Constructing Caveat Contracts from API Documentation

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
20 October 2019

to my parents Tuyet and Linh, and my sister Quyen

Acknowledgments

First, I would like to thank Dr. Zhenchang Xing for being an excellent supervisor. Thankyou for being patient, encouraging and supportive at all times. Without your guidance, this thesis would not have been completed.
To Jiamou Sun and Xiaxue Ren, thank you for yours ideas and support. You helped me learn, contribute and extend your work.
Finally, I would like to thank my parents, Tuyet and Linh, along with my sister, Quyen, for their words of wisdom and positivism throughout my time at ANU.

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 follow 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 comprehension 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.

1.1 Motivation

Suppose a new developer was starting their journey on learning Java programming. 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 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 might 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 a `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 {
6     public static void main(String[] args)throws Exception
7     {
```

[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

```
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)

1.2 Main Research Challenges

The primary goal of this thesis is the investigation of methods for linking API caveats to code. Ideally, this allows API misuse cases to be detected and help developers understand correct usage of a given API. Previous work has focused on an indirect approach of linking where caveats sentences would be matched to community text surrounding code examples on Q&A websites such as Stack Overflow. This utilised NLP techniques involving sentence embedding to form information retrieval systems in which caveat sentences could be used as queries and code examples from the Q&A websites would be retrieved. For this thesis, the original objective was to determine whether the same approach could be applied to a different domain that is not a Q&A platform: GitHub. This is a complex task that mainly involves three components: (1) sentence embedding, (2) matching caveats to code examples, and (3) presenting code examples to developers. Sentence embedding is a modern approach in NLP that transforms sentences into vectors that retain semantic meaning Palangi et al. [2016]. However, it typically requires complex models (such as neural networks) to learn important features of sentences. Matching caveats to code examples is non-trivial given the semantic/lexical gap that exists between natural language and programming code. An example of this in terms of API caveats is the sentence "UnsupportedEncodingException - If the named charset is not supported" from the Java 12 Documentation (method `getBytes` of class `java.lang.String`). It is obvious that developing a generalised representation of this that can be understood by computers is difficult. It requires both consideration of the context that "charset" refers to the parameter "charsetName", alongside what exactly denotes a supported "charset". The third component of presenting code examples refers to the problem of highlighting code examples in a succinct manner that also offers guidance towards correct API usage. The scope of this thesis is focused on the first and second

component as experiments in Sun [2018] showed that even a simple Graphical User Interface (GUI) could help users understand API caveats. However, negative results were found using this approach and an alternative method was investigated. This resulted in additional research challenges involving sentence parsing and static code analysis.

Sentence parsing involves numerous problems in the process of interpreting meaning, requiring identification of individual words, contextual information, punctuation and subtle features such as intonation Mitchell [1994]. Numerous fields exist simply to determine some properties of a sentence such as named entity recognition where to identify what entity a word refers to. An example of this from Ratinov and Roth [2009] is a news headline that reads "SOCCER - PER BLINKER BAN LIFTED". Without prior knowledge, it is difficult to discern that "BLINKER" refers to a person. It is therefore clear that parsing a sentence to understand different types of constraints alongside finding a generalised representation for these constraints is a difficult problem. This thesis extends upon the work by Zhou et al. [2017] that uses sentence normalisation (where specific words and phrases are substituted with labelled words) and from their heuristic rules to parse and extract the constraints of caveat sentences. The project scope for this approach is strictly based on applying the parsing techniques used and providing alternative methods of parsing.
Static code analysis involves examining the source code of programs before execution of the program Baca et al. [2009]. This is typically used to "find bugs and reduce defects in a software application" Bardas et al. [2010], and has been found to be the best method for eliminating bugs. However, implementing static code analysers is non-trivial. Complex static analysis tools can detect vulnerabilities such as buffer overflows, but detection rates are not perfect and false positive alarms can be common Zitser et al. [2004]. Furthermore, performance of these tools must also be considered as high complexity analysis requires more computation time. A major challenge is to then implement a static code analyser that can detect as many coding errors as possible without the minimal amount of computation time and resources. Overall, the scope of static code analysis involves usage of an existing API (IntelliJ's Program Structure Interface) to develop a plugin that analyses source code.

1.3 Thesis Outline

An overview of API caveats and the background of this thesis is described in Chapter 2. Chapter 3 extends upon the work of Sun [2018] and Ren et al. [2018] for linking API caveat sentences to code examples in a different domain: GitHub. However, due to the negative results found, a more direct approach is attempted with the idea of transforming the natural language from API caveats to code contracts in Chapter 4. In Chapter 5, the findings of this thesis are discussed with remarks for future work.

1.4 Main Contributions

The main contributions of Chapter 3 is the discovery of a significant lexical gap for community text from GitHub and (Java 12) API caveat sentences. This impedes previous approaches used for linking API caveats to code examples.

The main contributions of Chapter 4 are:

- Parsing of a subset of API caveats that result in exceptions to formulate code contracts.
- A tool implementation that uses static code analysis to automatically check code contract compliance in real time.

Chapter 2

Related Work

This chapter provides an overview of the key works that serve as the background for API caveats, linking API caveats to code examples with NLP techniques, and the use of static code analysis to detect API errors or misuses.
Note that the term *API reference documentation* or *API documentation* is adopted throughout this thesis to refer to the set of *documents* that are indexed by an *API element* such as a class or method (Maalej and Robillard [2013]). For example, the Java 12 reference documentation consists of documents as web-pages with each describing a specific Java class (`String`, `ArrayList` etc.).

Section 2.1 references the papers that termed *API caveats* and extended upon this concept for linkage of API caveats to code.

Section 2.2 mentions notable works that have applied NLP techniques to the domain of API documentation and for general API misuse detection.

Section 2.3 references the papers that applied static code analysis to the domain of API documentation and for general API misuse detection.

Add motivation and high level picture of each chapter at the start

2.1 API Caveats

API reference documentation consists of a taxonomy of knowledge types that is defined by Maalej and Robillard [2013]. A particular knowledge type identified as *directives* that "specifies what users are allowed/not allowed to do with the API element" is identified as a notable component of API reference documentation by Li et al. [2018]. These knowledge types are particular interest as they can be used to highlight the accessibility of API documentation, which are essential parts of a framework because they describe functionality and usage of an API. The term *API caveat* is defined by Li et al. to refer to directives and all other forms of constraints described by an API document, and is the adopted term for this thesis. Li et al. conduct formative study of the Q&A website Stack Overflow that highlights the prevalence of

API caveats as issues for programmers, a set of syntactic patterns were provided that could be used to identify API caveats (see Table 3.1 for the complete list of syntactic patterns). API caveats have many practical applications such as the examination of the quality/validity of Stack Overflow answers Ren et al. [2018], construction of a knowledge graph that can be used for information retrieval purposes or entity-centric searches of API caveats Li [2018], and augmentation of caveats with code examples Sun [2018]. In particular, the work by Sun [2018] is the foundations for Chapter 3. It was shown that one solution to caveat linkage to code was via indirect approach. NLP techniques of mainly word2vec sentence embedding could be applied to sentences of API caveats and community text that surrounded code snippets. The cosine similarity of these vector outputs could then be used to determine the relatedness of API caveats to community text, and hence the relatedness of API caveats to the code examples.

