Making Sense of Online Code Snippets

Siddharth Subramanian, Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
s23subra@uwaterloo.ca, rtholmes@cs.uwaterloo.ca

***Abstract*—Stack Overflow contains a large number of high-quality source code snippets. The quality of these snippets has been verified by users marking them as solving a specific problem. Stack Overflow treats source code snippets as plain text and searches surface snippets as they would any other text. Unfortunately, plain text does not capture the structural qualities of these snippets; for example, snippets frequently refer to specific API (e.g., Android), but by treating the snippets as text, linkage to the Android API is not always apparent. We perform snippet analysis to extract structural information from short plain-text snippets that are often found in Stack Overflow. This analysis is able to identify 253,137 method calls and type references from 21,250 Stack Overflow code snippets. We show how identifying these structural relationships from snippets could perform better than lexical search over code blocks in practice.**

I. Introduction

Developers frequently reuse existing libraries and frameworks while creating new systems [1]. Unfortunately, figuring how to use many of these complex frameworks is not obvious [2]. This is often due to a lack of documentation or source code examples that illustrate the correct way to use the framework's APIs [3].

For example, while using existing libraries or frameworks, developers often know what type of object they need, but do not know the sequence of steps involved in accessing it [4], [5]. In these situations, developers often look for solutions on the web. Stack Overflow [1] is one popular web site to look for such usage examples [6]. It is common to encounter posts on Stack Overflow that contain source code snippets that demonstrate a solution to a particular programming task or the usage of a specific API. Stack Overflow also lets users annotate the best solution to a question based on whether the answer helped them with their original problem. This metadata provides insight into the quality and correctness of the code included in these answers. Thus, these code snippets represent high quality source code examples that demonstrate API usage.

Unfortunately Stack Overflow treats source code snippets as plain text; this causes searches to fail to utilize any structural information present in the code snippet and thus masks good API usage examples. For example, it is common for multiple API methods belonging to different classes to share similar names. Hence, a search for a particular method on Stack Overflow often returns results that are not relevant to the developer's query. Also, other information, such as type usage examples can be lost when method calls are chained together and the types are implicitly used but are never explicitly named. These facts can make it difficult for a user to identify relevant results without having a thorough understanding of the API.

To solve this problem, we perform static analysis on code snippets to extract structural information from them. Since snippets are usually only partial programs, we rely on an external oracle for additional API information to increase our ability to reason about the partial programs. We identify various API classes being used in the code and resolve method invocations and implicit type usage by inferring details from the snippets and correlating it with our oracle.

We present the results of our analysis on accepted answers containing source code snippets from Stack Overflow on questions that were tagged `Android`. We were able to successfully resolve 177,799 type references and 75,338 method invocations from 21,250 code snippets. On average, our approach identified eleven Android API elements from each code snippet.

This paper makes the following contributions:

- An analysis of the raw Stack Overflow source code snippets (Section III).
- A partial program analysis that uses an easily-satisfiable oracle to extract a structural model of code snippets in Stack Overflow source code examples (Section IV).
- A list of Android API types and methods that are most referenced in Stack Overflow snippets (Section V).

II. Motivation

The Stack Overflow search mechanism treats source code snippets as plain text; it does not attempt to leverage the underlying structural information from the snippets. Consider the following piece of code extracted from a Stack Overflow answer marked solved:

```
public View getView(final int position,
   View convertView, final ViewGroup parent) {
      ...
      parent.getChildAt(position)
            .findViewById(R.id.progressbar_Horizontal)
            .setVisibility(View.VISIBLE);
      ...
}
```

The Android API contains six methods named `setVisibility`. This means that a developer searching for example for a specific `setVisibility` method will frequently

[1] http://stackoverflow.com

encounter code snippets for versions of `setVisibility` declared in types they are not interested in. From the snippet alone it can be difficult to determine which specific API elements are being used. Using the class name along with the method name as a search string is not a viable solution (e.g., in the snippet above we cannot determine the type that `setVisibility` is being invoked on). Type information from a few method invocations often goes unnoticed due to the kinds of method chaining seen above (which is not uncommon in object oriented languages). Searching for the fully qualified name returns no relevant results since the textual code does not contain any fully-qualified names. Similarly, in the above snippet `findViewById` is also not unique; there are five methods with this name in the Android API as well.

