Live API Documentation

by

Siddharth Subramanian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

Statement of Contributions

In Chapter 1:

Contributor	Contributions
Subramanian, S.	Manuscript writing
Inozemtseva, L.	Manuscript editing
Holmes, R.	Manuscript editing and Figure 1.1

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.

This thesis describes 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.

Acknowledgements

I would like to thank my supervisor Reid Holmes for his support and guidance over the last two years. Our regular meetings not only gave me insights on conducting good research but also kept me on my toes and ensured that I completed my work on time. I would also like to thank Laura Inozemtseva for having worked with me on the two ICSE publications. By working with her, I learned a lot about good academic writing practices.

I would like to thank my readers, Ondřej Lhoták and Michael Godfrey for their valuable comments and suggestions which helped improve this thesis.

Finally, I would like to thanks MITACS for their generous financial support which has been of immense help during my time here at Waterloo.

Dedication

I dedicate this thesis to my parents. It is their love and encouragement that keeps me going.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reuse is a technique that is widely employed to reduce the effort required to build new systems [3]. Utilizing components of existing software to build new systems greatly expedites the software development process. These reusable software components are typically bundled into software libraries or frameworks, and are exposed through Application Programming Interfaces (API). APIs are powerful abstraction mechanisms that provide access to complex functionality, often involving low-level system calls through simple classes and methods. They allow developers to use sophisticated functions without having to modify or understand the underlying implementation details. By supporting abstraction, APIs also make client programs easier to understand and maintain. Thus, APIs play an integral role in the software development process.

Though APIs facilitate software development tasks, they are often complex and hard to learn [4]. APIs may contain hundreds of elements (classes, fields, methods and interfaces), and the dependencies between these elements can sometimes be overwhelming for a developer to comprehend [5]. Hence, to assist developers in learning new APIs, API authors provide documentation that describes the behavior of various API elements. Occasionally, additional resources like reference manuals and support channels are also provided. These resources play a vital role in determining the usability of libraries and frameworks [6].

However, creating and maintaining these resources requires considerable effort. Documentation often fails to keep up with evolving software, and this renders the documentation obsolete and untrustworthy [7, 8]. API learnability is further exacerbated by the lack of usage examples in the documentation [4]. Although other resources like support channels (e.g., mailing lists) try to address these issues, they are disconnected from the API and the corresponding documentation due to their unstructured and informal nature [9, 10].

For example, there is no cross-referencing between a library's mailing list threads and the API elements referenced in them. This loose coupling makes it difficult for a developer learning an API to identify relevant mailing list threads. Additionally, although documentation is common, not all libraries provide additional resources like support channels. The difficulties associated with learning APIs and the inadequacy of various learning resources in addressing them, increase the overhead associated with integrating a new API into an existing project.

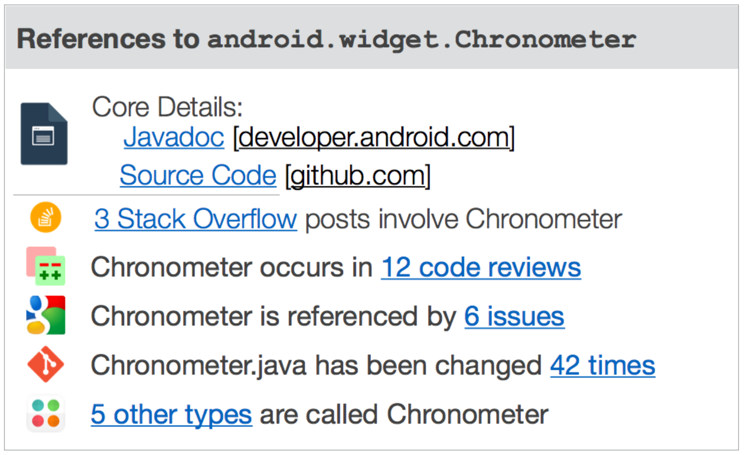
This division between the benefits offered by APIs and the shortcomings of conventional learning resources has led developers to look for better resources to learn from. Web-based resources like discussion forums and blogs that discuss software development have grown to bridge this gap [11]. Developers are increasingly starting to use these resources to find solutions to their API learning problems [12]. Stack Overflow [1] is one such discussion forum that is highly popular among developers and rich in insightful discussions [13]. Its discussions cover not only a large number of libraries and frameworks, but also a wide range of APIs within these libraries and frameworks [13]. A large volume of these posts are accompanied by code snippets that illustrate API usage [13, 2]. Additionally, the high quality of posts is maintained through a strict set of community guidelines that favor factual and informative answers [14]. By overcoming many of the shortcomings of conventional API documentation, Stack Overflow has turned into a worthy extension to API documentation, if not a replacement.

Effectively harnessing Stack Overflow to improve API learnability requires precise identification of API elements referenced in posts. Due to frequent naming collisions among source code identifiers, this is a difficult task. However, every language provides a mechanism to reference external API, be it through namespaces, packages or global objects that have specific names. This allows every API element to be uniquely identified by its *Fully Qualified Name* (FQN), which is the name of the identifier prefixed by the name of its package or namespace. The use of FQNs avoids naming collisions among identifiers and allows distinct source code elements to share the same short identifier names. For example, though the Android library has 32 different methods named `onCreate` belonging to 15 different packages, each of these methods can be uniquely identified by their FQNs (e.g., `android.app.Application.onCreate()`, `android.content.ContentProvider.onCreate()`) . Hence, mapping source code elements to their respective API documentation requires identification of the FQNs of the API elements they reference.

In fact, other online resources including the element's source code, bug reports and code review, can also be tied together using the FQN [15]. Generating these *traceability links*

[1] http://stackoverflow.com

to various related resources satisfies the developer's information needs to a large extent and helps in thorough understanding of the API. Figure 1.1 depicts the different kinds of resources that can be linked together using the FQN of an API. In this thesis, we focus on mapping Stack Overflow posts to their respective API documentation.



References to `android.widget.Chronometer`

Core Details:
Javadoc [developer.android.com]
Source Code [github.com]

3 Stack Overflow posts involve Chronometer

Chronometer occurs in 12 code reviews

Chronometer is referenced by 6 issues

Chronometer.java has been changed 42 times

5 other types are called Chronometer

Figure 1.1: A mockup of what could be achieved with an effective traceability linking strategy across a number of information sources.

However, identification of FQNs of elements in plain text resources like Stack Overflow is difficult. The primary challenge involved is the inherent ambiguity of natural language text. Previous techniques that have tried to identify source code elements in non-code resources [16, 17, 18, 10] have limitations. For example, some techniques expect the target library to be specified beforehand and all other external library references are ignored [18]. Some techniques only provide partially qualified API elements [10], which are not useful to cross-reference API documentation with API usage.

Fortunately, web-based learning resources contain high quality code snippets that support discussions [13]. In our preliminary study [2], we found that the structural information in these code snippets can be used to identify API elements being referenced in them. This information can, in turn, help us cross-reference the API elements to relevant discussions and examples on the web and vice-versa, thereby enabling *live documentation*.

This thesis proposes a deductive linking technique that leverages structural information to uniquely identify fine-grained API type and method references in incomplete snippets of code through static program analysis, and map them to their respective API documentation. We demonstrate the generality of this approach by providing implementations for both typed (Java) and dynamic (JavaScript) languages through a tool called Baker. We also evaluate the ability of Baker to cross-reference code elements contained in snippets extracted from Stack Overflow to the corresponding API documentation over five widely-used Java libraries and four JavaScript libraries.

1.1 Motivating Scenario

Consider the Java code snippet shown in Figure 1.2. 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 elements in bold characters. If we can identify what API types and methods are being referenced by these elements, 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 documentation pages into the Stack Overflow post. These two additions to the API documentation would make it easier for developers to learn how to use this class.

```
public FirstPanel() {
  History.addHistoryListener(this);
  String token = History.getToken();
  if (token.length() == 0) {
    History.newItem(INIT_STATE);
  } else {
    History.fireCurrentHistoryState();
  }
  .. rest of code
}
```

Figure 1.2: A Java code snippet representing a Java API (GWT) usage.

Next, consider the JavaScript snippet in Figure 1.3, where a developer is trying to make a web application 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, identifying the right API can help in cross-referencing this code with the corresponding API documentation, thereby improving API learnability.

```
$("#addphoto").on('click',
  function() { useGetPicture();}
);
function useGetPicture() {
  var cameraOptions = { ... };
  navigator.camera.getPicture(onCameraSuccess,
    onCameraError, cameraOptions);
}
function onCameraSuccess(imageData) {
  var image = document.getElementById("..");
  image.src = "data:image/jpeg" + imageData;
}
function onCameraError(message) {
  alert("Failed: " + message);
}
```

Figure 1.3: A JavaScript code snippet containing Cordova, JQuery and JavaScript DOM API usage.

The code snippets in Figures 1.2 and 1.3 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.

1.2 Benefits of Recovering Traceability Links

Enabling cross-referencing between API documentation and relevant online resources like Stack Overflow has a number of advantages, including:

- **Better API Understanding**
  Stack Overflow posts have proven to be highly effective at answering framework (e.g., jQuery, ruby-on-rails) and environment (e.g., Android, iOS) related questions [19]. Developers learning a new API can utilize these resources to gain a thorough understanding of how various elements in the API interact with each other. The related discussion can provide better contextual information about what the code is trying to achieve, in contrast to code search engines (CSE), which just provide example snippets of code that use the API.