2.2 Natural Language Processing for API Documents

Beside the work of Sun [2018] explained in the previous section, other NLP-centered works for linking API documentation in general to code have been proposed. For example, CROKAGE (Crowd Knowledge Answer Generator) is a tool that takes a query in natural language form and returns programming solutions consisting of both code and explanations. This is achieved with several NLP models and word embedding approaches including the use of a Lucene Index, FastText, IDF and an API inverted index. The relevance scoring methods also utilise the BM25 function and TF-IDF. Note that BM25 and TF-IDF (and IDF) are explained in further detail in Chapter 3 (Section 3.2). An explanation of the other methods will not be presented here as those models were not used in this thesis. Another solution is proposed by Huang et al. [2018] named BIKER (Bi-Information source based KnowledgE Recommendation). This mainly tackles the problem of helping developers find APIs that appropriate for their programming tasks, and uses a combination of language models involving IDF and word2vec. Similar to the other works mentioned, relevant posts from Stack Overflow are ranked using similarity functions for a given query and returned to users. The advantage of deep learning with word2vec was also used by Van Nguyen et al. [2017] for example code searching by computing the similarity of query vectors with source code as vectors.
In terms of parsing semi-structured natural langauge, Pandita et al. [2012] proposes an approach using part-of-speech tagging, and phrase and clause parsing to identify sentences that describe code contracts via first order logic (FOL) expressions with precision and recall of 91.8% and 93% respectively. Jdoctor is proposed by Blasi et al. [2018] that performs sentence normalisation techniques to translate Javadoc comments to executable Java expressions. In other words, the tool generates procedure specifications that are analogous to code contracts and boast an overall precision and recall of 92% and 83% respectively. This approach is extended upon in Chapter 4 to construct code contracts that can be applied to a static code analyser.

2.3 Linkage Methods using Code Analysis

A code-analysis centered solution for linkage of API documents and code was proposed by Subramanian et al. [2014] with Baker. The method proposed uses deductive linking, where the abstract syntax tree (AST) of a particular code snippet from online sites is analysed, relevant API elements are detected, then linked to the associated API documents. Another approach combines NLP and code analysis to detect errors such as obsolete code samples Zhong and Su [2013]. For this, the names of API elements from code samples represented as natural language or as code are compared to the set of all API element names for some API. Mismatches in the names are then be reported as documentation errors. A different approach for detecting API misuse does not use any API documentation, but data-mines the most common patterns used from a large set of code repositories like GitHub Zhang et al. [2018]. This is accomplished by traversing the AST of code samples generating API call sequences that represent important information about a single API call and relevant expressions/statements surrounding it. The correct usage patterns are then applied to Stack Overflow to determine how reliable the code examples were. Zhou et al. [2017] provides a method of detecting defects between API documents and their code implementations by extracting the constraints from directives and code as FOL expressions, then comparing them with an Satisfiability Modulo Theories (SMT) solver. In particular, this work defines and focuses on several categories of constraints for directives, which represent the largest portion of API documentation at 43.7% Monperrus et al. [2012a]. Heuristic patterns and regular expressions used from this work are the building block of API caveat contracts construction in Chapter 4. The concept of mutation analysis has also been suggested for exposing API misuse by Wen et al. [2019]. Mutation analysis involves many, small modifications to a program called *mutants* that are then executed with a given test suite. Execution data from the stack trace of these programs can then be used to determine whether a mutant introduced an API misuse and recognise patterns misuse. Finally, an example of the application of static code analysis for detecting bugs is shown by Bae et al. [2014] for JavaScript web applications. Note that JavaScript is a dynamically typed programming language, but static type analysis can be used to perform type-based analysis. This uses fact that Web APIs are typically specified in a certain format known as Interface Definition Language, which exposes function semantics.

2.4 Summary

This chapter provides an overview of related works for API caveats, NLP applied to API documentation, and the use of static code analysis for API misuse detection. Related works that are the foundations of this thesis are identified and key terminology used is defined to avoid ambiguity for readers. In Chapter 3, the techniques in Sun [2018] are applied to GitHub data to investigate linking API caveats to a different community platform domain (GitHub).

Chapter 3

Locating API Misuse Examples in GitHub Data

Section 3.1 provides an overview of linking API caveats to code examples on Q&A websites and the same approach applied to GitHub for this thesis. Section 3.2 describes the use of TF-IDF, word2vec and BM25 as information retrieval systems. This is followed by the extraction of GitHub data and API caveats from the Java 12 API documentation (Section 3.3). The complete process used for sentence matching of GitHub comment sentences and the Java 12 API caveat sentences is also described. Finally, Section 3.4 describes the results of sentence matching and Section 3.5 summarises the findings of this chapter.

3.1 Introduction

One approach for linking API caveats to code examples is to utilise real code examples from community Q&A websites such as Stack Overflow. Those websites allow users to post questions that may contain some "buggy" code that requires fixing. The idea for linkage then involves analysing the natural language surrounding this code example, which can be from the answers or the question itself. The approach taken in Sun [2018] and Ren et al. [2018] used sentence embedding to show that sentences in answer posts typically have high similarity to sentences of API caveats. This chapter therefore aims to extend upon this approach by applying it to another domain: GitHub. GitHub is a website that is abundant with community text and code examples, but its central focus is for providing an interface to Git, a version control system that allows developers to maintain their projects over time. It is therefore commonly used for hosting free and open-source software. According to GitHub's internal statistics[1], the website includes over 31 million developrs and 96 million project repositories in 2018. A key component of GitHub is its *issues* features, which lets developers post questions, suggestions or report bugs for a repository. These issues can contain multiple *comments*, typically posts from other users, that answers the questions of the issue. It can therefore be seen that GitHub shares some similarities to Q&A websites with the addition of significantly larger collection of example

[1] https://octoverse.github.com/

code for analysis. The same approach from Sun [2018] and Ren et al. [2018] can then be applied to GitHub to investigate if the API caveat linkage to code examples can be improved.
I extract 1,855,870 GitHub comments related to Java throughout 2018 via the GitHub Archive project. I perform sentence tokenisation and filtering to these comments to find those associated code examples, resulting in 629,933 comment sentences. I then use the Java 12 API documentation to extract 73,831 API caveat sentences. I then perform similar sentence preprocessing and filtering from Sun [2018] and create 4 information retrieval systems consisting of TF-IDF, word2vec, BM25 and word2vec + BM25 to match the API caveat sentences with GitHub comment sentences. However, all of the information retrieval systems performed considerably worse in comparison to the studies on Stack Overflow by Sun [2018] and Ren et al. [2018], with 1% or less precision. A significant lexical gap was identified as the cause for the poor results from, a consequence of many generic caveat sentences in the Java 12 documentation in combination with GitHub data that relate to many different APIs, and because of the differences in platforms of GitHub and Stack Overflow.

3.2 Design

Section 3.2.1 provides an overview of the TF-IDF statistic and its use in information retrieval, followed by a description of the word2vec model for sentence embedding (Section 3.2.2) and Okapi BM25 function (Section 3.2.3). For the purposes of this thesis, *documents* is used interchangeably with *sentences* and *terms* is used interchangeably with *words*.

3.2.1 TF-IDF Sentence Matching

TF-IDF is short for *term frequency-inverse document frequency*. TF is based on the frequency of a term within a document while IDF refers to the inverse of the number of documents that contain the given term Robertson [2004]. TF-IDF is computed as $TF \cdot IDF$, the multiplication of two measures. The formal definition of IDF is shown in Equation 3.1, where t is a token and D is the documents in the corpus. The overall motivation for TF is based on the assumption that meaningful words may appear more often as keywords for a particular document. However, this assumption means common terms such as “the” and “a” would be incorrectly emphasised as more meaningful. The IDF is therefore used to minimise the weights of words that appear frequently across a corpus.
The TF-IDF heuristic is thus a widely used technique for information retrieval systems that has been successful in several cases Havrlant and Kreinovich [2017]. For sentences, TF-IDF typically follows a bag-of-words approach in which vectors have length equal to the vocabulary size and each component of the vector corresponds to

the TF-IDF score of a particular term.

$$idf(t, D) = log \frac{\mid D \mid}{1 + \mid \{d \in D : t \in d\} \mid} \tag{3.1}$$

3.2.2 Word2Vec Embedding

Word2vec is a computationally efficient neural network that is trained to learn and produce word embeddings. The original motivation for this was to capture contextual information such that words could be predicted in a sequence of words Mikolov et al. [2013a]. The two models that can be used with word2vec is Continuous Bag-of-Words (CBOW) and Skip-Gram, though only an overview of Skip-Gram will be provided here as it was the model used for sentence embedding. The Skip-Gram model is used to predict surrounding words given the current word, with the property that even "simple vector addition can produce meaningful results" Mikolov et al. [2013b]. An example of this provided in Mikolov et al. [2013b] is vec("Germany") + vec("capital") results in a vector close to vec("Berlin"). A simple method for applying the word embedding to sentences is to use the sum the word vectors to represent the overall sentence. This is the method used for sentence similarity computations in 3.3.6.

