

**A METHODOLOGY FOR INTENTIONAL  
SPECIFICATION AND USER-CENTERED  
DOCUMENTATION OF OBJECT-ORIENTED APIs**

Seham Abdulrahman A Alharbi

Doctor of Philosophy

University of York  
Computer Science

March 2025



## **Abstract**

The ideal companion to any application programming interface (API) is its documentation, which can take various forms, such as API reference documentation, user guides, and tutorials. One of the essential resources for learning APIs, commonly found in many types of API documentation, is code examples that demonstrate common API usage. However, writing and maintaining effective API usage examples is often a demanding and repetitive process for API developers. This is because API users ideally expect these examples to be simple, standalone, and linear. This PhD research addresses this challenge by providing an approach to assist API developers in producing and maintaining effective API usage examples. The approach also aims to support the development of code examples that align with API users' expectations, thereby promoting API learnability.

Additionally, this research proposes a method to assess coverage of API code examples. The proposed approach provides API developers with a dedicated API description language that enables them to concisely describe intended APIs. This API description is then utilised to report code example coverage both textually and visually.

The evaluation conducted in this research demonstrates that the proposed techniques can reduce the effort required to write and maintain API code examples by minimising repetition in example code and generating linear code. A controlled user study further shows that linear code examples are easier to comprehend and are preferred by API users. The findings also indicate that the proposed code example coverage tool and its accompanying API description language can simplify the process of describing intended APIs for API developers by providing features that facilitate formal and concise API specifications.

# List of Contents

<b>Abstract</b>	<b>3</b>
<b>List of Contents</b>	<b>8</b>
<b>List of Tables</b>	<b>10</b>
<b>List of Figures</b>	<b>13</b>
<b>List of Listings</b>	<b>15</b>
<b>List of Algorithms</b>	<b>16</b>
<b>Acknowledgments</b>	<b>18</b>
<b>Author Declaration</b>	<b>19</b>
<b>1 Introduction</b>	<b>20</b>
1.1 Thesis Contributions . . . . .	22
1.2 Thesis Structure . . . . .	23
<b>2 Background</b>	<b>25</b>
2.1 Application Programming Interfaces (APIs) . . . . .	25
2.2 API Documentation . . . . .	27
2.3 Code Examples in API Documentation . . . . .	29
2.3.1 Characteristics of Effective API Code Examples . . . . .	30
2.3.2 Coverage of API Code Examples . . . . .	31

2.4	API Code Example Generation Approaches . . . . .	33
2.5	Program Comprehension . . . . .	35
2.5.1	Attributes Influencing the Comprehension of Source Code . . . . .	37
2.5.2	API Code Example Comprehension . . . . .	38
2.6	Source Code Metrics . . . . .	38
2.6.1	Size . . . . .	39
2.6.2	Maintainability . . . . .	39
2.6.3	Complexity . . . . .	41
2.6.4	Comprehension . . . . .	42
2.6.5	Source Code Linearity . . . . .	42
2.7	Code Refactoring . . . . .	44
2.8	Domain-specific Languages (DSLs) . . . . .	46
2.9	Summary . . . . .	49
<b>3</b>	<b>Analysis and Hypothesis</b>	<b>51</b>
3.1	Analysis . . . . .	51
3.2	Research Objectives . . . . .	53
3.3	Research Questions . . . . .	54
3.4	Research Hypothesis . . . . .	56
3.5	Research Scope . . . . .	57
3.6	Summary . . . . .	60
<b>4</b>	<b>Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples</b>	<b>61</b>
4.1	Methodology . . . . .	62
4.1.1	Research Questions . . . . .	62
4.1.2	Study Design . . . . .	63
4.1.3	Independent Variables . . . . .	63
4.1.4	Dependent Variables . . . . .	65

## List of Contents

4.1.5	Participants . . . . .	65
4.1.6	Material . . . . .	67
4.1.7	Experiment Procedure . . . . .	73
4.1.8	Data Analysis . . . . .	73
4.2	Results and Discussion . . . . .	75
4.3	Threats to Validity . . . . .	79
4.3.1	Construct Validity . . . . .	79
4.3.2	Internal Validity . . . . .	82
4.3.3	External Validity . . . . .	82
4.4	Summary . . . . .	83
<b>5</b>	<b>Linear API Usage Example Synthesis</b>	<b>84</b>
5.1	Linear Code Synthesiser’s Features . . . . .	85
5.1.1	Code Analysis . . . . .	88
5.1.2	Code Transformation . . . . .	90
5.1.3	Code Processor and Generator . . . . .	91
5.2	Synthesiser’s Constraints . . . . .	96
5.3	Evaluation . . . . .	97
5.3.1	Data Collection . . . . .	97
5.3.2	Data Cleaning . . . . .	102
5.3.3	Data Analysis and Similarity Detection . . . . .	102
5.3.4	API Code Examples Rewriting . . . . .	103
5.3.5	API Code Examples Evaluation . . . . .	104
5.4	Results and Discussion . . . . .	104
5.5	Limitations and Observations . . . . .	110
5.5.1	Template-based Code Synthesis . . . . .	111
5.5.2	Interactive API Usage Example Generation . . . . .	112
5.6	Threats to Validity . . . . .	113
5.6.1	Construct Validity . . . . .	113
5.6.2	Internal Validity . . . . .	114

5.6.3	External Validity . . . . .	114
5.7	Summary . . . . .	114
<b>6</b>	<b>Coverage of API Code Examples</b>	<b>116</b>
6.1	APIExCov's Features . . . . .	117
6.1.1	Intended-API Description Language . . . . .	118
	The API Description Language ANTLR-based Custom Grammar . . . . .	124
6.1.2	Static Analysis and Parsing . . . . .	127
6.1.3	Intended-API Description Validation . . . . .	128
6.1.4	Coverage Analysis . . . . .	133
6.1.5	Textual Coverage Report . . . . .	138
6.1.6	Visual Coverage Report . . . . .	141
6.2	Constraints . . . . .	146
6.3	Evaluation . . . . .	146
6.4	Results and Discussion . . . . .	152
6.4.1	Qualitative Comparison of APIExCov and Code Coverage Tools . . . . .	155
6.5	Limitations and Observations . . . . .	157
6.6	Threats to Validity . . . . .	158
6.6.1	Construct Validity . . . . .	158
6.6.2	Internal Validity . . . . .	159
6.6.3	External Validity . . . . .	159
6.7	Summary . . . . .	160
<b>7</b>	<b>Conclusions and Future Work</b>	<b>162</b>
7.1	Summary . . . . .	163
7.2	Thesis Contributions . . . . .	163
7.3	Research Results . . . . .	164
7.4	Future Work . . . . .	168

List of Contents

<b>Appendices</b>	<b>171</b>
<b>Appendix A</b>	<b>172</b>
<b>Appendix B</b>	<b>177</b>
<b>Bibliography</b>	<b>202</b>



# List of Tables

4.1	API code examples and their variants and metric values. Variants sharing the same colour belong to the same treatment category. . . . .	64
4.2	Participant demographics. . . . .	67
4.3	The study results. Variants sharing the same colour belong to the same treatment category. The imbalance in the number of responses (N) between two variants of the same example is due to our exclusion of responses with inaccurately reported break times. . . . .	78
4.4	Study results for the linear API code examples. . . . .	79
4.5	Study results for the non-linear API code examples. . . . .	79
5.1	Statistics of the selected Java libraries and packages. . . . .	101
5.2	Summary of the evaluation results. . . . .	106
5.3	Number of API code examples with template-suitable code similarity patterns. . . . .	113
6.1	Usage information for two API elements, extracted from an API code example. . . . .	135
6.2	Overview of key statistics for the subject Java libraries, retrieved from GitHub on October 30, 2023 by Monce et al.[1] and December 9, 2024. . . . .	148

## List of Tables

6.3	Comparison of API element counts, character totals, and percentage decreases in full and shortened API descriptions for the evaluated Java libraries and frameworks. . . . .	154
-----	--	-----

# List of Figures

1.1	Workflow illustrating the main contributions of the thesis. . .	24
2.1	The architecture of Docio [2]. . . . .	34
2.2	Casdoc [3] document, with (1) block, (2) anchors to additional explanations, (3) pop-up annotations, (4) navigation aids, and (5) injected API documentation. . . . .	35
3.1	A concept map illustrating the thesis’s scope within the field of Software Engineering. . . . .	58
4.1	Study design (between-subjects) and the distribution of participants. . . . .	64
4.2	Example of a code comprehension task given to participants.	74
4.3	Example of a code reuse task given to participants. . . . .	74
4.4	The time spent on (a) comprehending, and (b) reusing the API code examples used in the study. Each boxplot represents the responses for one version (linear or non-linear) of a single example. . . . .	80
4.5	Participants’ subjective ratings of API code example comprehension (a) and reusability difficulty (b). . . . .	81
5.1	Linear code synthesiser architecture. . . . .	90
5.2	API code examples collection method. . . . .	98

## List of Figures

5.3	Distributions of the evaluated API code examples and the percentage decrease in their Lines of Code (LOC). Each boxplot aggregates the API examples of each Java library. . . . .	108
5.4	The extracted utility methods and their calls stacked by each Java library. Each stacked bar represents a single utility method and is labelled with its number of calls. . . . .	110
5.5	A scatter plot showing the relationship between the number of extracted utility methods and the number of API code examples containing an applicable code similarity. . . . .	111
6.1	Architecture of API code example coverage tool. . . . .	118
6.2	Class diagram of the Java classes representing API elements. These classes are used to create an in-memory representation of the API and its coverage details. . . . .	137
6.3	An example of a textual report showing a subset of API code example coverage information for the Jsoup Java library. . . .	139
6.4	Coverage filtering options in generated textual reports. . . .	141
6.5	Code city metaphor metrics for the API code example coverage visual report. . . . .	144
6.6	Hover information displayed over buildings in the visual API coverage report. . . . .	145
6.7	Hover information displayed over connecting lines in the visual API coverage report. . . . .	145
6.8	A visual report (code city) illustrating the API coverage for the Jsoup Java library. The coverage information is based on the API code examples available in the library's GitHub repository. . . . .	147
6.9	Workflow of the approach proposed by Monce et al. [1]. It illustrates the analysis of client sources to generate a syntactic usage footprint (SUF) of a library. . . . .	149

6.10	Comparison of API elements in full and shortened API descriptions for the evaluated Java libraries and frameworks. Percentages represent the proportion of shortened descriptions relative to full descriptions for each Java project. . . . .	154
6.11	Comparison of API element character counts in full and shortened API descriptions for the evaluated Java libraries and frameworks. Percentages indicate the proportion of shortened descriptions relative to full descriptions for each Java project.	155

# List of Listings

2.1	A semi-linear API code example showing one use of the Chronology API. Programmers are required to jump once when following the execution flow. . . . .	43
4.1	A linear code example using the Joda-Time API (DateExample.java). . . . .	68
4.2	A non-linear code example using the Joda-Time API (DateExample.java). . . . .	69
4.3	A linear code example using the Joda-Time API (ChronologyExample.java). . . . .	70
5.1	Vonage API code example (1) - SendDtmfToCall.java . . . . .	85
5.2	Vonage API code example (2) - SendTalkToCall.java . . . . .	85
5.3	Utility method - Vonage API. . . . .	86
5.4	Documentation method - Vonage API - SendDtmfToCall.java . . . . .	88
5.5	Jackson API code example (1) - DisableDateAsTimestamps.java . . . . .	93
5.6	Jackson API code example (2) - DisableFailOnEmptyBeans.java . . . . .	94
5.7	Utility method - Jackson API. . . . .	94
5.8	Rewritten Jackson API code example (1) - DisableDateAsTimestamps.java . . . . .	95
5.9	Rewritten Jackson API code example (2) - DisableFailOnEmptyBeans.java . . . . .	96
5.10	PDFBox code example (1) - FieldTriggers.java. . . . .	112
5.11	PDFBox code example (2) - FieldTriggers.java. . . . .	112

## List of Listings

6.1	A subset of an API description that defines the API of the Jsoup Java library. . . . .	122
6.2	ANTLR-based custom grammar for the intended API description language. . . . .	125
6.3	Java method for processing and validating method signatures extracted from the intended API description YAML file. . . .	129
6.4	An example of an API description with errors. This description defines a subset of the API of the Apache Commons CLI Java library. . . . .	132
6.5	Examples of the API description language validation error messages. . . . .	132
6.6	API Code Example from the Jsoup Java Library (ListLinks.java).	135
6.7	A subset of the full API description for the Jsoup Java library.	151
6.8	A subset of the shortened API description for the Jsoup Java library. . . . .	152
B.1	Full intended API description of the Jsoup Java library. . . .	177
B.2	Shortened intended API description of the Jsoup Java library.	194

# List of Algorithms

1	Algorithm for linear code synthesis . . . . .	89
2	Algorithm for measuring and reporting API code example coverage . . . . .	119



To my Mum – my first teacher, my role model, and the guiding light of my  
life.

## **Acknowledgements**

First and foremost, I would like to begin by expressing my deepest gratitude to Almighty God, whose blessings and guidance have enabled me to undertake this PhD journey. I would also like to thank my supervisor, Professor Dimitris Kolovos, for his invaluable support, thoughtful guidance, constructive feedback, and for always being approachable. I am deeply grateful to have worked with such an extraordinary supervisor and researcher, from whom I have learned so much, not only academically but also personally. My thanks also go to Dr Simos Gerasimou, my assessor, for his valuable insights and fruitful discussions during our TAP meetings.

A big thanks to all the ASE research group members at the University of York for the wonderful time we shared and for their feedback during the seminars. I appreciate and thank Dr Sondess Missaoui for motivating me during challenging times and for our delightful conversations over coffee.

A special thank you to my husband, Abdulmajeed, for leaving everything behind and sharing this journey with me. Thank you for always cheering me up. Thank you for being the calm in the storm of my PhD. I am also grateful to my son, Maher, the love of my life, for all the laughter and joyful moments that nourished me. I am truly sorry for all the playtimes I missed.

I am thankful to my family and friends for their prayers, motivation, and support in every aspect of my life. Thanks to my Mum for our long Sunday calls, which have always been the fuel that kept me going throughout the week. A special thanks to my brothers, Dr Meshari Alharbi and Dr Basil Alharbi, for their unforgettable care and kindness during the health difficulties I faced in my final year. Your continuous check-ins and encouragement were the wind beneath my wings, helping me complete this work.

Finally, I thank Qassim University for funding my PhD studies and my beloved country, Saudi Arabia, for this enriching opportunity.

## Author Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Parts of this thesis have been previously published by the author. The following research papers have been primarily written by the PhD candidate.

[1] S. Alharbi and D. Kolovos, “Exploring the Impact of Source Code Linearity on the Programmers Comprehension of API Code Examples”, In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2024, pp. 236-240.

[2] S. Alharbi, D. Kolovos, and N. Matragkas, “Towards Generating Maintainable and Comprehensible API Code Examples”, In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 830-834.

[3] S. Alharbi, D. Kolovos, and N. Matragkas, “Synthesising Linear API Usage Examples for API Documentation”, In *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 607–611.

# Chapter 1

## Introduction

An Application Programming Interface (API) serves as an abstraction layer that enables developers to access the core components of a library or framework to perform development tasks while hiding internal complexity [4]. Although the use of APIs can improve the quality and efficiency of software systems, their size and complexity can make them difficult to learn and use even for expert programmers [5, 6]. For instance, the Java Platform, Standard Edition API Specification contains over 4,000 classes and more than 35,000 methods, and Microsoft's .NET Framework provides approximately 140,000 classes, methods, properties, and fields [7]. In addition to the size and complexity of APIs, the effectiveness of using an API depends significantly on the availability of adequate resources for learning it; thus, APIs come with documentation. API documentation can take many forms, such as API code examples, user guides, tutorials and API reference documentation [8]. However, as API documentation is a product in itself and thus requires effort to develop and maintain, its quality cannot be taken for granted [9, 10]. Recent work on best practices for API code examples [11] emphasises the effort put into creating API documentation and the importance of its up-to-dateness and maintenance.

Several studies on the quality of API documentation have reported a

number of issues such as incompleteness, ambiguity, and the need for more complex usage examples [9, 5, 12, 4]. In particular, the shortage of code examples in API documentation has been shown to be a major obstacle commonly faced by users when learning a new API [4]. This is because API code examples play important roles in a variety of learning tasks, and most API users consider working through code examples as the first step to take [8, 11].

When official API documentation lacks examples, API users often spend considerable time searching for code snippets on the web. They typically turn to sources such as code search engines or AI-powered tools to get assistance. While these sources can be useful, they also pose risks, as the generated code may introduce API usage violations, contain incorrect patterns, or fail to follow best practices for API usage. For instance, one of the popular online sites that users usually extract code examples from is StackOverflow, which in turn could be a source of buggy and outdated code snippets [13]. Such risks can negatively impact the quality and maintainability of software.

Although the reason behind the shortage of API usage examples in official API documentation is not entirely clear, it has been shown that API developers have remarkably little tool support that could help them properly document their APIs [14, 2]. The lack of such support could be a reason for the shortage of code examples in API documentation. Another reason is possibly the repetitive and demanding process of writing effective API code examples [15, 16]. Also, maintaining and updating those code examples and matching the actual use of APIs are considered significant challenges for API developers [15].

To mitigate the issue of the shortage of code examples, several systems have been proposed for mining different types of resources, such as online sites and test code, and extracting API usage examples [17, 18]. However, to the best of our knowledge, little attention has been given by previous research

to the root causes of the shortage of code examples in API documentation. Similarly, only a few approaches have directly addressed the challenges that make the task of writing and maintaining code examples unappealing for API developers [19, 20]. In addition, existing research does not seem to have fully explored the impact of several source code structures on users' comprehension in the context of API code examples. Also, it is not entirely clear how much of an API should be covered in usage examples, nor how many examples are enough.

This thesis contributes to addressing the problem of the shortage of code examples in API documentation by investigating its underlying causes and understanding the barriers that prevent API developers from producing more code examples for their API documentation. It also explores the API users' expectations and preferences regarding various structures of these API usage examples. In short, the main goal of this thesis is to support API developers with tools and techniques for writing and maintaining API code examples that meet the API users' preferences and needs. This goal is achieved by enabling API developers to reduce the repetition involved in documenting APIs by providing them with suitable techniques for creating API code examples that comprehensively cover their intended API. In turn, this would minimise the effort required to create and maintain the API documentation materials that meet the users' expectations.

### 1.1 Thesis Contributions

The main contributions of this thesis are outlined below and illustrated in Figure 1.1, which presents them as a workflow.

- **A controlled code comprehension experiment** designed to examine how various structural characteristics of source code; specifically, its level of linearity and length, affect API users' comprehension of

API code examples. The aim of this user experiment was to provide API developers with insights into the effective structuring of API code examples.

- **A linear API code example synthesis approach** tailored for the Java programming language. This approach aimed at alleviating the effort required by API developers to write lengthy, repetitive, and costly-to-maintain API code examples. This, in turn, enables them to produce more code examples, thus, enhancing API learnability for API users.
- **A domain-specific language for intended API specification**, which enables API developers to formally and precisely define the source code elements, such as classes and methods, that comprise their intended APIs. This DSL offers a range of features designed to facilitate the process of describing an API, making it more efficient.
- **A coverage tool for API code examples** built on top of the proposed API description DSL. This API code example coverage tool measures the extent to which a given set of code examples covers an intended API. Coverage reports are generated in two formats, textual and visual.

## 1.2 Thesis Structure

The remainder of this thesis is structured as follows. **Chapter 2** presents a comprehensive overview of the background on the main topics related to this PhD research, including the role of APIs and API documentation in the field of software engineering. It also discusses the current state-of-the-art API code example generation approaches and highlights challenges in creating API documentation.

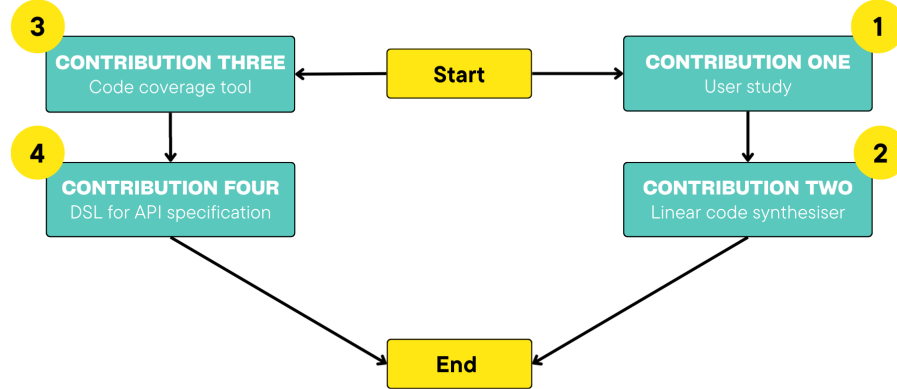


Figure 1.1: Workflow illustrating the main contributions of the thesis.

**Chapter 3** provides an analysis of the main research problem and states the objectives, questions, hypotheses, and scope of this PhD research.

**Chapter 4** presents novel results from a controlled user experiment conducted to examine the impact of various levels of source code linearity and other source code structures on the comprehensibility and reusability of API code examples.

**Chapter 5** presents the architecture, features, implementation, and evaluation of a linear code synthesiser; a novel approach for transforming non-linear, less repetitive API code examples into linear, easy-to-comprehend ones.

**Chapter 6** presents a novel API description language for specifying the intended surface of an API and a tool for reporting the coverage of API code examples based on the specified API elements. This chapter also includes an evaluation of the proposed tool and a discussion of its limitations.

**Chapter 7** summarises the thesis contributions and highlights its notable results. It also proposes possible future directions and discusses areas that could benefit from further investigation.



## Chapter 2

# Background

This chapter provides some definitions and background material for the research context. It also provides a review of relevant literature and related studies that have been conducted to solve similar problems. The chapter is structured as follows: Section 2.1 describes the background required to understand the concept of Application Programming Interfaces (APIs). Section 2.2 provides an explanation of API documentation and its purpose. Section 2.3 presents the roles and characteristics of API code examples. Section 2.4 outlines the existing approaches that have been proposed to help document APIs with code examples. Section 2.5 explains the concept of program comprehension. Finally, Sections 2.6, 2.7, and 2.8 describe some key concepts that contribute to the implementation and evaluation of this PhD research.

### 2.1 Application Programming Interfaces (APIs)

Reuse-based software development is a major concept in the field of software engineering. Reuse is fundamental in many software projects because it helps to effectively and accurately build desired functionalities, facilitate programming tasks, and save developers' time. Software reuse can take many forms, such as copying and pasting code, or using external software libraries or

## Chapter 2. Background

frameworks by integrating their application programming interfaces (APIs) [21, 22]. An API is an interface that provides high-level abstractions and allows developers to use its internal functionality to implement some software development tasks [4]. For example, the `java.util.List` interface in the Java standard library exposes methods like `add()`, `remove()`, and `get()` to manipulate collections. Furthermore, APIs have been used for a long time, most likely since the advent of computers, and they were initially used to exchange data between two or more programs [23].

There are many benefits of using APIs in software development. In addition to their easy and predictable use, APIs increase the productivity of software developers and save time by allowing them to perform a particular task by only writing client code instead of creating the entire required functionality from scratch [24, 6]. However, in practice, correctly using and accessing APIs requires understanding and following their usage patterns and rules [21]. Ignoring such rules can be the reason behind many software crashes, bugs, flaws or even significant security problems [25, 7]. In addition, some APIs can grow extremely large, causing some complications and challenges to software developers when trying to use them.

According to Stylos et al. [26], developers occasionally find APIs tedious and difficult to learn, and as a result, they often spend significant time trying to properly apply them. A multi-phased study that was conducted on over 440 professional Microsoft developers to analyse the obstacles they faced when learning new APIs has shown that some of the serious obstacles are related to poor API documentation [5]. In addition, some API documentation is incomplete, inconveniently organised, and lacking high-quality and complex usage examples or a consistent visual design that can ease the process of the navigation [27, 28]. As a result, software developers often prefer to use informal learning resources, such as community forums like StackOverflow, although they contain some incorrect API uses [29]. Furthermore, the

## 2.2. API Documentation

misleading naming convention of some API methods and classes can place a barrier in terms of the correct use of APIs. For example, the correct way to create a new thread in Java is to call a `Thread.start()` method, however, developers often misuse this approach by calling a `Thread.run()` method, which should not be used to create a new thread [30].

