Locating API Misuse Examples in GitHub Data

Thien Bui-Nguyen

A thesis submitted for the degree of
Bachelor of Advanced Computing (Honours) at
The Australian National University

October 2019

Except where otherwise indicated, this thesis is my own original work.

Thien Bui-Nguyen
10 October 2019

to my xxx, yyy (yyy is the people you want to dedicated this thesis to.)

Acknowledgments

Who do you want to thank?

Abstract

Put your abstract here.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An Application Programming Interface (API) provides a set of functions to interact with some software. However, APIs often contain numerous constraints that developers must abide for correct usage of a given API. Misuse of an API can lead to severe bugs for developers such as software failures. These constraints are documented by the developers of an API in their formal documentation, which are usually created manually by the API developers or automatically generated from source code comments such as with the Javadoc tool for the Java programming language. A recent study by Li et al. researched a particular subset of constraints that focused specifically on correct and incorrect API usage that are referred to as *API caveats*. Furthermore, Li et al. proposed an extraction method for these API caveats at the sentence level by identifying common keywords between them, improving the accessibility of these API caveats. In addition to this, an approach by Sun to simplify these API caveats for developers and improve understanding was proposed using Natural Language Processing (NLP) techniques. This involved using word/sentence embedding and comparing sentence similarity of sentence vectors across community text from Q/A forums such as Stack Overflow to link the API caveats with actual code examples. With this, API caveats are augmented with "real" code examples by developers that can help them understand how to use an API. However, this approach requires a large corpus of code examples alongside the assumption informal text by communities containing high lexical similarity to sentences of formal API documentation. Besides this, it also makes the assumption that text surrounding code examples are relevant to the code examples provided. This thesis seeks to extend upon the word by Li et al.; Sun; Ren et al. to investigate alternative methods of improving the understandability of API caveats. In particular, I focus on translating natural language of API caveats into code *contracts* that can be used directly by program analysis tools to check for API misuse in real time.

JIAMOU and XIAOXUE ref

are also proposed and used by

A specific subset of API usage constraints are classified by Maalej and Robillard [2013] as *directives*, which "specify what users are allowed/not allowed to do with the API element".

This subset of constraints was further researched by Li et al.

Suppose a new developer was starting their journey on learning Java or program-

ming in general. One of the first topics they might be interested in could be file handling (i.e. how to read a text file into their program). A common strategy to solve this problem involves using a search query on Google such as "java how to read files". The first search result of this is a HTML page from GeeksforGeeks[1] that provides multiple examples for reading a file in Java. The first example described is shown in Listing 1.1. The next step a developer would take is to copy-paste the relevant section into their program (i.e. the code within `main`). Finally, the developer might try to execute their modified program, but this would result in `FileNotFoundException` to be thrown. This is because the file path in the `File` constructor call (line 11) has not been changed to the appropriate file path for the developer. In an integrated development environment (IDE) such as IntelliJ, information about the exception such as which line the exception was thrown from will also be displayed. With further investigation, the developer will find a confounding result: IntelliJ reports the exception is not associated with the `File` constructor line (where the file path is set), but with the `FileReader` constructor in line 13. In other words, the file path appears to be accepted by `File` but not by `FileReader`. Searching for the reference documentation of these classes[2]. Specfically, the documentation of the relevant constructor for `FileReader` says "Throws: FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading". Furthermore, the documentation for the constructor of `File` does not mention the consequences of an invalid path. Rather, the developer might notice that `File` has an `exists()` method that can be used to verify whether the file exists. Only then could the developer understand the source of the problem encountered alongside correct usage of both `File` and `FileReader` for reading files in Java applications.

Although the above example presents a contrived view of how a new developer would approach using a feature of an API for the first time, we observe the constraints associated with an API introduce a significant problem for all programmers that is best explained with the common phrase: "you don't know what you don't know". This indicates that the usage of an API requires considerable understanding of the different components involved alongside how its functions will behave in different scenarios. This is both time-consuming and a significant endeavour due to the number and size of APIs that exist across all programming languages and computer science fields.