3.2.3 BM25 Ranking

BM25 is a ranking function that outputs the relevance of documents in comparison to each other for a given query. It is computed from Equation 3.2, where $f(q_i, D)$ is term frequency of the term q_i, which is the i^{th} in document D, $|D|$ is the document length, k_1 and b are tuning variables, and IDF' is an alternative version of IDF that is defined by Equation 3.3 Manning et al. [2008]. In particular, $b = 0.75$ and $k_1 \in [1.2, 2]$ are reasonable values reported by Manning et al.. The values $b = 0.75$ and k_1 were used in 3.3.6. BM25 was considered a state-of-the-art model for information retrieval in previous years Pérez-Iglesias et al. [2009]. It is noted that since the BM25 scores are only comparable for a specific query, they can be normalised to the range 0-1 with 0 representing the least amount of relevance and 1 representing the highest.

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}'(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})} \tag{3.2}$$

$$\text{IDF}'(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \tag{3.3}$$

3.3 Implementation

Implementation of the data retrieval and extraction for GitHub comments alongside collection of the Java 12 API documentation and extraction of its API caveats was completed in Python 3.6. In particular, the Gensim library was used to apply

word2vec sentence embedding to the GitHub comment and Java API caveat sentences. The Sci-kit library was used to obtain sentence vectors from TF-IDF sentence embedding. Rankings for BM25 were computed manually using its associated function. Section 3.3.1 describes the collection of GitHub data and its extraction for sentence embedding and matching. Section 3.3.2 provides an overview of the Java 12 API documentation and extraction of its API caveat sentences. Section 3.3.3 explains the sentence preprocessing steps performed, followed by filtering steps to ease the scope of sentence matching required for the large sets of sentences (Section 3.3.4). Section 3.3.5 illustrates the tools used for sentence embedding, and finally, Section 3.3.6 describes how the Java 12 API caveat sentences were matched with sentences from GitHub comments.

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.[2] 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

[2]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].

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. 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 all GitHub comments results in 1,855,870 tokenised sentences. It is noted here that in Sun [2018], two additional steps are performed for the sentences: (1) coreference resolution, where pronouns such as "it" and "this" are substituted with the closest API names, and (2) lexicon construction in which an API lexicon is constructed from the code of question and answer posts of Stack Overflow. Both of these steps were intended to reduce the complexity and noise of sentence embedding. However, an initial search of the GitHub data via Google found that finding examples of API misuse from GitHub data was much more difficult than with Stack Overflow data. It was suspected that perhaps the API caveats of Java were too generalised for GitHub as it pertained to the standard library of a programming language rather than a specific API. For simplicity, the first step was therefore ignored and the second step was reduced to keyword matching in 3.3.4 for determining the relevant API elements for the GitHub data.

3.3.4 Candidate Filtering

The caveat sentences that were associated with deprecated API elements or not from the description, parameter or exception sections of an API element were filtered. The sentences for deprecated elements were ignored because those caveats are usually obvious to developers from IDEs, which mention when an API element is deprecated. Note that this is accomplished using the `@Deprecated` annotation for Java and is tagged onto deprecated methods by API developers. The sentences from description, parameter and exception sections of API elements were selected in particular as they were the main sentences that described caveats about the elements from initial observation.

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. To counteract this problem, caveat sentences that exceeded an arbitrary (generous) limit of 400 characters were excluded from further analysis and sentence matching. In addition to this, it was noted that the large number of GitHub issue comment sentences would require significant processing time and power. The potentially relevant API caveats for each GitHub comment were then identified by concatenating the text of each comment (which consisted of one or more sentences), transforming the string to lowercase characters only, and performing a sub-string search for the (lowercase) class name associated with API caveats.

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)]
```

Hence, sentence matching would only be performed in later steps if the enclosing GitHub comment was identified as relevant to a given API element. Caveats that were potentially relevant to more than 1000 GitHub comments were also restricted to those 1000 GitHub comments to further reduce computation. This restriction involved 213 of the 21,932 API elements from the Java 12 documentation. We note that this limits the search area of sentence matching for those API caveats considerably, but is acceptable for an initial attempt for sentence matching. The total number of caveat sentences after preprocessing and filtering totalled 73,831.

For GitHub comments, those that were not attached to an issue containing a code block were filtered. This is because the purpose of sentence matching is to indirectly link API misuse in code to the API caveat sentences. From the dataset, 85,318 issues contained a code block, which had a total of 290,019 relevant comments. Those comments contained 629,933 sentences in total. The next step for sentence matching was to perform sentence embedding for the caveat sentences and GitHub comment sentences.

3.3.5 Sentence Embedding

Sentence embedding provides a method of comparing the similarity of two sentences by representing them in vector space and computing the cosine similarity between the two vectors. Representing variable length sentences as fixed length vectors has been a major research topic related to deep learning approaches for NLP **??**. For TF-IDF, the Scikit-learn library was used. This involved importing the class `TfidfVectorizer` and calling its `fit_transform` function to create vectors for all API caveat sentences. For word2vec sentence embedding, the `word2vec` class from the `gensim` library was imported and used. Note that the input for sentence embedding was the preprocessed, tokenised and filtered forms of caveat sentences from the previous sections. Overall, sentence vectors were generated via TF-IDF and word2vec functions/algorithms.

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);
        }
}
```

3.3.6 Candidate Matching

Sentence matching was accomplished using cosine similarity for the TF-IDF and word2vec sentence embedding vectors. For BM25, its function already output a ranking score and did not require further computation. To evaluate the cosine similarity of the vectors, each of the 73,831 caveat sentence vectors was compared to a subset of the 629,933 sentence vectors from GitHub comments corresponding to sentences deemed potentially relevant to the given caveat sentence in 3.3.4. The equation in 3.4 was then used to compute the similarity scores. This score is then reversed such that 1 represents perfect similarity between two sentence vectors, while 0 represents complete dissimilarity between two sentence vectors. Given that BM25 ranks the sentence vectors relevant to each other, the scores were then normalised to the range 0 and 1, with 1 being representing highest similarity. Thus, scores were computed for each caveat sentence against their subset of potentially relevant GitHub comment sentences for TF-IDF, word2vec and BM25. A weighted combination of BM25 and word2vec was also considered (with equal weightings for both scores) since using both approaches could be expected to create a balance between their advantages and disadvantages.

$$\cos(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{AB}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} \mathbf{A}_i \mathbf{B}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{A}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{B}_i)^2}} \tag{3.4}$$

To evaluate the performance of each of the approaches, a statistical sampling size was estimated via 4.1 using a 95% confidence interval, 5% error margin and a population size of 73,831 (i.e. the number of preprocessed and filtered caveat sentences). The estimated sampling size population was 383, though 384 was used for consistency with Ren et al. [2018] in which notably more API caveat sentences from the Android documentation were analysed (160,112). The precision at K metric was chosen for a simplistic, initial evaluation of a query for the information retrieval systems, where K was set to 3 to reduce the amount of manual labelling required for all approaches (TF-IDF, word2vec, BM25 and word2vec combined with BM25). This would then be combined with the mean average precision (at k) to quantify overall performance of the systems across multiple queries. Therefore, a random sample of 384 caveat sentences was selected for each of the information retrieval systems and the top 3 results were returned for each query. Note that a random sample was computed for each system since we are mainly interested in relevant API caveat sentences are to GitHub data, and less on the performance of the systems. The different approaches are only used for a general idea of different approaches that can be compared to Sun [2018] where API caveat sentences from the Android documentation were linked to community answers on Stack Overflow. Considering cases in which the information retrieval systems can return less than 3 results, random sampling resulted in 959, 936, 1129 and 1051 results for the TF-IDF, word2vec, BM25 and word2vec + BM25 information retrieval systems for manual labelling.