- **Easier Documentation Maintenance**
  Since content in web-based resources like Stack Overflow is constantly on the rise, enabling this kind of coupling provides automatic API documentation. New discussions are mapped to their respective API documentation as and when they are posted. Moreover, APIs that are frequently used in practice are documented better on Stack Overflow than ones that are used less often [13]. Thus, these discussions and relevant code snippets serve as effective learning resources for developers, and can easily be augmented with existing documentation.

- **Enhanced Code Reuse**
  A majority of web-based resources contain code snippets [11]. These code snippets are of high quality, since they are tested and constantly improved through discussions. In fact, websites like Stack Overflow allows users to test and choose the answer that best solves the problem. A large volume of these *accepted answers* contain code snippets [2]. Since they are succinct and accurate, they can be readily reused in client code.

1.3 Challenges

Our technique tries to identify and map API references in source code snippets to their FQNs. Identifying such fine grained API references in incomplete code snippets requires the ability to parse these code snippets. Since code snippets are typically incomplete, and hence ambiguous, parsing them is hard. Of the four types of ambiguities identified by Dagenais and Robillard in their study on identifying code elements in plain-text [18], two are relevant to our task of parsing code snippets. These are *Declaration Ambiguity* and *External Reference Ambiguity.*

- **Declaration Ambiguity**
  Code snippets are inherently incomplete; they contain just the amount of information required for a human reader to comprehend the code. For example, code statements are generally not enclosed within methods or classes, declaration statements are omitted, and classes are unqualified (missing package information). Unfortunately, multiple API elements, often belonging to different libraries, share the same name. This makes it difficult to precisely identify the elements being referenced in a piece of code. Analysis is further hampered by the frequent use of syntactically incorrect constructs like '...' to elide functionality.

- **External Reference Ambiguity**
  References to types and methods belonging to JDK (Java Standard Library) and other external libraries is common. Previous studies have overcome this by ignoring references to external elements. However, these could be valuable examples that depict API usage for the external library and cannot be outrightly ignored.

Unlike previous approaches, our technique handles these ambiguities through a reduced dependency on the parser and through the use of an oracle.

Since code snippets are syntactically incomplete, generating a proper Abstract Syntax Tree (AST) for these snippets is not possible. As a solution, we modify every code snippet appropriately so that the parser can at least generate an incomplete AST. While traversing this AST, the only information we extract at every node is the node type (e.g., *Method Invocation* node). Maintaining this minimal dependency on the parser allows us to deal with ambiguous code snippets.

The oracle, on the other hand, is a large database containing information about APIs belonging to various libraries and frameworks. When Baker encounters an ambiguous code element, such as the `History` class in Figure 1.2, 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.

More information about ambiguous code snippets we encountered on Stack Overflow during this study is presented in Appendix A. Chapters 3 and 4 discuss the need for an oracle, its design and its contents. Additionally, a quantitative study of naming ambiguities among various libraries in the oracle is presented in Chapter 4.

1.4 Contributions

The contributions of this thesis are as follows:

- A quantitative analysis of code snippets in Stack Overflow posts.
- A constraint-based, iterative approach to deduce the fully qualified names of code elements in source code snippets. This approach works with both statically and dynamically typed languages.

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

1.5 Outline

Chapter 2 discusses existing work in the area of improving API learnability. We then present our deductive linking technique in Chapter 3. Chapter 4 describes the implementation of the deductive linking technique for Java and JavaScript in a tool called Baker. Chapter 5 then presents our evaluation of Baker. This is followed by concluding remarks with some related discussion and future work.

Chapter 2

Background and Related Work

Software engineering researchers have long recognized the difficulties associated with learning APIs and the shortcomings of API documentation in addressing them. Consequently, a number of approaches have been proposed to improve documentation. In one of the earliest documented studies in this field [20], McLellan et al. advocated for the practice of *usability testing* of APIs to identify their flaws and make appropriate changes to their design and documentation. Researchers have also called for API design that ensures that the API is self-documenting [6]. Code that uses such self-documenting API can easily be read and written without having to refer to the documentation. Nevertheless, fairly recent studies on API usability have found that APIs are still hard to use and that documentation is still lacking [4].

Lately, researchers have started to investigate techniques that support existing documentation by leveraging information from related resources like source code repositories and mailing lists. By combining multiple information sources, these techniques provide developers with a comprehensive view of the API. Previous work in this field can be broadly classified into two categories. The first category contains techniques that extract examples relevant to a developer from code-based resources like source code repositories (Example Recommendation Systems), while the second contains techniques that identify API references in plain-text resources like mailing lists and link them to their respective API elements (Traceability Link Recovery). Both of these categories are challenging in their own right. While the former deals with challenges in estimating an example's relevance to the developer's context, the latter has to overcome various natural language ambiguities to identify API references in plain text. Our approach lies at the intersection of these two categories. By mapping Stack Overflow posts to their respective API documentation, we recover traceability links between these two resources which are otherwise independent, but, we do so by analyzing source code snippets in these posts.

The following sections describes various tools and techniques belonging to the two categories and their relation to our approach. This is followed by a discussion on the potential of Stack Overflow as a source for high quality snippets and discussions.

2.1 Example Recommendation Systems

Example Recommendation Systems are tools that assist developers in using new libraries by recommending relevant reusable code snippets. These snippets are extracted from software projects that use the library, typically belonging to source code repositories. Source code repositories are archives of software projects that cover a wide range of libraries and frameworks. By recommending relevant code examples, these systems make it easier for a developer to use complex libraries and frameworks.

A number of different example recommendation systems have been proposed in the past. Holmes and Murphy proposed Strathcona [21], an Eclipse plugin that helps developers learn a new framework by recommending contextually relevant code examples. Strathcona extracts structural information like the container class, parent class, implemented interfaces and method calls from the developer's code by parsing its AST. Then, using a set of heuristics, it matches the extracted contextual information against a collection of projects that use the corresponding framework, and identifies and recommends the most relevant examples to the developer. However, Strathcona requires that all projects in the example database use the target library or framework. In other words, it requires prior knowledge about the possible libraries that the code might use. This is a difficult requirement to satisfy. Additionally, maintaining an up to date example database which is specific to a framework is challenging.

Prospector [22], proposed by Mandelin et al., is a tool that recommends reusable *jungloids*. A *jungloid* is the sequence of object creations and method calls required to obtain an object of type τ_2 from an object of type τ_1. In their study, they identified that a majority of API usage obstacles involved difficulties in finding the *jungloid* required for a task. As a solution, they use signatures of the library's API to construct a directed acyclic graph with API types connected by API methods. The appropriate jungloids for a query (τ_1, τ_2) are paths in the graph from τ_1 to τ_2. Parseweb [23] is another tool that solves the problem of synthesizing task-specific jungloids. Parseweb retrieves source code files through a code search engine (CSE) and mines them for relevant jungloids. It identifies method invocation sequences required to convert type τ_1 to τ_2 using the data flow graph of the source code files. This dependency on real-world code, as opposed to API signatures, helps Parseweb produce more accurate results than Prospector. However, the task of finding jungloids is only one sub-problem of many API usability issues. A better solution would be to identify code examples that illustrate a variety of different API usages patterns.

To address this issue, Zhong et al. proposed MAPO [24], which identifies frequent patterns of API usage in code. MAPO leverages existing CSEs to find relevant source code files and then uses data mining techniques to identify frequently used sequences of methods and classes, thereby helping developers in understanding how APIs are used. However, a major drawback with such tools is their tight dependency with CSEs. Most CSEs treat code as plain text. Hence, improperly framed queries return irrelevant results due to naming collisions among API elements belonging to various libraries.

More recently, McMillan et al. proposed Portfolio [25] that tried to overcome many of the shortcomings of these previous approaches. Much like a textual search engine, Portfolio is a search engine that retrieves relevant functions based on user queries and provides visualizations to illustrate their dependencies with other functions. It uses Natural Language Processing (NLP) techniques to identify relevant source code files from a database of open source projects and employs a combination of PageRank and spreading activation network (SAN) algorithms to model user behavior and to identify functions that best solve the developer's programming task. Portfolio is robust in handling searches since it adopts techniques that have proven to be successful in plain text search. Nevertheless, this reliance on text-based techniques rather than the structural information in code does not allow identification of fine-grained API references.

Although example recommendation systems have been more or less successful at identifying contextually relevant examples, the techniques they use identify only snippets that are structurally similar to the developer's context. Their goal is not to precisely identify API references in code. However, to link documentation to code examples, we require identification of fine-grained API references in code. Additionally, most of these tools require prior knowledge about the libraries or frameworks being used. This dependency restricts these tools from leveraging arbitrary software projects. In techniques where this knowledge is not required, the tools rely on text-based approaches. Such tools cannot identify fine-grained references (FQNs). For example, consider the snippet of Java code in Figure 1.2 that uses methods from the `History` class of the GWT library. Unless it is known beforehand that this snippet refers to API from the GWT library, these tools cannot disambiguate the History class to `com.google.gwt.user.client.History` from among the 58 different `History` classes belonging to various Java libraries and frameworks. Our technique overcomes these obstacles by using an oracle that contains API information about various libraries. This use of a single global oracle enables us to identify API references from snippets pertaining to a number of different libraries and does not require any prior knowledge. Thus, our technique eliminates many of the constraints that the previous techniques imposed.

