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
9 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 that can even lead to software failures. These constraints are documented by the developers of an API in their formal documentation, which are usually created manually by developers or automatically generated from source code comments such as with the Javadoc tool for the Java programming language. These constraints are formally referred to as *API caveats* by Li et al. [2018]. A specific subset of API usage constraints classified by Maalej and Robillard [2013] as *directives* "specify what users are allowed/not allowed to do with the API element".

are researched by Li et al.

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

[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

structor 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 can observe the constraints associated with an API introduce a significant problem for all programmers, which 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
5  {
6      public static void main(String[] args)throws Exception
7      {
8          // We need to provide file path as the parameter:
9          // double backquote is to avoid compiler interpret words
10         // like \test as \t (ie. as a escape sequence)
11         File file = new File("C:\\Users\\pankaj\\Desktop\\test.txt");
12
13         BufferedReader br = new BufferedReader(new FileReader(file));
14
15         String st;
16         while ((st = br.readLine()) != null)
17         System.out.println(st);
18     }
19 }
```

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

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 Background

3.3 Design

3.4 Implementation

3.5 Results

3.6 Summary

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

Chapter 4

API Contracts Generation and Static Code Analysis

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

4.1 Introduction

4.2 Background

4.3 Design

4.4 Implementation

4.5 Results

4.6 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 page 1)

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.

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)

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)