All of the above-mentioned API learning obstacles can impact the usability of APIs and increase the chance of incorrect use, which, in turn, could be mitigated by providing API users with rich API documentation and effective API code examples. The following sections shed more light on this topic and discuss some existing techniques and approaches proposed to improve API learnability.

## 2.2 API Documentation

As with any other complex technical tool, the effective and correct use of an API relies significantly on the appropriate support for learning and using it, and in order to fulfil this goal, APIs come with documentation. This documentation acts as a reference guide for any developer who wants to use the API. It gives developers guidance on how to use it, what to expect from using it, and any other assistance they may need [9, 31]. It also provides some details on the methods, classes, return types, arguments, and more, sometimes supported by concise tutorials and usage examples. However, an API document is a product in itself, and as such its quality cannot always be guaranteed. In light of this, it could be said that high-quality documentation can help to increase the efficiency of a developer's work, and their interest in using an API, while also reducing the chances of misusing it. By contrast, APIs with poor-quality documentation can affect the quality of the software built on them and are usually abandoned because of the difficulty in understanding them [9].

High-quality API documentation helps prevent common mistakes users

## Chapter 2. Background

make when using APIs. However, for API documentation to succeed in this way, they should meet the users' expectations by providing the information they need to easily, correctly, and effectively use the API [8, 9]. This can be achieved by ensuring that the API document contains an explanation of all the assumptions and constraints of using the API. A good illustration of this is the documentation of the Java standard library API and the useful examples, usage scenarios, and contexts that it provides [31].

There are a number of methods and tools used to automate the process of generating API documentation and make their future updates and maintenance much easier. For instance, in Java, the Javadoc<sup>1</sup> tool is used as a document-generator tool for producing API documentation. It parses the declaration and special documentation comments found in Java source files and produces a set of HTML pages that describe all the classes, interfaces, constructors, fields, and methods [32, 33]. Similarly, Sphinx,<sup>2</sup> which is used by the Python community, generates documentation for the APIs written in Python by converting the reStructuredText sources into HTML, PDF, and many other formats [34].

Since API documentation plays an important role in understanding and correctly using APIs, several approaches have been proposed to improve the quality of this documentation, Dekel et al. [35], for instance, have proposed eMoose, an Eclipse plug-in that allows developers to be aware of any directives associated with the targets of the method invocations that appear in their code. It highlights these method calls by surrounding them with a box and marking their lines with an icon. This plug-in is useful for the developers because the rules or caveats of APIs are usually lost within the lengthy documentation. Similarly, Treude and Robillard [36] have proposed SISE, a machine-learning approach that enhances the API documentation by augmenting them with useful information extracted from StackOverflow. This

---

<sup>1</sup><https://www.oracle.com/java/technologies/javase/javadoc-tool.html>

<sup>2</sup><https://www.sphinx-doc.org/en/master/>

## 2.3. Code Examples in API Documentation

information is not found in the API documentation and is highly related to the targeted API.

## 2.3 Code Examples in API Documentation

According to Robillard and DeLine [5], listings of varying lengths that demonstrate the use of an API are called API usage code examples. Furthermore, it is evident that API code examples are an essential learning resource. The results of the survey conducted by Robillard [4] have revealed that around 55% of the respondents prefer turning to code examples when learning a new API. Generally, API code examples can be categorised into four main groups: *small code snippets*, which are often written to demonstrate a certain feature of the API; *tutorials*, which are a series of combined and short code examples that explain the implementation of a small but complex API functionality; *sample applications* that are usually small in size but complete and independent; finally, the *production code* of the API itself, which can also be used to illustrate an API usage [5].

It has been demonstrated that API code examples are an essential and popular learning resource, as they can be more informative and easier to understand than the descriptive text found in documentation [8]. This stems from the possibility that text in documentation could be outdated or inaccurate compared to code [37]. This can also be associated with the fact that code examples are usually easy to adjust when assembling code, thus serving the programming needs of API users [38]. In addition, most API users consider code examples as the first step to take when learning a new API. On average, around 20% of their time is spent online searching for usage examples [39].

### 2.3.1 Characteristics of Effective API Code Examples

For API code examples to be effective, they need to meet certain characteristics that align with the users' expectations. Based on the qualitative analysis conducted by Nasehi et al. [40], users often prefer concise code examples that do not contain irrelevant details or code that is easy to figure out. This is because most concise examples where the unnecessary parts are left as either comments or ellipses tend to have a better structure, which makes them a good skeleton for a solution. In addition, highlighting the key code elements in the usage examples or hyperlinking them to other useful resources, such as the Javadoc of a certain API element, seems to be desired by the users. However, beginners usually prefer code examples that use a well-known domain and are divided into small parts where each part demonstrates a certain detail. Additionally, it is preferable for code examples to provide information about the possible alternative solutions that use a different class, API, or API version. Meng et al. [41] suggested that API code examples should clearly demonstrate the intended use of the API. Also, API users generally prefer complete code examples as they are more convenient to work with; for instance, via copy and paste.

Another way of exploring the characteristics of effective code examples is to study the extraction metrics that are used by the existing API code search approaches and websites, such as ProgramCreek,<sup>3</sup> and Tabnine,<sup>4</sup> which automatically mine, rank, and recommend API usage examples for API users. For example, in APISonar [42], some of the metrics that have been used to assess the relevance and usefulness of the extracted code examples are code readability, which refers to how easily the code is to be read and understood by users, while, code reusability assesses the potential for reusing the functionality provided by a specific piece of code.

---

<sup>3</sup><https://www.programcreek.com>

<sup>4</sup><https://www.tabnine.com>

## 2.3. Code Examples in API Documentation

To the best of our knowledge, very little work has been done to develop a set of quality metrics that are made specifically for API code examples and that API developers could refer to during the process of creating their API documentation. Moreover, it could be argued that the lack of such metrics could be one of the reasons behind the shortage of usage examples in API documentation, since the API developers do not have enough information on what is considered a good example, or on how many examples are generally enough. However, Radevski et al. [43] have worked on investigating the quality of several existing and publicly available API usage examples and have given an initial knowledge base for developing an automated API usage example metrics analysis tool. Furthermore, this work has suggested some plausible metrics for code examples, such as example coverage, age, usefulness, and repeated calls to the same methods.

### 2.3.2 Coverage of API Code Examples

Several approaches and tools for Java and many other programming languages<sup>5,6</sup> have been proposed to report the degree of code coverage and to measure how much of the source code has been tested. These tools offer a variety of types and levels of coverage, such as function coverage and statement coverage, and can integrate seamlessly with many IDEs (Integrated Development Environments). However, to the extent of our knowledge, little progress has been made in developing tools that specifically measure how much of an API is covered by a set of examples.

An example of research in this area is the work proposed by Radevski et al. [43] to investigate the API code examples coverage by examining a set of usage examples provided by seven popular API libraries for Java. The examples were first turned into unit tests and then run using a test code coverage tool, i.e. Atlassian Clover tool. The results show that the average coverage of

---

<sup>5</sup><https://www.jacoco.org/jacoco/>

<sup>6</sup><https://www.atlassian.com/software/clover>

## Chapter 2. Background

the analysed examples is only 30%. Moreover, the large number of examples provided by a certain library does not necessarily indicate good coverage of the API, since code examples can include a lot of repetition and many API functionalities can be left out. In addition, Radevski et al. [43] highlight that unit test coverage tools might not be the ideal technique for retrieving coverage of code examples. It could be said that test cases are designed to systematically validate program behaviour against expected outputs, whereas code examples are illustrative and intended to demonstrate usage patterns, typical workflows, or best practices for API users. Furthermore, usage examples do not define assertions, inputs, outputs, or error conditions, which are essential for applying conventional testing metrics. Therefore, API example coverage is best assessed differently, with a focus on the representativeness of API usage rather than the correctness of functionality.

In addition, Monce et al. [1, 44] proposed a conceptual framework consisting of two syntactic models that aim to help API developers and maintainers understand the boundaries of their APIs and the real-world uses of these APIs in client code. These models were implemented in a tool called UCov (Usage COverage) which utilises static analysis to extract them from Java source code. One of these models is called Syntactic Usage Footprint (SUF), which as its name indicates, reports the coverage and all the actual uses of all the API elements that are found in client code including API code examples. These reported API elements are given and defined via another model called Syntactic Usage Model (SUM) which consists not of the intended API uses as envisioned by the API developers but rather all the legal, programmatically possible, and allowed uses of the API.

While the optimal number of API code examples and the ideal percentage of API coverage they should provide is still not clear and further research on this topic is required, it is undoubted that code examples are crucial learning material and any inadequateness of coverage or shortage in them can place



## 2.4. API Code Example Generation Approaches

a significant obstacle for API users [4].

## 2.4 API Code Example Generation Approaches

As mentioned in Section 2.3, API code examples are a key resource for reusing and learning APIs, since they can save the users' time and increase productivity. However, the practice of explicitly documenting APIs with effective and working code examples is not ubiquitous in software development [45, 46, 47], and not all software libraries and frameworks provide adequate code examples in their official documentation [18, 48]. Consequently, there is a significant shortage of effective API usage examples in API documentation, which forces API users to seek other online resources like StackOverflow to learn new APIs. Although such websites can be very valuable for both API users and researchers, the code snippets available on them can be toxic, outdated, and contain some vulnerabilities and violations of software licenses [13, 49].

To compensate for the shortage of API code examples in API documentation, several approaches have been proposed to generate them. Furthermore, a recent systematic mapping study on API documentation generation approaches [14] has found that the most popular approaches are those that generate API usage examples or templates for using APIs. For example, Buse and Weimer [15] have proposed an approach that automatically mines and synthesises well-typed representative API usage examples. This approach incorporates techniques such as data-flow analysis and clustering to extract and discover the necessary details for constructing usage examples before using them as API documentation. In addition, to supplement formal API documentation, other approaches benefit from publicly available API unit tests [18, 19, 50]. Using unit tests as API usage examples has been shown to be efficient for supplementing formal API documentation, particularly for newly released APIs that lack the availability of client code.

## Chapter 2. Background

Some success has been achieved in filling in the gap between reference and example-based API documentation. This was achieved by using various API code examples found on various online sites, such as Stack Overflow and GitHub Gists, and embedding them into formal API documentation [17, 51]. These approaches typically employ various mining, clustering, and ranking techniques to extract API code examples that best satisfy the intent of the user when browsing API documentation.

Other approaches have focused on helping API users construct their code. For example, Jadeite [52] allowed API users to share their aggregate experience and collaboratively add placeholders to API documentation to indicate the expected classes and methods. In addition, Docio [2] is a system that helps API users understand the actual/dynamic input and output values of API functions. It also helps API developers who use the C programming language to write and add API examples into API documentation with actual and concrete input and output values. The architecture of Docio is shown in Figure 2.1.

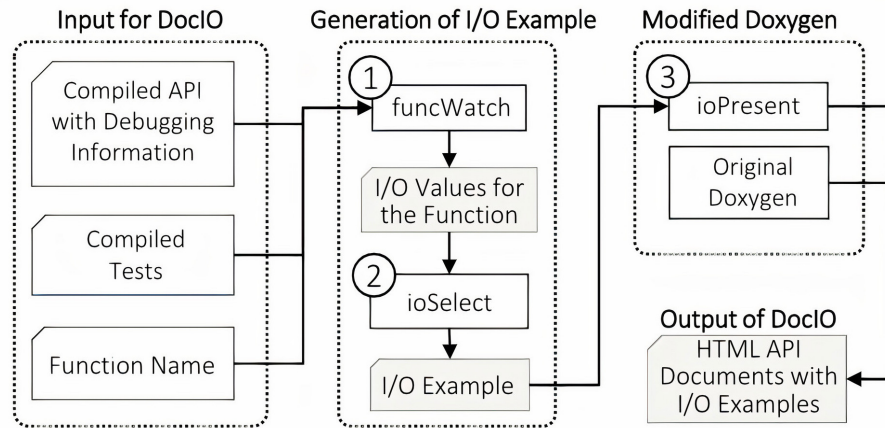


Figure 2.1: The architecture of Docio [2].

Recommending source code examples by submitting queries against API calls was also proposed to help both API developers [53] and users [54, 55].

## 2.5. Program Comprehension

In addition, some work has focused on improving the format in which code examples are viewed and read by API users [3]. This was achieved by nesting textual explanations linked to specific code elements within the usage example. The rationale behind the implementation of this documentation format (Casdoc) is that the plain text explanations that usually accompany the code examples are as necessary to API users as the code itself. However, writing explanations that are brief but comprehensive is a tricky balancing act for API developers. An example of a code snippet generated using this documentation format is shown in Figure 2.2.

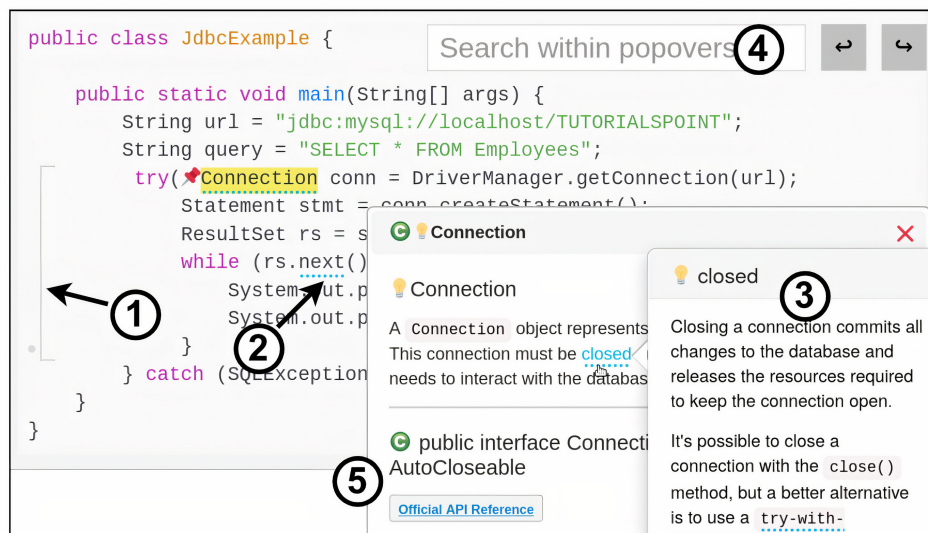


Figure 2.2: Casdoc [3] document, with (1) block, (2) anchors to additional explanations, (3) pop-up annotations, (4) navigation aids, and (5) injected API documentation.

## 2.5 Program Comprehension

Code comprehension can be defined as the activity through which software engineers understand how a software system or a part of it works [37]. This activity involves developing mental models of the software's structure, func-

## Chapter 2. Background

tionality, behaviour, rationale, and construction [56]. Furthermore, the process of comprehending code is highly impacted by several attributes of the code itself, such as its complexity and readability. Consequently, the majority of existing IDEs incorporate a large number of functionalities to facilitate code comprehension including references finder, and multiple files view [57].

Another approach that is commonly used to simplify comprehension of source code is employing visualisation techniques. Several studies have demonstrated that visualisation approaches, especially in large software systems, can indeed help developers in many software engineering activities and in understanding, exploring, and maintaining source code generally [58, 59, 60]. This is true since code visualisation gives great levels of abstraction, viewpoints, and filters that can facilitate the comprehension of complex software architecture. In addition to understanding software architecture, visualising code can be done for many other purposes, such as seeking knowledge about software evolution and behaviour.

Visualisation techniques can come in many forms; however, the most predominant ones are those that are either hierarchical or graph-based [60]. Another common type of visualisation that is used to represent software artefacts in a non-abstract way is the code city metaphor [61, 62]. This visualisation technique is a 3D representation of software components in a city-like metaphor, where classes are shown as buildings, packages as districts, and other software metrics, such as size and number of methods, are represented by the colour and height of the buildings. Unlike other techniques, the code city metaphor provides developers with a visual representation that is easier to understand since it uses a familiar concept and is more navigable [61, 63, 64].

### 2.5.1 Attributes Influencing the Comprehension of Source Code

A large number of studies investigated the impact of various source code characteristics on programmers' comprehension of code. For instance, one of these characteristics is the use of intermediate variables which are proven to help comprehend code, especially in cases where they are given meaningful names [65]. Moreover, certain syntactic structures such as `for` loops, particularly those counting down or with unusual bounds, are considerably more difficult to understand than `ifs` [66]. Also, different identifier names [67] as well as naming conventions [68, 69], length [70, 71], and style can improve code comprehension. For example, using single letters or abbreviations for identifier names is likely to make the code harder to comprehend compared to using full short words [72, 70, 71]. Also, developers, particularly novices, tend to prefer and better understand code that uses the camelCase identifier style, i.e. `firstName`, compared to the snake-case style, i.e. `first_name` [69]. Some research has found that indentation and style can significantly influence the comprehensibility of source code. According to Miara et al. [73], an indentation of two to four spaces is considered optimal for enhancing code comprehensibility, while six or more spaces may potentially reduce readability.

Not only can the content and structure of source code influence its understandability, but some studies have also explored the effects of more global factors, such as the order of methods and code regularity. For instance, one study showed that a class that conforms to the CLCT (Calling+Connectivity) method ordering strategy, i.e. related methods are grouped together and within each group, invoking methods come before invoked methods, tends to require less time to read and comprehend [74]. Similarly, research has found that code regularity; characterised by the repeated use of the same structures, reduces code complexity, making it easier to under-

stand, even if the code is lengthy or has a higher MCC (McCabe’s Cyclomatic Complexity) [75]. Lastly, source code linearity,<sup>7</sup> which refers to the use of interdependent methods or classes that force programmers to regularly jump to their definitions, has a significant impact on the programmers’ reading order and comprehension of code [76]. Also, this study showed that novice programmers tend to read code more linearly; as natural text, whereas experts prefer to follow its execution order.

### 2.5.2 API Code Example Comprehension

There are no comprehension factors that are specific to the source code of API code examples. All the attributes discussed in Section 2.5.1 and Section 2.3.1 also apply to API code examples. All in all, to boost the comprehensibility of these examples, they should be accompanied by well-structured and comprehensive documentation. Also, effective API learning requires adding some precise explanations of the example’s purpose, constraints, anticipated results, and potential errors.

## 2.6 Source Code Metrics

*“You Cannot Manage What You Cannot Measure.”*

- Peter Drucker

One of the aims of this PhD project is to ensure that the API code examples that are generated from the proposed approaches are of good quality. However, as mentioned in Section 2.3.1, there are no existing metrics concerning the quality of API code examples, neither those written manually by API developers nor generated automatically by tools [43]. As such, it is essential to shed some light on the standard source code metrics, their measurement, and

---

<sup>7</sup>Peitek et al. [76] define source code linearity as how closely the reading order of code matches its written, top-to-bottom order, without many jumps between different parts of the code. Further details are provided in Section 2.6.5.

## 2.6. Source Code Metrics

their role in assessing several aspects of software quality, since such metrics are used in the evaluation phase of this project. In addition, the selection of the following quality attributes is based on the code characteristics we target in this research, which we believe are of principal concern for both API developers (e.g. source code complexity and maintainability) and API users (e.g. source code size and comprehension).

### 2.6.1 Size

Program-size oriented metrics are one of the static code metrics and are often used to measure certain aspects of software development, such as code complexity, code maintainability, software reliability, and developer productivity and effort [77]. For example, line of code (LOC), which can also be referred to as source lines of code (SLOC), is a software metric that is used to measure the size of a software program by calculating the number of lines in the program's source code. There are two main types of the SLOC metric: physical SLOC, which is commonly defined as the count of all lines in source code excluding comments and blanks, and logical SLOC, which only computes the lines of the executed statements [78]. Therefore, physical SLOC tends to be more sensitive to various formatting and style conventions.

### 2.6.2 Maintainability

Software maintenance is a broad activity that often refers to the processes of modifying, optimising, correcting, or updating a software product after it has been launched [79]. On the other hand, software maintainability refers to the ease with which a software product can be maintained [77]. Moreover, software engineers have long considered software maintenance to be a challenging task [80], and maintenance expenses usually go beyond the initial development costs [81]. As such, several code metrics and indicators have been used to estimate and predict future software maintenance efforts

## Chapter 2. Background

and costs. For example, the change-proneness and bug-proneness of various source code components, such as classes and methods, can indicate their future cost of maintenance [82]. However, the use of these indicators has been subject to debate, since they are not usually obtainable until after the software system is released. As a result, other static source-code metrics like program size (Section 2.6.1) and McCabe Complexity (Section 2.6.3) have been used to estimate software maintainability.

It is crucial to understand that measuring software maintainability is not a trivial task, as software engineers cannot always rely on a single metric to evaluate whether a piece of software is more maintainable than another. This is because maintainable software has been recognised to have a combination of several characteristics including modifiability, understandability, and testability [83]. Therefore, the well-known maintainability metric (Maintainability Index), proposed by Oman and Hagemeister in 1994 [84], reuses other existing source-code metrics, namely Halstead Volume, Cyclomatic Complexity, SLOC, and Comments Ratio, to calculate the overall maintainability of the software. The Maintainability Index (MI) is calculated using the following formula:

$$MI = \max[0, 100 \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L + 50 \sin \sqrt{(2.4C)}}{171}] \quad (2.1)$$

Where  $V$  is the average Halstead Volume per module,  $G$  is the average Cyclomatic Complexity per module,  $L$  is the average number of Source Lines of Code (SLOC) per module, and  $C$  is the average number of comment lines per module [84].



### 2.6.3 Complexity

Several code complexity metrics have been proposed in the field of software engineering to predict certain aspects like software quality, defects, and understandability. Moreover, the effectiveness of a software project is significantly impacted by the complexity of its code. As such, measuring it can bring many significant benefits including reducing the risk of introducing more bugs, and lowering maintenance costs. One of the well-known metrics for calculating code complexity is Cyclomatic Complexity [85], which is a quantitative measure of linear independent paths in the source code. The fewer and less complex the paths in the code, the lower and better the Cyclomatic Complexity. This metric was proposed by Thomas J. McCabe in 1976 and is computed using the Control Flow Graph of the program. Once the graph is produced, its Cyclomatic Complexity is calculated using the following formula, where  $E$  is the total number of edges, and  $N$  is the total number of nodes:

$$V(G) = E - N + 2 \quad (2.2)$$

Another complexity metric introduced by Thomas J. McCabe is Essential Complexity [85], which measures the structure and quality of the code by calculating three values: the number of entry points, termination points, and nondeductible nodes [86]. The program is considered well-structured if the value of its Essential Complexity is closer to 1. Furthermore, this metric is often used to estimate the effort required for software maintenance.

One of the well-known metrics for measuring code complexity was developed by Maurice Halstead [87]. This metric works by breaking down the source code into a sequence of tokens, which are later identified as operators or operands. The number of these operators or operands is primarily used to calculate several measures and aspects of the code complexity, such as

effort, difficulty, vocabulary, and volume [88].

#### 2.6.4 Comprehension

There is a large body of work on finding and analysing the correlations between existing code complexity metrics and code understandability [89, 90]. As a result, new control-flow complexity metrics like Cognitive Complexity were introduced to provide fairer relative evaluations of code comprehension, and to remedy the shortcomings of Cyclomatic Complexity in measuring code understandability [91]. In addition, Beyer and Fararooy [92] have developed a complexity measure called DepDegree, which considers the low-level data flow of the program by calculating the number of dependency edges in its UseDef graph. The value resulting from this DepDegree measure can be used as an indicator for code readability and comprehension.

