the core JavaScript API. These libraries contain over 1,600 API object properties including functions, properties and event handlers. These libraries were chosen by gauging the popularity of the libraries' Github repositories and related activity on StackOverflow.

To dynamically add new JavaScript libraries to the oracle, we rely on *npm*[5], which is a package manager for *node.js*. We use *npm* to fetch the source code of said library. Baker then analyzes the source code to populate API methods and properties and add them to the existing database of API signatures.

4.3 Naming Ambiguities

Fully qualified names are heavily used in computer programs to reduce the likelihood that program identifiers (e.g., type names, method names, and field names) will conflict between different programs and libraries. For example, while `Log` is a common unqualified type name (occurring 284 times in the Java oracle), developers use fully qualified names to identify the `Log` they are interested in (such as `org.apache.tomcat.util.log.Log` vs. `org.eclipse.jetty.util.log.Log`).

Method and field identifiers can be partially qualified if their identifier contains the type in which they are declared. For example, while the unqualified method name `getId()` occurs 27,434 times in the oracle, `org.neo4j.graphdb.Node.getId()` and `jsx3.xml.Node.getId()` can be used to differentiate between two different `getId()` declarations. Class names cannot be partially qualified without considering either package identifiers or namespaces.

Naming ambiguity is common; Dagenais and Robillard previously found that 89% of method names are ambiguous and the average method name conflicts with 13 other methods [9]. We extended their result to 1.6 million types and extended the analysis to include types, methods, and fields. We also investigated the differences between fully

---

Table 2: Baker's overall Java precision (0.98) and recall (0.83). Only exact matches ($cardinality = 1$) were considered.

System	TP	FP	$\text{FN}_{c=1}$	$\text{FN}_{c>1}$
Android	40	1	8	1
GWT	43	0	7	0
Hibernate	37	0	13	0
Joda Time	44	3	3	0
XStream	40	0	10	0
Total	204	4	41	1

Table 4: Baker's overall JavaScript precision (0.97) and recall (0.96). Only exact matches ($cardinality = 1$) were considered.

System	TP	FP	$\text{FN}_{c=1}$	$\text{FN}_{c>1}$
JSCore/DOM	48	2	0	0
JQuery	47	2	1	0
Phonegap	46	2	2	0
Webworks	43	0	5	2
Total	184	6	8	2

qualified names, partially qualified names, and unqualified names. The results are detailed in Table 1.

We found the same result as Dagenais and Robillard: 89% of unqualified method names collided. We also found that one-third of unqualified types and one-third of partially qualified methods collide. These results confirm our earlier statement that unqualified names are insufficient to link a code element to the correct document. Some methods, like `getId()`, have thousands of unique fully-qualified declarations in the oracle that all conflict when unqualified.

5. EVALUATION

In evaluating Baker, we wanted to answer two research questions: first, can Baker accurately identify API elements in code snippets; and second, does Baker work on a variety of systems, or is it limited to just a few libraries?

5.1 Linker Accuracy

To answer our first question, we manually examined a number of the matches produced by Baker to see if the tool could correctly identify API elements. We did this for both Java and JavaScript code snippets.

We first populated Baker with a number of source code examples. We obtained our snippets from the Stack Overflow data repository provided for the 2013 MSR Challenge [3]. We augmented these by pulling from a few repositories on GitHub that were aimed at collecting source code examples. Baker analyzed 1,000 JavaScript source code snippets and 4,000 Java source code snippets.

In this context, precision is much more important than recall. Since the web contains tens of thousands of snippets, we would rather suffer a false negative result (a failure to infer a link that should have been identified), than a false positive (incorrectly linking one element to another). To this end, we also only analyzed Baker's recommendations that had a cardinality of 1; that is, we only examined the results that the tool was sure were correct. While the other results could be useful for the developer, we would not display them to the developer by default.

