Constructing Code Contracts from API Caveats

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
11 October 2019

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

Acknowledgments

1. Thank Prof. Zhenchang
2. Thank Jiamou
3. Thank Xiaoxue

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.

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 programming in general. One of the first topics they might be interested in could be file han-

dling (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
5 {
```

[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

```
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 }
```

1. Provide 3 caveat examples from Java 12 API (ArrayList.subList, String.indexOf, HashMap concurrency)

1.1 Main Research Challenges

1. Similar to Jiamou's report

1.1.1 Sentence Parsing for Contracts Generation

1.1.2 Source Code Analysis

1.2 Thesis Outline

1.3 Main Contributions

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

Discovered lexical gap for GitHub data...

1. Mention some API element dependency parsing if some time to complete

Chapter 2

Related Work

2.1 Related work

2.2 API Documentation and API Caveats

1. Sun [2018] - Jiamou report
2. Li et al. [2018] - Knowledge graph
3. Ren et al. [2018] - trustdoc
4. Li [2018] - knowledge graph thesis
5. Uddin and Robillard [2015] - API doc study

2.3 NLP focus for API

1. Subramanian et al. [2014] - BAKER - link source code to API documentation
2. Zheng et al. [2017] - Code to comments
3. Van Nguyen et al. [2017] - using w2v on code for finding code examples
4. Silva et al. [2019] - CROKAGE
5. Li et al. [2018] - API Caveat Explorer

2.4 API Usage Misuse

1. Zhang et al. [2018] - API call sequence mining on GitHub
2. Zhou et al. [2017] - Directive defects
3. Wen et al. [2019] - Mutation analysis
4. Amann et al. [2016] - API misuse detector benchmark MuBENCH

5. Kapur and Sodhi [2018] - defectiveness of code on GitHub
6. Bae et al. [2014] -API misuse detection on web apps

2.5 Summary

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

Basic description of TF-IDF

3.2.2 Word2Vec Embedding

Basic description of W2V

3.2.3 Okapi BM25 Ranking

Basic description of BM25

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

[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].

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, 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

It is observed that several sentences extracted are invalid caveat sentences or contain snippets of code within the context of a particular sentence. Example of this is shown in Listing 3.1, 3.2 and 3.3.

Listing 3.1: An example of a caveat sentence extracted from the `javax.swing.Spring` documentation containing some snippets of code or mathematical expressions

```
If we denote Springs as [a, b, c], where a <= b <= c,
we can define the same arithmetic operators on Springs:

[a1, b1, c1] + [a2, b2, c2] = [a1 + a2, b1 + b2, c1 + c2]

-[a, b, c] = [-c, -b, -a]

max([a1, b1, c1], [a2, b2, c2]) = [max(a1, a2), max(b1, b2), max(c1, c2)]
```

Listing 3.2: An example of a caveat sentence extracted from the `java.security.cert.X509CRL` documentation explaining the structure of a `TBSCertList` object.

```
The ASN.1 definition of tbsCertList is:

TBSCertList ::= SEQUENCE {
        version         Version OPTIONAL,
        -- if present, must be v2
        signature        AlgorithmIdentifier,
        issuer         Name,
        thisUpdate       ChoiceOfTime,
        nextUpdate       ChoiceOfTime OPTIONAL,
        revokedCertificates   SEQUENCE OF SEQUENCE {
        userCertificate     CertificateSerialNumber,
        revocationDate     ChoiceOfTime,
        crlEntryExtensions   Extensions OPTIONAL
        -- if present, must be v2
        } OPTIONAL,
        crlExtensions      [0] EXPLICIT Extensions OPTIONAL
        -- if present, must be v2
}
```

Listing 3.3: An example of a caveat sentence extracted from the `java.text.BreakIterator` documentation that contains some sample code.