#### 2.6.5 Source Code Linearity

As presented in Section 2.5.1, several source code factors can impact the way programmers read and comprehend source code. One of these factors is the linearity of the structure of the source code itself and whether the developers are likely to read it more sequentially and linearly as a natural text or not. According to Peitek et al. [76], source code linearity ( $i$ ) can be defined using the following metric that measures the order in which source code is executed:

$$i = \frac{\Delta}{\bar{\Lambda}}, \quad \text{where} \quad \Delta = \sum_{i=1}^{|M|} \delta_{m_i} \quad \text{and} \quad \bar{\Lambda} = \frac{1}{|M|} \sum_{i=1}^{|M|} \lambda_{m_i}, \quad \text{for } m_i \in M \quad (2.3)$$

With

$$\begin{aligned}
 \lambda_m &:= \text{length of a method} \\
 M &:= \{m \mid m \in P, m \text{ is a method}\} \\
 \delta_m &:= |C_m - D_m| \\
 D_m &:= l_e \quad \text{for } e := \text{Declaration}(m) \\
 C_m &:= l_e \quad \text{for } e := \text{Call}(m) \\
 l_e &:= \text{index}(e) \quad \text{for } e \in P \\
 P &:= \Omega_{\text{Program}}
 \end{aligned}$$

This formula shows that the linearity  $i$  of any piece of code is the relationship between the distances of jumps  $\Delta$  and the average length of methods  $\bar{\lambda}$ . Furthermore, a method's  $m$  jump  $\delta_{m_i}$  is defined by calculating the distance between the method call ( $C_m$ ) and the method declaration ( $D_m$ )[76]. To illustrate this, let us consider the code snippet in Listing 2.1 which shows a use of the Chronology API<sup>8</sup> of Joda-Time Java library.

```

1  public class ChronologyExample {
2      public static String getDateWithChronology(DateTime date,
3          Chronology chronology) {
4          DateTime dt = date.withChronology(chronology);
5          int year = dt.getYear();
6          int month = dt.getMonthOfYear();
7          int day = dt.getDayOfMonth();
8          return year + "-" + month + "-" + day;
9      }
10
11     public static void main(String[] args) {
12         DateTime currentDate = new DateTime();
13         System.out.print("Date in Buddhist chronology: ");
14         System.out.println(getDateWithChronology(currentDate,
15             BuddhistChronology.getInstance()));
16     }

```

---

<sup>8</sup><https://www.joda.org/joda-time/apidocs/org/joda/time/Chronology.html>

Listing 2.1: A semi-linear API code example showing one use of the Chronology API. Programmers are required to jump once when following the execution flow.

This API code example is mostly readable in a more linear way as developers are expected to follow its execution flow starting from line 10 and then jumping up only once from line 13 to line 2 where the method `getDateWithChronology()` is declared. As this piece of code contains only two methods, one is 7 lines and one is 5 lines long, its average method length  $\bar{\Lambda}$  is 6. Moreover, the calculated distance of jumps  $\Delta$  is 11 since the code example involves only one jump from a method call to its declaration. Therefore, the result of dividing this jump distance  $\Delta$  by the average of method lengths  $\bar{\Lambda}$  is  $i=1.83$ , which indicates that this piece of code is relatively linear since its calculated linearity value is fairly lower.

Furthermore, in line with Peitek et al. [76], linearity scores are best understood on a relative rather than an absolute scale; lower values indicate more linear and, therefore, more easily readable code. However, there is no empirically established threshold distinguishing ‘good’ from ‘bad’ linearity; instead, scores are intended for comparative analysis between different pieces of code. In addition, the definition of linearity provided in Equation 2.3 does not account for invocation depth or nested method calls, as each call-declaration pair is treated independently.

## 2.7 Code Refactoring

Since the implementation of the proposed approaches in this research project requires reusing some code refactoring techniques to reduce duplication/repetitiveness in API code examples and increase their maintainability, this section provides a brief overview of software refactoring, its purpose, and its

types.

**What is Refactoring?** The maintenance of a large software system can be a challenging task for many software developers. However, one of the ways that can reduce the technical costs of software maintenance is code refactoring, which is, as defined by Fowler [93], the process of altering software code to improve its internal structure without changing its function or behaviour. In other words, refactoring means cleaning up previously written software code to make it more efficient and maintainable and to reduce the possibility of introducing future bugs. Other purposes of code refactoring can be to improve the design of software, make it easier to read and understand, and help developers write code more quickly [93, 94].

**Code Smells.** Code smells are one of the source code surface indications that developers usually face when working with large and complex systems. These visible and tangible indications usually correspond to deeper issues in the underlying code of the systems, that could eventually result in severe breakdowns. Furthermore, while code smells are not necessarily bugs; they are often the result of poor programming. As such, they can significantly decrease software maintainability [95].

Smelly code can be complex, inefficient, and difficult to maintain. Code smells can therefore be an indicator that refactoring is needed. For example, duplicated code that has similar structure and functionality but appears in more than one place is one type of code smell [93]. Such duplication can be eliminated by the *Extract Method* refactoring that works by unifying similar/duplicated bits of code in a single method and then replacing the exacted code with a call to the newly created method. There are so many other refactoring techniques that have their own motivation and can fall under one of the major refactoring categories, such as *Composing Methods*, *Moving Features Between Objects*, and *Simplifying Methods* [93].

**Refactoring in Java.** Most of the refactoring techniques that Fowler extensively explained in his book [93] have been implemented by the Eclipse Java development tools (JDT)<sup>9</sup> for the Java programming language. The Eclipse JDT is an Eclipse project that provides Java developers with a variety of plug-ins that support Java code parsing, analysis, validation, manipulation, and several refactoring operations. Moreover, these JDT plug-ins provide a set of refactoring APIs that can facilitate their extension and reuse by other plug-in developers [96, 94].

**When Not to Refactor?** Having described all the benefits that refactoring can bring to software systems, it is worth mentioning that code refactoring is not advisable in some situations. For example, in the case of messy code, it could be much more efficient and less costly if it was entirely rewritten from the start, rather than refactored [93]. Beyond this, the results of a recent study on object-oriented code refactoring [97] have shown that some refactoring techniques, such as *Extract Method*, could have a negative impact on some software quality attributes, e.g. cohesion and coupling.

## 2.8 Domain-specific Languages (DSLs)

According to Tomassetti [98], a Domain-specific Language (DSL) is a language that is built to fulfil a single and specific purpose. Unlike general-purpose languages (GPLs) (e.g. Java, Python, and C) that are used to build all types of applications, a DSL can only solve one type of problem. One good example of a DSL is the DOT language, which is a language that is designed to describe graphs. Another example is the Structured Query Language (SQL), which is used to conduct certain operations (e.g. insert, select, or delete) on relational databases. Finally, a very successful example of a DSL is HTML, which is a language that is mainly used to define web documents.

---

<sup>9</sup><https://www.eclipse.org/jdt/>

## 2.8. Domain-specific Languages (DSLs)

Given the wide variety of GPLs and the high development costs that were involved in building them, one may wonder why a limited and specific language should be developed and used instead of a generic and strong one. Below is a summary of common arguments regarding the motivations and advantages behind the development of DSLs, as outlined by Tomassetti [98] and Mernik [99].

**Analysis.** Because DSLs are limited in the type of tasks that they can conduct, they are easier to analyse compared to GPLs. For instance, one cannot guarantee the expected full behaviour of a program written in Java or C, and whether it will terminate or end in an infinite loop, whereas, with DSLs, it is much easier to analyse such behaviours.

**Productivity.** DSLs are easy to learn because they are designed to tackle problems in a particular domain, and many users spend less time and training in learning them. Moreover, DSLs can enhance the maintenance of the programs written in them because it is easy for the domain engineers to understand the specific domain errors produced by the DSL programs and fix them.

**Readability.** In certain domains, it is not possible to use a GPL to define some domain-specific notations and special keywords. One good example of that is the difficulty in defining an infix mathematical notation using a programming language. Therefore, using a DSL for such a domain can increase the readability of the programs written to solve problems in that domain.

**Safety.** Unlike GPLs, DSLs can be safer to use because they are less likely to produce complex errors. This is crucial, particularly when working with critical systems, such as health or financial software.

Given all of these advantages of using a DSL, it is evident that designing

## Chapter 2. Background

a DSL in a proper way can be extremely useful and sometimes essential for certain domains. This is because DSLs give expression capabilities targeted directly to users' domains, and this makes them much easier to understand and use. However, DSLs also present certain weaknesses and challenges, such as limited applicability to broader or cross-domain tasks, and high maintenance costs as they evolve to reflect changes in their domain.

DSLs can come in two main forms: external and internal. As defined by Fowler [100], an external DSL is a standalone language and can have either its own custom syntax or the syntax of another format such as XML. On the flip side, an internal DSL is a language that is built on top of another general-purpose host language, such as Java. This means that internal DSLs are always constrained by the valid syntax and expressions of the host language. An internal DSL is usually referred to as the stylised use of a general-purpose language for a domain-specific purpose, or an embedded DSL. One of the main advantages of internal DSLs is the availability of tools and features, such as error handling and reporting, because they follow the same syntax rules of the host language. An internal DSL can easily integrate with the projects that are developed using the same host language [101]. Moreover, it is evident that using an internal DSL by some domain users (e.g. programmers) is easier than using an external one since they use a syntax of a GPL that they are already familiar with [100].

The approaches used to design a DSL can be categorised based on the type of DSL that one wants to build. A DSL can have different concrete syntaxes: textual, graphical, or projectional editors. Textual DSLs are the most common type of DSLs. They are easier to support and can be used in a variety of contexts [98]. One of the frameworks that is used to develop tooling for textual DSLs is Xtext<sup>10</sup>. Starting with the grammar definition, Xtext generates not only a parser but also a smart editor for the language.

---

<sup>10</sup><https://www.eclipse.org/Xtext/>



## 2.9. Summary

Xtext greatly utilises the Eclipse Modelling Framework (EMF)<sup>11</sup>. In fact, the model produced by the Xtext parser is an EMF model. Graphical DSLs, on the other hand, use nodes and edges in order to express intent.

According to Kolovos [102] and Paige et al. [103], there are three main components of a DSL, abstract syntax, concrete syntax, and semantics. An abstract syntax is the expression of the concepts and relationships offered by the language. The abstract syntax is also known as “metamodel” because a metamodel is simply a description of the abstract syntax of the language, and is built using modelling infrastructure [103]. The concrete syntax of a DSL can be graphical or textual. The former is represented using visual symbols of the language concepts, and the latter uses textual grammar and rules. Finally, the semantics of the language are the meanings of its concepts.

To the best of our knowledge, no proposal has been made for building a DSL to describe an API as intended by its developers in the same way as the one proposed in this PhD thesis. However, several DSLs have been proposed as solutions to certain issues in the field of software engineering, such as defining traceability metamodels [104] and improving the process of developing and deploying applications in the Cloud [105].

## 2.9 Summary

Section 2.1 and 2.2 of this chapter introduced the key concepts of Application Programming Interfaces (APIs) and explained their role in reuse-based software engineering. They also highlighted the importance of including code examples in API documentation to assist developers in understanding and using APIs effectively. Later, Section 2.3 explored the main characteristics of effective API code examples and provided a brief overview of the literature on the notion of code example coverage. Section 2.4 concluded with a discussion of the available tools and approaches proposed for generating API

---

<sup>11</sup><https://projects.eclipse.org/projects/modeling.emf.emf>

## Chapter 2. Background

code examples and templates for API documentation.

Section 2.5 of this chapter focused on the concept of program comprehension and the factors that influence the understandability of source code, such as its structure and regularity. Furthermore, it examined these comprehension attributes specifically in the context of API code examples.

Section 2.6 of this chapter began with an overview of various standard source code metrics commonly used to measure different aspects of source code and ensure software quality. Section 2.7 provided a discussion on the notion of code refactoring, a distinct yet closely related topic to this PhD research. Finally, the chapter concluded in Section 2.8 with an overview of domain-specific languages (DSLs) and their advantages.

## Chapter 3

# Analysis and Hypothesis

This chapter summarises existing API documentation approaches reviewed in Chapter 2 and discusses their limitations. Section 3.1 highlights challenges identified in related literature, which contributed to formulating the hypotheses of this work. The research objectives, questions, hypotheses, and scope are discussed in Sections 3.2, 3.3, 3.4, and 3.5.

### 3.1 Analysis

One of the most effective resources for learning application programming interfaces (APIs) is code examples, which can take various forms such as short code snippets, tutorials, or self-contained sample applications [5]. API users often prefer these code examples over traditional reference documentation, as they tend to be more informative and practical [8]. However, research indicates that explicitly documenting APIs with adequate code examples is not common practice in software development [45, 46, 47] and that not all software projects provide effective API code examples in their documentation [18, 48]. Furthermore, writing API code examples that cover most of an API's essential functions is a challenging and demanding task for documentation writers [19, 20]. This difficulty is often due to the significant

amount of code repetition and structural similarity commonly found in API code examples, making them burdensome to write. These observations help explain the notable shortage of API code examples in API documentation.

Repetitiveness and duplication are undesirable source code characteristics for API developers. This is because duplicated code, a common type of code smell, can significantly reduce software maintainability [95]. However, while eliminating duplication is generally desirable in software development, it may not always be beneficial when writing API code examples. This is because repeating certain code, e.g. boilerplate code, can help ensure that each code example remains self-contained and independent. In turn, this can enhance the comprehensibility and usability of code examples for API users.

In addition, there may be a general belief that non-linear code (i.e. code consisting of multiple interdependent methods or classes) is more comprehensible, as it is split into shorter methods whose names help abstract and convey the semantics of the underlying code. However, to the best of our knowledge, no studies have investigated whether this is true in the context of API code examples, nor whether the length and linearity of an API code example impact its understandability.

In general, existing approaches for API code example generation do not appear to have addressed the shortage of API code examples by directly tackling the underlying challenges of writing them. Nybom et al. [14] highlight the need for ‘tools to create and maintain API documentation’ and emphasise that such tools ‘help API users produce better software more quickly.’ In addition, a substantial body of research has examined API users’ learning experiences [4], including the role of API documentation as a key resource [9] and how documentation can be enhanced to provide more guidelines and support for API users [106]. Other studies have explored API users’ preferences regarding the types of information provided in API documentation [8, 107]. However, existing literature on API documentation does not appear

### 3.2. Research Objectives

to have thoroughly investigated the preferred characteristics of effective API code examples from the perspective of API users.

Furthermore, API developers need a set of quality metrics to assess the effectiveness and coverage of API code examples, whether generated by documentation tools or manually written. Radevski et al. [43] explicitly state that ‘if API developers have access to example metrics, they can get feedback on the information they provide to potential users’ and suggest an initial knowledge base for developing an automated API code example metrics analysis tool. The availability of such metrics could improve the quality of code examples in API documentation, thus, positively impacting users’ comprehension of code examples.

The gaps and challenges highlighted above have contributed to defining the research objectives outlined in Section 3.2 and formulating the research questions and hypotheses presented in Sections 3.3 and 3.4.

In summary, this thesis aims to provide a middle-ground solution for the automatic generation of API code examples that are both easy to write and maintain for API developers and highly comprehensible and easy-to-use for API users. It also seeks to offer insights that introduce a new layer of thinking on how API code examples can be well-structured and cover most of an API’s surface.

## 3.2 Research Objectives

The first aim of this research is to alleviate the burden for API developers and documentation writers of writing and maintaining repetitive and lengthy API code examples for API documentation, thereby empowering them to produce more API usage examples. The second aim is to ensure that these API code examples are not only easy to understand but also comprehensively cover the intended API. This can facilitate API learning and code comprehension for API users, which in turn can significantly reduce API misuse and increase

## Chapter 3. Analysis and Hypothesis

the users' productivity.

The above aims can be subdivided into more specific and detailed objectives as follows:

- (**RO**<sub>1</sub>) Conduct a controlled user-based experiment to assess the impact of different source code structures on API users' performance and comprehension of API code examples.
- (**RO**<sub>2</sub>) Create a dataset of API code examples extracted from multiple open-source Java projects to help understand and specify the extent and nature of code repetition in API code examples.
- (**RO**<sub>3</sub>) Develop a code synthesis prototype for the Java programming language that enables API developers to write less repetitive, more maintainable, and comprehensible API code examples for API documentation.
- (**RO**<sub>4</sub>) Develop a Domain-Specific Language (DSL) that allows API developers to concisely describe API elements intended for external use.
- (**RO**<sub>5</sub>) Develop a static code analysis tool, built on top of the proposed DSL, to help API developers assess the coverage of their API code examples.

### 3.3 Research Questions

The research questions outlined below have been formulated to achieve the research objectives (ROs) presented in Section 3.2.

- (**RQ**<sub>1</sub>) How does the linearity of an API code example impact a programmer's performance in terms of correctness and time spent in tasks that require code comprehension?
- (**RQ**<sub>2</sub>) What effects does the length of a linear API code example have on its comprehensibility and reusability?

### 3.3. Research Questions

- (**RQ**<sub>3</sub>) Does the degree of linearity in a non-linear API example affect its comprehensibility and reusability?

Answering the three research questions above will help achieve **RO1**, i.e. understanding the impact of certain source code structures (linearity and length) on API users' comprehension of code. Also, addressing this question will contribute to motivating the achievement of **RO3**, which aims to develop a linear code synthesis prototype.

- (**RQ**<sub>4</sub>) How often do API code examples contain duplicate or near-duplicate<sup>1</sup> code that can be eliminated using the proposed linear code synthesis approach?

- (**RQ**<sub>5</sub>) How often are duplicate code fragments repeated across different API code examples?

Answering RQ4 and RQ5 will help achieve **RO2** by creating a dataset of API code examples and analysing it for code duplication and repetition. Furthermore, understanding the extent and nature of such code repetition will contribute to both motivating and evaluating the benefits of **RO3**, i.e. the development of a linear code synthesiser. In addition, as explained in Section 2.7, duplicated code may present challenges for API developers and documentation writers from a maintenance perspective; however, duplication in API examples may not be inherently negative for API users, as repeating the same setup across multiple scenarios can reinforce consistency and lower the learning curve. Nonetheless, this topic requires further empirical investigation.

- (**RQ**<sub>6</sub>) How much reduction of duplicate code is achieved by the proposed linear code synthesis approach?

**RO3** will be achieved by answering this research question, as it will

---

<sup>1</sup>Unlike duplicate code, which is identical, near-duplicate code has minor changes like renamed variables or small edits but keeps the same structure and logic.

help assess the usefulness of the proposed linear code synthesis prototype in reducing code repetition and enhancing the maintainability of API code examples.

(RQ<sub>7</sub>) How can an intended API be specified in a formal and concise manner?

**RO4**, i.e. developing an API description language, will be achieved by addressing this research question.

(RQ<sub>8</sub>) What insights can be extracted by using a formal intended API specification and the proposed API code example coverage tool?

Answering this research question will help achieve **RO5** by developing a static code analysis tool to assess the coverage of API code examples and present the results using user-friendly representations.

### 3.4 Research Hypothesis

This research has two hypotheses. The first hypothesis (**H<sub>1</sub>**) is stated as follows:

**H<sub>1</sub>**: *API users prefer code examples to be linear. Conversely, linear API code examples contain significant code repetition which makes them challenging for API developers to write and maintain. This gap can be bridged by a code synthesis tool that automatically transforms **non-linear example code** into **linear code**.*

The terms in bold are defined as follows:

**Non-linear code:** Source code that is less repetitive and more maintainable due to its reliance on multiple reusable methods. Such code is typically read in a non-linear manner by following its execution flow.

**Linear code:** Source code that is primarily read in a linear order, as it does not rely on interdependent methods or classes, which would otherwise



### 3.5. Research Scope

force API users to frequently jump between definitions to fully comprehend the API code example.

The second hypothesis (**H<sub>2</sub>**) is stated as follows:

**H<sub>2</sub>:** *A domain-specific language (DSL) for API specification can enable API developers to concisely describe the **intended surface of their APIs**. An API description written using this DSL can serve as the foundation for an API code example coverage tool that provides useful **insights** specifically tailored to API code example coverage.*

The terms in bold are defined as follows:

**Intended surface of APIs:** Unlike a factual API, which includes all publicly accessible code elements, an intended API consists only of the API elements that are deliberately designed for external use. However, it is important to note that there are different types of API users, such as clients, extenders, and maintainers, each of whom may have different skills and expectations. This hypothesis primarily concerns API clients and the quality of the API code examples written specifically for their use. By contrast, API extenders or maintainers may require deeper insights into the factual API and therefore different forms of documentation.

**Insights:** Coverage information that may be of interest to an API developer, such as the frequency of different API element usages and the number of API elements covered by a single API code example.

## 3.5 Research Scope

As the thesis title indicates, this research focuses on enhancing the learnability and discoverability of object-oriented APIs by proposing automated techniques for user-centred API documentation. It aims to improve the experiences of both API developers and users in documenting and utilising APIs. However, as illustrated in Figure 3.1, this research does not examine

### Chapter 3. Analysis and Hypothesis

all aspects of API documentation but rather focuses on a specific component, i.e. code examples.

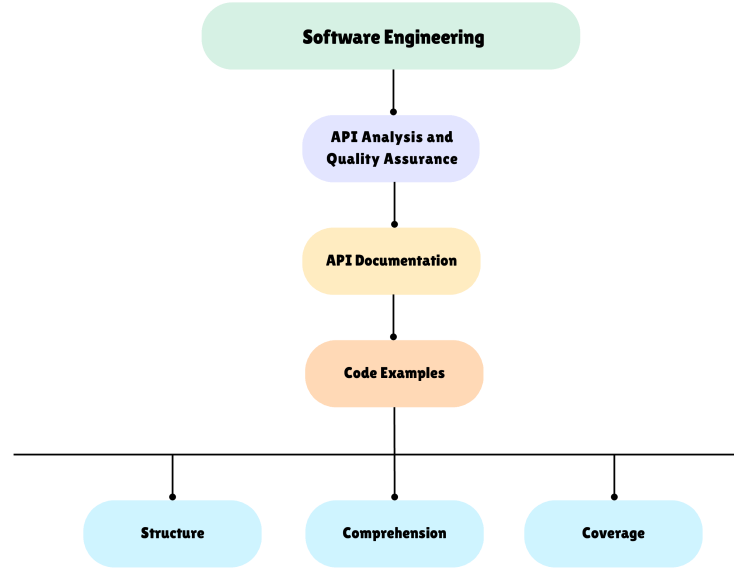


Figure 3.1: A concept map illustrating the thesis’s scope within the field of Software Engineering.

Moreover, all proposed approaches, the API specification DSL, and the conducted code comprehension user study, are limited to Java-based APIs and do not extend to APIs of other general-purpose programming languages. Supporting multiple programming languages would be a highly unrealistic task given the time constraints of this research project. However, due to the syntactic and semantic similarities between Java and other general-purpose programming languages such as Python and C, the proposed approaches are expected to be highly portable and adaptable. To fully validate this portability, confirm the generalisability of the findings, and address any language-specific challenges that may arise, further investigation and research is required.

In addition, as shown in Figure 3.1, this research focuses only on three

### 3.5. Research Scope

main aspects of API code examples: their structure, comprehension, and coverage. Only two structural factors are examined, i.e. the linearity and length of code examples. Other source code factors, such as code complexity and formatting, fall outside the scope of this thesis. Furthermore, while one of the objectives is to assess users' comprehension of linear and non-linear API code examples using specific tasks and metrics, it is not within the research scope to evaluate users' long-term retention or learning curves when working with linear API code examples or vice versa.

In the context of API usage, this research considers only direct method invocations when investigating linearity; transitive invocations across multiple levels of method calls are not included. Furthermore, hierarchical relationships such as inheritance and polymorphism fall outside the scope of the proposed approaches. In the intended API specification language, each concrete type and its members must be explicitly described. Specifying a superclass does not automatically include its subclasses or their members; these must also be explicitly declared if needed.

All datasets of API code examples, both newly created and reused in this research, are derived from open-source Java projects available in public repositories such as GitHub.<sup>2</sup> Proprietary and closed-source APIs also fall outside the scope of this research project.

Finally, the proposed static code analysis tool, built on top of the proposed API description language, is designed to evaluate API code example coverage. It is not intended to serve as a general-purpose code quality assessment tool; rather, it focuses only on measuring the extent to which code examples cover a described intended API.