Listing 3.4: Example of a GitHub comment containing an error log from `https://github.com/ChrisRM/material-theme-jetbrains/issues/863`

```
similar problem just now, Linux (CentOS), upgrading from 2018.1 to 2018.2.
(I use the Darcula theme, and this stack trace is
    suggestive...)\r\n\r\njava.lang.ClassCastException: java.lang.Boolean cannot
    be cast to com.intellij.openapi.actionSystem.ex.ComboBoxAction\r\n\tat
    com.intellij.ide.ui.laf.darcula.ui.DarculaButtonUI.
getComboAction(DarculaButtonUI.java:75)\r\n\tat
    com.intellij.ide.ui.laf.darcula.ui.DarculaButtonUI.
getDarculaButtonSize(DarculaButtonUI.java:236)\r\n\tat
    com.intellij.ide.ui.laf.darcula.ui.DarculaButtonUI.
...
```

3.4 Results

Labelling for each of the information retrieval systems' results was conducted by manually checking if the given API caveat sentence query appeared to be relevant to the GitHub comment containing the matched sentence. The entire GitHub comment was used to include contextual information regarding the sentence, ensuring that the sentence was actually relevant to the API caveat. Thus, results were annotated with either *relevant* or *non-relevant* based on whether they referred to the API caveat query. It was found that a significantly small percentage of the results were relevant to their API caveat query, with only 9, 16, 5 and 11 results identified as relevant for the TF-IDF, word2vec, BM25 and word2vec + BM25 information retrieval systems. In other words, only about 1% of the query results for the random samples were found to be relevant to their API caveats. The mean average precision at 3 for all of the information retrieval systems are therefore approximately 0. It was noted that using a similar version of the code on another (currently unpublished) work[3] based on Stack Overflow and Android API caveats showed promising results. The results were therefore inspected to understand why the systems performed poorly for GitHub data.

From further analysis of the results, it was found that numerous GitHub comments contained code or error logs as text. In particular, error logs are common occurences as GitHub contains repositories for many different APIs and projects, though formatting of these are not regulated. An example of an error log for a GitHub comment in its raw text form is shown in 3.4. Note that the example has been truncated and long lines were split for better viewing. This would result in some of the code words to be processed as natural language words for the information retrievals and would also be difficult to clean. In addition, it was noted that GitHub contains a collection of text regarding many different APIs. In comparison to Stack Overflow, which allows users to tag their questions under a certain topic (such

[3] `https://xin-xia.github.io/publication/ase192.pdf`

as Android), GitHub does not provide that functionality. This means it is much harder to differentiate the GitHub comments by API relevance. The consequence of this was seen from many cases where the correct API method name was identified, but because the name was quite generic (e.g. `truncate`, `compareTo`, `getOffset`), the information retrieval systems would provide positive results for comments concerning caveats from other Java related APIs. It is noted that perhaps linking caveats for a more specific API (such as specific Java libraries like ReactiveX or OkHttp) would yield better results for GitHub since those projects have GitHub repositories that allow users to ask project/API related questions contained within their repositories. Furthermore, many of the caveat sentences from the Java API could be considered too generic. For example, the `set` and `get` methods of the `java.util.ArrayList<E>` class both share the same sentence: "IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())". An example where unrelated classes and methods of the Java API contain similar caveat sentences is from the constructor of `java.util.HashMap<K,V>` that says "NullPointerException - if the specified map is null" and from the `addAll` method of `java.util.ArrayList<E>` that says "NullPointerException - if the specified collection is null".

Another possible reason that GitHub was considerably worse for linking API caveat sentences is because its community text serves a different purpose compared to Q&A websites such as Stack Overflow. In particular, community text regarding API caveats would likely be in the format of a bug report, and answers to specific caveats can already be found on platforms catered to answering those caveats (i.e. Stack Overflow). Ultimately, it can be seen that due to the usage of same/similar sentences for different API elements, the inclusion of many different APIs on GitHub, and different purpose of GitHub results in a significant lexical gap for the Java 12 API caveat sentences and community text from GitHub. The poor results from this approach lead to the idea for a more direct approach of linking the API caveat sentences to source code described in Chapter 4.

3.5 Summary

This chapter described the work from Sun [2018] and Ren et al. [2018], and focuses on extending their solutions to a different domain: GitHub. The data of GitHub is of interest for API caveat linkage because it is another community-centered website that contains both text and code that could be used to highlight examples of API misuse. Furthermore, it is inherently different from the focus of Sun [2018] and Ren et al. [2018], which is strictly Q&A community websites. The Java 12 API documentation instead of the Android API documentation was chosen for analysis as it could be generalised to more projects on GitHub and was also studied in Zhang et al. [2018] and Zhou et al. [2017] for API usage errors and API implementation discrepancies respectively. After extraction of Java 12 API caveat sentences using the keywords/-patterns in Li et al. [2018], preprocessing and filtering, 73,831 caveat sentences were

found. For GitHub data, the GitHub Archive project was used to collect all GitHub events in 2018. The complete list of Java related projects that had at least 2 stars was also collected using the GitHub REST API, which found 291,152 projects. A total of 1,855,870 GitHub comments related to Java in 2018 were then identified. The sentences underwent data preprocessing, filtering, and sentence embedding (TF-IDF and word2vec) to perform sentence similarity computations between the two datasets of GitHub comment sentences and Java 12 API caveat sentences. Information retrieval systems was created based on TF-IDF, word2vec, BM25 and word2vec + BM25 to perform sentence matching, where caveat sentences were used as queries and query results were relevant GitHub comment sentences.

Manual labelling was performed to measure the performance of this approach and the information retrieval systems. This was conducted with statistical random sampling with a 95% confidence interval and 5% error margin. However, 1% or less of the query results were found ot relevant for all of the information systems used. The cause of this was discussed and suspected as a result of generic caveat sentences in the Java 12 documentation in combination with GitHub data relating to many different APIs, and because of the different purpose of GitHub data in comparison to Q&A websites.

In Chapter 4, I take a more direct approach to linking API caveat sentences to source by transforming API caveat sentences into *contracts* and conducting static code analysis.

Chapter 4

API Contracts Construction with Static Code Analysis

Section 4.1 provides background information for code contracts and an overview of the work conducted for this Chapter: constructing code contracts from API cavaeat sentences and using static code analysis to apply the contracts to code in real time. Section 4.2 describes the design framework for generating and using code contracts. This includes a statistical analysis on the Java 12 API caveats, a description of the idea used for extracting information from caveat sentences for code contracts construction, and the concept of checker programs that can utilise the code contracts. Section 4.3 explains the implementation of API contracts construction and static code analysis in an IntelliJ proof-of-concept plugin. This is followed by the output of the plugin (Section 4.4) and a summary of this chapter in Section 4.5.

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. [2012b]. 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

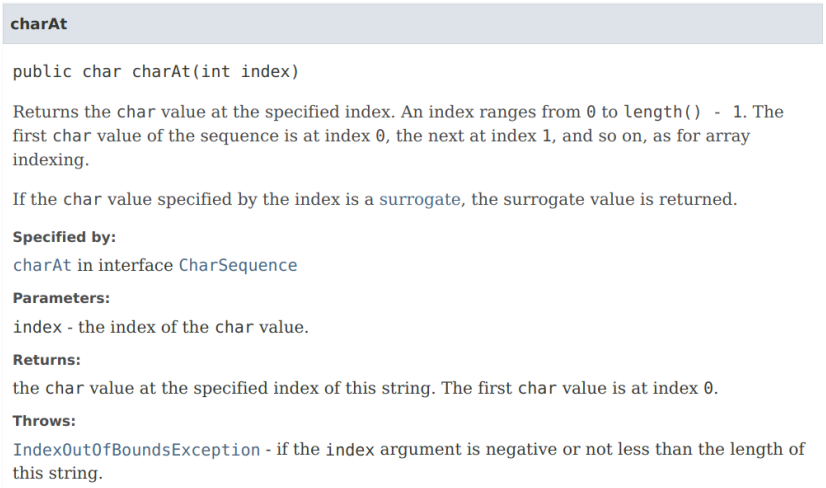
To better understand the prevalence of these API caveat categories identified by Zhou et al. [2017] in the Java 12 documentation, a statistical sampling method from Singh and Mangat [2013] is applied. Specifically, the minimum number of samples required to ensure the estimated population mean is within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



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

	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.

