Augmenting Stack Overflow with API Usage Patterns Mined from GitHub

Tianyi Zhang* Myoungkyu Song† Miryung Kim*
*University of California, Los Angeles, USA †University of Texas at Austin, USA
tianyi.zhang@cs.ucla.edu, mksong1117@utexas.edu, miryung@cs.ucla.edu

ABSTRACT

Programmers often consult Q&A websites like Stack Overflow to learn new APIs. However, such online code examples are not always complete or reliable in terms of API usage. To assess and augment Stack Overflow examples, we present an interactive approach, MAPLE that (1) mines and renders API usage patterns from over 7 million GitHub projects, (2) detects API usage violations by matching the mined patterns against Stack Overflow examples, and (3) solicits user feedback on the usefulness of learned patterns. With the assistance of MAPLE, programmers do not need to cross-check multiple examples for proper API usage reference, and can also build confidence by learning how many other GitHub projects also follow the same practice.

Categories and Subject Descriptors

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

General Terms

Keywords

1. INTRODUCTION

Programmers often search for code examples question-and-answer sites like Stack Overflow to learn new APIs. However, the examples on Stack Overflow are not always complete or reliable in terms of API usage, which can be misleading and potentially dangerous. We have observed that over 50% of 31,801 Stack Overflow posts contain API misuse that could produce symptoms of program crashes and resource leaks if reused in a target system. Neglecting to close an input stream, for example, could lead to data leakage or generally unreliable code.

To assess code quality of online code examples, an API usage mining approach, Maple, was designed and implemented (ref to Maple paper). Given a code example from Stack Overflow, Maple extracts API call sequences from the example and contrasts the sequence with API usage patterns learned from 7 million GitHub projects. If the API call sequence of the example do not follow the commonaly practiced patterns, the example is flagged for potential API misuse.

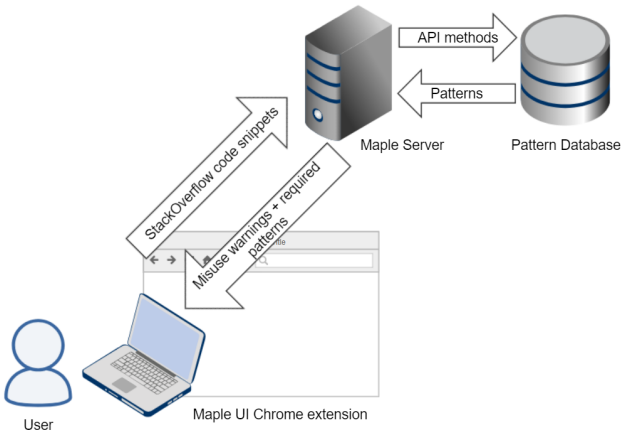


Figure 1: An overview of the system flow for Maple's user-interface

In order to give programmers access to Maple, we present a user interface designed with the intention of integrating user feedback into the application in order to enrich the mined patterns. Maple returns API violations and violation patterns to the user, which they may upvote or downvote based on its applicability to the code example being assessed and helpfulness to the user. Maple also provides users with the number of GitHub examples that also follow the same pattern, which helps a user to build confidence on the commonality of an API usage pattern.

A user of this tool would benefit from not needing to cross-reference multiple sites for proper API usage reference, and will be able to continue using Stack Overflow to learn APIs with the added advantage of seeing which usage patterns a post may have left out of its explanation. This could result in more complete, reliable code with minimal added time or effort on the part of the programmer.**TODO**: The description here does not sound very appealing. We need to rework this paragraph.

This paper's main contribution is to design and implement the interactive user-interface for Maple. The Chrome plug-in is available for download at `https://sites.google.com/a/utexas.edu/critics/`.

FSE'14 , November 16–22, 2014, Hong Kong, China

For better understanding, let me give you an example:

```
import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonPrimitive;

public class GsonTest {

    public static void main(String[] args) {
        JsonElement jsonElement = new JsonPrimitive("foo");

        System.out.println(jsonElement.toString());
        System.out.println(jsonElement.getAsString());

        jsonElement = new JsonPrimitive(42);

        System.out.println(jsonElement.toString());
        System.out.println(jsonElement.getAsString());

        jsonElement = new JsonPrimitive(true);

        System.out.println(jsonElement.toString());
        System.out.println(jsonElement.getAsString());

        jsonElement = new JsonObject();
        ((JsonObject) jsonElement).addProperty("foo", "bar");
        ((JsonObject) jsonElement).addProperty("foo2", 42);

        System.out.println(jsonElement.toString());
        System.out.println(jsonElement.getAsString());
    }
}
```

Figure 2: A code snippet that does not properly check `JsonElement.getAsString`.[1]