We chose the systems to analyze for our precision evaluation by identifying the union of the systems evaluated in the RecoDoc [9] and ACE [16] papers and in Parnin's Stack Overflow study [5]. The five libraries used in these studies are listed in Table 2. We then randomly selected code snippets from our repository that Baker had analyzed and had been annotated with a tag indicating it should contain a question of relevance to the project under study. Whenever Baker claimed that it had identified an API element from one of the five libraries in Table 2, we manually examined the snippet to determine if the result returned by Baker a) correctly matched the API intended by the developer (true positive [TP]) or b) incorrectly matched the API (false positive [FP]). We also examined the snippet to see if there were tokens not associated with any links at all but that we would have expected to see a result (false negative [FN]). We stopped once we had examined 50 code elements for each system in this way. The overall Java precision is 0.98 with a recall of 0.83. When we included any result with a cardinality > 1 (that is, where the correct element was found but could not be uniquely identified), the recall increased to 0.96.

Table 3: Number of matched elements from 4,000 Java code snippets extracted from Stack Overflow and Github. The types and method cells are split (total # matches / unique # elements).

System	# types	# methods
Android	272/64	175/104
Apache	178/79	108/97
Eclipse	104/41	53/45
GWT	149/47	122/69
Hibernate	389/133	378/199
JDK	14,252/632	7,483/1,981
Other	5,956/487	1,339/747
Total	21,300/1,483	9,658/3,242

Table 5: Number of matched elements from 1,000 JavaScript code snippets extracted from Stack Overflow and Github. The object and properties cells are split (total # matches / unique # elements).

System	# properties
JSCore/DOM	6,467/107
JQuery	1,793/96
Phonegap	126/27
Webworks	244/52
Other	1,297/300
Total	9,927/582

Table 6: Example of an imprecise Baker match. The top row represents the correct answer.

com.thoughtworks.`xstream.io.xml.AbstractDocumentReader`
com.cloudbees.shaded.thoughtworks.`xstream.io.xml.AbstractDocumentReader`
com.ovea.jetty.session.internal.`xstream.io.xml.AbstractDocumentReader`
cucumber.runtime.`xstream.io.xml.AbstractDocumentReader`
org.pitest.`xstream.io.xml.AbstractDocumentReader`

For JavaScript we applied the same procedure for analyzing the snippets and assessing true positives, false positives, and true negatives. Since none of the previous papers investigated JavaScript, we just chose four Stack Overflow tags for which there were a large number of associated questions. The JavaScript precision was 0.97 while the recall was 0.96. We believe the difference in the recall between the Java and JavaScript analyses was that the Java oracle had millions of entities in it, while the JavaScript oracle had only thousands. That said, we believe the Java oracle demonstrates that even with a huge breadth of API elements to choose from the approach still delivers reasonably high recall.

5.2 Example Diversity

In addition to assessing Baker's ability to identify links between source code examples and the API they represent, we looked further into the links identified by the tool to see the breadth of the systems it was able to generate links for.

JavaBaker parsed 4,000 source code snippets. It identified over 30,000 links to 4,500 unique API elements. Table 3 describes the elements that were identified in more detail. To get an idea of the projects that were referenced, we looked at the packages that were linked to. We then aggregated these and considered only those that had the same two initial tokens (e.g., all `org.eclipse` references would count as 1). This resulted in 188 unique second-tier packages for which we have examples. If we considered third-tier packages (the same first three tokens), 347 different packages were referenced.

JSBaker parsed 1,000 source code snippets and identified almost 10,000 references to over 500 unique elements. A brief overview of the systems identified are shown in Table 5. Looking into the elements in the 'other' category, we see a variety of popular JavaScript frameworks like Angular, Ember, Underscore, Require, Backbone, and so on. Since JavaScript programs tend to 'mash up' many libraries, we find that even if the exact library being asked about is not in the oracle, elements from other libraries are often found interspersed with these references.

5.3 Quantifying High-Cardinality Matches

As mentioned previously, our deductive linking method returns more than one match when there is not enough information to uniquely identify the FQN of a method or type. However, this is relatively rare. To quantify this, we recorded the cardinality of the result for each of the 4,000 snippets described in Table 3; this data is plotted in Figure 4. As can be seen in the figure, the majority (69%) of elements can be precisely identified, though there was a long tail that had high cardinality values. We have removed references to JDK types and methods from this figure since many of these results had cardinality 1 and we wanted to ensure this was not the cause of the long tail effect. The results with them included are even better (85% precisely

identified). This result supports the use of the tool for live API documentation, since in the majority of cases we can link documents that discuss the same source code element precisely.