2.2 Traceability Link Recovery

A large volume of software development artifacts like requirements and design documents consist of natural language text. Several techniques have been proposed to map such non-code resources to their respective source code elements. Antoniol et al. [16] proposed and evaluated two Information Retrieval (IR) based techniques, a probabilistic model and a vector space model, to recover traceability links between free text documentation and code. Similarly, Chen [26], De Lucia et al. [27] and Hsin-Yi et al. [28] used information retrieval techniques to do coarse granularity linking (e.g., linking an entire document to a source class). A major drawback of using IR techniques to identify code elements is that they cannot identify fine-grained references, which are necessary to identify correct uses of an API element and hence, they cannot be used in our task. Techniques that use regular expressions to identify and match text terms to API elements also face the same problem [9].

The two systems most similar to ours are RecoDoc [18] and ACE [10]. RecoDoc uses partial program analysis (PPA [29]) to identify code-like elements in support channels and emails and link them to their respective API elements. Like RecoDoc, we use a PPA like technique and an oracle as part of our link finding approach. However, unlike PPA, we use a much bigger oracle and do not require prior knowledge about the target libraries. The use of a bigger oracle also allows us to infer external API references (API not belonging to the target libraries) in the code. Additionally, our technique can be implemented for 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, our technique ranks the output based on expected relevance. However, ACE uses an island grammar [30] instead of PPA and cannot do documentation linking because the results are not fully qualified.

2.3 Stack Overflow as a Data Source

Lately, developers have started to use web-based resources like discussion forums and blogs to complement for the inadequacy of API documentation. These resources document software-related discussions over time and act as a thorough reference guide for developers. Parnin and Treude [11] performed an exploratory study to quantify API documentation on the web. In an analysis of the jQuery JavaScript library [1], they found that over 85% of the its API methods are covered by blog posts and discussion forums. This *crowd documentation* [13] is sought after by developers not only because it is thorough, but also because it is up-to-date and improved over time. This was confirmed by a study conducted by Li et al. [12] which tried to understand how developers sought help while solving software engineering tasks. They observed that online crowd-sourced knowledge in discussion forums and blogs made a significant contribution.

Of these online resources, Stack Overflow has had a notable impact. The phenomenal success of Stack Overflow has been attributed to the website's exceptional technical design and tight involvement of the design team with the user community [14]. A strict set of community guidelines and a carefully built reputation system ensures that the answers are always of high quality. In fact, Stack Overflow has proven to be particularly effective at answering framework and library related questions [19].

A major impetus for our research comes from a detailed study conducted by Parnin et al. [13]. To understand the coverage and dynamics of API related discussions on Stack Overflow, they examined the extent of the Java (JDK), Android [2] and GWT [3] library APIs in Stack Overflow posts. Through a set of natural language rules that took advantage of the Java language specifics, posts were approximately mapped to API classes. They found that 77% of Java classes, 87% of Android classes and 54% of GWT classes had related discussions on Stack Overflow posts. Moreover, they found that over 66% of the classes in these libraries had relevant code examples in posts, a majority of which were in accepted answers. This study motivated us to exploit these code examples and associated discussions to improve existing documentation.

However, though there is enough evidence on the effectiveness of web-based resources like Stack Overflow as substitutes for conventional API documentation, not much has been done in effectively harnessing them to improve documentation. Bacchelli et al. [31] tried to integrate Stack Overflow search into the Eclipse IDE, but this is only of minimal use to a

[1] `http://jquery.com`
[2] `http://developer.android.com/reference/packages.html`
[3] `http://gwtproject.org/javadoc/latest/index.html`

developer. Effectively harnessing Stack Overflow requires identification of fine grained API references in the posts and this is a challenging task. Use of programming language specific rules along with natural language processing techniques allows us to differentiate code-like terms from non-essential text. However, mapping these code-like terms to their respective API elements requires identification of fully-qualified API references in the posts. This is often not possible due to insufficient information and naming collisions among APIs. However, in our preliminary study of Android posts on Stack Overflow, we found that the structural information in code snippets can be used to identify fully qualified API references. In this thesis, we build on this idea to develop a system that utilizes this structural information in code snippets contained in Stack Overflow posts to map the posts to their relevant API documentation pages.

Chapter 3

Methodology

Ambiguities associated with code snippets make their analysis challenging. Any attempt to infer API references in code snippets must work around these challenges. In this chapter, we propose a deductive linking technique to overcome these challenges and map API references in code snippets to their respective fully qualified API elements.

In the deductive linking technique, we first generate an incomplete abstract syntax tree (AST) for the code snippet being analyzed. Over multiple depth-first traversals of the AST, we examine all nodes involved in declaration, invocation, and assignment statements. In each of these nodes, we extract information about the class or method being referenced from the AST and query the oracle for a set of potential API element matches for the node (candidate API elements). These candidates are then used to disambiguate other nodes and are updated in subsequent iterations as new facts are uncovered. This process is repeated until an iteration fails to improve the results for any element (viz. the *fixed point*). Eventually, every API reference node in the tree is associated with the fully qualified API element it refers to.

The following sections present more details about the contents of the oracle and the deductive linking technique. The algorithm is presented in Algorithm 1 and is then demonstrated over Java and JavaScript code snippets in Section 3.3.

3.1 The Need for an Oracle

To map source code identifiers to their Fully Qualified Names (FQN), we require a reference list containing information about APIs belonging to various libraries that may be referenced in the code snippets. We call this list the oracle. The information contained in this oracle is used to locate identifiers that reference API elements in the code, and subsequently map them to their respective FQNs. Thus, our technique's capacity to handle new libraries and frameworks is determined by the availability of the corresponding API information in the oracle.

In our approach, we implement the oracle as a database that contains API signatures of classes, methods and fields from a wide variety of libraries and frameworks. The API signatures in the oracle are preprocessed and stored such that the relationships between various elements of a library are retained. This allows us to query the oracle and extract a variety of facts. For example, the oracle can be easily queried for a list of all possible API classes that are named `History`, the methods contained in each of them, and their return types.

A few traceability tasks do not require an oracle. For example, Recodoc [10] proposed by Rigby and Robillard does not use an oracle. Yet, it can extract code elements contained in various documents with high precision. However, without an oracle, it is generally impossible to identify the FQNs of the code elements in a snippet. These FQNs are essential to documentation linking tasks and thus, an oracle is required. Additionally, the use of a single global oracle allows us to work with multiple libraries and frameworks simultaneously, and eliminates the need to know the target library beforehand. As we will see in the next chapter, this oracle is not a difficult requirement to satisfy since the initial oracle can be constructed fairly quickly and subsequent updates can be done dynamically.

3.2 Deductive Linking Technique

Snippets of code, although incomplete, contain valuable structural information. In the deductive linking technique, we leverage this information to locate and disambiguate API references in code snippets.

In this technique, we first determine if the snippet is surrounded by valid class and method declarations. If it is not, we add dummy class and method wrappers to facilitate parsing. We then parse the code snippet to generate an AST. Due to the incomplete nature of the code snippets being analyzed, the nodes of the generated ASTs lack binding information. Nevertheless, most parsers are capable of disambiguating node types even in incomplete snippets, i.e., the parsers can identify and annotate AST nodes based on the source code construct the node represents (e.g., method invocation, class declaration, etc.). This is the only information we extract from the AST and this minimal dependency on the parser allows us to deal with malformed code snippets.

Next, two empty sets `localTypes` and `localMethods` are initialized to store facts about the AST. Specifically, these sets record information about classes and methods that are declared locally. Additionally, maps `candidateTypesMap` and `candidateMethodsMap` are initialized to store API facts inferred about various identifiers in the code.

The deductive linking approach is implemented over three stages involving depth-first traversals of the AST.

- **First Iteration:**
  In the first iteration, the AST is traversed, and classes and methods that are locally declared in the snippet are identified and stored in `localTypes` and `localMethods` respectively. The purpose of this pass is to differentiate API references from references to locally declared elements. Since we are interested only in API references, any references to locally declared classes and methods are ignored in subsequent passes. Our technique treats any class or method that is used but not declared locally as an API reference. In Java snippets, any extended classes or implemented interfaces are recorded to identify overridden methods.

- **Second Iteration:**
  The second iteration contains two short AST passes.

  In the first pass, type information is extracted from variable declaration, field declaration, static method invocation and method parameter nodes. For extracted types that do not belong to `localTypes`, a list of candidate API classes having this type information is retrieved from the oracle. These candidates are recorded against the identifier in `candidateTypesMap` along with the identifier's scope information.

  In the second pass, method invocation nodes are visited. At each method invocation node, one of the following actions is performed.

  - If the object invoking the method has candidate classes in `candidateTypesMap`, then those candidates that declare the invoked method are retrieved. The corresponding method is extracted from each of these candidate classes and stored against the invoked method in `candidateMethodsMap` along with the node's scope. The invoking object's mapping in `candidateTypesMap` is updated with the reduced set of candidates that declare this method.
  - If the object invoking the method is another method invocation (method invocation chain), then the list of return types of the candidate methods of the method invocation is extracted from `candidateMethodsMap`. Of these, the

candidates that declare the invoked method are retrieved and the corresponding method in each of these candidates is stored against the invoked method in `candidateMethodsMap` along with the node's scope. The candidate methods of the method invocation that invokes the current method are updated in `candidateMethodsMap` according to the reduced return types list.