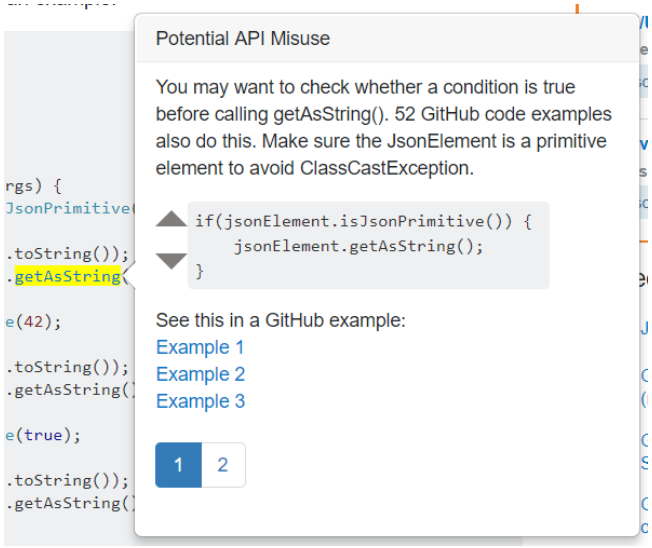
2. MOTIVATING EXAMPLES AND TOOL FEATURES

Consider Alice, a software developer who needs to develop a feature for a program using Google's Gson library for Java[2], which she is unfamiliar with. Alice uses an Internet search engine to look up how to use Gson's `JsonElement`'s `getAsString()` method and finds a post on Stack Overflow, as shown in Figure 2. However, this example does not use the `JsonElement` API completely correctly.

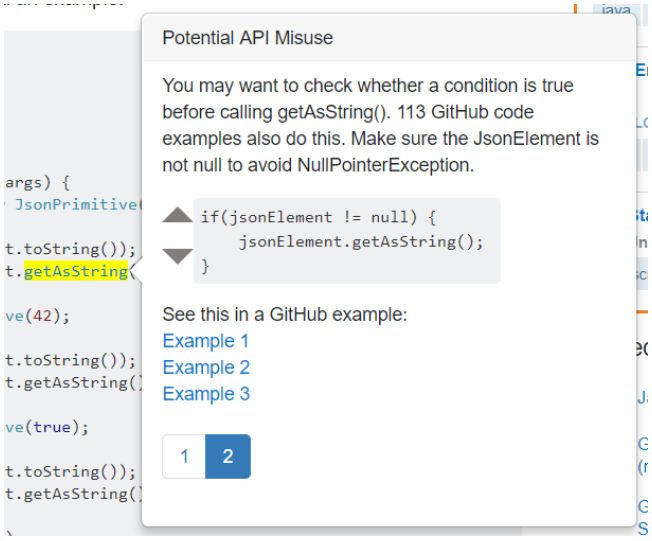
Stack Overflow Post View. Alice is running Maple's Chrome extension when she finds this post, and the extension highlights the potential API misuse in the code snippet, as seen in Figure 2. Note that the extension only highlights the first instance of this misuse although it occurs multiple times in the code snippet, to eliminate redundancy and avoid confusion. Alice is interested in learning more about the API and what specifically the code snippet did not include, so she clicks on the highlighted text.

Stack Overflow Popup View. Clicking on the highlighted text reveals a popup, as seen in Figure 3. The popup is populated with information about any required patterns in Maple's database this particular API call does not adhere to. Alice notices that there are two pages of the popup, indicating two different usage patterns that this call does not follow, as shown in Figures 3a and 3b.

Alice inspects the first page (Fig. 3a) and learns that she should check whether the `JsonElement` object is a primitive before calling `getAsString` in order to avoid a `ClassCastException`. She notices that 52 other GitHub code examples use this pattern, which gives her a qualitative measurement of how prevalent this pattern is in compiled, "real world" code. Alice then inspects the second page (Fig. 3b), and finds that it suggests a null check before calling `getAsString`. She notices that this pattern has more than double the support of the previous pattern.



(a) A page describing a way to avoid a `ClassCastException` by checking whether the `JsonElement` object is a primitive.



(b) A page describing a way to avoid a `NullPointerException` by checking whether the `JsonElement` object is null.

Figure 3: The two pages of a popup generated on `JsonElement.getAsString`.

[1]https://stackoverflow.com/questions/34120882/gson-jsonelement-getasstring-vs-jsonelement-tostring
[2]https://github.com/google/gson/blob/master/UserGuide.md

```
99
100         private static String getSelectedLauncherProfile(File minecraftDir) {
101             File path = new File(minecraftDir, "launcher_profiles.json");
102             JsonObject json = JsonUtils.parseJson(path);
103
104             if (json != null) {
105                 JsonElement element = json.get("selectedProfile");
106
107                 if (element != null && element.isJsonPrimitive()) {
108                     return element.getAsString();
109                 }
110             }
111
112             return null;
113         }
114
```

(a) The first GitHub example for Figure 3a.

```
175     @Override
176     public void stringProperty(PropertyName name, JsonObject context) throws IOException {
177         JsonElement prop = property(context, name);
178         if (prop == null) {
179             return;
180         }
181         builder.add(name, prop.getAsString());
182     }
```

(b) The second GitHub example for Figure 3b.

Figure 4: The highlighted GitHub examples redirected to from the links provided in the popup.

Curious to see the first pattern in context, Alice returns to the first page of the popup and clicks on the first link provided to her under "See this in a GitHub example."