Listing 1.1: File reading java code example from GeeksforGeeks

```
1 // Java Program to illustrate reading from FileReader
2 // using BufferedReader
3 import java.io.*;
4 public class ReadFromFile2
```

[1] https://www.geeksforgeeks.org/different-ways-reading-text-file-java/

[2] FileReader: https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html File: https://docs.oracle.com/javase/7/docs/api/java/io/File.html, the developer could find the reference documentation for both of these classes

```
{
    public static void main(String[] args)throws Exception
    {
        // We need to provide file path as the parameter:
        // double backquote is to avoid compiler interpret words
        // like \test as \t (ie. as a escape sequence)
        File file = new File("C:\\Users\\pankaj\\Desktop\\test.txt");

        BufferedReader br = new BufferedReader(new FileReader(file));

        String st;
        while ((st = br.readLine()) != null)
        System.out.println(st);
    }
}
```

1.1 Main Research Challenges

1.1.1 Sentence Parsing for Contracts Generation

1.1.2 Source Code Analysis

1.2 Thesis Outline

How many chapters you have? You may have Chapter 2,

1.3 Main Contributions

Parsing of API caveat sentences to formulate contracts (basic proof-of-concept) Developed checkers that can output warnings from contracts

Discover lexical gap for GitHub data...

Chapter 2

Related Work

At the begging of each chapter, please introduce the motivation and high-level picture of the chapter. You also have to introduce sections in the chapter.

Section 2.1 yyyy.

2.1 Related work

You may reference other papers. For example: Generational garbage collection [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984] is perhaps the single most important advance in garbage collection since the first collectors were developed in the early 1960s. (doi: "doi" should just be the doi part, not the full URL, and it will be made to link to dx.doi.org and resolve. shortname: gives an optional short name for a conference like PLDI '08.)

2.2 API Caveats in Reference Documentation

2.3 Word & Sentence Embedding

2.4 Static-Code Analysis

2.5 Summary

Summary what you discussed in this chapter, and mention the story in next chapter. Readers should roughly understand what your thesis takes about by only reading words at the beginning and the end (Summary) of each chapter.

Chapter 3

Locating API Misuse Examples in GitHub Data

Same as the last chapter, introduce the motivation and the high-level picture to readers, and introduce the sections in this chapter.

3.1 Introduction

3.2 Design

3.2.1 TF-IDF Sentence Matching

3.2.2 Word2Vec Embedding

3.2.3 Okapi BM25 Ranking

3.3 Implementation

3.3.1 GitHub Data Extraction

There are three notable sources for GitHub data: (1) the GitHub REST API, (2) the GitHub Archive project, and (3) GitHub itself by cloning a repository. For this project, the GitHub REST API and GitHub Archive was utilised due to the physical memory requirements of cloning each repository of interest. However, the GitHub REST API contains several limitations that inhibits its usefulness for a large corpus of community text such as request rate limitations and provided query options. Moreover, GitHub Archive is a project that creates hourly archives of all GitHub data starting from 2/12/2011 as JSON objects. The archives includes over 20 event types that are provided by GitHub such as the creation of an issue or a comment on a particular issue. Despite this, it does not provide metadata about the programming languages utilised for repositories. Thus, we can combine the queries of the GitHub REST API to identify the repositories that are Java related, then link them with community text captured by GitHub Archive.

GitHub provides a REST API to allow developers to query for data on GitHub such as metadata about a particular repository, or the number of repositories containing a certain programming language. However, the REST API imposes a rate limit on user requests to prevent flooding of GitHub servers. The API of interest for data extraction is the "search" queries, which allows searching for repositories or issues given some conditions such as the time in which they were created. However, the API is limited and does not provide functionality to search for the comments of an issue for example. Furthermore, the API is limited to a maximum of 100 results for a single query. Combined with the rate limitations imposed, the API is only relevant for finding the repositories that contain Java code. Hence, a list of all Java related repositories are searched for by circumventing the rate limit with numerous HTTP GET requests. This specifically involves sending queries with a timer between each request to avoid the rate limit with slightly modified creation times for each day of interest. A time window of 2009 to 2019 was chosen alongside a restriction of at least 2 "stars" (at the time of the query) to reduce the scope of projects collected, and with Java as a language used in the repository. In particular, the number of "stars" a repository contains can be used to gauge its popularity. Thus, the 2 stars requirement ensures repositories that likely contain multiple issues and comments can be mined. The PyGitHub Python library in particular is used to implement this. Overall, collecting the repository names of Java related projects of interest yields 291,152 results.