  - If the method invocation is a static method invocation, then the candidate API classes of the invoking class are extracted from `candidateTypesMap`. Of these, the candidates that declare the invoked method are retrieved, and the corresponding method in each of these candidates is stored against the invoked method in `candidateMethodsMap` along with the node's scope.
  - If the object invoking the method does not have a list of candidate API classes in `candidateTypesMap` (missing declaration statement), or if the entry in `candidateTypesMap` is not within the method invocation's scope, then a list of candidate methods for the invoked method is extracted from the oracle and is stored against the invoked method in `candidateMethodsMap` along with the the node's scope. The declaring classes of each of these candidate methods is stored against the object reference in `candidateTypesMap` along with the object's scope.

For Java snippets, the inheritance hierarchy of classes is considered in assignment statements and when we look for methods in classes. Polymorphic methods are disambiguated based on the parameter types. If parameter types are not available, parameter count is used.

- **Third and Subsequent Iterations:**
In every subsequent iteration until a *fixed point* is reached, the AST is traversed and candidates that do not satisfy the set constraints are rejected, and the constraints are updated. More specifically, at every method invocation node, candidate API methods that do not have their return type in the list of candidate return types or their container class in the list of candidate classes of the object reference, are rejected. The candidate API types of the object reference (or candidate return types if the object reference is another method invocation) and the candidate return types of the method are updated accordingly.

Repeating this iteration helps propagate facts up the tree in method invocation chains. Hence, the number of iterations to reach a fixed point depends on the maximum length of method invocation chains in the code. In practice, we find that not more than two iterations are generally needed to reach the *fixed point*.

All through the traversal, the scope of all data being used is tracked to ensure that nodes are combined only if allowed by the scoping rules of the language. Scope at every node is computed using the list of AST parent nodes of the current node. A variable usage is within the scope of its declaration if it shares the same set of parent nodes, or if the parent nodes of the declaration node are a subset of the parent nodes of the usage node.

While the goal of the approach is to identify the sole fully qualified API element that a given identifier can represent, sometimes there is not enough information to choose from a set of candidates. In this case, we generate a match with *cardinality* 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.

3.3 Examples

The above technique can be applied to disambiguate API references in both Java and JavaScript snippets. However, the lack of type information makes it difficult to disambiguate references in JavaScript code. During the course of this study, we observed that JavaScript library code usually use an object literal as an implied namespace and make functions and variables properties of the object literal. We exploit this in our analysis to link JavaScript objects and functions to the API elements they reference. The following sections demonstrate the algorithm over snippets of Java and JavaScript code extracted from Stack Overflow.

3.3.1 Java Example

To describe the deductive linking technique more concretely, we will revisit the Java code fragment from Figure 1.2 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. In the first iteration, the AST of the snippet is traversed and `FirstPanel` on line 1 is recorded as a locally defined class in `localTypes`. Since there are no method definitions, `localMethods` is left empty.

Algorithm 1 The Deductive Linking Technique

```
//Initialization
Generate AST
INIT localTypes = Set()
INIT localMethods = Set()
INIT candidateTypesMap = Map()
INIT candidateMethodsMap = Map()

//First Iteration
for each node in AST do
    if node is class declaration node then
        STORE class name in localTypes.
    else if node is method declaration node then
        STORE method name and corresponding class name in localMethods.
    end if
end for

//Second Iteration
for each node in AST do
    if node contains type facts then
        if type not in localTypes then
            Obtain list of candidate API types from the oracle.
            STORE variable, scope and candidates in candidateTypesMap.
        end if
    end if
end for

for each node in AST do
    if node is method invocation node then
        if method not in localMethods then
            Identify candidate API types of object reference that declare the method.
            STORE method, scope and candidates in candidateMethodsMap.
            UPDATE candidate types of object reference in candidateTypesMap.
        end if
    end if
end for

```

```
//Subsequent Iterations
while fixed point is not reached do
    for each node in AST do
        if node is method invocation node then
            Identify candidate methods that satisfy updated object reference and return
            type candidates.
            UPDATE candidateMethodsMap.
        end if
    end for
end while
```

2a. In the first pass of the second iteration, type information is collected from declaration nodes. Since line 2 contains a static method invocation, the oracle is queried for API classes called `History`. The 58 candidate types that are returned are recorded in `candidateTypesMap` against `History`, along with the scope of the method call.

2b. Next, the variable `token` declared as a `String` is encountered on line 2. The oracle is queried for API classes named `String` and the 54 candidates that are returned are recorded in `candidateTypesMap` against `String`, along with the declaration's scope.

2c. Since there are no other nodes that contain type information, we now proceed to investigate method invocation nodes.

2d. `History.addHistoryListener(this)` on line 2 is the first expression that is encountered which requires analysis. Corresponding to the left-hand side of this expression, all elements named `History` are retrieved from the `candidateTypesMap` map. These 58 candidate types are investigated for `addHistoryListener(...)` that take a single object parameter. This results in 4 candidate methods which are added to `candidateMethodsMap`. Correspondingly, the left-hand side (`History`) is updated to reflect the number of candidates (reduced from 58 to 4).

2e. For the assignment on line 3, the right-hand side is considered first. Here, we assume that the name `History` refers to the same `History` class as the reference on line 2 and subsequently starts using its 4 candidates from `candidateTypesMap`; 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. The return types of `getToken()` can be used to reduce the number of candidates of the `String` object on the left hand side. Note that inheritance hierarchy of the class is considered while handling assignments.

2f. 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, we iterate again.

3a. Once we return to line 2 `History` can be identified as `com.google.gwt.user.client.-History` in `candidateTypesMap` 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.

3b. The return type of the now-resolved `getToken()` can be used to confirm that token is of type `java.lang.String`.

3c. Since all elements have been fully qualified, we does not need to do another pass.

Additional pieces of information are used to help identify elements whenever available. They include import statements (rarely present in example code), cast expressions, return statements, `super` invocations, `extends`/`implements` relationships, and parameter types.

3.3.2 JavaScript Example

We revisit the code snippet in Figure 1.3 to demonstrate the deductive linking for JavaScript in detail.

1a. In the first iteration, locally declared function objects are extracted. In this case, lines 4, 9 and 13 contain function object declarations. Information about these nodes is recorded in the `localMethods` set.

2a. In the first pass of the second iteration, there is no specific type information that can be extracted from the nodes since JavaScript is dynamically typed. Hence, this step is skipped.

2b. In the second pass, we first investigate line 1 which 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. The two maps are updated appropriately.

2c. The next function expression encountered is a call to `useGetPicture()`. Since this identifier is recorded in `localMethods` as a locally defined method, it is ignored.

2d. On line 5, the scope of `cameraOptions` is recorded to ensure that any constraints applied to it do not 'leak' outside its scope. When we reach 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.

2e. The next function expression is on line 10; `getElementById` matches 3 functions. Next, we check to see if any of these are defined as `document.getElementById`; this results in a single match.

2f. 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. We assumes that library code will not be dynamically augmented in this way.

2g. The function call to `alert` matches two elements, `window.alert` and `notification .alert`. Since `window` is the default namespace for JavaScript executed in the browser, we link `alert` to `window.alert` and update `candidateMethodsMap`.

3a. No new information has been learned about the exact type of `image` on lines 11 and 12 in the next iteration; 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.

Chapter 4

Baker: Deductive Linking System

We implemented the Deductive Linking technique detailed in the previous chapter in a tool called Baker. Baker identifies API references in code snippets and maps them to their respective fully qualified API elements. It then uses this data to generate HTML links between API documentation pages and relevant posts on Stack Overflow. Currently, Baker can handle Java and JavaScript code snippets.

Baker's architecture can be classified into three modules: a snippet extractor, an oracle generator and a snippet parser. The snippet extractor module crawls Stack Overflow for new Java and JavaScript posts. It then parses these posts to extract code snippets from them and subsequently stores these snippets in a database (snippet database). The oracle generator module constructs the initial oracle and handles subsequent additions to the oracle. The snippet parser module uses information in the oracle to analyze code retrieved by the snippet extractor to identify fine-grained API usage instances in them. It produces a JSON representation of fully qualified API references in the code. This intermediate representation of examples is stored in a database (example database) and is exposed through a web server for other applications to use, including the documentation linking module, which generates HTML links between API documentation and relevant Stack Overflow posts. The following sections describe the design and implementation of each of these modules.