We performed an informal analysis of the elements that Baker matched to with multiple targets. Surprisingly, the most common cause of these multiple matches (more than half) were situations where projects make internal clones of existing source code files to avoid having to include an external JAR file along with their project. An example of this can be seen in Table 6. In these cases, Baker's results contain the correct FQN; we are currently working on techniques for differentiating between these results.

The impact of high-cardinality matches depends on the intended use of the data; some tasks favour recall over precision while for other tasks the reverse is preferred. For example, when annotating a Stack Overflow example with FQN information, presenting a small number of possible type options for the developer to choose from would be reasonable as they could use their own intuition gained from the snippet text to make an informed selection. Conversely, when annotating official API documentation with usage examples we would only want to include exact matches as this context would not be present.
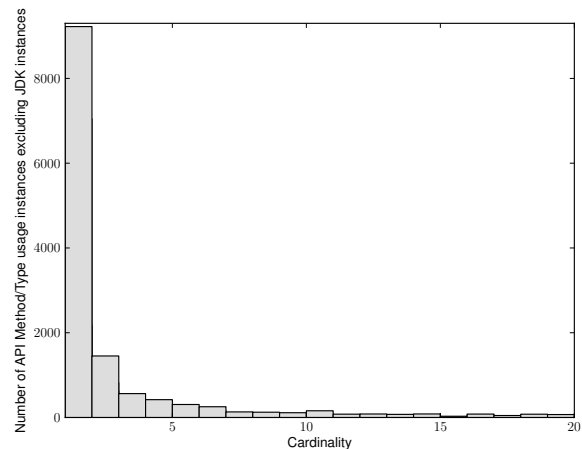

Figure 4: A histogram of the cardinality values obtained when the tool was run on the 4,000 code snippets described in Table 3.

5.4 Threats to Validity

The accuracy of our evaluation is subject to our ability to correctly identify each API usage in the code snippets we investigated. While the inherent ambiguity present in source code snippets sometimes obscured what the developer intended, since snippets generally exist to answer specific questions within a particular context, we were usually

able to identify the intended element. When we were not, or when Baker was incorrect, we conservatively flagged the recommendation as a false positive.

To reduce overfitting and increase generalizability, the Java systems we selected for the precision analysis were chosen by taking the union of systems evaluated for Recodoc [9], ACE [16], and Parnin's StackOverflow study [5]. Baker was executed in its default configuration for all studies. The only exception was that JavaScript snippets were only submitted to JSBaker while the Java snippets were sent to JavaBaker.

6. ENABLING LIVE DOCUMENTATION

We have applied the fully qualified API usages identified by Baker to create bidirectional links between snippets using an API and the documentation and source code represented by the API. As previously discussed, keeping documentation current is challenging and expensive. We therefore create links between source code examples and the official API documentation being used in the code examples that are automatically kept up to date without requiring any effort from documentation maintainers.

Our approach augments HTML-based source code examples and API documentation by adding relevant links between pages that are related by the APIs they use (or describe). These links can occur in multiple directions. For example, a Stack Overflow snippet can be augmented with links from a specific method call to the documentation for the API the call represents. The API documentation can also be updated with links back to source code examples demonstrating its usage. To do this, we use a browser extension that is able to monitor pages to see if they contain source code examples or are API documentation. If either of these is true, the pages are augmented by injecting new HTML elements into the page that represents links to other related resources.

To ensure our example repository is always current, we also continually monitor Stack Overflow to parse new source code examples as they are posted. Any API usage detected in these examples is added to a database we maintain so that usage links can be injected into API documentation. This means that as long as questions are being asked and answered about an API, the documentation will be updated. It remains to be seen if this could convince API owners to answer questions about their APIs in Stack Overflow knowing that their answers will be tied directly back to their own documentation.