---

<sup>2</sup><https://github.com>

### **3.6 Summary**

This chapter provided a comprehensive overview of the research and outlined its main challenges. After discussing the research background and related literature in Chapter 2, the analysis of the research problem domain was presented in Section 3.1. Furthermore, the research objectives were presented in Section 3.2. Section 3.3 introduced a detailed list of the main research questions and explained how addressing them aligns with the research objectives. The research hypotheses were stated in Section 3.4. Finally, the chapter concluded with a discussion of the research scope in Section 3.5.

## Chapter 4

# Impact of Source Code Linearity on the Programmers’ Comprehension of API Code Examples

Working through Application Programming Interface (API) code examples has been proven to be the most preferred learning strategy for both beginner and experienced API users [8]. Surprisingly, little is known about how the different source code structures in these examples affect their comprehensibility and reusability. Furthermore, existing work appears to be focused on examining the impact of several source code characteristics on comprehension only in the context of generic software. Therefore, to fill this gap, the study presented in this chapter focuses specifically on API code examples. Moreover, unlike existing studies, different source code constructs that illustrate the same API usage and functionality are examined, narrowing the evaluation focus to the constructs’ impact on comprehension. Also, an additional concept is assessed, which is the impact of the examined source code

## Chapter 4. Impact of Source Code Linearity on the Programmers’ Comprehension of API Code Examples

structures on the reusability of API code examples.

In this study, the interest is particularly on exploring the impact of two source code aspects: the degree of linearity and length. Linear source code refers to code that can be read primarily in a sequential order without interference from interdependent methods or classes. Considering the absence of jumps between method definitions in such a code, it is hypothesised that comprehension may be easier. In addition, linear API code examples may be easier to reuse and adapt into one’s codebase, as they typically contain a single self-contained method that can be copied and edited, as opposed to non-linear code examples that involve multiple methods.

Through this study, the aim is to help API developers understand how to structure their API code examples more effectively, thereby enhancing the examples’ comprehensibility and reusability. This, in turn, is expected to promote the learnability of their APIs.

The work discussed in this chapter has been published under the title: “Exploring the Impact of Source Code Linearity on the Programmers’ Comprehension of API Code Examples” in the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC), 2024.

All data collected or used in this chapter is available in the replication package.<sup>1</sup>

## 4.1 Methodology

### 4.1.1 Research Questions

The aim is to answer the following research questions:

**RQ1:** How does the linearity of an API code example impact a programmer’s performance in terms of correctness and time spent in tasks that require code comprehension?

---

<sup>1</sup>Replication package: <https://figshare.com/s/52e11ece2f39bac64bcb>

**RQ2:** What effects does the length of a linear API code example<sup>2</sup> have on its comprehensibility and reusability?

**RQ3:** Does the degree of linearity in a non-linear API example affect its comprehensibility and reusability?

### 4.1.2 Study Design

This study was conducted online to allow access to a large and more diverse pool of participants. The Gorilla platform [108] was utilised, a widely used online experiment builder, which provided all the features that were needed in the study (randomisation, counterbalancing presentation of formatted source code, accurate reaction times and integration with participant recruitment platforms). Java developers were recruited through Prolific<sup>3</sup> – a participant recruitment platform for online research – and assigned to two main groups: linear vs non-linear (between-subjects) as shown in Figure 4.1. Each group consisted of two sub-groups that correspond to the treatment categories shown in Table 4.1. Participant assignment to groups was fully randomised and balanced, with a 1:1 ratio. Each participant completed two code comprehension and reuse tasks from the same treatment category (highlighted in the same colour in Table 4.1). The order in which each participant received tasks was also randomised to eliminate any potential order effects. Ethical approval was obtained before the study was conducted.

### 4.1.3 Independent Variables

A single independent variable was considered i.e. the source code structure of API code examples. Two primary source code factors were systematically varied: code linearity, which is manipulated using the source code linearity metric (*i*) proposed by Peitek et al. [76] and explained in Section

---

<sup>2</sup>Please refer to (replication package → examples) for some more sample code illustrating linear and non-linear API code examples.

<sup>3</sup><https://www.prolific.com>

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

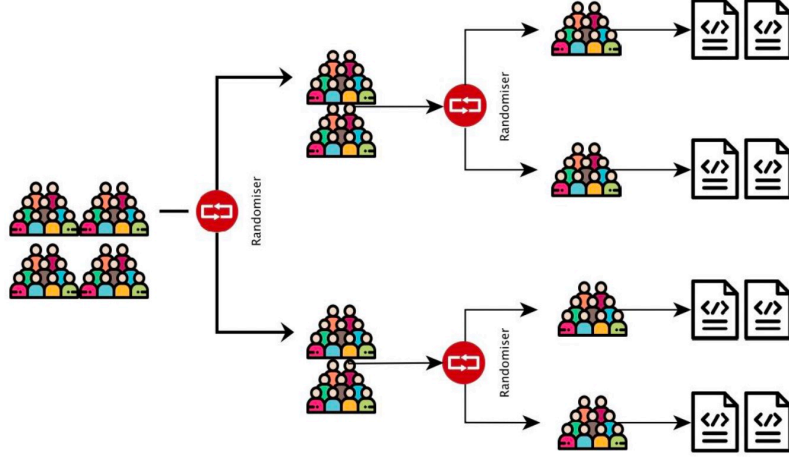


Figure 4.1: Study design (between-subjects) and the distribution of participants.

Table 4.1: API code examples and their variants and metric values. Variants sharing the same colour belong to the same treatment category.

API Code Example	Variant	Metrics			
		LOC	Complexity	Linearity ( $i$ )	# Method Calls
Date Example	Linear	31	7	0.00	0
	Non-linear	14	1	12.95	5
Chronology Example	Linear	25	4	0.00	0
	Non-linear	11	1	11.41	4
Duration Example	Linear	35	7	0.00	0
	Non-linear	21	2	19.47	6
Interval Example	Linear	46	7	0.00	0
	Non-linear	33	5	19.43	6

2.6.5; and code length, which is varied by adjusting the number of lines of code (LOC). As shown in Table 4.1, these two factors were combined to generate four treatment categories: **linear-short**, **linear-long**, and non-linear with varying levels of linearity ( $i$ ), ranging from  $(10.00 < i \leq 15.00)$  to  $(15.00 < i \leq 20.00)$ . These selected values reflect a diverse spectrum of



code linearity.

#### 4.1.4 Dependent Variables

Three dependent variables were measured: reaction time, correctness and subjective rating.

*Time Duration Marking.* For the comprehension phase, reaction time was defined as the amount of time that elapsed between a participant’s initial view of an API code example and submission of their overall comprehension rating. Similarly, for the code-reuse phase, reaction time was defined as the duration between a participant’s first view of the required code-reuse task (in which they were asked to modify the given API code examples to solve a specific programming problem and then optimise the code by deleting any unnecessary statements) and the submission of their solution.

*Judging Correctness.* Marking consistency was ensured by defining a set of correctness categories and criteria: correct (A), almost correct (B), partially correct (C), incorrect (D) and absent (F). Detailed criteria are available in Appendix A.

#### 4.1.5 Participants

*Pilot.* To validate the study design, a pilot with four participants (average age  $28.5 \pm 9.5$ ; average years of Java programming  $3.2 \pm 1.5$ ) was conducted. Based on the results of this pilot study, the number of tasks assigned to each participant was reduced from four to two to minimise the experiment’s overall duration. Participants also noted that the online IDE, Replit,<sup>4</sup> used in the pilot study, had a slow execution time. As a result, it was replaced with JDoodle<sup>5</sup> in the official version of the experiment. However, it is worth noting that, at the time of the study (October 2023), JDoodle, unlike desktop

---

<sup>4</sup><https://replit.com>

<sup>5</sup><https://www.jdoodle.com>

#### Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

IDEs such as Eclipse or IntelliJ IDEA, only provided basic features such as syntax highlighting, code execution, and code sharing. It did not support advanced capabilities commonly found in full IDEs, including code navigation, integrated API documentation lookup, and code completion.

In addition, the wording of the code optimisation question was improved. The data collected in the pilot study was not used in the final analysis.

*Pre-screening survey.* In addition to the pre-screeners provided by Prolific, a separate programming knowledge survey was created to assess participants' programming knowledge before participating in the study. This was essential since recent research revealed that, while recruitment platforms, such as Prolific, greatly mitigate self-selection bias and some security issues, their pre-screeners may not always be reliable [109, 110, 111]. Furthermore, in this survey, the basic knowledge questions and time limit recommended by Danilova et al. [109] were used. Participants who correctly answered all the questions within the time limit were manually invited to participate in the study.

*Study.* 61 Java developers from 14 countries with varying levels of programming experience were recruited. Only 19 (31%) participants stated that they previously used the Joda-Time Java library (a detailed discussion of the rationale for selecting this particular Java library is provided in Section 4.1.6). Among them, only eight (42%) said that they used it more than once, and none of them reported regular usage. The participants' prior programming experience was assessed using a validated questionnaire that is based on self-estimates [112, 113]. Each participant was compensated £10 for their time and effort. Additional information about the participant demographics is shown in Table 4.2.

Table 4.2: Participant demographics.

Category	n=61
<b>Student</b>	43 (70%)
<b>Professional Developer</b>	18 (30%)
<b>Programming Experience</b> (in Years)	$5.9 \pm 3.3$
<b>Java Programming Experience</b> (in Years)	$3.2 \pm 2.2$
<b>Familiarity with Joda-Time</b>	19 (31%)
<b>Male</b>	54 (89%)
<b>Female</b>	7 (11%)
<b>Age</b> (in Years)	$25.4 \pm 6.1$

#### 4.1.6 Material

*API code examples.* We required the code example to be from a Java library that met our selection criteria, which were as follows:

1. The library must not require prior domain knowledge that would pose an unnecessary challenge to participants;
2. It should be well-documented; and
3. It should not be so popular that an average Java developer would already be familiar with it.

All of these criteria were satisfied by the Joda-Time Java library,<sup>6</sup> which addresses a well-known concept, i.e. date and time handling. Furthermore, the intention was to utilise the code examples available on the Joda-Time documentation page.<sup>6</sup> However, these examples were not complex enough. Thus, custom code examples were created.

As shown in Table 4.1, four API code examples were developed, each of which demonstrated a distinct usage of Joda-Time. This variation of examples was important to minimise the risk of any potential learning effect arising from within the examples. Then, linear and non-linear versions of each example were created. Efforts were made to ensure that these examples appeared as natural as possible by: 1) properly documenting them and 2)

<sup>6</sup><https://www.joda.org/joda-time/>

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

making them depict real-world scenarios such as data manipulation or meeting scheduling. For the non-linear versions of the examples, the linear version was refactored by extracting some functionalities into a set of utility methods and replacing the extracted code with method calls using the extract method refactoring technique. The extracted utility methods were collected into a single utility class, i.e. `util.java`. This class contained only `public static` methods and was imported into the non-linear examples via static imports, ensuring that no object instantiation was required. For example, the non-linear version of the `DateExample` shown in Listing 4.1 is presented in Listing 4.2.

In addition, examples within the same treatment category were relatively comparable in terms of length, complexity, degree of linearity, and the number of utility method calls. For instance, two of the API code examples used in this study are shown in Listings 4.1 and 4.3. The first example computes future and past dates by skipping weekends starting from a given date and demonstrates date mutation using Joda-Time. The second example illustrates how to convert a date to the Buddhist calendar and calculate a period.

```
1 public class DateExample {
2
3     public static void main(String[] args) {
4         String dateString = "2023-07-04";
5         String pattern = "yyyy-MM-dd";
6
7         DateTimeFormatter formatter = DateTimeFormat.forPattern(pattern);
8
9         // Parse the input date string to create a LocalDate object
10        LocalDate date = formatter.parseDateTime(dateString).toLocalDate()
11        ;
12        LocalDate futureDate = date;
13        int addedDays = 0;
14
15        // Find the future date
16        while (addedDays < 10) {
```

## 4.1. Methodology

```
16     futureDate = futureDate.plusDays(1);
17     if (!(futureDate.getDayOfWeek() == DateTimeConstants.SATURDAY
18         || futureDate.getDayOfWeek() == DateTimeConstants.SUNDAY)) {
19         ++addedDays;
20     }
21 }
22
23 // Find the past date
24 LocalDate pastDate = date;
25 int subtractedDays = 0;
26 while (subtractedDays < 5) {
27     pastDate = pastDate.minusDays(1);
28     if (!(pastDate.getDayOfWeek() == DateTimeConstants.SATURDAY
29         || pastDate.getDayOfWeek() == DateTimeConstants.SUNDAY)) {
30         ++subtractedDays;
31     }
32 }
33
34 System.out.println("Future date: " + futureDate + "\nPast date: "
35     + pastDate);
36
37 // Demonstrate how to mutate a date
38 System.out.println("Mutate the future date: ");
39 DateTime immutable = futureDate.toDateTimeAtStartOfDay();
40 MutableDateTime mutableDateTime = immutable.toMutableDateTime();
41 mutableDateTime.setDayOfMonth(3);
42 immutable = mutableDateTime.toDateTime();
43 System.out.println(immutable);
44 }
```

Listing 4.1: A linear code example using the Joda-Time API (DateExample.java).

```
1 import static code.comprehension.experiment.tasks.joda.created.
    nonlinearExternal.Util.parse;
2 import static code.comprehension.experiment.tasks.joda.created.
    nonlinearExternal.Util.add;
3 import static code.comprehension.experiment.tasks.joda.created.
    nonlinearExternal.Util.subtract;
4
5 import org.joda.time.DateTime;
```

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

```
6 import org.joda.time.LocalDate;
7 import org.joda.time.MutableDateTime;
8
9 public class DateExample {
10     public static void main(String[] args) {
11
12         String dateString = "2023-07-04";
13         String pattern = "yyyy-MM-dd";
14
15         // Parse the input date string to create a LocalDate object
16         LocalDate date = parse(dateString, pattern).toLocalDate();
17
18         // Find the future date
19         LocalDate futureDate = add(date, 10);
20
21         // Find the past date
22         LocalDate pastDate = subtract(date, 5);
23
24         System.out.println("Future date: " + futureDate + "\nPast date: "
25                             + pastDate);
26
27         // Demonstrate how to mutate a date
28         System.out.println("Mutate the future date: ");
29         DateTime immutable = futureDate.toDateTimeAtStartOfDay();
30         MutableDateTime mutableDateTime = immutable.toMutableDateTime();
31         mutableDateTime.setDayOfMonth(3);
32         immutable = mutableDateTime.toDateTime();
33         System.out.println(immutable);
34     }
```

Listing 4.2: A non-linear code example using the Joda-Time API (DateExample.java).

```
1 public class ChronologyExample {
2     public static void main(String[] args) {
3         // Create a DateTime object with default chronology i.e. ISO
4         // Chronology
5         DateTime date = new DateTime(2023, 8, 15, 10, 30, 0, 0);
6         System.out.println("Date in default ISO chronology: " + date);
7
8         // Convert the date to the Buddhist chronology
```

## 4.1. Methodology

```
8      DateTime buddhistDate = date.withChronology(BuddhistChronology.  
        getInstance());  
9      int year = buddhistDate.getYear();  
10     int month = buddhistDate.getMonthOfYear();  
11     int day = buddhistDate.getDayOfMonth();  
12     System.out.println("Date in Buddhist chronology: " + year + "-" +  
        month + "-" + day);  
13  
14     // Calculate a period using a date with default chronology and  
        excluding weekend days  
15     System.out.println("Period while skipping weekend days: ");  
16     DurationFieldType[] types = { DurationFieldType.years(),  
        DurationFieldType.months(), DurationFieldType.days() };  
17     PeriodType periodType = PeriodType.forFields(types);  
18     LocalDate startDate = date.toLocalDate();  
19     int subtractedDays = 0;  
20  
21     // Loop to subtract an arbitrary number of days (e.g. 450) while  
        excluding weekend days  
22     while (subtractedDays < 450) {  
23         startDate = startDate.minusDays(1);  
24         int dayOfWeek = startDate.getDayOfWeek();  
25         if (!(dayOfWeek == DateTimeConstants.SATURDAY || dayOfWeek ==  
            DateTimeConstants.SUNDAY)) {  
26             ++subtractedDays;  
27         }  
28     }  
29  
30     // Format and print the period  
31     Period period = new Interval(startDate.toDateTimeAtStartOfDay(),  
        date).toPeriod(periodType);  
32     PeriodFormatter formatter = new PeriodFormatterBuilder()  
33         .appendYears().appendSeparator(" years ")  
34         .appendMonths().appendSeparator(" months ")  
35         .appendDays().appendSeparatorIfFieldsBefore(" days ")  
36         .toFormatter();  
37     System.out.println(period.toString(formatter));  
38 }  
39 }
```

Listing 4.3: A linear code example using the Joda-Time API (ChronologyExample.java).

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

*Tasks.* Oftentimes, when API users turn to code examples to learn a new API, they have a specific problem in mind. They are hoping that the code in the example they are reviewing will be reusable. If this is possible, they copy and paste the example, then modify its source code by adding or deleting statements to match their needs [8, 114]. In this study, the aim was to simulate this behaviour. Therefore, each code-reuse task had two parts: 1) code modification, in which participants were required to make changes to address a specific problem; and 2) code optimisation, in which participants were asked to remove any unnecessary code that did not directly contribute to their task solution. The tasks<sup>7</sup> were generally easy and designed to be solved with a few edits. Each API code example had a unique task that remained the same for both versions (i.e. linear and non-linear) of the example. For instance, the code-reuse task for the API code example shown in Listing 4.1 was defined as follows:

**Code-reuse Task:** Once you have reviewed the code below and developed a thorough understanding of its functionality, please proceed with the following tasks:

1. Using the Joda-Time Java library, modify the code to retrieve a future date that is 20 days ahead and a past date that is 10 days ago while skipping weekend days and Mondays.
2. Identify and remove any unnecessary code from your solution. Keep only the code parts that directly pertain to your solution.
3. Make a copy of your code solution and paste it back into the survey system.

---

<sup>7</sup>All tasks used in this study are available in the [replication package](#).



### 4.1.7 Experiment Procedure

After obtaining their consent, the participants were asked to complete a demographics questionnaire. Subsequently, each of them was randomly assigned two code examples from the same treatment category. This means that each participant completed two distinct code-reuse tasks.

Each task was designed to be completed in four sequential parts. The first was the comprehension part (shown in Figure 4.2), in which participants were asked to review the example and rate their own understanding. The next part pertained to instructions, in which participants were given a link to an online IDE<sup>8</sup> that contained a Java project of the example, with Joda-Time imported and ready to use. Participants were also instructed on how to download the example if they preferred using their own IDE. The third part was the code-reuse task (shown in Figure 4.3), in which participants answered a two-part question (as explained in Section 4.1.6) and pasted their solution code in a given text box. The final part involved post-task questions, in which participants were asked to rate how difficult it was to reuse the code, whether they employed the provided online IDE or their own, and report any break time (if any was taken).

The time spent on two of the four parts i.e. comprehension time (part 1) and reuse time (part 3), was only measured. The rationale for separating the comprehension and reuse of the same code example was to reduce participants' use of the 'as-needed' program comprehension strategy [115, 116, 117].

### 4.1.8 Data Analysis

The correctness of responses for each task was manually analysed. First, the categories mentioned in Section 4.1.4 were converted to numerical values (correct (A) = 100%, almost correct (B) = 70%, partially correct (C) = 40%; both incorrect (D) and absent (F) = 0%). The same scale was applied to both

---

<sup>8</sup><https://www.jdoodle.com>

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

After reviewing the API code example provided below (i.e. ChronologyExample.java) and gaining a thorough understanding of its functionality, please **rate** your overall comprehension of it. Then, click on **Next** to proceed.

```
ChronologyExample.java
import org.joda.time.DateTime;
import org.joda.time.DurationFieldType;
import org.joda.time.LocalDate;
import org.joda.time.PeriodType;
import org.joda.time.chrono.BuddhistChronology;

/**
 * This code example demonstrates the use of the Joda-Time library
 * and format dates in different chronologies.
 */
public class ChronologyExample {
    public static void main(String[] args) {
        // Create a DateTime object with default chronology i.e. ISO
        DateTime date = new DateTime(2023, 8, 15, 10, 30, 0, 0);
        System.out.println("Date in default ISO chronology: " + date);

        // Convert the date to the Buddhist chronology
        System.out.println(
            "Date in Buddhist chronology: " + Util.getDataWithChronology(
                date, BuddhistChronology.INSTANCE));

        // Calculate a period using a date with default chronology and
        // days
        System.out.println("Period while skipping weekend days: ");
        DurationFieldType[] types = { DurationFieldType.years(), DurationFieldType.months(),
            DurationFieldType.days() };
    }
}
```

```
Util.java
import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;

import org.joda.time.Chronology;
import org.joda.time.DateTime;
import org.joda.time.DateTimeConstants;
import org.joda.time.DateTimeFieldType;
import org.joda.time.Duration;
import org.joda.time.Period;
import org.joda.time.PeriodType;
import org.joda.time.format.DateTimeFormatter;
import org.joda.time.format.DateTimeFormatterBuilder;
import org.joda.time.format.PeriodFormatter;
import org.joda.time.format.PeriodFormatterBuilder;

/**
 * a utility class that contains a set of static utility methods
 * for Joda-time functionalities.
 */
}
```

Very Poor

Poor

Average

Good

Excellent

Next

Need Assistance? [Contact Us for Help!](#)

Figure 4.2: Example of a code comprehension task given to participants.

### Programming Task Questions

- Using the **Joda-Time** Java library, **modify the code** you just viewed to:
  - Display the year, month, and day of an additional chronology (e.g., Gregorian or Julian) alongside the existing Buddhist chronology code.
  - Identify and remove any unnecessary code from your solution. Keep only the code parts that directly pertain to your solution.
- Make a copy** of your solution code and **paste it** back into the text box below:

Next

Need Assistance? [Contact Us for Help!](#)

Figure 4.3: Example of a code reuse task given to participants.

## 4.2. Results and Discussion

parts of the code-reuse task (i.e. code modification and code optimisation). In the code modification part, participants were required to reason about a programming problem, adapt the provided code, and produce a working solution. By contrast, the code optimisation part only required simplifying the code by removing unnecessary statements. Because the first part demanded considerably greater cognitive effort and problem-solving ability, it was assigned a higher weight (90%) in the overall task score, while the second part accounted for only 10%. Responses with an overall score of 60% or higher were considered correct. This overall correctness threshold ensures that participants achieve at least 70% in the first part of the task.

When analysing reaction times automatically captured by Gorilla [108], only correct responses were considered. The Shapiro-Wilk test [118] was used to assess the normality of reaction times and correctness, as well as Levene’s test [119] to evaluate variance homogeneity. The findings indicated non-normality and unequal variances for both correctness and reaction times. Therefore, to test for statistically significant differences, a non-parametric test, the Mann–Whitney U test (Wilcoxon rank-sum test), was used with a significance level of  $\alpha = 0.05$ .

## 4.2 Results and Discussion

As shown in Tables 4.3, 4.4, 4.5 and Figure 4.4, participants generally spent less time comprehending and reusing linear code examples (both in mean and median reaction times). This observation suggests that the source code linearity in an API code example may affect a programmer’s performance. This influence has a greater impact on comprehension and is statistically significant when the linear API code example is also short (e.g. date and chronology examples). Moreover, in terms of reusability, there appears to be a trend towards significance in two of the API code examples (chronology and interval examples), as reflected by their moderate p-values of 0.10 and

#### Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

0.06, respectively. However, the impact on correctness (Mann–Whitney U test,  $W = 1690, p = 0.428$ ) and subjective rating (as shown in Figure 4.5) was not substantial (**RQ1**).

The Mann–Whitney U test revealed a significant difference in both comprehension ( $W = 125, p = 0.004$ ) and reusability ( $W = 99, p = 0.000$ ) between the groups that received linear-short and linear-long API code examples (**RQ2**). Similarly, participants spent less time reusing the non-linear code examples when the linearity value ( $i$ ) was lower ( $i < 15.00$ , Mann–Whitney U test:  $W = 68, p = 0.003$ ). Notably, unlike the comparison in RQ1, this comparison is based on code examples illustrating different API usage; thus, the significant differences in participants' performance could be due to variations in the implemented API functionality and required tasks (**RQ3**).