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

	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.

		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.

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 *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-verb-object (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 6.1.1. 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

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).

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.

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, which are replaced with a labelled word to facilitate dependency parsing. This process is a form of *sentence normalisation*. However, rather than using a dependency parser in combination with manually formulated heuristic rules catered to dependency parsing output, we can simply use heuristic rules that are catered to the normalised sentences given knowledge about the structure of SVO in English sentences. For example, a sentence in the exception section of the `ArrayBlockingQueue` constructor for class `java.util.concurrent.ArrayBlockingQueue<E>` says:

```
IllegalArgumentException - if capacity is less than c.size(), or less than 1
```

If we ignore the exception, and focus on the text after the hyphen ('-' symbol), we obtain:

Regular Expression	Normalisation Form
(not? (less | shorter) than) | ((greater | larger) than or equal to)	>=
(not? (greater | larger | longer) than) | ((less | shorter) than or equal to)	<=
(less | shorter) than	<
(greater | larger | longer) than	>
((is | are)? not negative) | ((be)? non-negative)	>=0
((is | are)? not positive) | ((be)? non-positive)	<=0
(is | are)? negative	<0
(is | are)? positive	>0
not equal(to)?	!=
equal to	==

Table 4.6: Regular expressions used to normalise mathematical phrases.

```
if capacity is less than c.size(), or less than 1
```

From this, we can then apply some additional heuristic rules that are loosely based on the formulated heuristics from Zhou et al. and common English phrases for mathematical expressions. The rules formulated for this project are shown in 4.6. In particular, these rules are used first in the normalisation process. For the given example, the first phase of normalisation results in:

```
if capacity is < c.size(), or < 1
```

The second phase of sentence normalisation involves the regular expressions in 4.7 and normalisation of additional white-space characters. This allows parsing based on heuristics that are based on the SVO structure of English rather than manual observation of Java API specific sentences used in Zhou et al.. The result of this is:

```
if @PARAM1 is @EXPR1, or @EXPR2
```

Here, "@PARAM1" represents "capacity", which is the first parameter of the method, and "@EXPR1" and "@EXPR2" represent the mathematical expressions within the sentence. Finally, a greedy parsing approach is used to extract the constraints from the sentence. We assume the linear nature of the SVO structure is used for most sentences, which suggests that each word of a sentence can be considered from left to right, to perform our parsing. Hence we tokenise the sentence such that we have a list of tokens and iterate over the list in a single pass. States are saved throughout the iteration process based on the token observed at each step of iteration. These states contain information such as whether a negating word (e.g. "not", "false") or expression was encountered. A simplistic rule for the parsing is to then construct a constraint whenever an expression token is encountered. The last parameter observed is then used to *complete* this expression. For the given example, both "@EXPR1" and "@EXPR2" are therefore completed with "@PARAM1" to form the

Type	Description	Regular Expression
Specific Values	0.0, 0.1f, etc.	\W(-)?[0-9]+(,[0-9]+)*((\.[0-9]+)?[a-z]*)\W
	Member value of objects e.g. Location.x	\W(^(java\.|javax\.|org\.))? ([A-Za-z_]+\w+\.)+[a-z_]+[a-z0-9_]*[^\.A-Za-z0-9_]
Class methods and static members	class methods, e.g., ClassA.func(Param1)	\W[A-Za-z_]+[A-Za-z_0-9]*(\.[A-Za-z_]+[A-Za-z_0-9 (#[A-Za-z_]+[A-Za-z_0-9]*)?[()]*\W
	Static member, e.g. Desktop.Action#OPEN	\W([A-Za-z_]+[A-Za-z_0-9]*(\.[A-Za-z_]+ [A-Za-z_0-9])*)?(#[A-Za-z_]+[A-Za-z_0-9]*)[^A-Za-z0-
	All upper case	\W(\w+\.)*([A-Z]+_)*[A-Z]+\W
	Class name	\W([A-Za-z_]+\w+\.)*[A-Za-z_]*[A-Z]+ \w+[^\.A-Za-z0-9_]
Expressions	A - B	\W\w+((\s+-)|(-\s+)|(\s+-\s+))\w+\W
	A + B	\W\w+\s*\+\s*\w+\W
	A * B	\W\w+\s**\s*\w+\W
	A..B	\W(?\s*\w+\s*)?\s*\.\s*\.\s*(?\s*\w+\s*)?\W
	[A, B]	\W[\s*\w+\s*,\s*\w+\s*]\W
	[A..B]	\W[\s*\w+\s*(\.\s*\.\s*)\s*\w+\s*]\W
	A <\<= B <\<= C	\W\w+\s*<=?\s*\w+\s*<=?\s*\w+\W
	A >\>= B >\>= C	\W\w+\s*>=?\s*\w+\s*>=?\s*\w+\W
	From A to B	\W(from\s+)?\w+\s+to\s+\w+\W
	A != B	\W\w+\s*!=\s*\w+\W
	Enumeration expression	\W (\s*\w+\s*)(,\s*\w+\s*)+,?\s*or\s*\w+\W

Table 4.7: The regular expressions identified in \cite{} for sub-sentence substitutions and dependency parsing.

constraints $capacity < c.size()$ and $capacity < 1$. We note that for the scope of this project, the rules used to construct these constraints are simplistic and only consider these mathematical constraints alongside logical "or" conditions as shown in the example above. API caveat sentences consist of other constraint types (such as temporal rules for API calls) that would require more complex parsing techniques. However, we mainly propose this approach as an alternative to the heuristics-based approach from Zhou et al. that requires significantly more effort with manual labelling.

4.2.3 Checker Programs

Describe IntelliJ plugin and Boa AST

To apply the constructed API caveat contracts to source code, it was assumed that program analysis tools exist that can utilise the caveat contracts constructed for a given API documentation. Fortunately, such tools exist and are contained within a field known as static code analysis Louridas [2006]. Static code analysis uses checkers that can perform checks on a program without executing them. A common data structure used by these tools is Abstract Syntax Trees (ASTs), which models the structure of source code as a tree. Syntactic details of the code are not represented which

enable analysis of code to be significantly simpler. Consider the following code for the Euclidean algorithm:

```
while b != 0
    if a > b
        a := a - b
    else
        b := b - a
return a
```

The AST of the program is shown in 4.2. In particular, ASTs are can be used to extract important information such as an API call or to generate API sequence calls as shown in Zhang et al. [2018]. Furthermore, static code checkers can be developed that use API caveat contracts as constraints to check for within the AST of an arbitrary program. Given some API caveat contract, we could traverse the AST of a program and identify any code expressions that correspond to the relevant API call. Checking if a code contract is satisfied then requires one or more of the following processes: observing the surrounding API calls, identifying which are relevant, inspecting the various control flows of the program, evaluating expressions, and inspecting the API call arguments used. The IntelliJ platform was chosen for development of a proof-of-concept plugin given that it was one of the three most popular Java IDEs (next to Netbeans and Eclipse) at the time of writing and provided a simple interface for accessing the AST of Java code. IntelliJ provides an interface known as the Program Structure Interface (PSI) that allows multiple approaches to navigation of an AST, making the development of a plugin relatively simple.

4.3 Implementation

Implementation of the Java 12 API caveat contract construction was completed purely using Python 3.6 and its standard libraries. The contracts generated were exported as a JSON array to be used in other applications (i.e. plugin) and can be found in the project repository (`./exception_range_rules_filtered.json`). The IntelliJ plugin was developed in Java 11.