One piece of data is missing in order for the browser extension to work: an explicit mapping between a fully-qualified name and the corresponding official API documentation. Fortunately, generating these links is easy to manage in practice. This is because the vast majority of API documentation is automatically generated and is very well formed. For example, augmenting the Android API documentation with examples simply requires a mapping from a Java package to a web location, e.g., `android.*` → `http://developer.android.com/reference/`. With this mapping, the browser extension can automatically determine the correct target page that should be annotated (in either direction) with either the source code example or API documentation link.

Figure 5 shows how our browser extension modifies a Stack Overflow post. Normally, the source code snippet in the post does not contain the underlined elements and is treated as a plain text block. In this case, the Baker extension detects that the user has navigated to a Stack Overflow post. If this post had not been previously parsed, it is now parsed on demand. Once the parsing is complete, Baker has a list of which API elements are present in the post and where they are used. After consulting the mapping, the extension modifies the code block to show all of the elements that Baker has identified by underlining them. In this case, Baker was able to correctly identify the fully-qualified name for `mChronometer` (in addition to several other types and methods). When the developer hovers their mouse over `mChronometer` (as they are in Figure 5), they are presented with a dynamic popup that contains links to the official Android `Chronometer` API documentation, to the source code for `Chronometer`, and links to 18 other Stack Overflow posts that also use `Chronometer`.

In the opposite direction, Figure 6 shows how the browser extension augments the official Android documentation with Stack Overflow examples. Once again, the browser extension detects from the mapping file that the user is visiting a page for which it has API usage examples. While Baker actively monitors new Stack Overflow for new examples, any snippet the browser extension has encountered can be included in the example list. It then checks the page to see if Baker has examples for any of the API elements on the page; if it does, it injects a small table into the page that describes the relevant source code examples. These example tables can be injected for types and methods.

Baker is able to parse source code snippets found in online repositories to identify fully qualified names that pertain to API usage. It is able to use these fully qualified names as a form of links that can be dynamically injected into web-based code resources. This improves the utility of the source code examples by enabling easy navigation to official API documentation for any given code element while simultaneously enhancing the API documentation by providing concrete usage examples that compliment the traditional descriptions of the API. The entire process is automatic, enabling injected markup to be dynamically updated whenever new resources are encountered. The Baker parser web service and the browser extension are both available online[6].

7. DISCUSSION

A number of opportunities exist for extending the utility of the Baker data. For example, the API elements on a page could be dynamically reordered based on how many different examples have been found for them. While API elements that have more examples associated with them could be interpreted as being more difficult, they might also indicate the key elements a developer should consider. PopCon [12] explores a similar concept, but leverages a large static analysis repository rather than code examples.

While developers frequently create APIs, it is not straightforward for them to receive feedback on the APIs' ease of use. Baker could allow developers to discover the common questions people have about their APIs. This feedback could be used to guide future API updates or simple documentation fixes.

In the future, we aim to document and fully open the web services that power Baker. This would allow anyone to add new code to the Java and JavaScript oracles, update mapping files, submit snippets to be parsed, and query Baker.

---

[6]`https://cs.uwaterloo.ca/~rtholmes/baker`

1 Answer

14

When I played with the chronometer awhile back I just used the setBase() method to set the base to the current time just before calling start(). Depending on your exact needs you may need to add some logic around whether to reset the chronometer or not before starting it.

```java
View.OnClickListener mStartButtonListener = new OnClickListener() {
        @Override
        public void onClick(View arg0) {
            mChronometer.setBase(SystemClock.elapsedRealtime());
            mChronometer.start();
        }
};
```

android.widget.Chronometer

Javadoc   [developer.android.com]

Source Code [github.com]

StackOverflow posts (18) involving Chronometer

share | edit | flag

Figure 5: Baker-augmented Stack Overflow post. Note that the Baker browser extension automatically augmented the Stack Overflow post without any intervention from the teams maintaining Stack Overflow, GitHub, or the Android documentation.