In addition, while some comparisons yielded statistically significant differences in comprehension times (e.g., Date and Chronology examples), other cases showed non-significant results in both the comprehension and reuse phases of the study. One possible explanation is that the effect of code examples' linearity on their comprehensibility and reusability may be influenced by factors beyond linearity itself, such as familiarity with the API or individual problem-solving strategies, which could reduce the sensitivity of the study's metrics. Furthermore, the cognitive demands of switching between code fragments in the non-linear versions may not have been substantial enough to noticeably affect performance, particularly when the overall task or code was relatively simple. Thus, the non-significant findings suggest that while linearity can matter, its impact may be context-dependent rather than universal.

Overall, the results of this study suggest that API users perform better and understand API code examples more effectively when they are structured and presented in a linear order. Also, it may be more beneficial for API

## 4.2. Results and Discussion

developers to avoid making their code examples overly lengthy, as this can negatively impact their comprehensibility and reusability.

Table 4.3: The study results. Variants sharing the same colour belong to the same treatment category. The imbalance in the number of responses (N) between two variants of the same example is due to our exclusion of responses with inaccurately reported break times.

API Code Example	Variant	N	Correctness	Comprehension		Reuse	
				Median Reaction Time	p-value*	Median Reaction Time	p-value*
Date Example	Linear	17	15 (88%)	44s	<b>0.01</b>	6m 11s	0.36
	Non-linear	13	8 (62%)	1m 38s		7m 14s	
Chronology Example	Linear	18	10 (56%)	43s	<b>0.03</b>	3m 37s	<b>0.10</b>
	Non-linear	14	13 (93%)	1m 49s		6m 15s	
Duration Example	Linear	16	9 (56%)	3m 31s	0.60	16m 18s	0.40
	Non-linear	12	5 (42%)	2m 17s		24m 59s	
Interval Example	Linear	14	10 (71%)	1m 14s	0.33	9m 21s	<b>0.06</b>
	Non-linear	12	9 (75%)	2m 49s		12m 56s	
Overall	Linear	65	44 (68%)	58s		7m 17s	
	Non-linear	51	35 (69%)	1m 50s		10m 13s	

\* Mann–Whitney U test.

### 4.3. Threats to Validity

Table 4.4: Study results for the linear API code examples.

API Code Example	Variant	N	Correctness	Comprehension Median Reaction Time	Reuse Median Reaction Time
Date Example	Linear	17	15 (88%)	44s	6m 11s
Chronology Example	Linear	18	10 (56%)	43s	3m 37s
Duration Example	Linear	16	9 (56%)	3m 31s	16m 18s
Interval Example	Linear	14	10 (71%)	1m 14s	9m 21s
<b>Overall</b>	<b>Linear</b>	65	44 (68%)	58s	7m 17s

Table 4.5: Study results for the non-linear API code examples.

API Code Example	Variant	N	Correctness	Comprehension Median Reaction Time	Reuse Median Reaction Time
Date Example	Non-linear	13	8 (62%)	1m 38s	7m 14s
Chronology Example	Non-linear	14	13 (93%)	1m 49s	6m 15s
Duration Example	Non-linear	12	5 (42%)	2m 17s	24m 59s
Interval Example	Non-linear	12	9 (75%)	2m 49s	12m 56s
<b>Overall</b>	<b>Non-linear</b>	51	35 (69%)	1m 50s	10m 13s

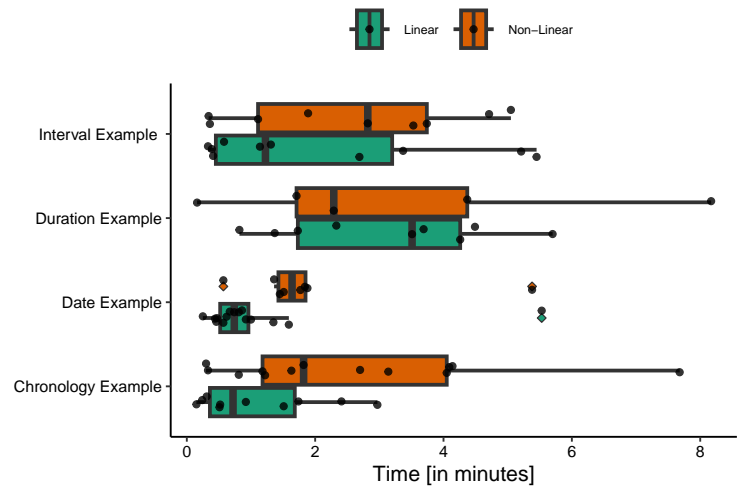
## 4.3 Threats to Validity

A number of validity threats may have arisen from decisions made during the design and implementation of the user study discussed in this chapter. The following sections outline these threats and describe the steps taken to minimise them.

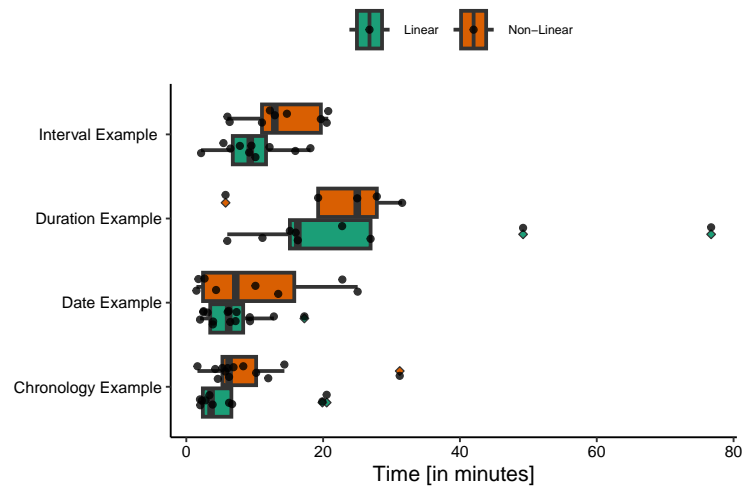
### 4.3.1 Construct Validity

To mitigate construct validity threats, specific metrics were used to manipulate the independent variables. Several API code examples and tasks were created to reflect common real-world programming scenarios, such as duration calculations and date/time manipulations. Additionally, a consistent correctness evaluation was followed. However, since the experiment was

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples



(a) Comprehension time

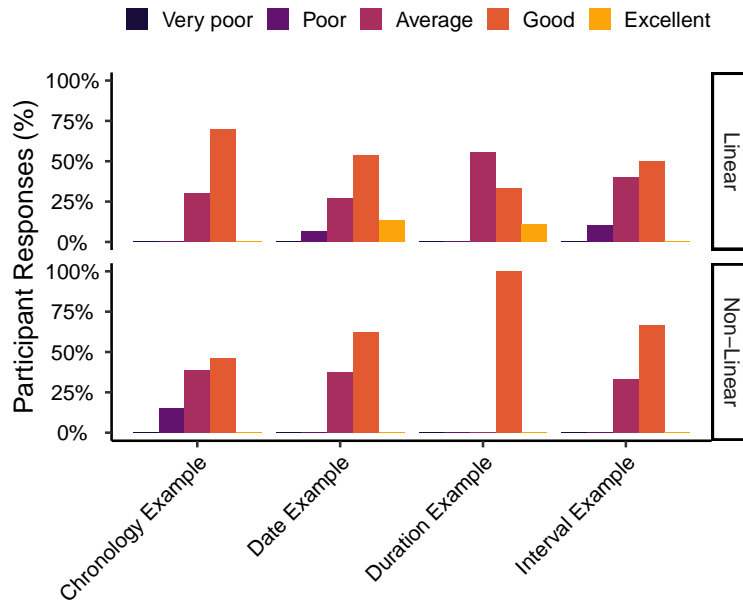


(b) Reuse time

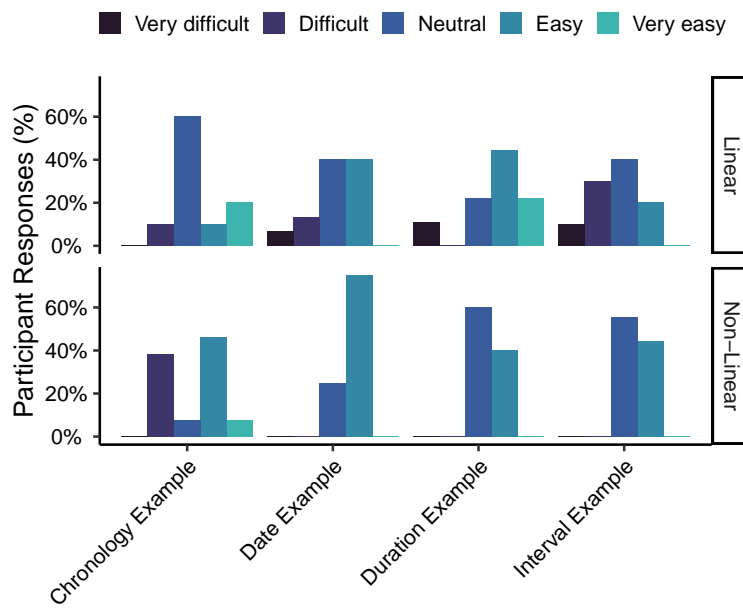
Figure 4.4: The time spent on (a) comprehending, and (b) reusing the API code examples used in the study. Each boxplot represents the responses for one version (linear or non-linear) of a single example.



### 4.3. Threats to Validity



(a) Comprehension Ratings



(b) Reusability Difficulty Ratings

Figure 4.5: Participants' subjective ratings of API code example comprehension (a) and reusability difficulty (b).

## Chapter 4. Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

conducted online, there was limited control over some participants' activities. To address this, participants were asked to self-report break times, which were subtracted from the time spent on solving the task, and disclose whether they used their own IDEs. While this approach provided insights into participants' behaviour, it was not entirely conclusive.

In addition, the weighting scheme of 90% for code modification and 10% for code optimisation was chosen based on the anticipated cognitive effort required for each part of the code-reuse task. However, this choice is subjective, and alternative weighting schemes could potentially affect the overall correctness scores and, consequently, the significance of the findings. A more comprehensive evaluation would require additional statistical testing.

### 4.3.2 Internal Validity

Internal validity threats in this study primarily included selection bias and learning effects. These threats were reduced by randomly assigning participants to treatment groups and randomising the order in which they viewed tasks. In addition, a validated programming experience questionnaire was administered; and the participants were also pre-screened for their programming knowledge.

### 4.3.3 External Validity

One potential threat to external validity was the study's limited scope. This study focused solely on the API of one Java library and included only a few code examples. Also, 70% of the participants were students, which limited the generalisability of the study's findings. Moreover, the study's virtual setting and the use of an online IDE, which lacks features such as auto-completion and error checking, may not fully reflect the conditions of a traditional coding environment. Furthermore, participants who worked with non-linear API code examples, in particular, and used the provided online

#### 4.4. Summary

IDE rather than their own full-featured IDE, may have found it more difficult to navigate between different parts of the code, as JDoodle did not support advanced code navigation capabilities at the time of the study. However, study data indicated that the proportions of participants who used their own IDEs in both main study groups (linear and non-linear) were relatively similar; therefore, the potential impact of this threat on the study's results was likely minimal.

### 4.4 Summary

The first part of this chapter described the methodology used to conduct a controlled user study aimed at exploring the impact of two main source code aspects: code linearity and length, on the comprehensibility and reusability of API code examples. It then provided an overview of the study procedure, including the steps and instructions followed by the participants during the study. This section concluded with a brief presentation of the key data analysis decisions and the statistical tests utilised to obtain the study results.

The second part of this chapter discussed the study results and concluded by addressing the main threats to the validity of the acquired results and the measures taken to mitigate them.

## Chapter 5

# Linear API Usage Example Synthesis

As explained in Chapter 4, API users tend to prefer linear API code examples, as they are generally easier to reuse and comprehend. However, such examples can be tedious to write and maintain for API developers, as they often contain repetitive code and structural patterns. Therefore, the linear code synthesiser proposed in this chapter aims at helping API developers with writing and maintaining less repetitive and more maintainable code examples while also keeping the examples linear and thus easy to follow for API users. It is implemented in a prototype for the Java programming language. It automatically synthesises linear API code examples from less repetitive and more maintainable versions by refactoring and inlining reusable methods.

The work discussed in this chapter has been published under the titles: “Towards Generating Maintainable and Comprehensible API Code Examples” in the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024, and “Synthesising Linear API Usage Examples for API Documentation” in the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022.

## 5.1. Linear Code Synthesiser's Features

All implementation code, data, results, and R scripts discussed or used in this chapter are available in the replication package.<sup>1</sup>

### 5.1 Linear Code Synthesiser's Features

To better illustrate the proposed Synthesis approach, let us consider the two real-world code examples shown in Listings 5.1 and 5.2. These two code snippets are extracted from the quickstart Java examples of the open-source Vonage Voice API<sup>2</sup> and are available on GitHub [120]. This API provides a simple means of constructing and manipulating cloud-based voice applications. For example, the code snippet in Listing 5.1 demonstrates to API users how to send dual-tone multi-frequency (DTMF) tones to an active call, and the example in Listing 5.2 shows how to play a text-to-speech message to a specified phone call.

```
1 final String ANSWER_URL = "https://nexmo-community.../long-tts.json";
2 CallEvent call = client
3     .getVoiceClient()
4     .createCall(new Call(TO_NUMBER, VONAGE_NUMBER, ANSWER_URL));
5
6 Thread.sleep(20000);
7
8 final String UUID = call.getUuid();
9 final String DIGITS = "332393";
10 client.getVoiceClient().sendDtmf(UUID, DIGITS);
```

Listing 5.1: Vonage API code example (1) - SendDtmfToCall.java

```
1 final String ANSWER_URL = "https://nexmo-community.../silent-loop.json";
2 CallEvent call = client
3     .getVoiceClient()
4     .createCall(new Call(TO_NUMBER, VONAGE_NUMBER, ANSWER_URL));
5
6 Thread.sleep(5000);
7
```

---

<sup>1</sup>Replication package: <https://figshare.com/s/ac8128c17420fa9c5d2e>

<sup>2</sup><https://www.vonage.co.uk/communications-apis/>

## Chapter 5. Linear API Usage Example Synthesis

```
8     final String UUID = call.getUuid();
9     final String TEXT = "Hello ... ";
10    client.getVoiceClient().startTalk(UUID, TEXT, var);
```

Listing 5.2: Vonage API code example (2) - SendTalkToCall.java

While the above code snippets illustrate two distinct API usages, they are very similar in structure and contain duplicate code. The only code statements that differ are those highlighted in the same colour in both examples.<sup>3</sup> Duplicated code can present several challenges in the context of software maintenance and evolution [121, 122]. Furthermore, the process of manually writing such repetitive usage examples can be a laborious task for API developers, especially when dealing with large APIs. Thus, moving repetitive code to a set of reusable/utility methods (e.g. `createCallEvent` method in Listing 5.3) that can be invoked as needed would be less repetitive and more maintainable.

```
1  public void createCallEvent(String URL, long threadMillis, String
    string, boolean isTalk) {
2      final String ANSWER_URL = URL;
3      CallEvent call = client
4          .getVoiceClient()
5          .createCall(new Call(TO_NUMBER, VONAGE_NUMBER, ANSWER_URL)
6              );
7
8      Thread.sleep(threadMillis);
9
10     final String UUID = call.getUuid();
11     final String STRING = string;
12
13     if (isTalk) {
14         client.getVoiceClient().startTalk(UUID, STRING,
15             TextToSpeechLanguage.AMERICAN_ENGLISH);
16     } else {
17         client.getVoiceClient().sendDtmf(UUID, STRING);
18     }
19 }
```

---

<sup>3</sup>Similar code also appears in further examples that show other different usages of the Vonage Voice API [120], such as muting or transferring a call.

## Listing 5.3: Utility method - Vonage API.

Making use of utility methods when writing API code examples can improve their maintainability and make them more modular and reusable; however, following and understanding such code examples might not be straightforward for API users, particularly newcomers. This is because API users will frequently need to jump between different utility method definitions to fully comprehend the illustrated API usage. In addition, users will have to copy and paste multiple methods (instead of a single, linear, self-contained snippet) to adapt the whole API usage into their codebases. Also, as discussed in Section 2.6.3, an increase in the number of independent paths, such as method calls, leads to higher code complexity, as measured by the Cyclomatic Complexity metric. Thus, as assessed in Chapter 4, the number of method calls or jumps that API users make has a direct impact on their comprehension and reuse of a given code example. Although linearising code involves inlining utility methods and can reduce modularity, this approach reflects the linear code principle, which favours code examples that can be read sequentially and reused without navigating between separate method definitions.

The objective of the proposed linear code synthesiser is to empower API developers to minimise the amount of repetition when writing API usage examples by allowing them to encapsulate shared behaviours into modular and reusable utility methods, and then automatically refactor and inline the calls to these utility methods to produce simple, linear and clean API code examples for API users.

The linear code synthesiser comes in the form of an Eclipse plug-in to facilitate its use. As described in detail in Algorithm 1, this synthesiser takes as input a single non-linear API code example, then, as shown in Figure 5.1, the code example (i.e. the Java source code file) passes through four main

stages: (1) code analysis, (2) code transformation, (3) code processing and (4) code generation. The following explains each of these stages in more detail.

### 5.1.1 Code Analysis

The proposed linear code synthesiser is based on the static analysis of annotated source code examples and abstract syntax tree (AST) parsing; thus, API developers are required to use two main Java annotations when writing non-linear usage examples, i.e. `@Documentation` to the methods that demonstrate a certain API usage and contain non-linear code (line 2 in Listing 5.4) and `@Utility` to the methods that encapsulate reusable code intended to be linearised (e.g. Listing 5.3). These Java annotations assist the static code analyser in distinguishing between the code segments (methods) that serve as documentation for specific API usage and those that are reusable and can be called in many API usage scenarios.

```
1 public class SendDtmfToCall {
2     @Documentation
3     public static void main(String[] args) {
4         configureLogging();
5         // ... some code
6         createCallEvent("https://nexmo-community.../long-tts.json",
7                         20000, "332393", false);
8     }
```

Listing 5.4: Documentation method - Vonage API - SendDtmfToCall.java

First, the selected API code example (a single .java file) is parsed to retrieve its type-resolved AST. Next, this AST is passed to a visitor i.e. `MethodDeclarationVisitor` that analyses and marks all the methods that are annotated with `@Documentation` or `@Utility` (e.g. line 3 in Listing 5.4 and line 1 in Listing 5.3), and it locates all the calls to utility methods, along with their bindings, that are found in the definitions of the marked methods.



## 5.1. Linear Code Synthesiser's Features

---

### Algorithm 1 Algorithm for linear code synthesis

---

```

1: Input: NonLinearJavaSourceCode
2: Output: LinearisedJavaSourceCode
3: Initialize ASTParser, MethodDeclarationVisitor, MethodDeclarationTransformer
4: procedure LINEARISEJAVASOURCECODE(NonLinearJavaSourceCode)
5:   DocumentationMethods  $\leftarrow$  PARSEJAVASOURCE(NonLinearJavaSourceCode)
6:   UpdatedASTs  $\leftarrow$  INLINEUTILITYMETHODS(DocumentationMethods)
7:   CleanedCode  $\leftarrow$  POLISHLINEARCODE(UpdatedASTs)
8:   LinearisedJavaSourceCode  $\leftarrow$  GENERATELINEARCODE(CleanedCode)
9:   return LinearisedJavaSourceCode
10: end procedure
11: procedure PARSEJAVASOURCE(NonLinearJavaSourceCode)
12:   Use ASTParser to parse the NonLinearJavaSourceCode to generate a type-resolved AST.
13:   Use MethodDeclarationVisitor to locate methods annotated with @Documentation.
14:   return DocumentationMethods
15: end procedure
16: procedure INLINEUTILITYMETHODS(DocumentationMethods)
17:   for each docMethod in DocumentationMethods do
18:     Traverse the method using MethodDeclarationTransformer to locate utilityMethodCall annotated with @Utility.
19:     for each utilityMethodCall in docMethod do
20:       if utilityMethodCall contains nestedUtilityMethodCalls then
21:         Inline the nestedUtilityMethodCalls first.
22:       end if
23:       Inline the utilityMethodCall into the docMethod.
24:     end for
25:   end for
26:   return UpdatedASTs
27: end procedure
28: procedure POLISHLINEARCODE(ASTs)
29:   Detect redundant or dead code generated by the inlining process.
30:   Eliminate temporary or intermediate variables created during the refactoring.
31:   Update the ASTs.
32:   return CleanedCode
33: end procedure
34: procedure GENERATELINEARCODE(CleanedCode)
35:   Remove unnecessary import statements and annotations from CleanedCode.
36:   Generate the final LinearisedJavaSourceCode from CleanedCode.
37:   Store LinearisedJavaSourceCode in a separate package.
38:   return LinearisedJavaSourceCode
39: end procedure

```

---

This Java source code parsing and analysis is done using Eclipse JDT.

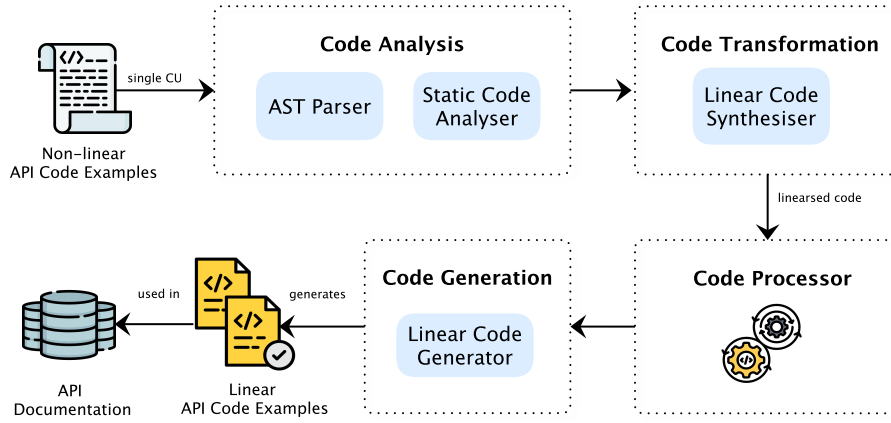


Figure 5.1: Linear code synthesiser architecture.

### 5.1.2 Code Transformation

The linear code synthesis is done by a `MethodDeclarationTransformer`, which takes each documentation method found in the received `CompilationUnit` and automatically inlines all the calls to utility methods found in its definition. The transformer works recursively, which means that it programmatically does the following:

1. It traverses the received documentation method.
2. Inlines the first encountered instance of a utility `MethodInvocation`.
3. Once the inlining is complete, it updates the `MethodDeclaration` AST node of the being-processed documentation method.
4. Updates the whole compilation unit since the offset positions of the next utility invocations (if any exist) have changed as a result of the newly inserted/inlined code.

## 5.1. Linear Code Synthesiser's Features

5. It repeats all previous steps for the next encountered utility method call, but this time on the updated version of the compilation unit.

During step 2, the transformer also checks whether the body of the called utility method contains any nested calls to other utility methods that need to be in-lined and processed first. All of these steps are done on a copy of the original `CompilationUnit`.

The transformer utilises the refactoring capabilities of Eclipse JDT (i.e. `InlineMethodRefactoring`) to inline utility method calls. This *inline method* refactoring technique works by replacing a method call with its body. The rationale for choosing JDT is that it is capable of ensuring that no syntax errors will result from this code transformation process. For example, if the definitions of the documentation and utility methods contain variables with identical names that would clash with each other after the inlining process is complete, it automatically renames one of them.

In addition, since the transformer reuses one of JDT's automated refactorings, it complies with several constraints imposed by JDT, such as preventing calls to recursive utility methods from being inlined, as this can result in an infinite loop or a stack overflow error. More on these constraints is discussed in the following sub-sections.