```
Creating and using text boundaries:

public static void main(String args[]) {
        if (args.length == 1) {
                String stringToExamine = args[0];
                //print each word in order
                BreakIterator boundary = BreakIterator.getWordInstance();
                boundary.setText(stringToExamine);
                printEachForward(boundary, stringToExamine);
                //print each sentence in reverse order
                boundary = BreakIterator.getSentenceInstance(Locale.US);
                boundary.setText(stringToExamine);
                printEachBackward(boundary, stringToExamine);
                printFirst(boundary, stringToExamine);
                printLast(boundary, stringToExamine);
        }
}
```

3.3.5 Sentence Embedding

3.3.6 Candidate Matching

3.4 Results

3.5 Summary

Describe embedding for all methods (TF-IDF, W2V and BM25)

Describe matching method with the manually labelling conducted

Add manual labelling table

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 an algorithm for analysing caveat sentences from this subset of API caveats to construct a total of 4,694 unique contracts. 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. Furthermore, we note that a subset of API caveats dealing with directives Zhou et al. [2017] contain explicit constraints that represent the largest portion (43.7%) of API documentation Monperrus et al. [2012]. From Zhou et al. [2017], it can also be seen that a set of heuristic rules can be used to obtain constraints of type: (1) nullness not allowed, (2) nullness allowed, (3) type restriction and (4) range limitation from the directives of an API document. Although other categories of API caveats can also potentially be transformed into contracts and applied to static code analysis, I focus primarily on the categories identified by Zhou et al. [2017] for the scope of this project.

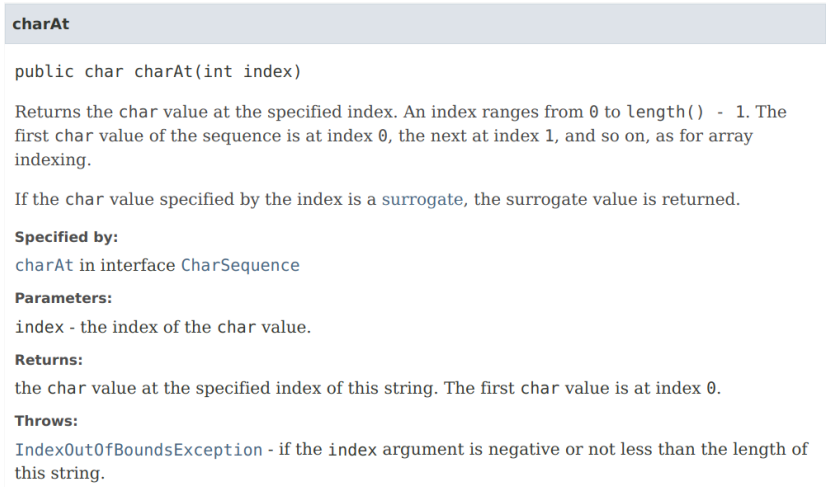
In addition, Zhang et al. [2018] proposes a method for mining API misuse by transforming source code to structure they refer to as API call sequences. These sequences essentially capture an API method call in addition to surrounding code elements such as guard conditions and method calls that are most relevant to the target API call. From this, we note that API contracts can be modified to apply to API call sequences given that contracts can be transformed to a structure resembling API call sequences.

4.2.1 Java 12 Caveat Statistics Analysis

To better understand the prevalence of these API caveat categories identified by Zhou et al. [2017] in the Java 12 documentation, a statistical sampling method from Singh and Mangat [2013] is applied. Specifically, the minimum number of samples required to ensure the estimated population mean is within a given confidence level and error margin can be calculated by the formula:

$$min = \frac{\frac{z^2 \times 0.25}{e^2}}{1 + \frac{(\frac{z^2 \times 0.25}{e^2} - 1)}{p}} \tag{4.1}$$

Where z is the z-score associated with the desired confidence level, e is the error margin and p is the population size. For a 95% confidence interval, 5% error margin and a population size of 115,243 caveat sentences, the minimum sample size is approximately 383. However, we observe that the directive constraints identified by Zhou et al. [2017] correspond to the parameter and exception sections of method API elements specifically. This is because the directives they analysed were sentences annotated with *@param, exception* and *@throws* tags within the comments of the JDK source code files, which are transformed into HTML via Javadoc. It is also noted that the heuristic rules formulated by the authors for caveat categories of not-null, nullness allowed, type restriction and range limitation are differentiated for sentences with the *@param* tag and for sentences with either *@exception* or *@throws* tags. Thus, we adopt a similar approach to reduce the scope of caveat sentences that require anal-



charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the `char` value.

Returns:

the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:

`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

Figure 4.1: API documentation for the `charAt` method of the `java.lang.String` class

ysis. We specifically focus on the parameter or exception sections only API methods and constructors, and regard the sentences from those sections separately.