4.1 Snippet Extractor

The Snippet Extractor module is responsible for retrieving appropriate posts from Stack Overflow and extracting code snippets from them to populate a snippet database. Stack

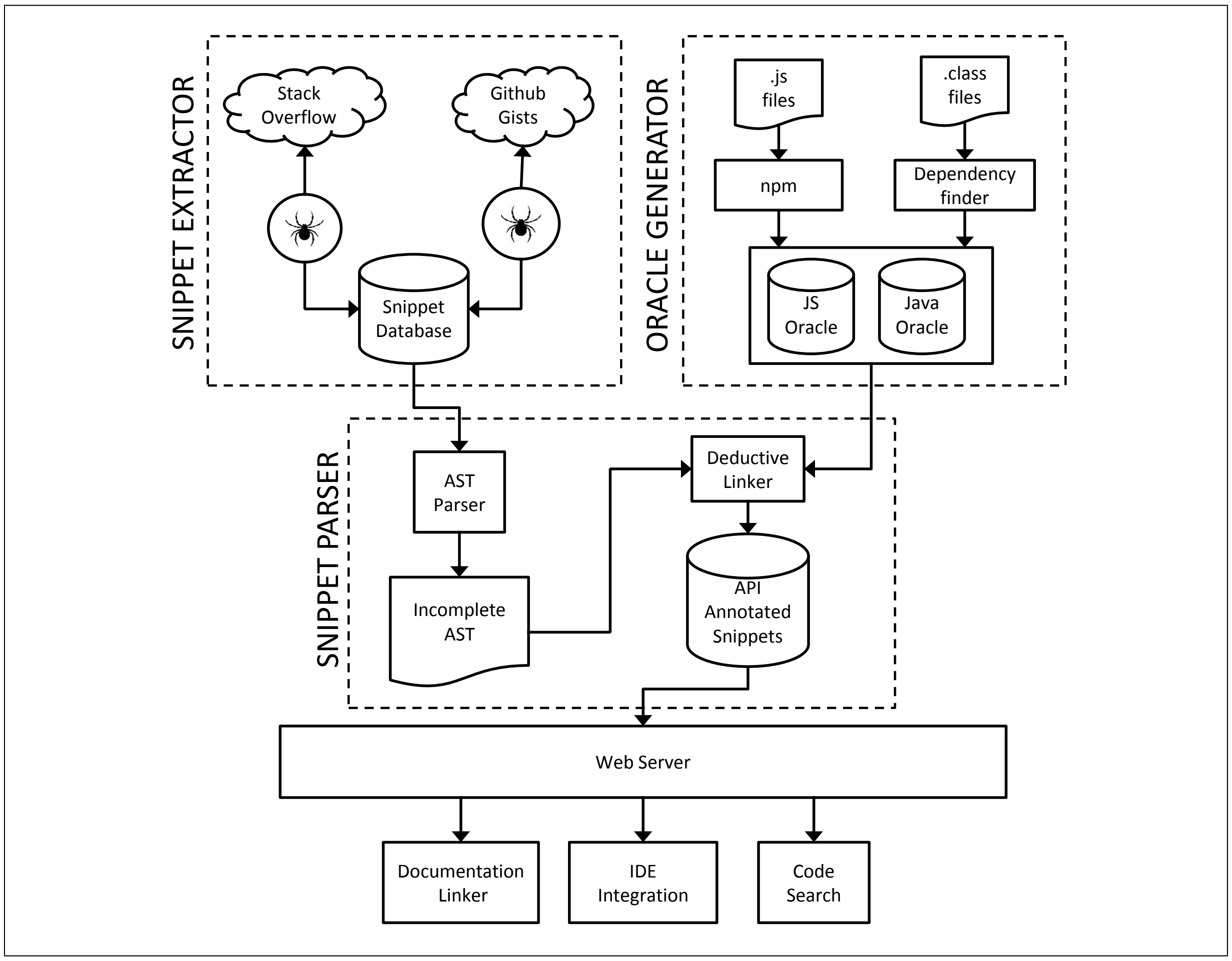


Figure 4.1: Baker's architecture depicting the Snippet Extractor, Oracle Generator and Snippet Parser modules.

Overflow uses XML to structure data within posts. In this format, blocks of code that are used in the posts are marked with `<code>` tags. We use an XML parser to extract these code snippets from the posts and add them to the snippet database. We consider only code snippets that are greater than 3 lines of code (LOC). This lower bound was chosen since manual investigation revealed that shorter snippets usually lacked the surrounding context necessary to well understand the API's usage. Additionally, we restrict this snippet database to code snippets extracted from *accepted answers* to ensure good quality of examples. Since we are interested only in Java and JavaScript snippets, we take advantage of Stack Overflow's tagging feature to identify posts associated with these languages. To quickly construct a database of snippets, we used the data dump that was provided for the Mining Challenge conducted at MSR 2013 [32]. This data dump, available as a PostgreSQL database, is a cleaned version of the official data dump released by Stack Overflow. Table 4.1 presents the distribution of code snippets in Stack Overflow posts that are tagged Java or JavaScript. The frequency of snippets of various sizes is illustrated in Figure 4.2.

	Accepted Answers	**Code Snippets**	**Median Size** (LOC)	**Mean Size** (LOC)
Java	110,516	69,794	10.0	16.60
JavaScript	121,113	89,737	9.0	12.57
Total	231,629	159,531	9.0	14.33

Table 4.1: Distribution of code snippets in posts tagged Java and JavaScript.

Once the initial snippet database is populated using the data dump, it is updated on a daily basis. The Stack Exchange network provides access to the Stack Overflow data through a RESTful API [1]. We utilize this API to update our local database with the latest Java and JavaScript discussions posted to Stack Overflow. Since we are interested only in code snippets contained in the post, we do not store the contents of the entire post locally. Instead, we extract code snippets from the post and store them along with appropriate metadata that would allow us to trace back the original post (e.g., Question ID or Answer ID). In fact, even the code snippets can be discarded once they are processed and their results are stored. They can be queried on-demand using the Stack Exchange API and appropriate metadata.

4.2 Oracle Generator

As demonstrated by the two detailed examples in the previous chapter, Baker's oracle is the key to its success. In this section, we describe how we design and populate our oracle.

We use two separate oracles, one each for Java and JavaScript. The oracles are databases implemented on a server and can be accessed as web services, allowing them to be updated dynamically by any user or program. New development resources can be augmented through `HTTP POST` requests to the service. These resources are automatically

[1] `http://api.stackexchange.com`

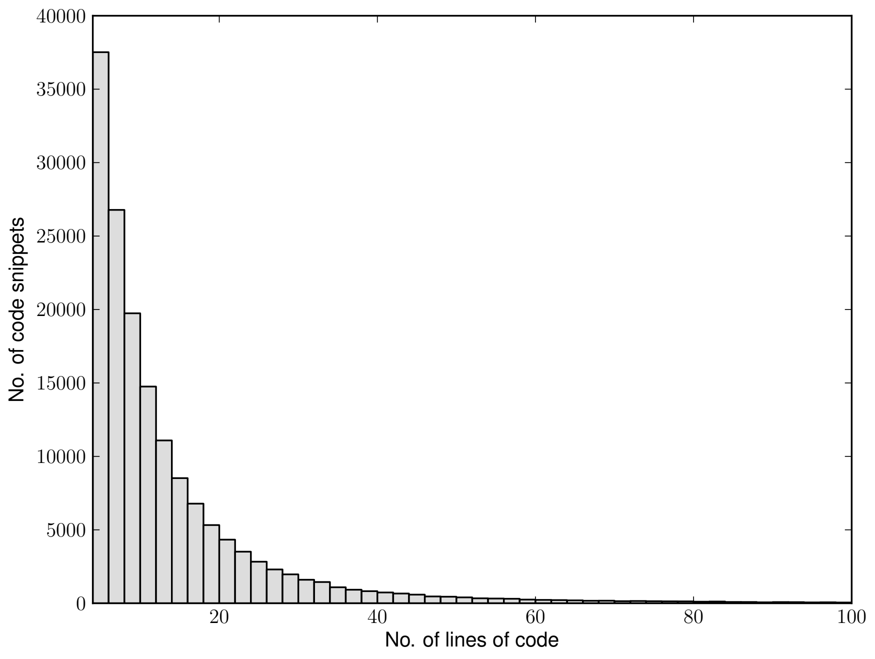


Figure 4.2: Frequency vs Size of code snippets in Stack Overflow posts tagged Java or JavaScript (> 3 LOC)

analyzed and incorporated into the oracle. Similarly, the oracle can also be shrunk on demand to consider only a smaller set of target libraries and frameworks in the analysis. For example, if a project is known to use only the JDK and Android API, the oracle can accordingly be minimized to this subset of APIs. This results in more precise results.

4.2.1 Java Oracle

The Java oracle is a database containing API class, method and field signatures from a large number of Java libraries and frameworks. We chose a graph database called Neo4j [2] to implement the Java oracle.

We chose a graph database for this purpose because the graph data structure makes it easy to represent various relationships and hierarchies between code elements that an

[2] http://neo4j.org

object-oriented language like Java offers. In this graph, every class and method is represented as a node. These nodes hold information about the code element, including their fully qualified name (FQN) and visibility (viz. public, private or protected). Class nodes contain additional information to identify abstract classes and interfaces. Class nodes are connected to the classes they extend and the interfaces they implement through appropriate edges. Similarly, method nodes are connected to their declaring class nodes and parameter class nodes through appropriate edges. This database runs on a server and can be queried through a RESTful API service that we implemented. Each of these nodes is indexed based on their FQNs and their unqualified names to enable fast querying. Figure 4.3 presents an overview of how information and various relationships between API elements are preserved in the graph.

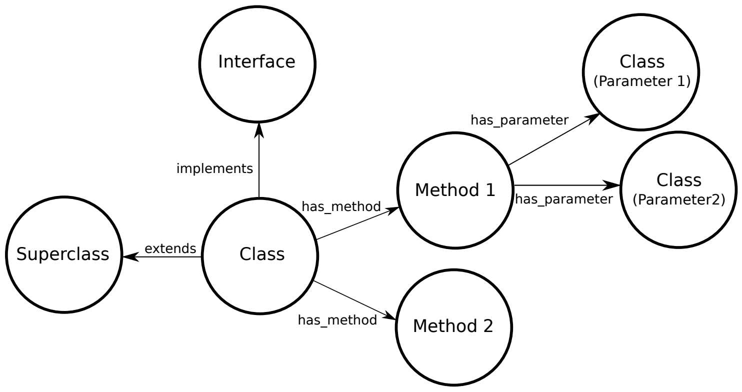


Figure 4.3: Template followed by the graph oracle.