Ideally, we would like to be able to search on the text snippet above and identify all the relevant information about the API usage present. For the snippet above, we would like to know that `android.view.ViewGroup.getChildAt` method takes an `int` argument, returns a `View` object. We would also like to know that this `View` has a method `findViewById` which can be resolved to `android.view.View.findViewById(int)` and similarly `setVisibilty` can be resolved to `android.view.View.setVisibility(int)`.

Being able to resolve this level of detail from source code snippets increases their utility because it can identify information that is not present in the text itself. The remainder of this paper describes how this can be achieved.

III. Stack Overflow Snippets

Online sources play an important role in API documentation. Sources such as blog posts and Stack Overflow achieve a high level of API coverage [6]. A study by Parnin et. al. [7] revealed that 87% of all Android API classes and 77% of all Java API classes had at least one related post on Stack Overflow. Moreover, 56% of the Android API classes were covered by code examples in accepted answers. We analysed the Stack Overflow data dump [8] (repository) to further investigate these source code snippets. Of all posts, we observed that 65% of the 2.1 million accepted answers had code snippets in them.

Among posts tagged `Android`, 65,095 accepted answers out of 111,733 had code snippets in them (58%). Considering only code snippets that had at least 3 lines of code (LOC), we observed 39,000 source code snippets in the repository with a mean size of 16.4 LOC and a median of 9 LOC. We set the lower bound at 3 LOC because manual investigation found that shorter snippets often lacked surrounding context that was essential to help really understand an API (in contrast to just answering a question or providing a short fragment of pseudo-code). Figure 1 provides a histogram showing the frequency of each sized code snippet ($>$ 2 LOC).

From the 39,000 code snippets we analysed, we observed that only 6,766 (17%) were complete files with class and method declarations. 6,302 (16%) code snippets were just method bodies devoid of class declarations. The remaining 66% contained standalone source code statements. As these standalone statements were so prevalent we tailored our approach to be able to handle these cases. In our approach, we identify API examples from code snippets while treating all of these cases with equal importance.

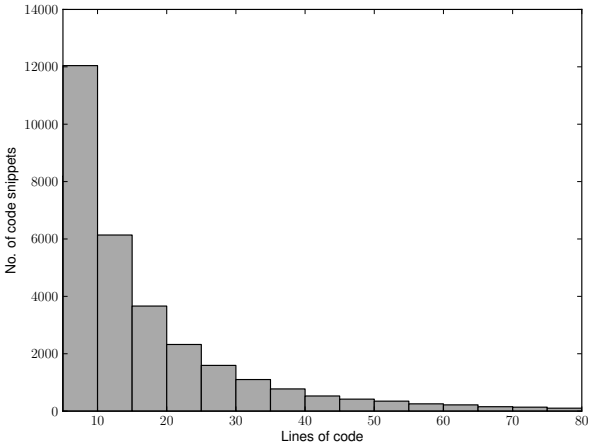

Fig. 1: Distribution of code in posts tagged `Android` ($>$ 2 LOC)

IV. Approach

Source code snippets are by their nature *incomplete*. Most answers extend on details provided in the question; because of this, certain aspects of the snippet, like variable declarations, are often skipped. These snippets are also often supplemented code with explanations, providing a more complete answer than the snippet alone. These explanations are often provided inline, that is, large snippets of code are broken into smaller ones spread across several code blocks. This also complicates analysing the snippets as it is not possible to gauge if the snippets belong together or not. In our approach, we treat each code block as a separate code snippet.

The core of our approach is a snippet analysis framework, much like the partial program analysis framework PPA, that was built by Dagenais et. al. [9]. While PPA parsed individual complete files, our approach instead works on arbitrarily small code fragments. Unlike PPA, which works on individual files alone, we supplement our framework with a simple oracle that describes the API space likely to be found in the code fragments under analysis.