An observation on the parameter and exception sections of the Java 12 documentation is the consistent structure used for sentences. Sentences form the parameter section follow a template of "`param - description`", where `param` is the name of the parameter for a given method/constructor and `description` is the actual sentence describing some information about `param`. Exceptions also follow a similar template of "`exception - description`", where `exception` is the exception class thrown and `description` describes conditions required for the exception to be thrown. An example of this is shown in Figure 4.1. It is therefore trivial to separate the `description` sub-parts of a caveat sentence for analysis as we are only interested on the constraints imposed upon a particular parameter and the exception conditions. Moreover, we can filter the corpus of exception and parameter sentences to obtain a unique set of sentences for both as identical sentences can be mapped to the same contracts (but with different target parameters or API elements). Hence, two separate random samples of caveat sentences are collected: one from the parameters section of methods and constructors which consists of 2,654 unique sentences, and one from the exception sections of methods and constructors which consists of 4,870 unique sentences. Using equation 4.1, the sample sizes required are 336 and 356 respectively.

Manual labelling is then required for the samples to identify the prevalence of different caveat types for the sentences. In particular, the categories identified in Zhou et al. [2017] of *not-null*, *range limitation* and *type restriction* are used as labels with the addition of *ambiguous* to account for sentences that do not match any of the former classes. The *not-null* category are defined as sentences which specify some parameter

	Labels			
	ambiguous	**not-null**	**range limitation**	**type restriction**
Count	291	26	19	0

Table 4.1: Manually labelled results for classes of 336 randomly sampled parameter level caveat sentences.

	Labels			
	ambiguous	**not-null**	**range limitation**	**type restriction**
Count	242	73	46	1

Table 4.2: Manually labelled results for classes of 356 randomly sampled exception level caveat sentences.

cannot be null. The *range limitation* category specifies some numerical limitation on a parameter such as a non-negative requirement. Finally, the *type restriction* category indicates that a parameter must a particular class type or one of several types. The *nullness allowed* category is not considered from Zhou et al. [2017] because it only describes an acceptable condition for contracts, whereas the other categories describe a stricter condition for contracts that must not be broken. Overall, the results of this for the parameter sentences sample is shown in Table 4.1, while the results for exception level sentences is shown in Table 4.2. We note that for 4.2, the counts contribute add up to more than 356 because 6 of the labelled caveat samples fit both the *not-null* category and the *range limitation* category.

From Table 4.1, it can be seen estimated that approximately 8% of unique parameter caveat sentences impose a *not-null* constraint and approximately 6% of the unique parameter caveat sentences impose a *range limitation* constraint. Despite the small percentage of sentences in parameter sentences sample that fit these categories, it its important to note that they represent one the most important type of API caveats: caveats that can cause software failures from exceptions. Furthermore, these caveat types are found to require (generally) explicit constraints that have little dependencies on other API elements, which makes them an adequate baseline for constructing code contracts. In contrast, the results from Table 4.2 show that considerably larger subset of 20% of unique exception caveat sentences specify a *non-null* constraint. This is also observed for the *range limitation* category with approximately 13% of sentences labelled.

Analysis of other categories must also be considered to determine what API contracts can be constructed and the approaches that are available. In particular, the categories identified in Zhang et al. [2018] are are derived from API misuse patterns data-mined from code snippets on Stack Overflow, but can be mapped into contracts. For example, *missing control constructs* can be represented by an API contract that defines the control structure around some API call as a requirement. The same concept can also be applied to *missing or incorrect order of API calls* and *incorrect guard conditions*. However, it the subcategories of *missing control constructs*, which include

		Labels			
		Ambiguous	**Control**	**Temporal**	**Guard**
Sentence Location	**Constructor**	264	21	6	32
	Method	360	9	4	5
	Parameter	257	2	2	75
	Return	351	0	1	0

Table 4.3: Manually labelled categories for caveat sentences found in different locations of the Java 12 API documentation.