The Java oracle can be dynamically updated by analyzing the appropriate JAR. We use a tool called Dependency Finder [3] to analyze and extract class, method and field signatures from the `.class` files contained in the JAR. These signatures are then parsed and augmented to the existing Neo4j oracle graph. We implement this as a web service to make the oracle update process as seamless as possible.

The initial oracle was quickly populated using API signatures extracted from the Maven repository [33]. This data dump contains over 14 million method signatures and 3 million field signatures belonging to 1.5 million classes. This information, when updated into the database, results in a graph with 32 million nodes and 89 million relationships.

[3] https://bitbucket.org/rtholmes/depfind_uw

4.2.2 JavaScript Oracle

The JavaScript oracle is built by statically analyzing the source files of JavaScript libraries. JavaScript libraries are often minified to reduce their size. In such cases, the source code is accompanied with a source-map that contains the mapping used to obfuscate the identifiers. When the libraries are obfuscated, we use this mapping to reconstruct the original source files and use them to build the oracle.

We use ESPRIMA [4] to parse the source code of each library and extract APIs. We build an Abstract Syntax Tree (AST) for each source file and walk this tree in a depth-first fashion to 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 have to be used to access these functions. Since object assignments in JavaScript are pass-by-reference, additional traversals of the AST are performed to map non-trivial and indirect Function Expression assignments.

As an example, consider the snippet of code from Backbone.js in Figure 4.4. A first pass is done to fetch all FunctionExpression variables, in this case, `extend`. Additional traversals of the AST are performed to identify and resolve transitive aliases like `History.extend` and `View.extend`, which inherit all properties of the `extend` object. With these additional passes, most 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. Since JavaScript does not have type information for objects returned from functions, our ability to reason about these assignments is lost. For this reason, we only follow such statements one level deep.

```
_.extend(History.prototype, Events, {

    getHash: function(window) {
        ...
      },
});
var extend = function(protoProps, staticProps){
    ...
};
View.extend = History.extend = extend;
```

Figure 4.4: A snippet from the source code of Backbone.js

[4] http://esprima.org/

The dynamic nature of JavaScript makes it harder to extract APIs from the source files through static analysis. Hence, whenever available, we make use of annotations provided by JSDoc [5] and other similar documentation tools. Furthermore, unlike Java, JavaScript lacks visibility annotations. The code elements are not marked as `public` or `private`. Private methods are hidden through various design patterns. This makes it difficult to distinguish private methods from public methods through static analysis and we might accidentally encounter false positives. However, they only slightly reduce Baker's accuracy since private methods are not used in code snippets and hence will rarely be matched.

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 Stack Overflow.

To dynamically add new JavaScript libraries to the oracle, we rely on *npm* [6], which is a package manager for *node.js* [7]. We use *npm* to fetch the source code of said library. Baker then analyzes the source code to populate API methods and properties, and adds them to the existing database of API signatures. Since the number of libraries analyzed is smaller than in the Java oracle, we currently use only a key-value store for the JavaScript oracle. However, we intend to implement the JavaScript oracle also as a graph database in the future.

4.2.3 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`).

In Java, method and field identifiers can be partially qualified if the identifier contains the type but not the package in which it is declared. For example, while the unqualified method name `getId()` occurs 27,434 times in the oracle, a few of these methods may have the same partially qualified names (`Node.getId()`). However, each of these methods can be uniquely identified by their fully qualified names (`org.neo4j.graphdb.Node.getId()`,

[5] https://github.com/jsdoc3/jsdoc
[6] http://npmjs.org
[7] http://nodejs.org

`jsx3.xml.Node.getId()`). Though method and field names can be partially qualified, class names cannot be partially qualified without considering the package identifier.

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 [18]. We extended their result using our oracle to 1.6 million types and extended the analysis to include types, methods, and fields. We also investigated the differences between fully qualified names (FQN), partially qualified names (PQN), and unqualified names (UN). Our analysis revealed that 80% of the API elements are ambiguous when unqualified. In fact, even partially qualified API elements, that a few previous approaches used, are ambiguous 37% of the time. Thus, they can only be disambiguated based on their fully qualified names. Table 4.2 provides an overview on the prevalence of naming collisions in Java API.

With respect to method names, 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. Moreover, APIs involved in naming collisions are commonly used in applications. Appendix B investigates the frequency of naming ambiguities in the applications that implement the Android android.

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

Table 4.2: Analysis of naming ambiguities in the Java oracle.

4.3 Snippet Parser

We use different parsing engines to statically analyze Java and JavaScript code to account for the differences in the languages. In both these engines, the code is wrapped around synthetic classes or methods as per the parser's requirements.

4.3.1 Java Parser

To create ASTs from Java snippets, we built a parser using the Eclipse Java Development Tools (JDT) project [8]. Since the Eclipse parser is robust to badly formed input, it was able to manage many of the problems associated with source code snippets. We also built a web service for the parser that enables us to use HTTP POST to send snippets of code to the Java Parser and get a JSON response with the results.

4.3.2 JavaScript Parser

JavaScript snippets are parsed by the ESPRIMA [9] parser. ESPRIMA is very tolerant of malformed input because it is frequently found in JavaScript code. We implemented the JavaScript parser as a web service so code snippets could easily be parsed from a variety of applications.

4.4 Example Database

The parsing framework parses code snippets extracted from Stack Overflow and maps code elements to the fully qualified API elements they reference. This information is output as a JSON file and can readily be used by other applications (like the documentation linking tool). These analyzed snippets and their results are stored in a SQL database (example database), and this data is exposed as a web service for external applications to access. The JSON representation of the output contains a list of all candidate elements for each code element along with the exact location of the code element in snippet, thereby facilitating further analysis of the snippet if required. Figure 4.5 represents a part of the output generated for the example Java snippet that is presented in Figure 1.2.

4.5 Enabling Live Documentation

We utilize Baker's ability to identify fully qualified API references in code snippets to create bi-directional links between API documentation and relevant source code examples on Stack Overflow. As previously discussed, keeping documentation current is challenging

[8] `http://eclipse.org/jdt/`
[9] `http://esprima.org`

```
"api_elements": [
    ...,
    {
        "cardinality": "1",
        "elementType": "api_type",
        "name": "History",
        "candidates": [
            "com.google.gwt.user.client.History"
        ],
        "locationInCode": "46",
        "lineNumber": "2"
    },
    ...
]
```

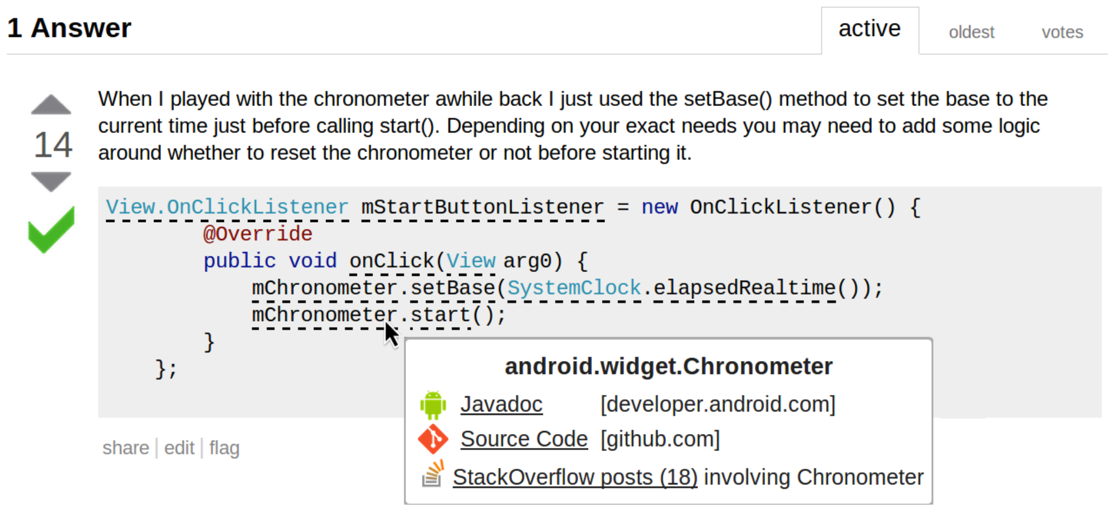
Figure 4.5: The JSON output generated by Baker for the Java code snippet in Figure 1.2.

and expensive. Therefore enabling these links automatically keeps the document up to date without 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 represent links to other related resources.

The snippet extractor module ensures that the example database is always current by continually monitoring Stack Overflow for new source code examples. Any API usage detected in these examples is added to the example database 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 is an explicit mapping between a FQN 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 4.6 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 the API elements that are present in the post and their locations. After consulting the mapping, the extension modifies the code block and inserts information to show all of the elements that Baker has identified by underlining them. In this case, Baker was able to correctly identify the FQN for `mChronometer` (in addition to several other types and methods). When the developer hovers their mouse over `mChronometer` (as they are in Figure 4.6), they are presented with a dynamic pop-up 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`.



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.