GitHub Example View. When Alice clicks on one of the GitHub links, the file opens in a new tab and the view scrolls to where the API is called in the file, and the method in which this occurs is highlighted so Alice can easily find it, as seen in Figure 4. The addition of a compilable code example that demonstrates the pattern in context can aid Alice in understanding how to use the pattern if it is unfamiliar to her. In this case, Alice finds herself redirected the method in a GitHub project seen in Figure 4a. She notices that the example uses a null check in conjunction with the primitive check, which makes sense to her after seeing that both were missing from the Stack Overflow code snippet.

Returning to the popup in Stack Overflow, Alice clicks on the second link provided for the second page to compare usage patterns in context. This link opens up to the GitHub method seen in Figure 4b. Unlike the one in Figure 4a, this example does not use the null check in conjunction with the primitive check.

After seeing these two examples, Alice can infer that a null check is more necessary and is more common than the primitive check, based on the GitHub examples she has seen as well as the GitHub support indicated by the popup message. She upvotes the null check's pattern by clicking on the up-arrow on its page (see Figure 3b) to send the server her feedback on the patterns it gave her.

3. IMPLEMENTATION

This section describes the implementation details of the Maple plugin. It is implemented as an Chrome extension and consists of five components: (1) Stack Overflow code snippet extraction, (2) API misuse detection, (3) popup generation, (4) GitHub example alteration, and (5) a user feedback interface.

Code Snippet Extraction. When a user loads a page in the Stack Overflow domain, the plug-in side of the Maple interface extracts any text in answer posts that appears within `<code>` tags, and sends them in a JSON message to the server, as seen in Figure 1.

API Misuse Detection. When the server receives a message from the plug-in, it parses the snippets into API call sequences, abstracting away irrelevant statements and syntactic details. An API call sequence consists of relevant control constructs and API calls, where the API calls are annotated with the number of arguments as well as any guard conditions associated with it.

Once the code snippets are parsed, Maple searches a MySQL database for the API calls present in each API call sequence. It receives the required and alternative patterns associated with these calls, and checks whether the code snippet's call sequence satisfies one of the alternative patterns and all required patterns. A code snippet's call sequence satisfies a pattern if it is subsumed by it.

Additionally, while this checking process is occurring, the guard conditions in the code snippets are generalized before checking their logical implications using the SMT solver Z3. Z3 is used to check the logical equivalence between two guard conditions so that Maple can merge logically-similar clusters of guard conditions, making it able to prove the semantic equivalence of conditions like `arg0<arg1` and `arg1>arg0` regardless of their syntactic similarity [much of this is lifted pretty directly from p4 of the Maple paper...].

If Maple finds that an API call sequence does not satisfy the necessary patterns, it generates violations for each potential API misuse. The server collects these violations, along with the required pattern being violated, and maps them to the API call that generated the violation. This data is wrapped into a single JSON message and returned to the plug-in.

Popup Generation. Using the data from the server's JSON message, the plug-in searches the identified code snippet for the API call in question, highlights it, and generates a Bootstrap popover on it, as seen in Figure 3. The popover is populated with a violation message describing the pattern being violated and including the GitHub support for the required pattern, in terms of number of supporting projects as well as three links to relevant GitHub pages using that pattern correctly. The plug-in generates one popover for each API call, and generates pages of the popover for each different pattern being violated by that call. The provided example of the pattern is currently hard-coded.

GitHub Example Alteration. When the user clicks on a provided GitHub example in the popup, the plug-in's main script writes the name of the method call associated with that link to a shared storage space for the Chrome extension, using the chrome.storage API. When the GitHub page is loaded, the plug-in's secondary script is activated and checks the storage for any messages. If a method name has been written there, the secondary script searches the GitHub page for a method declaration of the same name. This is not necessarily the name of the API call the popup has been generated on; this is a method in a GitHub code file that uses that API call in a pattern that the Stack Overflow code snippet does not adhere to. The script highlights the entire method and scrolls the view down so the user is easily able to find it, as seen in Figure 4.

User Feedback. Users are able to give feedback on the patterns the popup shows them by voting "up" or "down"

on them. When a vote is registered, the plug-in sends the server a message with the pattern's specific ID and a vote of +1 or -1, which the server uses to update the database. This will be used in the future to rank patterns that are sent to the plug-in, and learn which ones users find helpful or unhelpful.

4. RELATED WORK

5. SUMMARY

The Maple user interface is an approach to enriching code examples on Q&A sites like Stack Overflow that incorporates API usage mining and user feedback. Programmers often use sites like Stack Overflow to understand API usage, but the reliability of the code examples on these forums is under question. This will reduce the need to cross-check sites for API usage questions and takes advantage of the benefits of using easy-to-understand code examples as opposed to API documentation for learning API usage by enhancing the examples already present in Stack Overflow while also providing other, more reliable ones.

For future work, we would like to qualify the benefits of the Maple UI with a user study, as well as generate code examples for the popups based on the abstract patterns we have in the database.

6. ACKNOWLEDGEMENT

7. REFERENCES