Figure 6: Here we can see that the Baker browser extension has injected a list of relevant Stack Overflow posts into the official Android API documentation. These links dynamically update without any intervention from the documentation team.

In addition, we will be releasing the browser extension so that other researchers and developers can try the tool and provide feedback.

8.   RELATED WORK

As described in Section 1, previous work has shown that writing and maintaining documentation is difficult [8, 9]. Consequently, researchers have explored ways to make this task easier. One way of doing this is to automatically add links to the documentation that direct developers to other relevant artifacts. For example, XFinder maps tutorial steps to the classes involved in the tutorial [7].

Linking API documentation to examples of correct use is a special case of this idea. Bacchelli et al. [1] used regular expressions to match text terms to method names. This technique can produce some matches, but cannot resolve the ambiguity that results from having many methods with the same name in the API. Chen [4], De Lucia et al. [11] and

Hsin-Yi et al. [13] used information retrieval techniques to do coarse granularity linking (e.g., linking an entire document to a source class). These techniques are useful, but cannot do the fine-grained linking necessary to identify correct uses of, for instance, a method in an API.

The two systems most similar to ours are RecoDoc [9] and ACE [16]. RecoDoc uses partial program analysis (PPA [6]) to infer links between documentation and an API. Like RecoDoc, we use PPA and an oracle as part of our link finding approach. Unlike RecoDoc, Baker uses a much bigger oracle and does not need to be told which API corpus to use; a single oracle is used for all queries. Moreover, Baker can be used with dynamically typed languages.

ACE is a linking system that tries to relax two of RecoDoc's key assumptions: that there must be an oracle, and that each mention of a code element in the documentation has equal relevance to a problem. Like ACE, Baker ranks the output based on expected relevance. However, ACE uses an island grammar instead of PPA and cannot do documentation linking because the results are not fully qualified.

9. CONCLUSION

Maintaining API documentation is a challenging, time-consuming task; consequently, the documentation is frequently out of date. This paper presented a method and tool for automatically generating links between API documentation and source code examples. We demonstrated that our tool, Baker, has high precision (0.97) and is able to successfully link code snippets to thousands of different Java classes and methods along with hundreds of JavaScript functions. Baker's results can be automatically integrated into web pages for both the source code examples and the official API documentation. This will increase the timeliness of the API documentation while providing valuable reference links for source code examples.

10. ACKNOWLEDGEMENTS

11. REFERENCES

[1] M. L. Alberto Bacchelli and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 375–384, 2010.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions of Software Engineering*, 28(10):970–983, 2002.

[3] A. Bacchelli. Mining challenge 2013: Stack overflow. In *The Working Conference on Mining Software Repositories*, 2013.

[4] X. Chen. Extraction and visualization of traceability relationships between documents and source code. In *Proceedings of the International Conference on Automated software Engineering (ASE)*, pages 505–510, 2010.

[5] L. G. Chris Parnin, Christoph Treude and M.-A. D. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical Report GIT-CS-12-05, Georgia Tech, 2012.

[6] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 313–328, 2008.

[7] B. Dagenais and H. Ossher. Automatically locating framework extension examples. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 203–213, 2008.

[8] B. Dagenais and M. P.Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 127–136, 2010.

[9] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 47–57, 2012.

[10] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage. *Empirical Software Engineering*, 18(6):1195–1237, 2013.

[11] A. De Lucia, R. Oliveto, and G. Tortora. Adams re-trace: Traceability link recovery via latent semantic indexing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 839–842, 2008.

[12] R. Holmes and R. J. Walker. A newbie's guide to Eclipse APIs. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 149–152, 2008.

[13] H. Jiang, T. Nguyen, I.-X. Chen, H. Jaygarl, and C. Chang. Incremental latent semantic indexing for automatic traceability link evolution management. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 59–68, 2008.

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 125–135, 2003.

[15] C. Parnin and C. Treude. Measuring API documentation on the web. In *Proceedings of the International Workshop on Web 2.0 for Software Engineering (Web2SE)*, pages 25–30, 2011.

[16] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 832–841, 2013.

[17] J. Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 139–145, 1998.

[18] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 85–88, 2013.