missing exception handling, *missing if checks* and *missing finally*, all require explicit explanations for usage of these control structures. This is because usage of a control structure such as `if` or `finally` cannot be inferred without those keywords being described. Furthermore, *incorrect guard conditions* could be considered a superset of the constraints analysed previously in Zhou et al. [2017] for the Java API documentation, but sentences that do not fit a category by Zhou et al. [2017] could then be expected to be rare occurrences. We therefore focus on *missing control structures* and *missing or incorrect order of API calls* as categories to analyse. Using a similar approach to before, the estimated sample size required is calculated with Equation 4.1 for sentences from the method/constructor description, parameter section or return value section, which are 321, 377, 336 and 353 respectively. The exception section sentences are not considered as they could all be categorised as descriptions of *missing exception handling* or as examples of *incorrect guard conditions*. In other words, all exception level sentences can be regarded as important, but extracting essential information from those sentences is non-trivial given their diversity. We therefore focus on a subset of these types of API caveats to reduce the scope of the project. The labelled results are shown in Table 4.3. Note that *Control* refers to *missing control structures*, *Temporal* refers to *missing or incorrect order of API calls* and *guard* refers to *incorrect guard conditions*.

The results in Table 4.3 show the size of *Control* and *Temporal* caveat sentences is notably small, indicating that perhaps API documents rarely contain API methods that require specific call orders or additional control structures. This is an interesting result given that 31% of 217,818 Stack Overflow posts studied in Zhang et al. [2018] were found to have a potential API misuse based on the categories mentioned previously. This suggests that API documentation may not contain sufficient information for developers to handle API caveats of these categories in particular.

4.2.2 API Contracts Construction

Given the results found from statistical analysis in 4.2.1, the next step is to collect a set of API caveat sentences and attempt different approaches to extract important information for a given caveat. As a baseline study, the API caveats contained within the *not-null* and *range limitation* categories are the main caveats researched for this

thesis. This is because the 64 heuristic rules and 29 regular expressions from Zhou et al. [2017] could be used as one approach for parsing an API caveat sentence. Their approach utilised these heuristics and regular expressions to parse the constraints of a caveat into a first-order logic (FOL) formula that can be passed to an satisfiability modulo theories (SMT) solver. However, their work (and the artefacts produced) are based on 429 documents of the JDK 1.8. In comparison to the Java 12 API documentation, 4,712 documents were data-crawled. Furthermore, large amounts of manual analysis is required to formulate these heuristic rules such that they can be generalised to multiple sentences for a given corpus. Hence, a simpler approach is proposed based on observations of the heuristics and some sample caveat sentences that are *not-null* or *range limitation* constraints. Thus, we can attempt to find a generalised approach for analysing sentences of these categories for different APIs and perhaps across other programming languages.

To design a simpler method for parsing API caveats with a *not-null* constraint, we observe that sentences must mention the term "null" to either specify if a `null` value is allowed or not allowed in code. Furthermore, given the structural information of the Java 12 API documentation, two notable sections already exist for each method/constructor of the API: the parameter list and exceptions list. These sections contain structural consistency in their sentences which follow a template described in 4.2.1. Therefore, it is trivial to obtain information such as the subject of a sentence can obtained without the need for dependency parsing or part-of-speech (POS) tagging. Another observation made based on the heuristics from Zhou et al. [2017] is the prevalence of the subject-object-verb (SVO) ordering for a given sentence Dryer [2005]. Specifically, English is known to follow SVO ordering despite other possible logical orderings such as subject-object-verb used in Japanese or subject-verb-object in Mandarin. This structure can be seen from rules 1 and 17 of 5.3.1. For rule 20, we observe the use of "non-null' as a predicative adjective to the subject, which is the only *non-null* heuristic formulated that has "null" appearing before the subject. The final observation for this category of caveats is that whether some parameter for an API method call can be null is a boolean condition. In other words, this category of caveats represents the simplest form of an API