GitHub Archive captures a particularly useful event for community text: the "IssueCommentEvent" which contains information about a comment on a particular GitHub issue alongside information such as the associated repository and URL of the issue. Firstly, a script is used to download the archives for each hour of 2018. Note that only data from 2018 is collected to reduce the amount of time required for querying the entire dataset and as an initial study. Multiprocessing is used in particular with a Python script to perform concurrent downloads of archived data within several hours. After this, extraction of relevant "IssueCommentEvent" objects is performed by testing if the associated repository of a comment is Java related based on the list attained from using the GitHub REST API. Notable information of these objects such as the text body of an issue comment and its title are then mined for natural language processing. Overall, this extraction process results in 627,450 GitHub issues and 1,855,870 issue comments to be collected.

3.3.2 Java 12 Documentation Caveat Extraction

To start extracting API caveats for a given API document, the API document must first be collected. At the time of writing, the Java Standard Edition 12 was chosen as the latest Java version and documentation available. Its documentation consists of HTML pages for each class of the Java Development Kit (JDK) 12. In particular, the API documentation has a HTML page that lists the complete class hierarchy tree of

the Java standard library with URL links to each class.[1] This information is utilised to data crawl the entire Java API documentation. First, the URL of all classes are mined from the class hierarchy page by collecting all hyperlink references on the page that are found within the appropriate HTML `section` element. The relative URLs of each class are found by locating list item elements (`li`) then anchor elements (`a`) residing within. From this, absolute URLs are constructed for 4,865 classes. Next, the HTML pages of each class is collected by recursively sending HTTP GET requests for each of the URLs generated and saved locally for data mining. A total of 4,712 classes are found. It is important to note that the documentation of Java SE 12 is well-structured. For example, the parameters and possible exceptions for a method are consistently placed within certain HTML elements across all HTML pages. Hence, it is relatively simple to extract all sentences for each API element alongside additional information such as whether a method is deprecated from the existence of a `div` element with the "deprecationBlock" class for example.

Extraction of the caveat sentences is performed by first creating a set of keywords and patterns based on those found by Li et al. [2018]. For each pattern, a regular expression is used to represent it and allow searching for exact matches within an arbitrary string. An API sentence is then regarded as an API caveat sentence if at least one regular expression match is found. This is executed recursively for all of the HTML pages, in which 115,243 caveat sentences are found. Of these sentences, 9,964 are regarded as "class level sentences", which are sentences that appear in the description section of a given API element and describes general information about that API element. Each method, field and constructor of a Java class is also identified using the structural HTML information, where 37,578 sentences are found within the description section of these elements. Specific sentences to an API element such as the parameters for a method and their possible exceptions comprise 67,701 sentences. It is also noted that 1,522 API elements are identified as deprecated elements.

Add several good caveat examples alongside GitHub issues and StackOverlfow posts concerning them

3.3.3 Data Preprocessing

Data cleaning and preprocessing is required to allow NLP techniques such as word embedding to perform and behave correctly. This step involves filtering the issue comments such that only those associated with an issue of interest (i.e. those containing some code) are analysed. A total of 85,318 issues of interest are found alongside 290,019 associated comments. Furthermore, several text preprocessing techniques are utilised to transform the sentences within a comment to tokens for usage in word embedding. In particular, the comments are in markdown format, allowing simple removal of unwanted elements such as code blocks. Hence, the preprocessing process involves removing all code blocks, URLs, additional white space characters,

[1] See the class hierarchy page at: https://docs.oracle.com/en/java/javase/12/docs/api/overview-tree.html

Category	Subcategory	Syntactic Pattern Examples
Explicit	Error/Exception	"insecure", "susceptible", "error", "null", "exception", "susceptible", "unavailable", "not thread safe", "illegal", "inappropriate",
	Recommendation	"deprecate", "better/best to", "recommended", "less desirable" "discourage"
	Alternative	"instead of","rather than","otherwise"
	Imperative	"do not"
	Note	"note that", "notably", "caution"
Restricted	Conditional	"under the condition", "whether ...", "if ...", "when ...", "assume that ..."
	Temporal	"before", "after"
Generic	Affirmative	"must", "should", "have to", "need to"
	Negative	"do/be not ...", "never"
	Emphasis	"none", "only", "always"

Table 3.1: API caveat categories and syntactic patterns from Li et al. [2018].

apostrophes, punctuation (except full stops appearing after a word). Sentence tokenisation is then performed based on the full stops in the preprocessed comment sentences. Finally, tokenisation simply involves splitting words based on white space characters due to the removal of additional white space characters from the preprocessing step. An extra data cleaning step performed involves removing all tokens that contain only a single character or those that are English stopwords provided by the NLTK library. Overall, preprocessing of the GitHub issue comments results in 1,855,870 tokenised sentences.