### 5.1.3 Code Processor and Generator

The main purpose of the code processor is to polish the resulting linear code by detecting and eliminating any redundant or dead code (i.e. source code that is never executed). This is necessary since the inlining process done by the code transformer can generate some pieces of unreachable code that would negatively impact the readability of the generated API usage example. For example, consider inlining the method call in line 6 in Listing 5.4; this will result in the body of the if-statement in the `createCallEvent` method (lines 12–14) in Listing 5.3 being dead since the received boolean value is

‘false’. Similarly, if the passed boolean value were ‘true’, the body of the else-statement would be unreachable.

Since detecting and eliminating dead code is a common compiler optimisation process, the linear code synthesiser utilises the problem reports noted by JDT’s compiler, which include a rich description of all the standard Java problems, to retrieve the locations of all dead/unreachable code problems. Then, to eliminate the code causing these problems, the code processor replicates the JDT’s implementation of the dead code removal quick-fix and rewrites the whole AST after the problems are resolved.

To improve the readability of the generated linear usage examples, the code processor also discards all unnecessary code statements. This is essential because in some cases, JDT’s *inline method* refactoring may create temporary/intermediate variables to hold the return value of the inlined method call when it is assigned to another variable. For example, consider the following simple `getFullName` utility method that takes two string parameters representing a person’s first and last name and returns their full name:

```

1 @Utility
2 public static String getFullName(String firstName, String lastName) {
3     String fullName = firstName + " " + lastName;
4     return fullName;
5 }

```

If the return value of this method is assigned to a variable named `myName`:

```

1 String myName = getFullName("John", "Smith");

```

Then, applying the *inline method* refactoring to replace the above method call with the actual body of the method will generate the following:

```

1 String fullName = "John" + " " + "Smith";
2 String myName = fullName;

```

The variable `fullName` in the above code snippet is not needed and can be eliminated altogether, and the concatenated string can be used directly. This

## 5.1. Linear Code Synthesiser’s Features

removal can make the code more concise and easy to read, especially in more lengthy and complex API usage examples. Furthermore, this code readability enhancement can be done by analysing the code example to identify any variables that are assigned but never used or variables that are used only once and can be replaced with the value they hold.

Once all documentation methods are inlined and polished, including the removal of no-longer-needed import statements and annotations, the entire example is forwarded to a linear code generator (code generation stage in Figure 5.1) that generates and stores the linear API code example in a separate package under the source folder of the active Java project.

Another set of real-world API code examples from the collected corpus of usage examples<sup>4</sup>, restructured using the proposed linear code synthesis approach is shown in Listings 5.5 and 5.6.

```
1 public class DisableDateAsTimestamps {
2     public static void main(String[] args) throws IOException {
3         defaultOutput();
4         disableDateAsTimestamps();
5     }
6
7     private static void defaultOutput() throws IOException {
8         Date date = new Date();
9         ObjectMapper om = new ObjectMapper();
10        String s = om.writeValueAsString(Map.of("myDate", date));
11        System.out.println(s);
12    }
13
14    private static void disableDateAsTimestamps() throws IOException {
15        Date date = new Date();
16        ObjectMapper om = new ObjectMapper();
17        om.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
18        String s = om.writeValueAsString(Map.of("myDate", date));
19        System.out.println(s);
20    }
```

---

<sup>4</sup>This dataset was created from real-world API code examples to evaluate the proposed approach and is available in the [replication package](#). More details are provided in Section 5.3.

21 }

Listing 5.5: Jackson API code example (1) - DisableDateAsTimestamps.java

```

1  public class DisableFailOnEmptyBeans {
2      public static void main(String[] args) throws IOException {
3          defaultOutput();
4          disableFailOnEmptyBeans();
5      }
6
7      private static void defaultOutput() throws IOException {
8          MyEmptyObject myObject = new MyEmptyObject();
9          ObjectMapper om = new ObjectMapper();
10         String s = om.writeValueAsString(myObject);
11         System.out.println(s);
12     }
13
14     private static void disableFailOnEmptyBeans() throws IOException {
15         MyEmptyObject myObject = new MyEmptyObject();
16         ObjectMapper om = new ObjectMapper();
17         om.disable(SerializationFeature.FAIL_ON_EMPTY_BEANS);
18         String s = om.writeValueAsString(myObject);
19         System.out.println(s);
20     }
21 }

```

Listing 5.6: Jackson API code example (2) - DisableFailOnEmptyBeans.java

The two examples shown above are from the Jackson Java API<sup>5</sup> and are very similar in both structure and logic, containing duplicated code (i.e., lines 9–11 and 16–19 in both Listings 5.5 and 5.6). By applying the linear code synthesis approach, the duplicated code was extracted into a single utility method, as shown in Listing 5.7, and the examples were rewritten and shortened, as presented in Listings 5.8 and 5.9.

```

1  @Utility
2  public static void serialize(Object value, boolean withfeature,
        SerializationFeature sFeature, boolean withJsonInclude, boolean
        withFilter, SimpleFilterProvider filterProvider) throws
        JsonProcessingException {

```

<sup>5</sup><https://github.com/FasterXML/jackson>

## 5.1. Linear Code Synthesiser's Features

```
3   ObjectMapper om = new ObjectMapper();
4   if (withfeature && sFeature != null) {
5       om.disable(sFeature);
6   }
7   else if (withJsonInclude) {
8       om.setDefaultPropertyInclusion(JsonInclude.Include.NON_DEFAULT);
9   }
10  else if (withFilter) {
11      om.setFilterProvider(filterProvider);
12  }
13  String jsonString = om.writeValueAsString(value);
14  System.out.println("-- after serialization --");
15  System.out.println(jsonString);
16 }
```

Listing 5.7: Utility method - Jackson API.

```
1  public class DisableDateAsTimestamps {
2
3      public static void main(String[] args) throws IOException {
4          defaultOutput();
5          disableDateAsTimestamps();
6      }
7
8      @Documentation
9      private static void defaultOutput() throws IOException {
10         Date date = new Date();
11         serialize(Map.of("myDate", date), false, null, false, false,
12                     null);
13     }
14
15     @Documentation
16     private static void disableDateAsTimestamps() throws IOException {
17         Date date = new Date();
18         serialize(Map.of("myDate", date), true, SerializationFeature.
19                     WRITE_DATES_AS_TIMESTAMP, false, false, null);
20     }
21 }
```

21 }

Listing 5.8: Rewritten Jackson API code example (1) - DisableDateAsTimestamps.java

```

1  public class DisableFailOnEmptyBeans {
2
3      public static void main(String[] args) throws IOException {
4          defaultOutput();
5          disableFailOnEmptyBeans();
6      }
7
8      @Documentation
9      private static void defaultOutput() throws IOException {
10         MyEmptyObject myObject = new MyEmptyObject();
11         serialize(myObject, false, null, false, false, null);
12     }
13
14     @Documentation
15     private static void disableFailOnEmptyBeans() throws IOException {
16         MyEmptyObject myObject = new MyEmptyObject();
17         serialize(myObject, true, SerializationFeature.
18             FAIL_ON_EMPTY_BEANS, false, false, null);
19     }
20 }

```

Listing 5.9: Rewritten Jackson API code example (2) - DisableFailOnEmptyBeans.java

## 5.2 Synthesiser's Constraints

When writing or calling utility methods, API developers should keep in mind a few cases where the inlining and generation of linear code examples may not be possible due to some constraints imposed by the reused JDT refactoring technique (i.e. *inline method*). These cases include inlining recursive methods (e.g., `factorial(n)`), which JDT automatically and internally detects by analysing the selected method body and resolves by aborting the inlining



process, leaving the invocation unchanged and producing the error message ‘Method declaration contains recursive call’. Similarly, nested method invocations (e.g., `printSum(calculate(x,y))`) and invocations of methods that have multiple return statements cannot be inlined. This is because inlining such methods would be complicated or could lead to an infinite inlining process.

### 5.3 Evaluation

The linear code synthesiser was evaluated on two aspects: (1) its effectiveness in reducing repetition in API code examples; and (2) the benefits of the proposed approach for API developers, assessed by analysing the extent of duplicate code in the evaluated API code examples. Particularly, the aim was to answer the following research questions (RQs):

**RQ1:** How often do API code examples contain duplicate or near-duplicate code that can be eliminated using the proposed linear code synthesis approach?

**RQ2:** How much reduction of duplicate code is achieved by the proposed linear code synthesis approach?

**RQ3:** How often are duplicate code fragments repeated across different API code examples?

To answer these RQs, the five-step evaluation process shown in Figure 5.2 and explained in more detail below was followed.

#### 5.3.1 Data Collection

To evaluate the linear code synthesis approach, a dataset of API code examples was required. These examples needed to be self-contained, compilable and grouped based on the Java library to which they belonged. Additionally,

## Chapter 5. Linear API Usage Example Synthesis

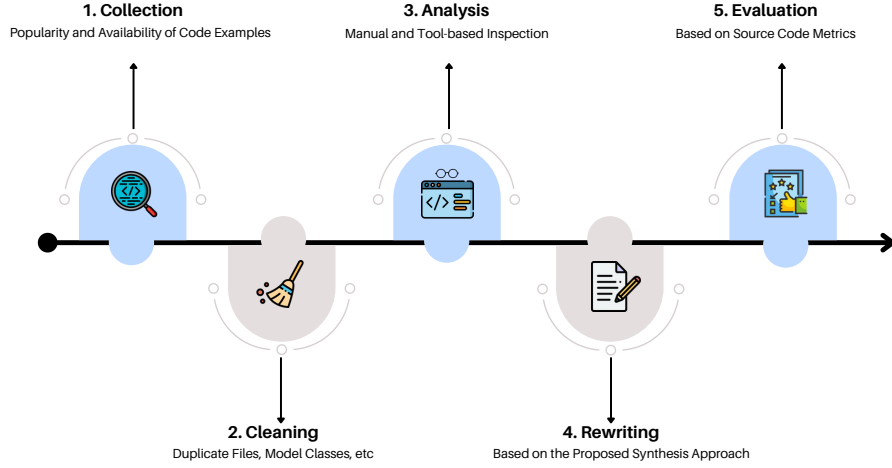


Figure 5.2: API code examples collection method.

these examples needed to be explicitly provided or referenced by the developers of the API to demonstrate certain API usages. This is because the evaluation aimed at understanding the current state of API code examples and measuring the amount of code repetition found in them. By doing so, some insight could be gained into the effort required to create and maintain such examples, as well as into assessing how the proposed approach could contribute to mitigating this effort.

Existing code example datasets (e.g. CodeSearchNet[123]), however, did not meet the evaluation criteria because they were extracted either from client applications available on public repositories or from online sites, such as Stack Overflow[49, 124], and not solely from code examples that are specifically written to document an API. Therefore, these datasets were unsuitable, since the evaluation aimed at exploring the state of API code examples only. For the same reasons, the code examples extracted from code search engines, such as SearchCode [125], were not used. Therefore, a new evaluation dataset was built.

To build a suitable dataset, API code examples from several popular

### 5.3. Evaluation

and open-source Java projects available on GitHub [126], as well as other similar documentation sources were collected. As GitHub contains a large number of inactive/personal projects, which cannot be an adequate data source for software engineering studies [127], the library selection and the example retrieval were based on the following criteria: (1) the popularity and activity level of the library, (2) the availability of complete/compilable code examples in its repository and (3) its domain.

The popularity of a library can be measured by the number of stars on its GitHub repository or the number of artefacts that use it on Maven Central Repository; therefore, only Java projects with at least 50 stars or those used by at least 30 other projects were considered. Moreover, to select only the Java projects likely to be active, only those with a commit within the past three months (prior to this study) and with at least five contributors were extracted. The code examples were extracted from three sources: (1) the official website of the library, (2) the ‘samples’ or ‘examples’ folder found inside the library’s source code directory on GitHub or (3) the external usage tutorials listed in the description of the library’s GitHub page. This ensured that the selected libraries were mature and well-established with good documentation and usage information. Finally, the diversity of the selected libraries was ensured by targeting different domains, such as database access, web development, and data processing.

To facilitate the search for GitHub projects and due to the limitations of GitHub search API [128], such as the limited number of requests and search results, the GitHub Search (GHS)[129], a recently published and well-known search engine<sup>6</sup> and dataset for mining GitHub repositories, was used. GHS contains 735,669 repositories written in 10 programming languages and provides a set of 25 characteristics, such as the number of stars, commits and contributors and whether the project has a wiki, all of which can be used

---

<sup>6</sup><https://seart-ghs.si.usi.ch>

## Chapter 5. Linear API Usage Example Synthesis

as search filters. After applying the evaluation selection criteria on GHS, the resulting projects for any code example folders or tutorial links were inspected. Finally, 7 Java libraries from different domains were randomly selected from the resulting corpus. Also, two more standard Java APIs (i.e. JDBC and Java Applets),<sup>7</sup> which are widely recognised as popular Java APIs and frequently used in existing literature [130, 36], were selected. Some information and statistics about the selected Java libraries and packages are summarised in Table 5.1.

---

<sup>7</sup>Despite the fact that Java Applets are an obsolete technology, it can still be considered an extensively documented API.

Table 5.1: Statistics of the selected Java libraries and packages.

Library	Source of Examples	Stars	Used by	# of Examples (raw)	# of Examples (after cleaning)	$\approx$ Median Example Size (in LOC)*
Vonage	<a href="#">GitHub</a>	82	-	98	98	29
Jackson	<a href="#">GitHub</a> and <a href="#">LogicBig</a>	8.1k	25.4k	179	81	20
JAXB	<a href="#">GitHub</a>	165	5.5k	80	49	23
Eclipse Epsilon	<a href="#">GitHub</a>	-	256	18	16	32
JavaParser	<a href="#">GitHub</a>	4.6k	558	29	20	22
Java Applet	Oracle <a href="#">Java Documentation</a>	-	-	20	17	40
JDBC	Oracle <a href="#">Java Documentation</a>	-	-	19	19	113
gRPC	<a href="#">GitHub</a>	10.4k	3.9k	56	45	70
PDFBox	<a href="#">GitHub</a>	1.9k	646	101	84	86
<b>Total</b>				<b>600</b>	<b>429</b>	

\* Lines of code (LOC).

### 5.3.2 Data Cleaning

Through the cleaning process, the aim was to reduce noise in each subset (Java library) of the dataset and focus the analysis on relevant API usage examples only. Thus, unnecessary testing code and identical copies, as well as all the Java source code files that had no behaviour or did not demonstrate a certain usage of an API were removed. For example, model classes, interfaces, package-info files, class files with minimal code (e.g. a `toString()` method) and class files that were only created to be parsed or manipulated were all excluded. Furthermore, since some libraries provided examples in the form of Java projects, some Java file names were recurrent; thus, renaming or labelling such files made them more easily identified. All of these actions were essential to ensure that the dataset was clean and consistent, which was important for facilitating the subsequent analysis and evaluation steps.

### 5.3.3 Data Analysis and Similarity Detection

In this step, each subset was individually inspected. The aim was to look for duplicate and near-duplicate code that was similar in structure, and functionality and could benefit from the proposed synthesis approach. This means that only the code instances/fragments that might have been copied, modified and reused in more than one API code example were detected. To facilitate this similarity detection step, a code similarity detector (JPlag),<sup>8</sup> which compares sets of source code files to find similarities between them (pairwise comparison), was used. JPlag [131, 132] is an open-source token-based similarity detector for Java and many other programming languages. It can detect exact and modified code clones at various levels of granularity and provides a web-based interface for viewing and navigating the generated results. Moreover, to reduce the probability of getting false positives, a

---

<sup>8</sup><https://github.com/jplag/JPlag>

similarity threshold of 60% was used, as it is one of JPlag’s cutoff threshold values that can yield the best results [131]. Experimentation with different threshold values revealed that high values (e.g. 100%) can yield only exact copies rather than adapted ones, while those between 70% and 90% can miss some genuine clones. Also, JPlag’s default minimum token match value for Java (i.e. 9) was kept. After running JPlag on the selected subset, a manual inspection of the resulting clones was completed to determine if they could be refactored and used as input for the proposed linear code synthesiser.

#### 5.3.4 API Code Examples Rewriting

After analysing and inspecting the API code examples for similarities, copies of those examples containing applicable similarities<sup>9</sup> were taken and manually rewritten based on the structure of the code synthesis approach. This means that the repetitive/similar code was factored out in a set of reusable/utility methods (in a standalone utility class) which were then called wherever needed. The required Java annotations (i.e. `@Documentation` and `@Utility`) were also added.

During the rewriting process, the same coding style used in the original examples was maintained. This is important because once the examples are linearised again (when passed as input and automatically linearised using the proposed code synthesiser), they should match the original versions as closely as possible. However, some examples within the same subset used different coding flavours interchangeably (e.g. creating objects using builders, factories or direct construction). In such cases, the most dominant coding style was chosen when similar code was factored out to a utility method, as long as the code functionality remained unaltered.

---

<sup>9</sup>The phrase ‘applicable similarity’ is used to refer to the type of code similarity that can be eliminated using the API code example synthesis approach.

### 5.3.5 API Code Examples Evaluation

In this step, the conciseness of the rewritten versions of the examples was evaluated. This was achieved by comparing the rewritten examples to the original ones. This comparison was based on computing the relative percentage decrease (Formula 5.1) in the non-comment, non-blank lines of code (LOC) in each API code example. LOC (also known as SLOC) is a common metric for measuring the size of a software program [78], which was calculated using *cloc*,<sup>10</sup> an open-source command-line tool for counting the physical LOC in software projects.

$$\text{PercentageDecrease} = \frac{LOC_{\text{Original}} - LOC_{\text{Rewritten}}}{LOC_{\text{Original}}} \quad (5.1)$$

This evaluation can indicate the amount of repetition in API code examples that has been eliminated as a result of applying the API code example synthesis approach, while still retaining the examples' functionality.

## 5.4 Results and Discussion

As discussed in Section 5.3, the evaluation was based on three main research questions. RQ1 is mainly concerned with identifying the number of API code examples that include the type of similarity that can be eliminated by the proposed linear example synthesiser. This means that only the API code examples with similar code fragments that had been slightly modified by API developers to adapt and document new usage examples were searched for. Furthermore, it was also important to notice whether the values of the code elements that differ between these code fragments can either be passed as arguments to methods in Java, such as primitive data types and `String`

---

<sup>10</sup><https://github.com/AlDanial/cloc>



## 5.4. Results and Discussion

(more on this in Section 5.5) are controllable using a conditional statement.<sup>11</sup> As listed in Table 5.2, four out of the nine scanned Java projects contained over 40% API code examples with duplicate/near-duplicate code that could be factored out and reduced using the proposed synthesis approach. The duplicate code manifested several patterns, such as similar interface implementation and type instantiation. The mean percentage of such examples (across all the evaluated Java libraries) is 36.33% and the median percentage is 36%. These results show that the percentage of examples containing applicable similarity varied across the evaluated Java libraries, thus making it difficult to draw a definitive conclusion about which characteristics make a library more likely to benefit from the proposed synthesis approach. This is true since the repetitiveness in code examples is highly impacted by the coding style/flavour API developers prefer when writing code examples. It is also worth mentioning that four of the selected Java libraries (Epsilon, Vonage, PDFBox and JDBC) already contained a set of utility methods<sup>12</sup>,<sup>13</sup> for some repetitive functionalities, which could indicate API developers' desire for reducing repetitiveness in API code examples.

**Answer to RQ1:** The percentages of the API code examples that contained duplicate code that could be eliminated using the proposed approach ranged between 18% and 56% in the selected Java libraries.







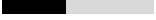
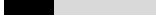
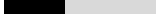
---

<sup>11</sup>As shown in the code example in Section 5.1.

<sup>12</sup><http://bit.ly/43nwQ10>

<sup>13</sup><http://bit.ly/4kVmGz4>

Table 5.2: Summary of the evaluation results.

Library	% Applicable*	% Median Decrease (in LOC)	# of Extracted Utility Methods	# of Utility Method Calls (Total)
Vonage	18 (18%) 	15%	9	22
Jackson	29 (35%) 	19%	4	34
JAXB	18 (36%) 	14%	2	28
Eclipse Epsilon	9 (56%) 	26%	3	11
JavaParser	4 (20%) 	8%	3	6
Java Applet	8 (47%) 	11%	5	28
JDBC	8 (42%) 	9%	4	18
gRPC	15 (33%) 	3%	4	19
PDFBox	34 (40%) 	7%	9	60

\* Linear examples with repetitive code.

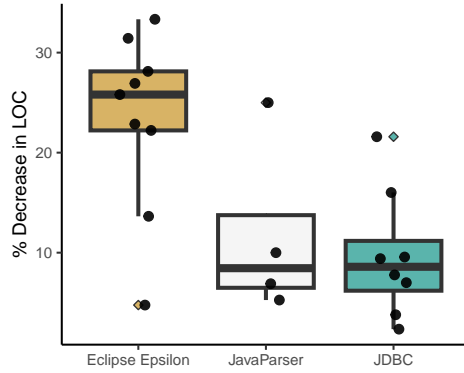
## 5.4. Results and Discussion

For RQ2, the interest was in measuring the decrease in duplicate code that is achieved by the code synthesis approach. After obtaining all the API code examples that can benefit from the approach, each subset (i.e. each Java library set of examples) was inspected to calculate the reduction in the size of each of its code examples which resulted from applying the code synthesis approach. As shown in Table 5.2 and illustrated in Figure 5.3, the percentages of decrease in the LOC of many of the evaluated API code examples are scattered between 15% and 37% in four of the selected Java projects (Eclipse Epsilon, JAXB, Vonage and Jackson). Precisely, over 50% of the evaluated API code examples in three of these Java libraries (Eclipse Epsilon, Vonage, and Jackson) had their LOC reduced by more than 15%, while only 33% of the examples in JAXB showed a similar reduction.

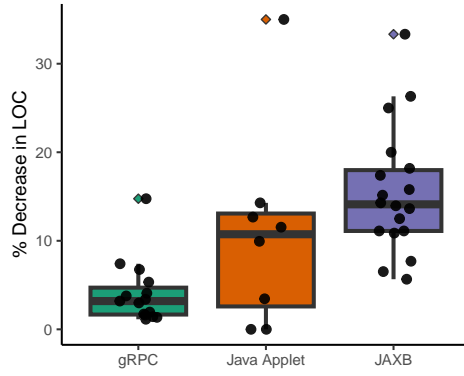
**Answer to RQ2:** The proposed linear code synthesis approach brought more than a 30% decrease in LOC in five of the nine evaluated Java projects.

It is not only the amount of reduction of duplicate code in a single code example that matters but also the frequency of that duplicate code within the set of examples. In some cases (e.g. code examples in the JDBC subset), the elimination of some duplicate code fragments resulted in a low amount of reduction in the size of the examples, but those code fragments were extensively recurrent across the whole subset. Thus, the aim for RQ3 was to understand the number of times a single repetitive code fragment is repeated across the other API code examples within the same set of examples (i.e. across the API code examples of the same Java library). This information is important since it can partially indicate the effort the API developers made and the number of times they changed those code fragments to make them

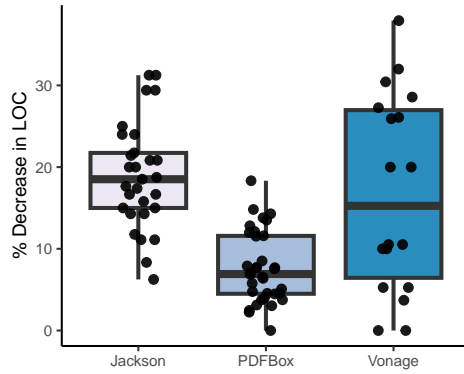
## Chapter 5. Linear API Usage Example Synthesis



(a)



(b)



(c)

Figure 5.3: Distributions of the evaluated API code examples and the percentage decrease in their Lines of Code (LOC). Each boxplot aggregates the API examples of each Java library.

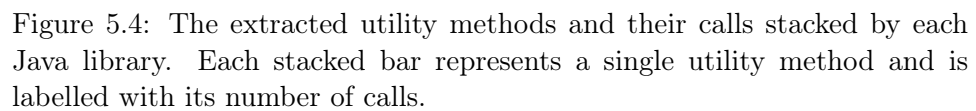
## 5.4. Results and Discussion

fit a new API code example. It could also provide more granular indicators of the need for these examples to be refactored and made more modular and maintainable as well as easy to modify when the API evolves over time. This information can in turn reinforce the significance of the API code example synthesis approach.