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

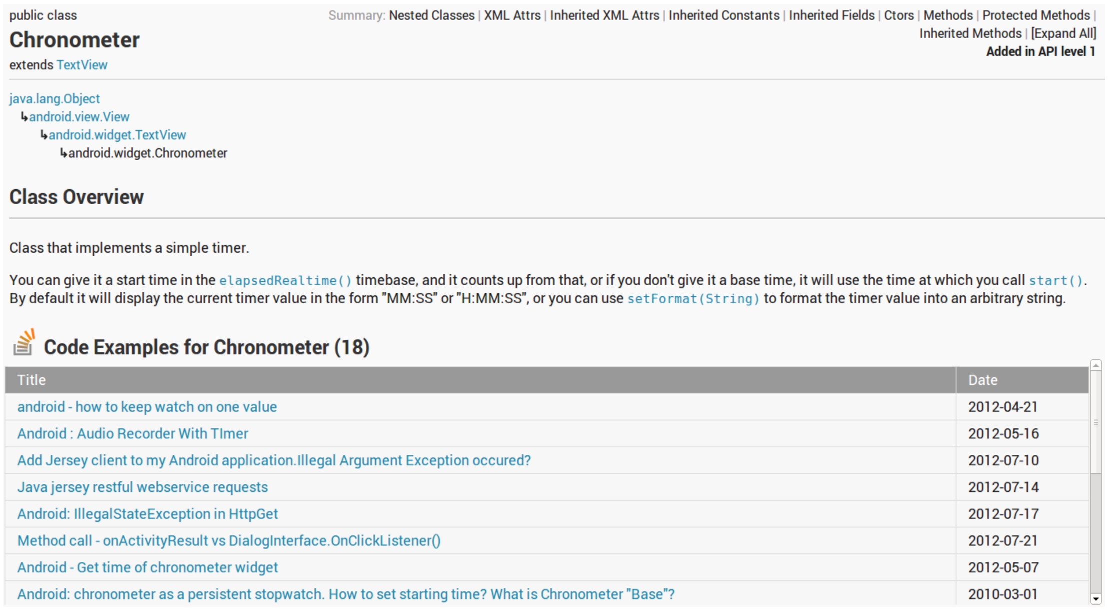
android.widget.Chronometer

Javadoc	[developer.android.com]
Source Code	[github.com]

StackOverflow posts (18) involving Chronometer

Figure 4.6: 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.

In the opposite direction, Figure 4.7 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 posts for code 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 any 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 both types and methods. Baker



public class
Chronometer
extends TextView

Summary: Nested Classes | XML Attrs | Inherited XML Attrs | Inherited Constants | Inherited Fields | Ctors | Methods | Protected Methods | Inherited Methods | [Expand All]
Added in API level 1

java.lang.Object
 ↳android.view.View
  ↳android.widget.TextView
   ↳android.widget.Chronometer

Class Overview

Class that implements a simple timer.

You can give it a start time in the `elapsedRealtime()` timebase, and it counts up from that, or if you don't give it a base time, it will use the time at which you call `start()`. By default it will display the current timer value in the form "MM:SS" or "H:MM:SS", or you can use `setFormat(String)` to format the timer value into an arbitrary string.

Code Examples for Chronometer (18)

Title	Date
android - how to keep watch on one value	2012-04-21
Android : Audio Recorder With TImer	2012-05-16
Add Jersey client to my Android application.Illegal Argument Exception occured?	2012-07-10
Java jersey restful webservice requests	2012-07-14
Android: IllegalStateException in HttpGet	2012-07-17
Method call - onActivityResult vs DialogInterface.OnClickListener()	2012-07-21
Android - Get time of chronometer widget	2012-05-07
Android: chronometer as a persistent stopwatch. How to set starting time? What is Chronometer "Base"?	2010-03-01

Figure 4.7: The Baker browser extension automatically injects a list of relevant Stack Overflow posts into the official Android API documentation. These links dynamically update without any intervention from the documentation team.

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 complement 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 [10].

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

Chapter 5

Results and Evaluation

In our evaluation of Baker, we answer two questions.

- Can Baker accurately identify API elements in code snippets?
- Does Baker work on a variety of systems, or is it limited to just a few libraries?

5.1 Linker Accuracy

To answer the first question, we manually examined Baker's results to check if API elements in Java and JavaScript code snippets are correctly disambiguated.

Firstly, we populated the snippet database using the data from the MSR Challenge [32] as described in the previous chapter. Additionally, we augmented this data by pulling source code from a few repositories on Github. Of the resulting snippet database, Baker analyzed 1,000 JavaScript source code snippets and 4,000 Java source code snippets.

In our context of linking examples to documentation, 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 [18] and ACE [10] papers and in Parnin's Stack Overflow study [13]. The five libraries used in these studies are listed in Table 5.1. We then randomly selected code snippets from Baker's example database that had been annotated with a Stack Overflow tag appropriate to the project under study. Whenever Baker claimed that it had identified an API element from one of the five libraries in Table 5.1, we manually examined the snippet and Baker's result to classify the result based on the following rules.

- If the result returned by Baker correctly matched the API intended by the developer, it is classified as a true positive (TP).
- If the result returned by Baker incorrectly matched the API, it is classified as a false positive (FP).
- If the result contained tokens that were not associated with an API element at all but that we would have expected to see a result, it is classified as a false negative (FN).

We stopped once we had examined 50 code elements for each system in this way. Baker's overall precision with Java code snippets 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.

System	**TP**	**FP**	**FN**$_{c=1}$	**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 5.1: Baker's overall Java precision (0.98) and recall (0.83). Only exact matches ($cardinality = 1$) were considered.

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 studies 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.

System	**TP**	**FP**	**FN**$_{c=1}$	**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

Table 5.2: Baker's overall JavaScript precision (0.97) and recall (0.96). Only exact matches ($cardinality = 1$) were considered.

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 fully qualified names identified by the tool to see the breadth of the systems it was able to generate links for.

Baker parsed 4,000 Java source code snippets. It identified over 30,000 links to 4,500 unique API elements. Table 5.3 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.

Baker parsed 1,000 JavaScript 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 5.4. 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

System	**No. of types**	**No. of 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 5.3: Number of matched elements from 4,000 Java code snippets extracted from Stack Overflow and Github. The types and method cells are split (total no. of matches / unique no. of elements).

library being asked about is not in the oracle, elements from other libraries are often found interspersed with these references.

System	**No. of 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 5.4: Number of matched elements from 1,000 JavaScript code snippets extracted from Stack Overflow and Github. The object and properties cells are split (total no. of matches / unique no. of elements).

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 5.3; this data is plotted in Figure 5.1.

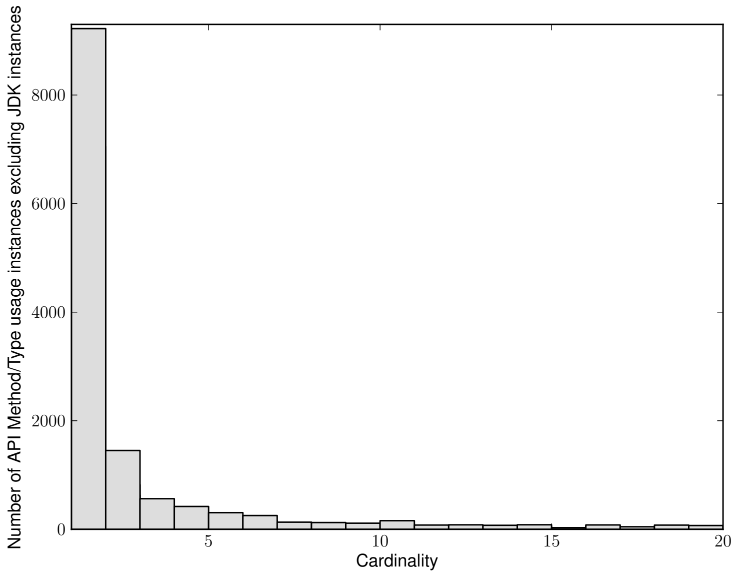


Figure 5.1: A histogram of the cardinality values obtained when the tool was run on the 4,000 code snippets described in Table 5.3.

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 5.5.

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`

Table 5.5: Example of an imprecise Baker match. The top row represents the correct answer.

In these cases, Baker's results contain the correct FQN; we intend to work on techniques for differentiating between these results in the future.

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.

Chapter 6

Discussion

The information extracted by Baker has a number of applications in addition to documentation linking. For example, the API elements in a documentation page can be reordered based on the number of API examples found on Stack Overflow for the element. 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. A similar concept is explored by Holmes et al. in PopCon [34], where they leverage a large static analysis repository rather than code examples.

Linking online discussions to API documentation can act as a feedback mechanism for library developers to identify various difficulties faced by developers in using their library's API. This information can be be used to appropriately improve the design or documentation of the API in future releases.

Since Baker manages to find fine-grained API references in code snippets, the resulting data can be used to build a code search engine that supports compound queries. For instance, we can query Baker for examples that use the method `elapsedRealtime` belonging to the `SystemClock` class of the Android framework, along with the `setBase` method of the `Chronometer` class. It can also help in finding *jungloids* (described in Chapter 2) relevant to a specific task.

Additionally, in cases where the libraries or frameworks used by a snippet are known beforehand, Baker's oracle can accordingly be reduced to contain just the relevant API. For example, Baker can use such a reduced oracle to analyze source code files from a collection of Android projects found on Github, thereby generating more precise results.