For example, in this paper we are exploring 'solved' Stack Overflow questions that have been given the `Android` tag. For these questions, we provide our framework with an oracle consisting of the Android API (this is generated by a simple program that examines the `android.jar` file that ships with the Android SDK). The oracle contains all of the types and methods present within the library under analysis.

A. Parsing Snippets

Our approach constructs an Abstract Syntax Tree (AST) for each code snippet. To ensure quality, we only considered

snippets that were greater than 2 LOC and were from answers marked as solutions by the developer who asked the question. Since the Android framework is based on Java, the snippets we encounter are pieces of Java code. While some of these snippets are compilable code fragments with complete class and method body declarations, the majority are not; while some contain method bodies, the majority consist of a free-standing code statements. To generate the AST, we use Eclipse Java Development Tools (JDT)[2] which provides a Document Object Model (DOM) representation of the AST; the Eclipse parser takes a 'best effort' approach to parsing code. We used a standalone implementation of the parser, enabling us to programmatically invoke the JDT parser and parse arbitrarily source code snippets. As the JDT can only parse code which contains class and method declarations, we wrap free standing snippets accordingly before parsing them.

Since the code is usually not complete, information present in the code is often not sufficient to resolve API method accesses.

B. Inferring API Usage

To deal with incomplete information from the parsing step we employ a simple set of heuristics. Our approach works as follows:

1) We traverse the AST, collecting type information at variable declaration and field declaration nodes. We then query the oracle to retrieve a list of fully qualified classes that each variable can possibly belong to and annotate the variable with its set of type possibilities. We also keep track of type information for parameters used in method declarations.
2) We then use the variable type information to resolve method calls. When a method invocation is encountered, we retrieve the list of candidate types from the corresponding object reference. We query the oracle to identify which of these types declare methods with the same signature as the one being invoked. We annotate each method invocation in the AST with its set of candidate fully qualified methods. A list of fully qualified candidate return types is also maintained to resolve chained method invocations. We use similar techniques to identify API elements from anonymous class declarations. Finally, we identify overridden methods to infer information about interfaces and superclasses.
3) If a method invocation node does not have a corresponding object reference, we query the oracle for a list of all possible candidate API elements. Based on the information collected at every node, we update our beliefs about the types and methods resolved previously. This way, we predict type information for objects that cannot normally be resolved.

For `Android`-tagged examples in the repository, our inference method is able to identify 253,137 API classes and methods being used from 21,250 source code snippets which referenced the Android API. This contained 75,338 API method references and 177,799 API classes that have only one candidate element; that is, our approach yields an exact API match for these elements. Figure 2 shows a histogram depicting the distribution of API elements in code snippets that could be exactly resolved by our approach.

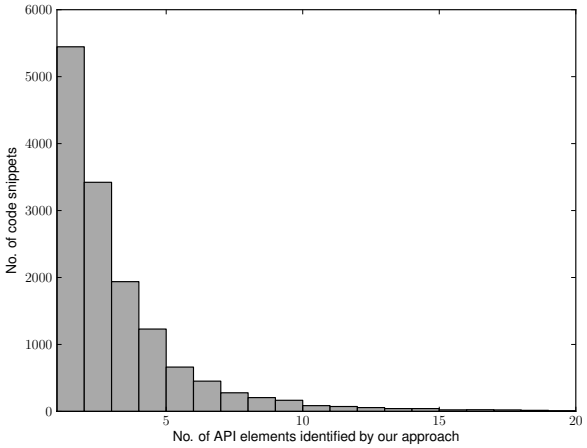

Fig. 2: Distribution of API elements in posts tagged `Android` ($>$ 2 LOC) as identified by our approach

V. Snippet Search Data

From the API usage information collected using our approach, we analysed the most commonly used API elements in code snippets on Stack Overflow. Table I and II list the most frequently used API types and methods, and the number of times they are used in code snippets in `Android` tagged posts on Stack Overflow.