4.3.1 Java API Caveat Contracts

The JSON objects for caveats of the Java 12 classes were reused from 3.3.2 and loaded as a Pandas library `DataFrame` object. The object consisted of rows of individual caveats while the columns represented various information about a caveat such as the sentence involved and which class it belonged to. This allowed filtering functions to be easily applied such that a subset of the caveats could be selected based on some condition. Constructing the caveat contracts involved following the algorithm described in 4.2.2. In particular, the regular expressions in 4.7 and 4.6 were written in Python's `re` library syntax. In addition to the algorithm described in 4.2.2,

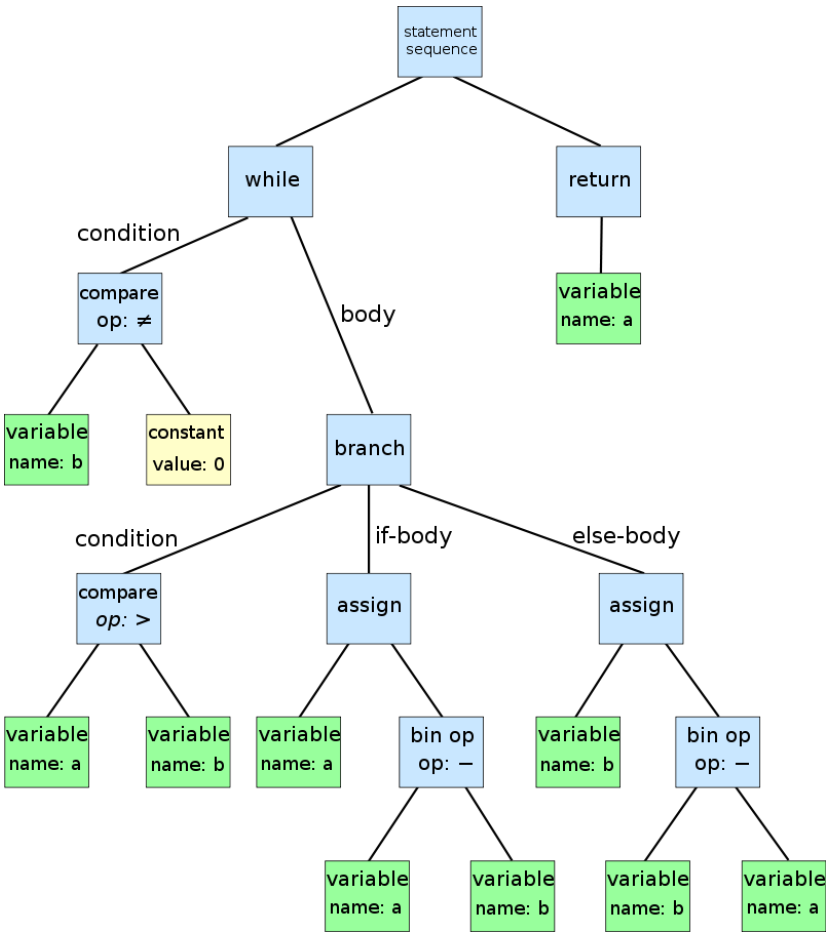


Figure 4.2: Example of an AST for the Euclidean algorithm. Copied from `https://en.wikipedia.org/wiki/Abstract_syntax_tree`

a simpler approach was also considered for *not-null* caveats. From observation of the exception level sentences of the Java 12 documentation, it was found that any mention of the term "null" would usually imply an exception to be thrown if some parameter was `null`. Therefore, an approach to capture all *non-null* caveats for exception level sentences involves a sub-string check to see if "null" appears within the sentence. It was assumed that the same logic could be applied to the caveat categories of interest (*range limitation* and *type restriction*). Furthermore, it was noted that these caveat sentences would require certain key terms to be considered at all for a category (e.g. "null" would have to appear within a sentence to specify whether `null` is allowed or not). Generalised regular expressions were then used to filter the caveat sentences into whether they potentially belonged to one of the caveat categories of interest. For example, to collect caveats that potentially issued a *non-null* constraint, "null" was used as a search term and the sentence was considered if it contained the sub-string "null". The complete list of regular expressions used can be found in the `APIDoc2Rules.ipynb` Jupyter notebook. Applying these filters resulted in significantly smaller sets of caveat sentences that required manual labelling to determine what constraints they described for future contract construction. Specifically, the unique potential *not-null* exception and parameter sentences totalled 835 and 193 respectively. For the *range limitation* category, the exception and parameter sentences totalled 193 and 149 respectively. Given the small sets of sentences obtained, a random sample of size 100 was collected for each group/filter. For the *not-null* categories of exception and parameter sentences, the subject parameter of the constraint was labelled. As an example, for the sentence "prefix - the prefix of the tag, may not be null", *prefix* would be marked as the constraint subject. Note that the parameter sentences follow a template as described in 4.2.1 which makes extracting the relevant parameter trivial whereas for an exception sentence such as "if the specified sorted set is null" from the constructor of the `java.util.TreeSet<E>` class, parsing is harder given that the name of the relevant parameter ("s" in this case) is not used.

For the *range limitation* sentences, the constraints were expressed as a mathematical expression. An example of this is with the sentence "IllegalArgumentException - if iv is null or (iv.length - offset < 2 * (wordSize / 8))" from the `RC5ParameterSpec` constructor of the `javax.crypto.spec.RC5ParameterSpec` class, the constraint of the sentence would be identified as $iv.length - offset < 2 * (wordSize/8)$. After manual labelling, 90% of the 100 random parameter sentences filtered for *not-null* were found to actually describe a *not-null* constraint. For the exception level sentences, 87% were found to describe a *not-null* constraint. Moreoever, for the *range-limitation* samples, 49% and 72% of the parameter and exception sentences were found to express an range limitation constraint, respectively. These results reveal that using basic sub-string searches for *not-null* constraints is a viable approach for the Java 12 API documentation, while more complex approaches are required for identifying (and extracting) *range limitation* caveats. Applying the sentence normalisation algorithm described in 4.2.2, the constraints were found to be correctly extracted for 41 of the 72 API caveat sentences with a range limitation constraint in the sample. An addi-

	Predicted Non-constraint	**Predicted Constraint**
Actual Non-constraint	25	2
Actual Constraint	19	54

Table 4.8: Confusion matrix for the sentence normalisation approach proposed.

tional 13 caveat sentences were partially able to extract the correct constraints (which would also cause an exception to be thrown). Overall, the results are represented in terms of a confusion matrix in 4.8 to allow computation of accuracy, and F-measure. It can be seen that the true positive (TP) is 54, true negative (TN) is 25, false positive (FN) is 2, and the false negative is 19. Using the equations 4.2, 4.3, 4.4 and 4.5, the accuracy is 0.77, recall is 0.73, precision is 0.96 and F1 score is 0.83. These results suggest that applying sentence normalisation is a viable approach for extracting constraints from sentences that describe exceptions thrown under certain conditions (or at least for the Java 12 API documentation). These extracted constraints were then used to represent API caveat contracts in static code analysis. Each caveat contract consisted of the full class name of the associated API caveat, the method name and its signature (containing its return type and list of parameters), and the constraint for one of the parameters. Overall, 4,694 unique contracts were constructed for the Java 12 API documentation using the sentence normalisation algorithm proposed (in regards to *not-null* or *range limitation* constraints).