contract as it only needs to specify whether a null value is accepted or disallowed for some dependent API element. Given this information, a general approach to identify whether an API caveat is of the *non-null* category is to first filter for caveat sentences containing the sub-string "null". Next, we can assume that API documentations aim to be simplistic and mentions of nullness within certain sections (i.e. exception sections) can be regarded as a "nullness not allowed" constraint. This is particularly true for the Java API documentation as the sentences in exception sections of methods/constructors is used to describe the conditions required for the relevant exception to be thrown. Hence, any mention of "null" inside this section could be assumed to indicate a null value will result in an exception. We do note however that this assumption does not necessarily hold for other API documentation.

Rule Number	Heuristic
1	[something] be/equals null
17	Value of [something] be/equals null
20	Non-null [something]

Table 4.4: Example of 3 of 20 heuristic rules for nullness not allowed from Zhou et al. [2017]. Note that the complete list can be found in 4.4 (Appendix).

Rule Number	Heuristic
1	[something] >/</= [value]
8	[something] be not negative/positive/false/true
20	[something1] equals [something2]

Table 4.5: Example of 3 of 23 heuristic rules for range limitations from Zhou et al. [2017]. Note that the complete list can be found in 4.5 (Appendix).

An alternative approach for parsing the *non-null* and *range limitation* API caveats is to utilise a sentence normalisation technique used in Zhou et al. [2017]. In particular, several regular expressions are identified by Zhou et al. to perform substitutions within a sentence prior to dependency parsing. These expressions are used to detect cases such as the names of variables, classes and mathematical expressions

4.2.3 Checker Programs

Describe IntelliJ plugin and Boa AST

4.3 Implementation

4.3.1 Java API Caveat Contracts

Describe contract construction in Python

4.3.2 IntelliJ Plugin with Static Code Analysis

Explain IntelliJ plugin and Boa AST code

4.3.3 Boa Programming Language & API Call Sequence Mining

Maybe remove this section if it doesn't add much

4.4 Results

Add screenshots of the IntelliJ plugin and table

Type	Description	Regular Expression
Specific Values	0.0, 0.1f, etc.	\W(-)?[0-9]+(,[0-9]+)*((\
	Member value of objects e.g., Location.x	\W(^(java\.|javax\.|or
Class methods and static members	class methods, e.g., ClassA.func(Param1)	\W[A-Za-z_]+[A-Za-z_
	Static member, e.g., Desktop.Action#OPEN	\W([A-Za-z_]+[A-Za-z_
	All upper case	\W(\w+\.)*([A-Z]+_)*[
	Class name	\W([A-Za-z_]+\w+\.)*
Expressions	A - B	\W\w+((\s+-)|(-\s+)|
	A + B	\W\w+\s*\+\s*\w+\
	A * B	\W\w+\s**\s*\w+\
	A..B	\W(?\s*\w+\s*)?\s*\.
	[A, B]	\W[\s*\w+\s*,\s*\w+
	[A..B]	\W[\s*\w+\s*(\.\s*\.
	A <\<= B <\<= C	\W\w+\s*<=?\s*\w
	A >\>= B >\>= C	\W\w+\s*>=?\s*\w
	From A to B	\W(from\s+)?\w+\s+t
	A != B	\W\w+\s*!=\s*\w+\
	Enumeration expression	\W (\s*\w+\s*)(,\s*\w

Table 4.6: The regular expressions identified in \cite{} for sub-sentence substitutions and dependency parsing.

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

1. Summarise work done with information retrieval
2. Summarise work done with sentence parsing
3.

5.1 Future Work

1. Mention mutation analysis paper
2. Talk about using knowledge graph / API element dependencies parsing
3. Applying same approach to other languages
4. Applying same approach to other API documentation
5. Looking at other caveat categories
6. Extending plugin

5.2 Final Remarks

5.3 Heuristic Rules from Zhou et al. [2017]

5.3.1 Not-null Heuristics

5.3.2 Range Limitation Heuristics

Rule	Nullness Not Allowed Heuristics (@exception or @throws)
1	[something] be/equals null
2	[something] be equal/equivalent to null
3	[something1] or [something2] be/equals null
4	[something1] or [something2] be equal/equivalent to null
5	[something] parameter be null
6	The specified [something] be null
7	Any/none of [something] be null
8	Either/neither/any/all/both/none parameter(s) be/equals null
9	Either/neither/any/all/both/none parameter(s) be equal/equivalent null
10	Either/neither [something1] or/nor [something2] be/equals null
11	Both [something1] and [something2] be null
12	The [type] be null
13	[parameter phrase] be/equals null
14	[parameter phrase] be equal/equivalent null
15	[something]âĂŹs value be/equals null
16	[something]âĂŹs value be equal/equivalent null
17	Value of [something] be/equals null
18	Value of [something] be equal/equivalent null

Table 5.1: Complete list of heuristic rules for exception nullness not allowed.

Rule	Nullness Not Allowed Heuristics (@param)
19	[something] can not be null
20	Non-null [something]

Table 5.2: Complete list of heuristic rules for parameter nullness not allowed.