Baker can also be implemented as a plugin for development environments (IDEs). The data generated by Baker from Stack Overflow snippets can be used to recommend succinct code examples that precisely use the set of API elements that the developer is interested in.

Chapter 7

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 [18], ACE [10], and Parnin's Stack Overflow study [13]. Baker was executed in its default configuration for all studies. The only exception was that JavaScript snippets were only submitted to JavaScript parser while the Java snippets were sent to the Java parser.

The parsers we use in our implementation cannot handle code that is extremely malformed. For example, code with missing braces cannot be parsed. However, the goal of this thesis is to identify examples that can help developers better understand APIs and such examples are not of much use to a developer. Hence, we discard such snippets from the snippet database.

In this study, we restrict our examples to code snippets found in accepted answers on Stack Overflow to maintain example code quality. Though Baker can be extended to analyze snippets of code on Github and other online resources, we cannot guarantee the correctness of these snippets. Baker's goal is not to assess the correctness of the code, but to infer API references in them while assuming the code is correct.

Chapter 8

Future Work

A number of heuristics can be employed to improve Baker's results. In practice, most API elements that are used in software projects are used to solve a specific task, and thus belong to a small set of libraries. In fact, in many cases, a lot of these elements belong to the same package. Thus, when Baker identifies that a particular method in the code refers to `android.os.SystemClock.elapsedRealTime()` in the Android framework, it is more probable that a method called `setBase` in the same code snippet belongs to `android.widget.Chronometer` class of the same Android framework and not `org.hibernate.cfg.IndexColumn` class of the Hibernate library. Thus, we can use Baker's results to predict what libraries or packages may have been used in the snippet to tune facts collected about other elements in subsequent iterations.

The deductive linking algorithm used by Baker tries to identify the API type of an object based on the constraints put forth by the methods it invokes. The algorithm described in this thesis analyzes these constraints in the order of their appearance in the code. However, every candidate type satisfies these constraints irrespective of the ordering of these constraints. Hence, the algorithm can be improved by re-ordering the constraints (methods) based on how efficiently they reduce the list of candidate types of the object. In other words, before analyzing the candidates, the methods can be reordered based on the number of candidates they have in the oracle. For example, methods like `fireCurrentHistoryState()` have fewer candidates than a method like `start`. Hence, they can used earlier in the analysis of an object to significantly reduce the object's candidate list, thereby making the algorithm more efficient.

The current implementation of Baker relies solely on code snippets in posts to map them to their documentation. However, discussions associated with the snippets often contain useful information that can help disambiguate the candidates when the cardinality of Baker's output is > 1. In future, we intend to employ natural language processing techniques to extract code-like elements from plain text and use this information to improve our linking technique.

Among all the code snippets that reference an API element, some snippets are better examples than the others. We intend to construct a mechanism to estimate the usefulness of a snippet in the context of a specific API element. This estimate can be used to present the examples in decreasing order of their usefulness to the developer.

Linking Stack Overflow discussions to API documentation is just one part of a framework that links together multiple web-based resources like source code repositories, issue trackers and mailing lists. Enabling this linking can help us generate composite pages for API elements that provide easy access to multiple artefact repositories (as illustrated in Figure 1.1) [15]. We intend to carry this research forward and build this framework.

Finally, we aim to document and fully open the web services that power Baker. This would allow anyone to add new APIs to the Java and JavaScript oracles, update mapping files, submit snippets to be parsed, and query Baker. In addition, we will be releasing the browser extension so that other researchers and developers can try the tool and provide feedback.

Chapter 9

Conclusion

Maintaining API documentation is a challenging and time-consuming task; consequently, the documentation is frequently out of date. This thesis presented a method and tool for automatically generating links between API documentation and source code examples on Stack Overflow by using structural information in the code snippets. 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.

APPENDICES

Appendix A

Ambiguous Snippets on Stack Overflow

This chapter presents a few ambiguous code snippets we encountered on Stack Overflow during the course of this research. Each code snippet presented below is associated with a PostID. The corresponding post can be found at `www.stackoverflow.com/questions/PostID`.

A.1 Missing Statements

To focus on the core functionality, variable declarations and field declarations are frequently omitted from the code snippets. Sometimes, this is also caused when code snippets are broken into multiple code blocks by the textual descriptions that support the snippet.

The code snippet in Figure A.1 presents a a piece of Java code, where the statement declaring **mChronometer** is missing. Hence it is not clear what type this variable belongs to.

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

Figure A.1: Stack Overflow PostID: 1916391

Additionally, the snippet only contains a block of statements. It lacks import statements and information about the class and method that surround these statements.

A.2 Use of Ellipses

Ellipses ("...") are frequently used in code snippets to elide functionality that is not relevant to the context of the post.

In the code snippet in Figure A.2, a large part of the code block is replaced with an ellipsis. This results in snippets that are syntactically incorrect and hinders their analysis.

```
private void doClear(int y, int x, JButton[][] bArray2, int gridy,int gridx)
{
    if (...already cleared...) {
       return;
    }

    ...
}
```

Figure A.2: Stack Overflow PostID: 6272964

Appendix B

An Analysis of Naming Collisions in the Android Framework API

To investigate if ambiguous APIs were frequently used in applications or if they were limited to less used API, we examined applications that used the Android framework. We obtained 47 example applications from the Android SDK that contained approximately 600 source files and 50,000 source lines of code. We examined the most-used non-constructor method names from this corpus. Table B.1 contains the top ten method names, along with the number of methods in the Android API having the same name (column 2) and the number of packages these declarations are spread across (column 3).

Method Name	No. of Collisions	No. of Packages
`onCreate`	32	15
`show`	17	2
`getResources`	9	6
`getContentResolver`	4	2
`getItemId`	7	2
`getAction`	6	3
`findViewById`	5	3
`getString`	34	12
`getData`	12	9
`close`	137	36

Table B.1: Naming collisions among the 10 most-used Android methods.

References

[1] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 643–652, ACM, 2014.

[2] S. Subramanian and R. Holmes, "Making sense of online code snippets," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pp. 85–88, ACM, 2013.

[3] L. P. Deutsch, "Design reuse and frameworks in the smalltalk-80 system," in *Software Reusability*, pp. 57–71, ACM, 1989.

[4] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Software*, vol. 26, pp. 27–34, Nov. 2009.

[5] G. Butler, R. K. Keller, and H. Mili, "A Framework for framework documentation," *ACM Computing Surveys (CSUR)*, vol. 32, p. 15, March 2000.

[6] J. Bloch, "How to design a good API and why it matters," in *Companion to the ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 506–507, ACM, 2006.

[7] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Softw.*, vol. 20, pp. 35–39, Nov. 2003.

[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)*, pp. 127–136, ACM, 2010.

[9] A. Bacchelli, M. Lanza, and R. Robbes, “Linking e-mails and source code artifacts,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 375–384, ACM, 2010.

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

[11] 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)*, pp. 25–30, ACM, 2011.

[12] H. Li, Z. Xing, X. Peng, and W. Zhao, “What help do developers seek, when and how?,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 142–151, IEEE, 2013.

[13] C. Parnin, C. Treude, L. Grammel, and M.-A. D. Storey, “Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow,” *Georgia Institute of Technology, Tech. Report*, no. GIT-CS-12-05, 2012.

[14] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, “Design lessons from the fastest Q&A site in the west,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2857–2866, ACM, 2011.

[15] L. Inozemtseva, S. Subramanian, and R. Holmes, “Integrating software project resources using source code identifiers,” in *Proceedings of the International Conference on Software Engineering (ICSE - NIER Track)*, pp. 400–403, ACM, 2014.

[16] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, “Recovering traceability links between code and documentation,” *IEEE Transactions of Software Engineering*, vol. 28, pp. 970–983, October 2002.

[17] 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)*, pp. 125–135, ACM, 2003.

[18] 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)*, pp. 47–57, ACM, 2012.

[19] C. Treude, O. Barzilay, and M.-A. D. Storey, "How do programmers ask and answer questions on the web?," in *Proceedings of the International Conference on Software Engineering (ICSE - NIER Track)*, pp. 804–807, ACM, 2011.

[20] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable APIs," *Software, IEEE*, vol. 15, pp. 78–86, May 1998.

[21] R. Holmes and G. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 117–125, ACM, May 2005.

[22] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid Mining: Helping to navigate the API jungle," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 48–61, ACM, 2005.

[23] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 204–213, ACM, 2007.

[24] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pp. 318–343, Springer-Verlag, 2009.

[25] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 111–120, ACM, May 2011.

[26] X. Chen, "Extraction and visualization of traceability relationships between documents and source code," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pp. 505–510, ACM, 2010.

[27] 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)*, pp. 839–842, ACM, 2008.

[28] 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)*, pp. 59–68, ACM, 2008.

[29] 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)*, pp. 313–328, ACM, 2008.

[30] L. Moonen, “Generating robust parsers using island grammars,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 13–22, IEEE, 2001.

[31] A. Bacchelli, L. Ponzanelli, and M. Lanza, “Harnessing stack overflow for the IDE,” in *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pp. 26–30, IEEE, June 2012.

[32] A. Bacchelli, “Mining Challenge 2013: Stack Overflow,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, ACM, 2013.

[33] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, “Software bertillonage,” *Empirical Software Engineering*, vol. 18, no. 6, pp. 1195–1237, 2013.

[34] R. Holmes and R. J. Walker, “A newbie’s guide to Eclipse APIs,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pp. 149–152, ACM, 2008.