As presented in Table 5.2 (column four) and illustrated in Figure 5.4, a considerable number of repetitive code fragments (i.e. utility methods) were extracted from the subsets of API code examples. For example, nine duplicate code fragments were extracted from Vonage, five from Java Applet and nine from PDFBox. However, the number of extracted utility methods does not necessarily indicate that the library example subset contains a large amount of repetitive code. For example, seven out of the nine extracted utility methods in Vonage API were only called twice; whereas, in JDBC for instance, the four extracted utility methods were called six, five, four and three times respectively. It is also worth mentioning that in some cases, the large number of calls to a single utility method (e.g. the utility method called 26 times in the Jackson example subset shown in Figure 5.4) is expected since the utility method encapsulates a main functionality of the library itself (i.e. serialising an object into a JSON string). In addition, as shown in Figure 5.5, it seems like there was no clear correlation between the number of extracted API code examples that contain applicable code similarities and the number of extracted utility methods.

**Answer to RQ3:** Overall, a substantial number of duplicate code fragments were recurrent in many of the API code examples subsets.

While no hard rules can be established regarding when to apply the transformation, some tentative guidelines can be drawn. In practice, the decision may rest with API developers. Based on the results, transformations appear more beneficial for relatively simple code examples, where linearisation can



Besides the JDT-imposed constraints discussed in Section 5.2, it was found that the current version of the proposed prototype is not fit for all the types of code similarity found in the evaluated API code examples, and that further

## 5.5. Limitations and Observations

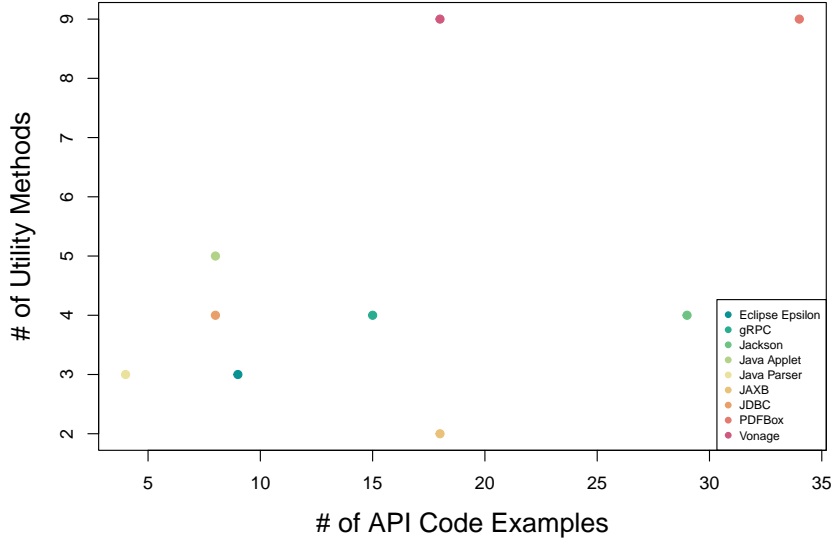


Figure 5.5: A scatter plot showing the relationship between the number of extracted utility methods and the number of API code examples containing an applicable code similarity.

extensions are needed to cover these types. The following sections explain these limitations in more detail and suggest suitable solutions.

### 5.5.1 Template-based Code Synthesis

The proposed prototype was designed to reduce any type of code repetition. Additionally, the more condensed examples (i.e. non-linear examples) used in the proposed approach were intended to be valid Java code, allowing API developers to benefit from error checking, code completion, and other similar features. However, due to constraints imposed by the syntax of Java, it was not possible for the proposed prototype to accommodate all the detected patterns of code similarity/repetition. For example, if we consider the two real-world code snippets shown in Listings 5.10 and 5.11, we can see that

they share a very similar pattern of code<sup>14</sup> and could benefit from some kind of encapsulation; however, the elements that differ between them (other than the strings passed at line 3 in both Listings) are the variable types and names as well as the method calls at line 4 in both listings.

```

1 // Create an action when exiting
2 PDActionJavaScript jsAction = new PDActionJavaScript();
3 jsAction.setAction("app.alert(\"On 'exit' ...\")");
4 annotationActions.setX(jsAction);

```

Listing 5.10: PDFBox code example (1) - FieldTriggers.java.

```

1 // Create an action when the field value changes
2 PDActionJavaScript jsAction = new PDActionJavaScript();
3 jsAction.setAction("app.alert(\"On 'recalculate' ..\")");
4 fieldActions.setC(jsAction);

```

Listing 5.11: PDFBox code example (2) - FieldTriggers.java.

These types of code elements can neither be controlled nor passed as arguments to a utility method; thus, it is not feasible for the proposed synthesis prototype to reduce such repetitive patterns. Therefore, an extension that allows a template-based linear example code synthesis is required to address this challenge and reduce this type of code repetition. More details are provided in Section 7.4.

A list of the evaluated API code examples that contained some patterns of a template-suitable code similarity is shown in Table 5.3.

### 5.5.2 Interactive API Usage Example Generation








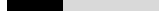

Another direction for exploration would be to allow API users to interactively specify some values against which they want to run the API code example generator. The assumption behind this interactive approach is that having a set of many pre-generated and static API code examples can make it difficult

---

<sup>14</sup>This code pattern appears ten times in the same code example (i.e. FieldTriggers.java in the PDFBox library example subset). Other types of template-suitable code patterns from the other evaluated Java libraries are available in the replication package.



Table 5.3: Number of API code examples with template-suitable code similarity patterns.

Library	# of API Code Examples	
Vonage	25 (25%)	
Jackson	16 (19%)	
JAXB	7 (14%)	
Eclipse Epsilon	6 (37%)	
JavaParser	4 (20%)	
Java Applet	4 (23%)	
JDBC	5 (26%)	
gRPC	17 (37%)	
PDFBox	6 (7%)	

for API users to find and reuse the examples that suit their needs. Having this interactive way of generating examples could facilitate this process, thus promoting API learnability. Further details are provided in Section 7.4.

## 5.6 Threats to Validity

Several threats to validity were identified during the evaluation of the proposed linear code synthesis prototype. The following sections discuss these threats and outline the steps taken to mitigate them.

### 5.6.1 Construct Validity

One potential threat to the construct validity is the possibility that the chosen similarity detector (JPlag) may have missed some instances of similarity (false negatives) between the API code examples. To mitigate this threat, a manual review of the detection results was conducted to identify any other instances of undetected similarity. Additionally, a common and widely accepted measure of program size (LOC) was used, which was clearly defined and consistently applied to measure the size of both original and rewritten API code examples. Also, the risk of subjective bias and human error was reduced by using an automated tool to perform the calculation of LOC in API code examples.

### 5.6.2 Internal Validity

One potential threat to internal validity arises from inconsistencies in coding and formatting styles between the two versions of the API code examples (i.e. original and rewritten), which could significantly influence the program size metric used in this study. This threat was mitigated by ensuring consistent coding and formatting styles across both versions of the API code examples.

### 5.6.3 External Validity

A possible threat to the external validity is the limited diversity of the dataset. To reduce this validity threat, API code examples were selected based on specific criteria and different domains in Java were targeted. This can help to promote the representativeness of the dataset and the generalisability of the findings. However, to gain more confidence in the generalisability of the results, the linear code synthesiser would benefit from further evaluation using a larger and more diverse dataset of API code examples. Additionally, more investigation is required to establish the applicability of the proposed synthesis approach to programming languages beyond Java.

## 5.7 Summary

This chapter introduced the main features of the proposed linear code synthesis approach, which aims to address the repetitiveness of API code examples. Key features, such as code transformation and linear code generation, along with the technologies used, were presented in Section 5.1 with real-world API code examples. The main constraints of the proposed approach were then briefly discussed in Section 5.2. Subsequently, the chapter outlined the key steps followed during the evaluation process of the proposed synthesis approach in Section 5.3, followed by a detailed discussion of the evaluation results in Section 5.4. Finally, the chapter concluded with an overview of

## 5.7. Summary

key observations and limitations, along with suggested solutions to overcome them, in Section 5.5.

## Chapter 6

# Coverage of API Code

## Examples

Since one of the aims of this PhD research is to alleviate the API developers' burden of writing API code examples for API documentation and empower them to produce more usage examples that can promote API learnability, we propose a method for defining what constitutes an intended API, along with an approach for measuring and reporting the coverage of this intended API by code examples. In a similar manner to unit test coverage, e.g. Java code coverage library (JaCoCo),<sup>1</sup> this coverage analysis tool measures the percentage of the API elements (e.g. methods and classes) that are illustrated in examples and highlights any instances that are not yet used in any examples. The main aim of these methods is to help API developers define their intended APIs using a proposed custom API description language and to ensure that most of the described API components are demonstrated in examples. This is particularly crucial because research [43] shows that producing a large number of API code examples does not necessarily ensure high API coverage, as these examples may contain significant repetition as explained in Section 2.3.2.

---

<sup>1</sup><https://www.jacoco.org/jacoco/>

All implementation code, data, results, and R scripts discussed or used in this chapter are in a publicly-available replication package.<sup>2</sup>

## 6.1 APIExCov's Features

As illustrated in Figure 6.1 and described in Algorithm 2, the proposed API code example coverage approach (APIExCov) requires a structured and concise description and representation of the intended API. It then needs a mechanism to verify the intended API specification against the actual codebase of the library to ensure that the specification is valid and consistent. Finally, it needs to measure and report coverage. Thus, the implemented tool takes as input three main components, which API developers wishing to check their API's code example coverage are expected to provide. The first component is the path of the API description, which is a YAML (YAML Ain't Markup Language) file containing all the API elements that are part of the intended API for a Java library or framework. These API elements are defined using a custom API description language developed specifically for this purpose. YAML was chosen as the data format for this API description due to its human readability and popularity as a commonly used format for writing configuration files and data exchange. The second and third components required as input for the APIExCov tool are the path to the source code of the Java library of interest, and the name of the package that contains the library's API code examples. Through all of these three components, API coverage is measured and reported.

After both the API description content and the Java project's source code are parsed and analysed, they are passed to another part of the APIExCov tool which is the *API Validator*. This important part is responsible for checking whether all described API elements in the YAML file of the API description are valid and actually exist in the actual source code of the

---

<sup>2</sup>Replication package: <https://figshare.com/s/f5eda91fedf91762b21c>

## Chapter 6. Coverage of API Code Examples

Java library of interest. It also ensures that the syntax used to describe the intended API adheres to the proposed custom API description language. Furthermore, once the API description is validated, an *Intended API Store* is generated and retained in memory to be later analysed and used, along with the API code examples, by a *Coverage Analyser*. As a final step, two types of coverage reports (i.e. textual and visual) are generated from an in-memory store of all the API code example coverage and detailed usage information. The following sections explain all of the APIExCov's components in more detail.

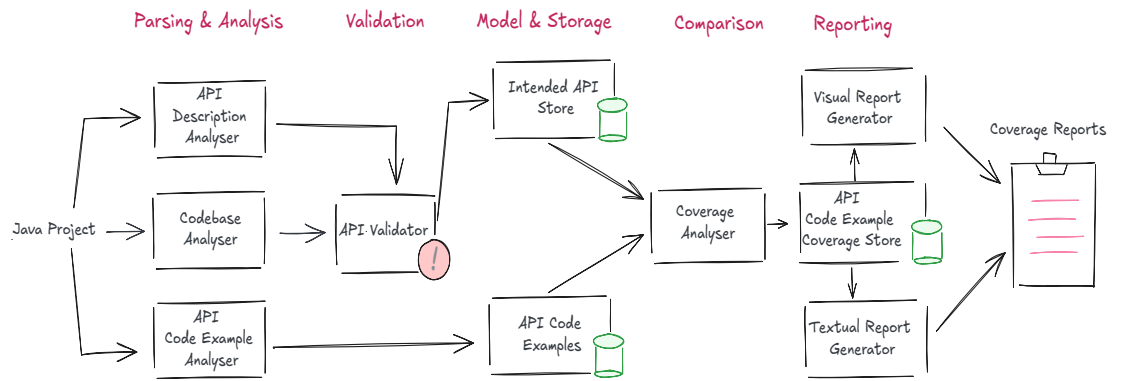


Figure 6.1: Architecture of API code example coverage tool.

### 6.1.1 Intended-API Description Language

One of the key challenges in this work is finding a way to define the source code elements, i.e. types, fields, constructors, and methods, that constitute

---

**Algorithm 2** Algorithm for measuring and reporting API code example coverage

---

```

1: Input: ProjectCodebase, APIDescription, APICodeExamples
2: Output: Textual and visual coverage reports
3: Initialize intendedAPIStore
4: procedure VALIDATEAPIELEMENTS(APIDescription, ProjectCodebase)
5:   Initialize logger
6:   for each apiElement in APIDescription do
7:     if apiElement syntax is valid according to custom ANTLR4 gram-
       mar then
8:       if apiElement exists in ProjectCodebase then
9:         Add apiElement to intendedAPIStore
10:      else
11:        Log error: 'apiElement not found in codebase' to logger
12:      end if
13:    else
14:      Log syntax error: 'invalid apiElement name' to logger
15:    end if
16:  end for
17: end procedure
18: Initialize apiCodeExampleAnalyser
19: for each code_example in APICodeExamples do
20:   apiCodeExampleAnalyser.parseAndAnalyse(code_example)
21: end for
22: Initialize coverageAnalyser
23: Initialize APICodeExampleUsageStore
24: coverageAnalyser.execute(intendedAPIStore, APICodeExamples)
25: for each apiElement in intendedAPIStore do
26:   for each code_example in APICodeExamples do
27:     if apiElement is found in code_example then
28:       APICodeExampleUsageStore.addUsageInfo()
29:     end if
30:   end for
31: end for
32: Generate a textual report based on APICodeExampleUsageStore
33: Generate a visual report based on APICodeExampleUsageStore

```

---

the API as intended by its developers. Moreover, differentiating between the “factual API”, which includes all the publicly accessible parts of a Java library, and the “intended API”, which refers to all the library components that are specifically created and documented for external use by the library’s clients, is not a straightforward task as they both can share some characteristics, e.g. being public. Therefore, a custom API description and specification language was developed to enable API developers to define the elements that comprise their APIs precisely. For this language, ANTLR4 (ANother Tool for Language Recognition)<sup>3</sup> is used as a parser generator. ANTLR is a robust tool that is widely used for creating parsers and interpreters for a variety of programming languages and structured text formats [133]. It allows developers to write a grammar that defines the structure and rules of their languages. Then, it generates code, i.e. a lexer, parser, and listener, that is based on the created grammar file (.g4), and that can read, analyse, and process any text written in the developed language. The reasons behind choosing ANTLR among the other parser generators available for Java are its popularity, flexibility in processing the parse tree, intuitive and easy-to-read grammar, and detailed documentation.

Unlike the internal DSL discussed in Chapter 5, an external DSL was chosen for this proposed API specification language. This decision was made to avoid polluting the API codebase with secondary, non-functional annotations. The aim was to keep production API source code clean and free from auxiliary metadata, while still enabling coverage descriptions through a separate specification artefact. In contrast, the use of an internal DSL for code generation directives in Chapter 5 was considered appropriate because these directives are embedded only in API code examples, which are relatively few in number and are already intended to include comments and explanatory content. Embedding annotations in examples was therefore re-

---

<sup>3</sup><https://www.antlr.org>



## 6.1. APIExCov's Features

garded as natural and not disruptive, whereas doing so in production API code would have unnecessarily cluttered the codebase.

In addition, this API specification language offers developers a concise syntax and a set of features designed to simplify the process of describing an API, making it less time-consuming. These features include wildcard matching, pattern matching, and inclusion and exclusion filtering with or without conditions. Moreover, these patterns and conditions are primarily used to describe API elements, such as methods and constructors, that can be difficult to identify and describe by name alone. This is because constructors and methods in Java are often overloaded and can have complex signatures that are tedious to specify manually. Java Types on the other hand are usually easy to refer to directly since their fully qualified names are singular, unique, and lack repetition.

As an example, Listing 6.1 shows a subset of the API description of the JSoup<sup>4</sup> Java library. This description contains the names of two packages (lines 2 and 26), one with two types (lines 4 and 18) and one with only one type (line 28). Names of methods and constructors that belong to each type are listed under their corresponding keys as list items. Furthermore, several items contain the asterisk symbol (\*) in their names (lines 8, 11, and 14), which serves as a wildcard, representing any sequence of characters. This wildcard can appear as a prefix (e.g. \*foo), postfix (e.g. foo\*), or even within the item name as an infix (e.g. foo\*bar). Also, this asterisk symbol (\*) is used to indicate that all the methods or constructors that belong to a particular type are included in the API description (e.g. line 20).

In addition, there are three descriptive options to distinguish between various versions of the same method or constructor (i.e. when an API element is overloaded). Firstly, the number of parameters can be represented using parentheses with comma-separated underscores, each denoting a single

---

<sup>4</sup><https://jsoup.org>

## Chapter 6. Coverage of API Code Examples

parameter (e.g. `Attribute (_,_,_)` at line 6). If the method to be described has no parameters, the parentheses can be left empty. Alternatively, if all versions of the same method are to be included in the description, listing the method name without parentheses is sufficient. Secondly, if the number of parameters is not enough to distinguish between two versions of the same API element, the parameter type can be used (e.g. `traverse(_,Node)` at line 31). Both simple and fully qualified parameter type names are supported. Finally, parameter types must be placed in the correct order to identify the right API element version and to distinguish between versions that have the same parameter types in a different order.

Listing the API element name, as explained above, indicates that it should be included in the API description. However, to exclude an API element, the minus sign (-) must be used as a prefix to the element's name (e.g. line 13). Moreover, if all the list items, i.e. all API element names, in a list start with a minus sign, it indicates that all the other API elements that belong to the described Java type, apart from those explicitly excluded, are included (e.g. lines 22-25). Another way to include or exclude API elements is through the use of special JavaScript-based conditions. These conditions are listed in the API description under a top-level key called `conditions`, which can contain a list of nested keys such as `excMethods`, `incMethods`, `incConstructors`, and `excConstructors`. The values of these keys are lists of JavaScript strings that describe the conditions to be applied (e.g., line 17).

```
1 packages:
2   - name: org.jsoup.nodes
3     types:
4       - name: Attribute
5         constructors:
6           - -Attribute(_,_,_)
7         methods:
8           - get*
```

## 6.1. APIExCov's Features

```
9          - hasDeclaredValue
10          - isBooleanAttribute
11          - set*
12          - toString
13          - -getValidKey
14          - html*
15      conditions:
16          - excMethods:
17              - name === 'html' && accessModifier === 'protected'
18      - name: Element
19      constructors:
20          - "*"
21      methods:
22          - -getElementsMatchingText(Pattern)
23          - -forEachNode
24          - -doClone
25          - -ensureChildNodes
26      - name: org.jsoup.select
27      types:
28          - name: NodeTraversor
29          methods:
30              - filter(_,Elements)
31              - traverse(_,Node)
```

Listing 6.1: A subset of an API description that defines the API of the Jsoup Java library.

The conditions explained above are particularly useful in cases where the order and type of parameters are not sufficient to identify the API element of interest; thus, additional attributes, such as the access modifier, can be used. For example, the method described on line 14 (i.e. `html*`) in Listing 6.1 indicates that all methods whose names start with ‘html’ will be included. However, the JavaScript-based condition specified on line 17 refines this inclusion and excludes the method with the `protected` access modifier from the list of included methods. The reasons behind choosing JavaScript as a format for these conditions are its simplicity, readability, and dynamic runtime evaluation.

## Chapter 6. Coverage of API Code Examples

In addition, API specifications written in the intended-API description language are validated at multiple levels. Since the language is YAML-based, its structure is enforced by the YAML parser. Semantic inconsistencies are also checked, for example when a method is declared under an omitted type, and vice versa. Moreover, invalid API element names that do not exist in the actual codebase are automatically flagged and reported. Further details are provided in Section 6.1.3.

Finally, it is important to note that the proposed API description language does not enforce a single description style. API developers may choose to specify either a common supertype or individual concrete types. There is no automatic expansion to ‘all subtypes’. Moreover, in the current design, method declarations are associated only with the exact type under which they are specified. During validation and coverage analysis, a method call is considered valid if and only if the method is declared in that type itself. Methods inherited from supertypes are not automatically included.

### The API Description Language ANTLR-based Custom Grammar

As presented in Listing 6.2, the API specification language grammar is particularly concerned with how Java method/constructor signatures should be described. It first declares the name of the grammar i.e. `APIDescription` followed by the `start` rule, which specifies that a valid description of an API should consist of a valid `methodSignature`. This method signature can take three forms, either a single method name e.g. `foo`, a method name followed by empty parentheses e.g. `foo()`, or a method name followed by parentheses with an optional parameter list e.g. `foo(int, String)`. The `parameterList` rule specifies that the method parameters are a comma-separated list of Java types or underscore symbols (`_`), which acts as a placeholder for any Java type. A valid parameter type is defined by the rule `javaType` (line 20), which states that it must be a `simpleType` followed by an optional array suffix (`[]`).

## 6.1. APIExCov's Features

This `simpleType` is defined to be either a reference type `typeName` with optional arguments for generic types (e.g. `List<T>`), or a (`primitiveType`) e.g. `byte`, `short`, or `char`.

In addition, this grammar defines generic type arguments, array suffixes, and type arguments based on their valid syntax in Java. Furthermore, the rule `typeName` states that a valid type name can be written in two ways; as a simple name e.g. `String` or a fully qualified name e.g. `java.lang.String`. This type name, as well as the method/constructor names, can include capital and small letters, numbers, and certain special characters i.e. an underscore (`_`) and an asterisk (`*`). Also, the dollar sign (`$`) is supported to allow the definition of nested classes, following Java's convention where inner classes are referenced as `OuterClass$InnerClass` in bytecode.

```
1 grammar APIDescription;
2 start
3     : methodSignature EOF
4     ;
5 methodSignature
6     :
7         IDENTIFIER
8     |
9         IDENTIFIER '(' ' ' ')'
10    |
11        IDENTIFIER '(' parameterList? ')'
12    ;
13 parameterList
14     : parameter (',' parameter)*
15     | parameter (',' parameter)* ','? varargsParameter
16     | varargsParameter
17     ;
18 varargsParameter
19     : javaType '...'
20     ;
21 parameter
22     : javaType
23     | '_'
24     ;
25 javaType
```

## Chapter 6. Coverage of API Code Examples

```
26      : simpleType arraySuffix*
27      ;
28 simpleType
29      : typeName genericTypeArguments?
30      |
31      primitiveType
32      ;
33 primitiveType
34      : 'byte'
35      | 'short'
36      | 'int'
37      | 'long'
38      | 'float'
39      | 'double'
40      | 'boolean'
41      | 'char'
42      ;
43 arraySuffix
44      : '[' ']'
45      ;
46 genericTypeArguments
47      : '<' typeArgument (',' typeArgument)* '>'
48      ;
49 typeArgument
50      : wildcard
51      | javaType
52      ;
53 wildcard
54      : '?'
55      | '? extends' javaType
56      | '? super' javaType
57      ;
58 typeName
59      : IDENTIFIER ('.' IDENTIFIER)*
60      ;
61 IDENTIFIER
62      : '-'? [a-zA-Z_0-9*$]+
63      ;
64 WS
65      : [ \t\r\n]+ -> skip
```

66 ;

Listing 6.2: ANTLR-based custom grammar for the intended API description language.

It is important to mention that the development of APIExCov was inspired by the work proposed by Monce et al. [1], which is described in more detail in Section 2.3.2. However, this work is particularly concerned with the *factual API elements*, i.e. all the public parts of a Java library. It also focuses on extracting API uses from client code to understand how APIs are used in the wild, which can help API maintainers improve their APIs’ tests and documentation [1, 44]. APIExCov, on the other hand, focuses on the *intended API elements* that can be described through the novel API specification language described above. This is crucial since public elements of a Java library may represent utility classes and internal functionality that is not meant to be part of the library’s API. Also, APIExCov only considers API code examples, making it particularly useful for newly released APIs that lack client code. Finally, APIExCov provides API developers with a comprehensive visual coverage report that can be easier to interpret than textual reports, which are also generated.