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.2}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{4.3}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{4.4}$$

$$\text{F1} = \frac{2 \cdot recall \cdot precision}{recall + precision} \tag{4.5}$$

4.3.2 IntelliJ Plugin with Static Code Analysis

IntelliJ's PSI provides the `AbstractBaseJavaLocalInspectionTool` class that can be extended for creating plugins involving static code analysis. This was used to define a *visitor* that would traverse the AST of a program periodically. In addition to this, classes were defined for the concepts of an API method, API class, caveat, and for storing a collection of all caveat contracts loaded from 4.3.1. Java's `HashMap` was used as the underlying data structure for storing the caveat contracts. This tree-like structure allowed searching for the contracts of an arbitrary method to consist of two simple steps: obtaining the methods attached to a certain class (via the full class name as a hash), finding the correct method by searching through the associated list of methods. It is noted that further optimisations could be used in the future

to quicken the retrieval of contracts for a given API method such as using the hash of the method signature to map directly to its caveat contracts. However, a simple design was chosen given that the plugin was a proof-of-concept application. Overall, the code analysis process involved implementing the visit function for the visitor such that each expression call in the program would be analysed. Specifically, the full class name, method name and argument types would be identified. The caveat contracts associated to the API call would then be retrieved. Each caveat contract would then be invoked and checked given the argument values provided to the API call. For the *not-null* constraints, this check simply required comparing the argument values to `null`. For the *range limitation* constraints, the logical operators ($<$, $\leq$, etc.) were also included alongside the comparing value in the contracts constructed from 4.3.1. This meant that an SMT solver was not required and we could simply applying the range constraints in Java code, though an SMT solver could be used in future for more complex constraints. We also note here that as baseline checker program, we only analyse argument values passed directly to an API call. Furthermore, the caveat categories of interest for this thesis are *not-null* and *range limitation*, which do not require further analysis of a program's AST even though more complex analysis can be conducted (such as evaluating expressions/variables that are passed as arguments to an API call). IntelliJ's PSI then provides the functionality to register a problem that will be displayed within the code editor of the IDE. Each caveat contract that was found to be violated would then result in a problem to be registered for the associated API call.

4.4 Results

From the construction of API caveat contracts for the Java 12 API documentation, and from the checker plugin implemented for IntelliJ, a proof-of-concept for applying natural language to source code was completed. To showcase an example of the plugin's functionalities, 4.3 illustrates several constraint-violating API calls for Java 12 API elements in IntelliJ. Using the plugin, the result is each constraint-violation is highlighted in IntelliJ with red squiggly lines as shown in 4.4. Hovering the cursor over any of these highlighted problems will bring show a pop-up that provides more detailed information about the caveat contract violation. An example of this is shown in 4.5. In these examples, IntelliJ version 2018.2.4 (community edition) was used.

It is noted that IntelliJ does implement its own version of contracts that require annotations following a specific syntax within the IDE. However, these contracts must be manually implemented for each method and were found to be mostly inconsistent. An example of this inconsistency is shown in 4.6 and 4.7. In 4.6, a *not-null* constraint is violated for the `isAfter` API call and correctly highlighted by IntelliJ's contracts. In 4.7, another *not-null* constraint is violated for the `add` method of a `PriorityQueue` object, which throws a `NullPointerException` if executed. Hence, it can be seen that the code contracts implementation has two major drawbacks: it requires manual implementation of IntelliJ contracts for each method, and it is inconsistent (poten-

```
public static void main(String[] args) {
    // Not null constraints violated
    SchemaFactory schemaFactory = SchemaFactory.newInstance(null);

    Font font = new Font( name: "TimesRoman", Font.PLAIN,  size: 12);
    font = Font.getFont( nm: null,  font: null);

    System.load( filename: null);

    // Range limitation constraints violated
    BasicStroke basicStroke = new BasicStroke( width: -1);

    Random r = new Random();
    r.nextInt( bound: -1);

    MessageInfo messageInfo1 = MessageInfo.createOutgoing( address: null,  streamNumber: -1);
    MessageInfo messageInfo2 = MessageInfo.createOutgoing( address: null,  streamNumber: 65537);
}
```

Figure 4.3: Example of IntelliJ's (lack of) problem highlights for obvious constraint violations.

```
public static void main(String[] args) {
    // Not null constraints violated
    SchemaFactory schemaFactory = SchemaFactory.newInstance(null);

    Font font;
    font = Font.getFont( nm: null,  font: null);

    System.load( filename: null);

    // Range limitation constraints violated
    BasicStroke basicStroke = new BasicStroke( width: -1);

    Random r = new Random();
    r.nextInt( bound: -1);

    MessageInfo messageInfo1 = MessageInfo.createOutgoing( address: null,  streamNumber: -1);
    MessageInfo messageInfo2 = MessageInfo.createOutgoing( address: null,  streamNumber: 65537);
}
```

Figure 4.4: Example of how the developed plugin handles API caveat contract violations.

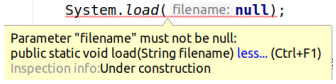


Figure 4.5: Example of displayed problem message for an API caveat contract violation using the plugin.

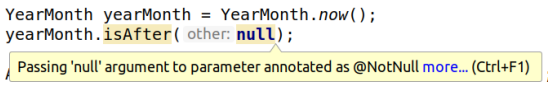


Figure 4.6: Example of IntelliJ's contracts.

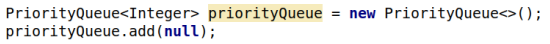


Figure 4.7: Example of IntelliJ's contract inconsistency.

tially as a result of the first problem). The plugin implemented is able to solve the first problem by automating the process of mapping sentences from the Java 12 API documentation to custom caveat contracts. Furthermore, it is able to solve the second problem with manual intervention as constraints for each individual API element are extracted and parsed into caveat contracts.

Overall, the plugin in its current iteration is able to highlight the explicit contract violations from the Java 12 API documentation related to *not-null* or *range limitation* constraints. It is clear that with the addition of other categories of API caveats mapped to contracts, the plugin could be used to highlight many errors and problems for developers in real-time, which could minimise API misuse and help developers learn correct usage of an API. In terms of constructing caveat contracts, further research could yield methods that both improve the precision and recall of constraints extraction. It could also be used to bridge the gap between developers of an API and users of the given APIs such that API misuses can be minimised. In terms of static code analysis, the plugin shows that natural language can be applied to source code to improve understanding of an API, avoid API misuses and potentially increase efficiency of learning/development for programmers.

4.5 Summary

This chapter focused on the concept of code contracts and associated it to API caveats. The idea of transforming API caveats to contracts was presented and implemented for a subset of API caveats that were related to *not-null* and *range limitation* constraints. A statistical analysis was also performed to determine the prevalence of these categories of caveats in the Java 12 API documentation. It was found that 20% of unique API caveat sentences appearing in the exception sections of the documentation imposed a *not-null* constraint. For the *range limitation* category, 13% of unique API caveat sentences were found to describe a constraint involving range. These categories were chosen for investigation and implementation of code contracts as they were explicit constraints that would result in an exception and could be considered the baseline API caveats for mapping natural language to code contracts. Furthermore, a sentence normalisation algorithm was proposed for extracting the range constraints from sentences and used to construct 4,694 unique contracts. Design and implementation of a proof-of-concept IntelliJ plugin that acted as checker

for the the caveat contracts was also presented. The potential of this approach for linking natural language to source code was also discussed. In particular, the plugin showcases that API misuses could be detected in real-time, improve understanding of APIs and direct developers to correct usage of an API.
In Chapter 5, the findings of this thesis are summarised and ideas for future work are presented. Lastly, some final remarks about the research focus of this thesis are expressed.

Chapter 5

Conclusion

In this chapter, I summarise the topic of investigation, the main challenges identified in Chapter 1 and how they relate to the main contributions of this thesis. Suggestions for future work are also presented.

Section 5.1 provides an overview of this thesis in terms of topic, challenges, contributions and findings.

Section 5.2 describes some future work ideas relevant to API caveats and constructing API caveat contracts.