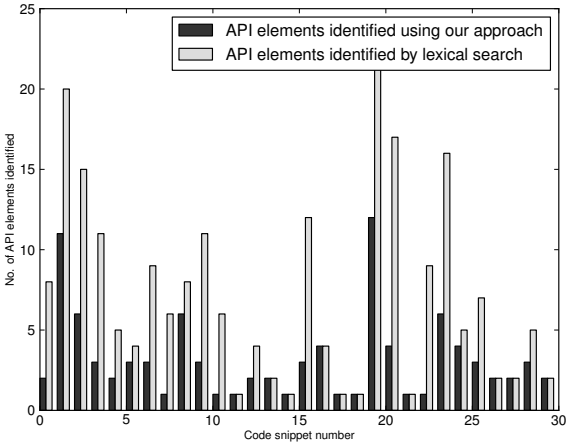

Fig. 3: Distribution of API elements in posts tagged `Android` ($>$ 2 LOC)

Our approach thus far has discussed only exact matches. Upon analysing the Android API we noted that the API

[2]http://www.eclipse.org/jdt/

TABLE I: Most-referenced Android API Types

API Type	Count
android.content.Intent	10550
android.view.View	8519
android.widget.TextView	5621
android.app.Activity	5473
android.os.Bundle	4503

TABLE II: Most-referenced Android API Methods

API Method	Count
android.view.View.findViewById(int)	1257
android.app.Activity.onCreate(android.os.Bundle)	1177
android.app.Activity.findViewById(int)	1174
android.util.Log.d(java.lang.String, java.lang.String)	1161
android.widget.Toast.show()	1063

consists of 24,545 total method declarations excluding constructors. Of these, 6,720 are unique. Of the remaining 17,825, each method name clashes with on average 33 other methods. For example, `describeContents` clashes with declarations from 158 different types. From our snippet analysis we found 23,239 instances where we mapped a call to a method name that we could not fully disambiguate. The search engine provides developers with a form of faceted search that enables them to investigate the alternative methods with the same name; this can allow them to more quickly identify the exact instance of the name that is relevant to them.

As the standard Stack Overflow search treats code snippets as plain text, we compared our approach to Stack Overflow results. To do this we randomly selected 30 code snippets from Stack Overflow and ran our analysis to identify the exact method names they contained. We then compared them to the number of different method names a lexical approach (such as that taken by Stack Overflow itself) would return. In fairness we opted to count methods that have more than 10 duplicate as one as we think it is unlikely that a developer would search for such a common name (e.g., `writeToParcel` has 172 declarations). The results of this comparison are in Figure 3. Ultimately, our snippet analysis approach is able to decrease mis-reported results by 51% compared to lexical approaches.

VI. Conclusion

In this paper we have explored source code snippets given in response to Stack Overflow questions. Specifically, we have explored the space of code snippets that utilize the Android API from accepted solutions. We believe that by focusing on accepted solutions we are more likely to identify 'best practice' API usage. We have developed an approach that can parse these short code snippets to effectively identify API usage. We found that Android-tagged Stack Overflow snippets contain on average 9 type references and 4 method calls that could be utilized to improve code search and link documentation with developer queries.

References

[1] L. P. Deutsch, "Software reusability," T. J. Biggerstaff and A. J. Perlis, Eds., 1989, ch. Design reuse and frameworks in the Smalltalk-80 system, pp. 57–71. [Online]. Available: http://doi.acm.org/10.1145/75722.75725

[2] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi, "Building more usable APIs," *IEEE Software*, vol. 15, no. 3, pp. 78–86, May/Jun.

[3] J. Singer, "Practices of software maintenance," in *Proceedings of the International Conference on Software Maintenance*, nov 1998, pp. 139 –145.

[4] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proceedings of the conference on Programming language design and implementation*, 2005, pp. 48–61. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065018

[5] 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*, 2007, pp. 204–213. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321663

[6] C. Parnin and C. Treude, "Measuring API documentation on the web," in *Proceedings of the International Workshop on Web 2.0 for Software Engineering*, 2011, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/1984701.1984706

[7] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey, "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow," Georgia Tech, Tech. Rep., 2012.

[8] A. Bacchelli, "Mining challenge 2013: Stack overflow," in *The Working Conference on Mining Software Repositories*, 2013, To Appear. [Online]. Available: http://2013.msrconf.org/challenge.php

[9] 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*, 2008, pp. 313–328. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449790