Rule	Range Limitation Heuristic (@exception or @throws)
1	[something] >/</= [value]
2	[something] be {not} less/greater/larger/equal/equivalent than/to [value]
3	[something] equals [value]
4	[something1] or/and [something2] be {not} less/greater/larger/equal/equivalent than/to [value]
5	Computing [expression] be not less/greater/larger/equal/equivalent than/to [value]
6	Computing either [expression1] or [expression2] be {not} less/greater/larger/equal/equivalent than [value]
7	Product/sum of [something1] and [something2] be not less/greater/larger/equal/equivalent than/to [value]
8	[something] be not negative/positive/false/true
9	[something1] or/and [something2] be not negative/positive/false/true
10	[something1] and [something2] be not the same
11	[something1] equals [something2]
12	[something] be not in/out of/outside of range [range value]
13	[something] be not in/out of bounds
14	[something] be not [value]
15	[range expression] (only the expression,like)
16	[something] be not between [value1] and [value2]
17	[something] be not [value set]
18	[something] be not one of [value set]
19	[something] be not one of following: [value set]
20	[something] be not one of supported data, which are [value set]

Table 5.3: Complete list of heuristic rules for exception range limitations.

Rule	Range Limitation Heuristic (@param)
21	[something] can/must not be negative/positive/non-negative/non-positive
22	[something] must be greater/less/larger than [value]
23	[something] be greater/less/larger than [value]

Table 5.4: Complete list of heuristic rules for parameter range limitations.

Bibliography

Amann, S.; Nadi, S.; Nguyen, H. A.; Nguyen, T. N.; and Mezini, M., 2016. Mubench: a benchmark for api-misuse detectors. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 464–467. IEEE. (cited on page 5)

Bae, S.; Cho, H.; Lim, I.; and Ryu, S., 2014. Safewapi: web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 507–517. ACM. (cited on page 6)

Dryer, M. S., 2005. 81 order of subject, object, and verb. *The world atlas of language structures, ed. by Martin Haspelmath et al*, (2005), 330–333. (cited on page 18)

Kapur, R. and Sodhi, B., 2018. Estimating defectiveness of source code: A predictive model using github content. *arXiv preprint arXiv:1803.07764*, (2018). (cited on page 6)

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, 5, 9, 10, and 14)

Li, J.; Sun, A.; Xing, Z.; and Han, L., 2018. Api caveat explorer–surfacing negative usages from practice: An api-oriented interactive exploratory search system for programmers. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, 1293–1296. ACM. (cited on page 5)

Li, S., 2018. Constructing software knowledge graph from software text. (2018). (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)

Monperrus, M.; Eichberg, M.; Tekes, E.; and Mezini, M., 2012. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17 (05 2012). doi:10.1007/s10664-011-9186-4. (cited on page 14)

Ren, X.; Xing, Z.; X; Xia; Lo, D.; Grundy, J.; and Sun, J., 2018. Trustdoc: Documents speak louder than votes. (2018). (cited on pages 1 and 5)

Silva, R. F.; Roy, C. K.; Rahman, M. M.; Schneider, K. A.; Paixao, K.; and Maia, M. d. A., 2019. Recommending comprehensive solutions for programming tasks by mining crowd knowledge. *arXiv preprint arXiv:1903.07662*, (2019). (cited on page 5)

Singh, R. and Mangat, N. S., 2013. *Elements of survey sampling*, vol. 15. Springer Science & Business Media. (cited on page 14)

Subramanian, S.; Inozemtseva, L.; and Holmes, R., 2014. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, 643–652. ACM. (cited on page 5)

Sun, J., 2018. Augment api caveats with erroneous code examples. (2018). (cited on pages 1 and 5)

Uddin, G. and Robillard, M. P., 2015. How api documentation fails. *IEEE Software*, 32, 4 (2015), 68–75. (cited on page 5)

Van Nguyen, T.; Nguyen, A. T.; Phan, H. D.; Nguyen, T. D.; and Nguyen, T. N., 2017. Combining word2vec with revised vector space model for better code retrieval. In *Proceedings of the 39th International Conference on Software Engineering Companion*, 183–185. IEEE Press. (cited on page 5)

Wen, M.; Liu, Y.; Wu, R.; Xie, X.; Cheung, S.-C.; and Su, Z., 2019. Exposing library api misuses via mutation analysis. In *Proceedings of the 41st International Conference on Software Engineering*, 866–877. IEEE Press. (cited on page 5)

Zhang, T.; Upadhyaya, G.; Reinhardt, A.; Rajan, H.; and Kim, M., 2018. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 886–896. IEEE. (cited on pages 5, 14, 16, and 17)

Zheng, W.; Zhou, H.-Y.; Li, M.; and Wu, J., 2017. Code attention: Translating code to comments by exploiting domain features. *arXiv preprint arXiv:1709.07642*, (2017). (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 pages xii, xv, 5, 13, 14, 15, 16, 17, 18, 19, 21, and 23)