5.1 Summary

This thesis presents the approach of using sentence embedding based on Sun [2018] for linking Java 12 API caveats to community text from GitHub alongside methods to construct caveat contracts from API documentation. Chapter 1 introduces the concept of API caveats and provided a realistic scenario to describe the motivation for linking API caveats to code. The main challenges of this thesis was also identified. Chapter 2 presents related work that defined the key concepts of this thesis: API caveats, using NLP techniques to augment API caveats with code examples, and the applications of static code analysis with API documentation. In Chapter 3, the process of extracting the documentation and caveats from the Java 12 API is described in detail. Furthermore, the approach for extracting GitHub community text data and the construction of multiple information retrieval systems based on TF-IDF, word2vec, BM25 is explained. The negative results of this approach is discussed and used to introduce an alternative method applied in Chapter 4. Furthermoer, Chapter 4 presents the idea of caveat contracts and the transformation of API caveats to form caveat contracts. The design of a checker program that can utilise the caveat contracts is given.
The main challenges of this thesis (Chapter 1) revolves around how API caveats can be linked to code. The challenges consist of effective sentence embedding and matching, sentence parsing techniques to extract constraints, and implementation of static code analysis that can use caveat contracts. In Chapter 3, I tackled the prob-

lem of sentence embedding and matching by extending the work of Sun [2018] with additional information retrieval system models. Specifically, information retrieval systems built upon TF-IDF, BM25 and word2vec + BM25 were also used in addition to just word2vec to investigate the differences of embedding models and overall approach applied to GitHub. I attempt to match 73,831 API caveat sentences against 629,933 GitHub comment sentences using the models mentioned, and perform statistical sampling with manual labelling to determine the performance of these information retrieval systems. However, a significant lexical gap was observed that resulted in all models to have precision less than or equal to 1%. The negative result of this reveals that sentence embeddings are non-optimal methods for linkage in the presence of large lexical gaps.
In Chapter 4, I tackle the challenges of sentence parsing for constraints extraction by utilising and combining ideas from Zhou et al. [2017] and Blasi et al. [2018] to propose a sentence normalisation approach. I conduct a statistical analysis of the Java 12 API documentation for constraints recognised in Zhou et al. [2017] as *not-null* and *range limitation* constraints to showcase their prevalence in the Java API documentation. I find that 20% of unique caveat sentences from exception sections of the documentation specify a *non-null* constraint, while approximately 13% of these sentences specify a *range limitation* constraint. I tackle the challenge of static code analysis simply by developing a proof-of-concept checker and showing how it can detect contract violations. Overall, I construct 4,694 unique caveat contracts and develop an IntelliJ plugin to automatically detect caveat contract violations in real time.

5.2 Future Work

Extending caveat contracts to include other caveat categories. Other interesting caveat types mentioned in Chapter 1 should also be considered for caveat contracts. These require much more complex parsing techniques. In addition, other methods to resolve dependencies of API caveats should be investigated. For example, the "add" method of "ArrayList" in Java specifies "IndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())". Identifying that "size()" refers to the `size` method of `ArrayList` is a complex problem given that cross class dependencies exist.
Detect bugs in existing software. Automatic detection of bugs remains a challenging problem. Two notable works that the methods proposed in this thesis can be applied to is API call sequences Zhang et al. [2018] and mutation analysis Wen et al. [2019]. The Boa programming language is used as a interface to a large number GitHub repositories in Zhang et al. [2018] to data-mine correct usage patterns of APIs. However, using caveat contracts, an interesting application would be to search for free, open-source software on GitHub that contain API misuses. Methods to effectively present these errors and suggest fixes to developers also requires further research as the approach of this thesis only identifies misuse after they occur: it does not teach developers about correct API usage beforehand. For Wen et al. [2019], I suggest the

topic of automatic caveat contract to test code mapping. Specifically, generating test suites for programs would be beneficial for all developers for code validation and development efficiency.

Testing other APIs and programming languages. Evaluating the practicality of caveat contracts with other APIs and programmings languages is another useful topic that should be researched. Differing results could be expected for analysis of dynamically typed programming languages for example. Furthermore, the different syntactic styles used for different API documentation presents a challenge for generalised approaches.

Chapter 6

Appendix

6.1 Heuristic Rules from Zhou et al. [2017]

6.1.1 Not-null 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 6.1: 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 6.2: Complete list of heuristic rules for parameter nullness not allowed.

6.1.2 Range Limitation Heuristics

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 6.3: 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 6.4: Complete list of heuristic rules for parameter range limitations.

Bibliography

Baca, D.; Petersen, K.; Carlsson, B.; and Lundberg, L., 2009. Static code analysis to detect software security vulnerabilities-does experience matter? In *2009 International Conference on Availability, Reliability and Security*, 804–810. IEEE. (cited on page 4)

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 9)

Bardas, A. G. et al., 2010. Static code analysis. *Journal of Information Systems & Operations Management*, 4, 2 (2010), 99–107. (cited on page 4)

Blasi, A.; Goffi, A.; Kuznetsov, K.; Gorla, A.; Ernst, M. D.; Pezzè, M.; and Castellanos, S. D., 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 242–253. ACM. (cited on pages 8 and 44)

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 30)

Havrlant, L. and Kreinovich, V., 2017. A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation). *International Journal of General Systems*, 46 (01 2017), 27–36. doi:10.1080/03081079.2017.1291635. (cited on page 12)

Huang, Q.; Xia, X.; Xing, Z.; Lo, D.; and Wang, X., 2018. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 293–304. ACM. (cited on page 8)

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, 7, 16, 22, and 26)

Li, S., 2018. Constructing software knowledge graph from software text. (2018). (cited on page 8)

Louridas, P., 2006. Static code analysis. *IEEE Software*, 23, 4 (July 2006), 58–61. doi:10.1109/MS.2006.114. (cited on page 33)

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 7)

Manning, C. D.; Raghavan, P.; and Schütze, H., 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA. ISBN 0521865719, 9780521865715. (cited on page 13)

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J., 2013a. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, (2013). (cited on page 13)

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J., 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119. (cited on page 13)

Mitchell, D. C., 1994. Sentence parsing. *Handbook of psycholinguistics*, (1994), 375–409. (cited on page 4)

Monperrus, M.; Eichberg, M.; Tekes, E.; and Mezini, M., 2012a. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17, 6 (2012), 703–737. (cited on page 9)

Monperrus, M.; Eichberg, M.; Tekes, E.; and Mezini, M., 2012b. 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 26)

Palangi, H.; Deng, L.; Shen, Y.; Gao, J.; He, X.; Chen, J.; Song, X.; and Ward, R., 2016. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 24, 4 (2016), 694–707. (cited on page 3)

Pandita, R.; Xiao, X.; Zhong, H.; Xie, T.; Oney, S.; and Paradkar, A., 2012. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, 815–825. IEEE Press. (cited on page 8)

Pérez-Iglesias, J.; Pérez-Agüera, J. R.; Fresno, V.; and Feinstein, Y. Z., 2009. Integrating the probabilistic models bm25/bm25f into lucene. *arXiv preprint arXiv:0911.5046*, (2009). (cited on page 13)

Ratinov, L. and Roth, D., 2009. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009)*, 147–155. Association for Computational Linguistics, Boulder, Colorado. https://www.aclweb.org/anthology/W09-1119. (cited on page 4)

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, 4, 8, 11, 12, 20, and 22)

Robertson, S., 2004. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of documentation*, 60, 5 (2004), 503–520. (cited on page 12)

Singh, R. and Mangat, N. S., 2013. *Elements of survey sampling*, vol. 15. Springer Science & Business Media. (cited on page 26)

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 9)

Sun, J., 2018. Augment api caveats with erroneous code examples. (2018). (cited on pages 1, 4, 8, 9, 11, 12, 17, 20, 22, 43, and 44)

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 8)

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 pages 9 and 44)

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 9, 22, 26, 29, 30, 34, and 44)

Zhong, H. and Su, Z., 2013. Detecting api documentation errors. In *ACM SIGPLAN Notices*, vol. 48, 803–816. ACM. (cited on page 9)

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, 4, 9, 22, 25, 26, 27, 29, 30, 31, 32, 33, 44, 47, and 49)

Zitser, M.; Lippmann, R.; and Leek, T., 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, vol. 29, 97–106. ACM. (cited on page 4)