3.3.4 Candidate Filtering

3.3.5 Sentence Embedding

3.3.6 Candidate Matching

3.4 Results

3.5 Summary

Same as the last chapter, summary what you discussed in this chapter and be the bridge to next chapter.

Chapter 4

API Contracts Construction with Static Code Analysis

4.1 Introduction

Code contracts are a concept derived from object-oriented principles in which preconditions, postconditions and invariants are defined for different software components. Specifically, the principle of "design by contract" suggests the use of specifications referred to as contracts. This provides a method for improving both code correctness and robustness as software components can only interact via obligations to contracts. Utilising this, we reduce the problem of linking API caveats to source code to the problem of mapping API caveats to code contracts. This allows us to simplify the problem of improving understanding of API caveat considerably by providing developers with immediate feedback while coding. Furthermore, finding a general method for transforming an arbitrary API caveat to a code contract is sufficient as we can assume the existence of programming analysis tools that can accept these contracts and locate patterns in source code (variations of such tools already exist). As a continuation of the previous chapter, I perform statistical analysis of several caveat types for the Java 12 API documentation based on work by Zhou et al. [2017]. I then propose a parsing technique to identify API caveats related to a significant subset of exceptions thrown and construct associated API contracts. This extends upon the parsing techniques used by Zhou et al. to collect a subset of API caveats related to explicit constraints such as range limitations for arguments of a method, or not-null requirements. I propose and utilise a new algorithm to the Java 12 caveat sentences to construct a total of 4,694 unique contracts for this subset of API caveats. Finally, I develop a proof-of-concept checker plugin for IntelliJ that can be used to highlight violations of these API contracts in real time.

4.2 Design

The first step to generating contracts for API caveats is the extraction of API caveat sentences as described in 3.3.2. We recall that caveat extraction using the approach described by Li et al. [2018] on the Java 12 reference documentation yields 115,243

caveat sentences, where a significant proportion of the sentences (67,701) are found within miscellaneous sections of an API element such as sentences in the parameters section for a method, or the description of a methods return value.

4.2.1 Java 12 Caveat Statistics Analysis

4.3 Implementation

4.3.1 Java API Caveat Contracts

4.3.2 IntelliJ Plugin with Static Code Analysis

4.3.3 Boa Programming Language & API Call Sequence Mining

Maybe remove this section...

4.4 Results

4.5 Summary

Same as the last chapter, summary what you discussed in this chapter and be the bridge to next chapter.

Chapter 5

Conclusion

Summary your thesis and discuss what you are going to do in the future in Section 5.1.

5.1 Future Work

Good luck.

5.2 Final Remarks

Bibliography

Li, H.; Li, S.; Sun, J.; Xing, Z.; Peng, X.; Liu, M.; and Zhao, X., 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 183–193. doi:10.1109/ICSME.2018.00028. (cited on pages xv, 1, 9, 10, and 11)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. (cited on page 5)

Maalej, W. and Robillard, M. P., 2013. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39, 9 (2013), 1264–1282. (cited on page 1)

Moon, D. A., 1984. Garbage collection in a large LISP system. In LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 235–246. ACM, New York, New York, USA. doi:10.1145/800055.802040. (cited on page 5)

Ren, X.; Xing, Z.; X; Xia; Lo, D.; Grundy, J.; and Sun, J., 2018. Trustdoc: Documents speak louder than votes. (2018). (cited on page 1)

Sun, J., 2018. Augment api caveats with erroneous code examples. (2018). (cited on page 1)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In SDE 1: *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, USA, Apr. 1984), 157–167. ACM, New York, New York, USA. doi:10.1145/800020.808261. (cited on page 5)

Zhou, Y.; Gu, R.; Chen, T.; Huang, Z.; Panichella, S.; and Gall, H., 2017. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17 (Buenos Aires, Argentina, 2017), 27–37. IEEE Press, Piscataway, NJ, USA. doi:10.1109/ICSE.2017.11. https://doi.org/10.1109/ICSE.2017.11. (cited on page 11)