### 6.1.2 Static Analysis and Parsing

Static source code analysis plays a significant role in the implementation of APIEXCov. To validate a described API, the source code of the given Java library, as well as the API specification YAML file, must first be parsed and analysed. To handle YAML, the SnakeYAML<sup>5</sup> Java library was used. This lightweight and popular open-source library provides an API for serialisation and deserialisation of YAML documents and is ideal for parsing both simple key-value pairs and complex nested YAML structures. In APIExCov, SnakeYAML was employed to deserialise the API description YAML content

---

<sup>5</sup><https://github.com/snakeyaml/snakeyaml>

into an in-memory representation (as shown in Figure 6.2) of the intended API, such as `API`, `JavaPackage`, `JavaType`, etc. This is done through a `YamlReader`, which stores the deserialised API object and exposes it through accessor methods, specifically `getApi()` and `setApi(API api)`.

To work with the source code of the Java project of interest, the Eclipse JDT (Java Development Tools)<sup>6</sup> was utilised. This Eclipse project offers a range of plug-ins to parse, analyse, validate, and manipulate Java source code. Through JDT, the Abstract Syntax Tree (AST) is created for each individual source file (.java) within the Java project. This process is managed by a custom `CodebaseParser`, which was implemented to also create a `Map<String, List<CompilationUnit>` for storing all parsed Java files. In this map, the `key` represents a package name, and the `value` is a list of all the parsed `CompilationUnit` objects, which are AST representations of the Java source files within the corresponding package. Each `CompilationUnit` stored in this map is later accessed and analysed by a `CodebaseVisitor`, which uses JDT's `ASTVisitor` to traverse each node in the AST. The visitor extracts and stores detailed information about various Java source code elements, such as classes, methods, and fields. It also handles nested types and separately identifies their associated members using a set of helper methods.

### 6.1.3 Intended-API Description Validation

Before reporting on how the intended API elements are covered and used in code examples, it is crucial to validate that these elements actually exist, are syntactically correct, and are properly structured. To achieve this, the described API elements are passed to an `APIValidator` which checks them against the actual elements parsed and extracted from the Java library's source code. Moreover, the validator processes each package found in the API specification YAML file, including its types, methods, constructors, and

---

<sup>6</sup><https://www.eclipse.org/jdt/>



## 6.1. APIExCov's Features

fields, one at a time. It also ensures that types belong to their corresponding packages and that methods, constructors, and fields are correctly associated with their respective types. It tracks every step of the validation process and logs issues it encounters, such as invalid element names or packages with no specified types, using the `logger.info()` method.

In addition, it is important to note that the basic file format for the intended API specification is YAML, as it allows the overall API description to be expressed in a clear and structured way. However, certain API elements, particularly method and constructor signatures, are more difficult to describe in YAML due to their parameter lists and modifiers. To address this, ANTLR was introduced specifically to parse method and constructor signatures embedded within the YAML specification. In this way, the YAML parser captures the general structure of the intended API, while the ANTLR parser complements it by accurately processing the syntactically rich components of the specification.

Validating package and type names can be done directly by checking them against the library's source code; however, validating methods and constructors requires an additional step to interpret the patterns and wildcards in their names before proceeding with validation. This is done with the help of a custom `PatternInterpreter` and a listener class, `MethodSignatureListener`, which extends the ANTLR auto-generated `APIDescriptionBaseListener` and overrides the method `enterMethodSignature()` as shown in Listing 6.3. Each method signature passed to this method is processed to extract information such as method name, parameter list, whether the method is excluded, and any patterns included in its name. The processed method signatures are then saved in a list of `JavaMethod` objects.

```
1  @Override
2  public void enterMethodSignature(APIDescriptionParser.
    MethodSignatureContext ctx) {
```

## Chapter 6. Coverage of API Code Examples

```
3
4     if (ctx.IDENTIFIER() == null) {
5         logger.warn("No valid identifier found for method signature.");
6         return;
7     }
8
9     JavaMethod method = new JavaMethod(ctx.IDENTIFIER().getText());
10    processExcludedMethods(method);
11
12    if (ctx.parameterList() != null) {
13        List<String> parameters = new ArrayList<>();
14        for (APIDescriptionParser.ParameterContext paramCtx : ctx.
15            parameterList().parameter()) {
16            parameters.add(paramCtx.getText());
17        }
18        method.setParameters(parameters);
19    }
20
21    processNameContent(ctx.getText(), method);
22    methods.add(method);
23 }
```

Listing 6.3: Java method for processing and validating method signatures extracted from the intended API description YAML file.

This list of interpreted method signatures is later processed and checked against the methods belonging to the same type but are extracted from the library's source code. This process involves interpreting wildcards in the method names using `java.util.regex.Pattern` to match their equivalents from the source code. Furthermore, a group of helper methods, such as `doParametersMatch()`, is used to compare parameter lists against each other. Once the matched methods are extracted, they are stored in a `Map<MethodDeclaration, Boolean> matchedMethods`, where the **key** represents the matched method, and the **value** is a boolean indicating whether the method should be included in the final `IntendedAPIStore`. The methods included in this `IntendedAPIStore` are those for which the coverage will be reported.

## 6.1. APIExCov's Features

In addition, the extracted matched methods are processed for inclusion or exclusion. As explained in Section 6.1.1, this process is carried out in two steps. First, the boolean value attached to each extracted method is checked, and the method is excluded if the value is `true`. However, if all the extracted methods belonging to the same Java type must be excluded, this indicates that every other method belonging to that Java type will be included in the final `IntendedAPIStore`. Second, the exclusion is also performed by inspecting the JavaScript conditions extracted from the API description YAML file.

To work with the conditions written in JavaScript and execute them within Java code, the GraalVM Polyglot API<sup>7</sup> was used. This API allows developers to embed, run, and seamlessly pass values between different languages, such as Java, JavaScript, Python, Ruby, and R. Specifically, two classes of this API were used; `org.graalvm.polyglot.Context` to create a context to execute JavaScript and manage bindings, and `org.graalvm.polyglot.Value` to hold the result of the evaluated script and convert the data type from JavaScript to Java.

To illustrate how this API description language validator works, let us consider the short API description example shown in Listing 6.4. This specification defines the API of the Apache Commons CLI Java library and contains four errors (highlighted in red in Listing 6.4). These errors must be detected by the validator and are as follows:

1. The method `parse()` at line 6 has only one parameter in its signature. No such method exists in the Apache Commons CLI codebase. All overloads of the `parse()` method in the `Parser` class have either two, three, or four parameters.
2. The type `ParseExceptions` at line 10 has an inaccurate name. The correct name is `ParseException`.

---

<sup>7</sup><https://www.graalvm.org/latest/reference-manual/polyglot-programming/>

## Chapter 6. Coverage of API Code Examples

3. The constructor at line 15 is missing a closing parenthesis in its declaration. This kind of syntax error is automatically detected and reported by the ANTLR-generated parser.
4. The method `hasArg()` at line 20 has an incorrect parameter type. It should be `boolean` instead of `String`.

```
1 packages :
2   - name: org.apache.commons.cli
3     types :
4       - name: Parser
5         methods :
6           - -parse(_)
7           - parse*
8           - processOption
9           - getOptions
10        - name: ParseExceptions
11          constructors :
12            - ParseException(String)
13        - name: MissingOptionException
14          constructors :
15            - MissingOptionException(String
16          methods :
17            - "*"
18        - name: OptionBuilder
19          methods :
20            - -hasArg(String)
```

Listing 6.4: An example of an API description with errors. This description defines a subset of the API of the Apache Commons CLI Java library.

When the above API description is processed by the validator, all the errors it contains are detected and reported as shown in Listing 6.5. The error message provides all the information needed to identify the exact error location, and explains exactly what the problem was. It also informs the user when none of the methods or constructors defined under a particular Java class matches the expected signatures.

## 6.1. APIExCov's Features

```
14:59:56.274 [main] ERROR uk.ac.york.cs.apiexcov.validation.  
    PatternInterpreter -- Invalid method name OR parameter list does  
    not match: parse in type: Parser in package: org.apache.commons.  
    cli  
14:59:56.307 [main] ERROR uk.ac.york.cs.apiexcov.validation.  
    APIValidator -- Invalid type name: ParseExceptions in package: org  
    .apache.commons.cli  
line 1:29 missing ')' at '<EOF>'  
14:59:56.380 [main] ERROR uk.ac.york.cs.apiexcov.validation.  
    PatternInterpreter -- Invalid method name OR parameter list does  
    not match: hasArg in type: OptionBuilder in package: org.apache.  
    commons.cli  
14:59:56.380 [main] INFO uk.ac.york.cs.apiexcov.validation.  
    PatternInterpreter -- All methods in type: OptionBuilder in API  
    package: org.apache.commons.cli in the API description file (.yaml  
    ) were not matched
```

Listing 6.5: Examples of the API description language validation error messages.

Finally, once all the API elements described in the API description YAML file are validated, they are stored in an in-memory representation of the intended API, i.e. `IntendedAPIStore`, which contains classes such as `IntendedAPIPackage`, `IntendedAPIType`, etc., as shown in the class diagram in Figure 6.2. This intended API is later used to detect API usages found in API code examples. The following sections explain how these API usages are extracted, how coverage analysis is performed, and what information the textual and visual reports contain.

### 6.1.4 Coverage Analysis

This part of APIExCov focuses on analysing API code examples to extract usage information. It works with two main components: the intended API store (described in Section 6.1.3) and API code examples provided by API developers. The first step in identifying usage information is to specify the type of API element for which coverage is being extracted.

## Chapter 6. Coverage of API Code Examples

As shown in Figure 6.2, five types of API elements are defined; **TYPE**, **METHOD**, **CONSTRUCTOR**, **FIELD**, and **ENUM**. Based on these element types, a corresponding set of suitable usage types is defined. For example, an API element of type **METHOD** may have usage types such as **INVOCATION** or **OVERRIDING**. Similarly, a **TYPE** can have several usage types, including **INSTANTIATION**, **REFERENCE**, and **INHERITANCE**. This classification of API usages is based on the interactions allowed by the Java programming language. It captures the most common ways in which an API user interacts with the exposed and described API elements.

Furthermore, the usage frequency, such as the number of times a method is invoked across all code examples, is also captured for each usage type. Another important piece of information that is extracted from code examples is the context in which an API element usage is found. This includes details such as the Java file name and line number. Providing this kind of information is crucial for helping API developers easily locate usage instances.

The coverage analysis process begins with a **CoverageAnalyser**, which iterates over each API element in the intended API store (described in Section 6.1.3) and extracts all its usage types from the provided set of API code examples, along with their contexts. This process is performed iteratively; the analysis starts with an **IntendedAPIPackage**, processes all its API elements of type **TYPE** one at a time, extracts their usage information, and then moves on to analyse the usage details of all the other API elements that belong to this type, such as elements of type **METHOD** or **CONSTRUCTOR**.

To further illustrate this coverage analysis, let us consider the API code example in Listing 6.6, which is one of the code examples provided by the developers of the Jsoup Java library and is available in the library’s GitHub repository.<sup>8</sup> Furthermore, let us assume that we wish to report the coverage details of the Jsoup’s class `org.jsoup.nodes.Element` and its method

---

<sup>8</sup><https://github.com/jhy/jsoup>

## 6.1. APIExCov's Features

`public String text()`, both of which are part of Jsoup's intended API as described in the API specification in Listing 6.1, and are used in the given code example. The usage information that the `CoverageAnalyser` would extract from the API code example in Listing 6.6 for the API elements described above is shown in Table 6.1.

Table 6.1: Usage information for two API elements, extracted from an API code example.

API Element	Type	Usage Type	Frequency	Context	
				File Name	Line Number
<code>org.jsoup.nodes.Element</code>	Type	Reference	3	ListLinks.java	23, 33, 38
<code>text()</code>	Method	Invocation	1	ListLinks.java	39

```

1 package org.jsoup.examples;
2
3 import org.jsoup.Jsoup;
4 import org.jsoup.helper.Validate;
5 import org.jsoup.nodes.Document;
6 import org.jsoup.nodes.Element;
7 import org.jsoup.select.Elements;
8
9 import java.io.IOException;
10
11 public class ListLinks {
12     public static void main(String[] args) throws IOException {
13         Validate.isTrue(args.length == 1, "usage: supply url to fetch"
14             );
15         String url = args[0];
16         print("Fetching %s...", url);
17
18         Document doc = Jsoup.connect(url).get();
19         Elements links = doc.select("a[href]");
20         Elements media = doc.select("[src]");
21         Elements imports = doc.select("link[href]");
22
23         print("\nMedia: (%d)", media.size());
24         for (Element src: media) {
25             if (src.nameIs("img"))
26                 print(" * %s: <%s> %sx%s (%s)",
27                     src.tagName(), src.attr("abs:src"), src.attr("

```

## Chapter 6. Coverage of API Code Examples

```
                width"), src.attr("height"),
27                trim(src.attr("alt"), 20));
28            else
29                print(" * %s: <%s>", src.tagName(), src.attr("abs:src"
                    ));
30        }
31
32        print("\nImports: (%d)", imports.size());
33        for (Element link: imports) {
34            print(" * %s <%s> (%s)", link.tagName(), link.attr("abs:
                href"), link.attr("rel"));
35        }
36
37        print("\nLinks: (%d)", links.size());
38        for (Element link: links) {
39            print(" * a: <%s> (%s)", link.attr("abs:href"), trim(link
                .text(), 35));
40        }
41    }
42
43    private static void print(String msg, Object...args) {
44        System.out.println(String.format(msg, args));
45    }
46
47    private static String trim(String s, int width) {
48        if (s.length() > width)
49            return s.substring(0, width - 1) + ".";
50        else
51            return s;
52    }
53 }
```

Listing 6.6: API Code Example from the Jsoup Java Library (ListLinks.java).

Finally, all the usage information is stored in an in-memory representation called `APICodeExampleStore`, which is then used to generate both textual and visual coverage reports. This is explained in more detail in the following sections.



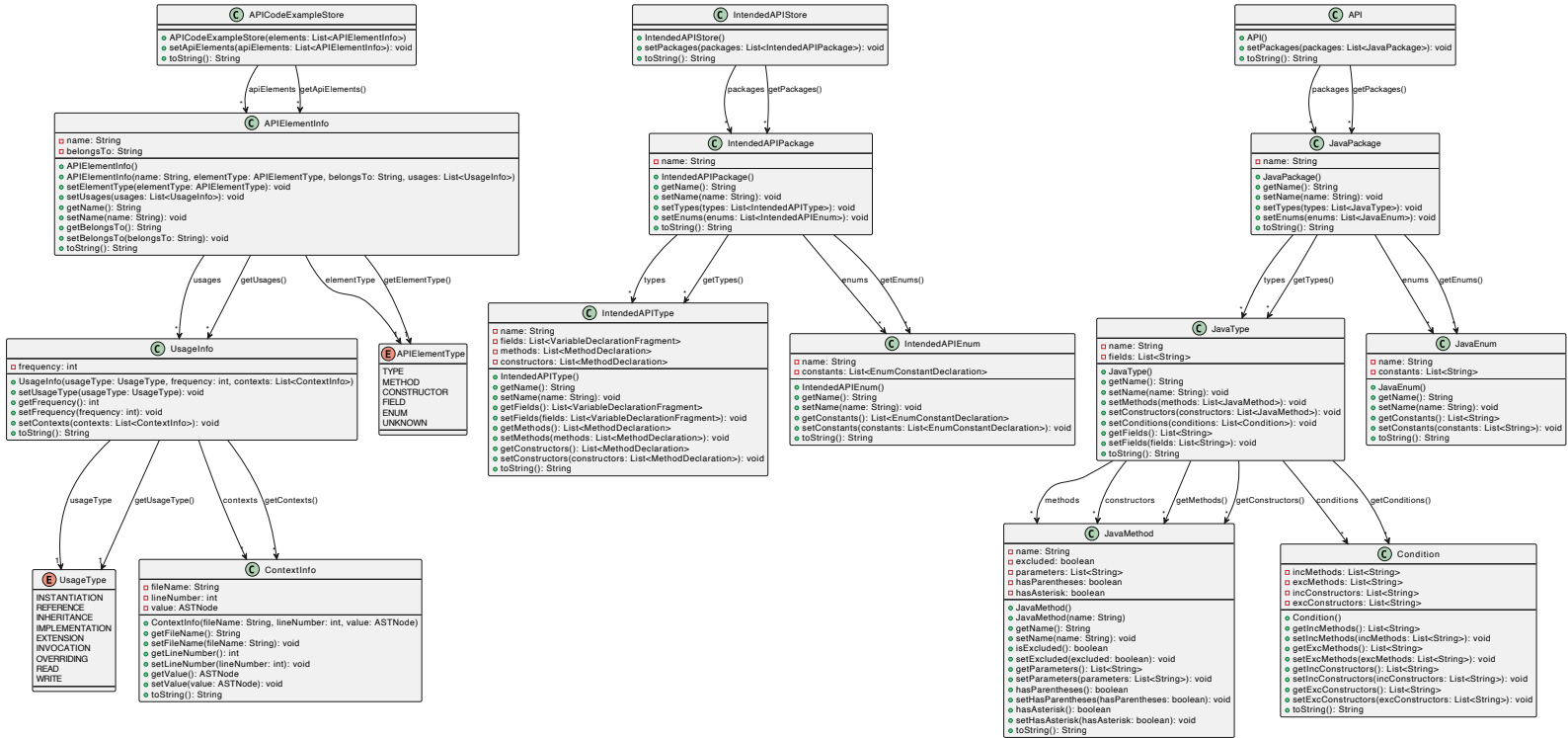


Figure 6.2: Class diagram of the Java classes representing API elements. These classes are used to create an in-memory representation of the API and its coverage details.

### 6.1.5 Textual Coverage Report

The proposed API code example coverage tool generates two main coverage report formats, both as static HTML files. These two formats show the same coverage information but with different levels of granularity. Furthermore, providing a textual coverage report was necessary because it provides more detailed and numerical information than a visual one. It presents information in a more structured and sorted way, which allows API developers to view fine-grained coverage details. Also, this kind of presentation includes some features, such as information filtering, that can facilitate navigation for API developers.

To explain the features of the generated textual report further, let us consider the report shown in Figure 6.3, which presents a subset of API code example coverage information for the Jsoup Java library. This textual report is presented in a tabular form and consists of six columns. The first column lists all the API elements for which coverage is being reported. The elements are sorted by the Java packages and types they belong to and are listed with indentation to improve the overall report's readability. Furthermore, special icons representing each API element type are listed next to the element's name. The use of these icons provides API developers with an intuitive visual cue that helps them immediately distinguish between different types of API elements. The second column shows the API element type such as packages, types, or methods.

Filter by Coverage: All

API Element	Type	Coverage	Usage Type	Frequency	File Name and Line Number
<b>org.jsoup</b>	Package	<div><div></div></div> 20.0%			
<b>HttpException</b>	Type	<div><div></div></div> 0.0%	n/a	n/a	n/a
public int getStatusCode()	Method	<div><div></div></div>	n/a	n/a	n/a
public java.lang.String getUrl()	Method	<div><div></div></div>	n/a	n/a	n/a
<b>Jsoup</b>	Type	<div><div></div></div> 5.3%			
public static java.lang.String clean(java.lang.String, java.lang.String, org.jsoup.safety.Safelist, org.jsoup.nodes.Document.OutputSettings)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.io.File, java.lang.String, java.lang.String) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.Connection connect(java.lang.String)	Method	<div><div></div></div>	Invocation	5	+ ListLinks.java + Wikipedia.java + MyOwnExample.java + HtmlToPlainText.java
public static org.jsoup.nodes.Document parse(java.lang.String, java.lang.String, org.jsoup.parser.Parser)	Method	<div><div></div></div>	n/a	n/a	n/a
public static boolean isValid(java.lang.String, org.jsoup.safety.Safelist)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.io.InputStream, java.lang.String, java.lang.String, org.jsoup.parser.Parser) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.net.URL, int) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static java.lang.String clean(java.lang.String, java.lang.String, org.jsoup.safety.Safelist)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.lang.String, java.lang.String)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.io.File, java.lang.String) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.io.InputStream, java.lang.String, java.lang.String) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static java.lang.String clean(java.lang.String, org.jsoup.safety.Safelist)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.lang.String, org.jsoup.parser.Parser)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.io.File) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.io.File, java.lang.String, java.lang.String, org.jsoup.parser.Parser) throws java.io.IOException	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parseBodyFragment(java.lang.String, java.lang.String)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parseBodyFragment(java.lang.String)	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.Connection newSession()	Method	<div><div></div></div>	n/a	n/a	n/a
public static org.jsoup.nodes.Document parse(java.lang.String)	Method	<div><div></div></div>	n/a	n/a	n/a
<b>UncheckedIOException</b>	Type	<div><div></div></div> 0.0%	n/a	n/a	n/a
public void (java.io.IOException)	Constructor	<div><div></div></div>	n/a	n/a	n/a

Figure 6.3: An example of a textual report showing a subset of API code example coverage information for the Jsoup Java library.

## Chapter 6. Coverage of API Code Examples

Moreover, to effectively represent API example coverage, two types of visual percentage indicators are used, as shown in the third column in Figure 6.3. The first indicator is a percentage bar, which is used only for package and type coverage. The coverage percentage of a package is based on how many Java types within that package are covered with API code examples, regardless of the coverage of the other API elements within these types. Similarly, the coverage percentage of a Java type is determined by the number of API elements, such as methods, constructors, and fields, within that type that are covered. For example, if a Java type has ten described methods in the intended API description and only five of these methods are exercised in code examples, then its coverage percentage is 50%, assuming no other API elements within this type are described in the intended API specification. Furthermore, the second type of visual indicator consists of two symbols representing the coverage of individual API elements. A red cross mark (✗) indicates that an element is not covered in API code examples, while a green check mark (✓) signifies that the element is covered and used at least once in the examples.

The textual report also shows other important pieces of information, such as the type of usage and its frequency across all API code examples, in the fourth and fifth columns, respectively. The last column provides some contextual coverage information, such as the file name and line number. These pieces of information are presented in an expandable list to enhance the textual report's overall readability and organisation. Moreover, the textual report provides a filtering feature with different options, as shown in Figure 6.4, which enhances the report's usability and allows API developers to easily identify coverage issues, locate API elements that need to be covered with code examples, and navigate lengthy reports.

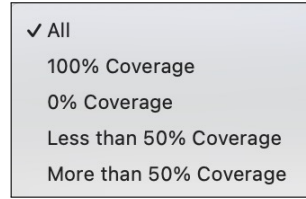


Figure 6.4: Coverage filtering options in generated textual reports.

### 6.1.6 Visual Coverage Report

Unlike the textual version of the API code example coverage report, the visual report provides greater insight into the overall status of API coverage at a glance. This is because visual representations make it easier to spot patterns, outliers, and coverage issues that may not always be apparent in tables filled with numerical values. Furthermore, research has demonstrated that visual representations often communicate information more effectively [134, 135] and more precisely [136, 137] than text. This may be attributed to the Picture Superiority Effect (PSE),<sup>9</sup> as visually presented information is more likely to be remembered [138, 139]. In addition, visual API coverage reports can be more engaging and dynamic than static textual reports, since they allow API developers to navigate and zoom into specific API elements. However, providing both textual and visual reports offers complementary insights into the API coverage.

To visually represent API code example coverage information, the city metaphor visualisation technique was chosen. This form of representation is navigable, three-dimensional, and intuitive, as it depicts real-world objects, with software packages represented as districts and classes as buildings. It was first proposed by Wettel and Lanza [61, 63, 62] to facilitate large-scale analysis and comprehension of complex software systems. This visualisation technique has since been employed in various areas in the software engineering field. For example, it has been utilised to understand system evolution

<sup>9</sup>A well-known phenomenon in cognitive psychology and neuroscience.

[140] and analyse memory behaviour [141].

To better illustrate API code example coverage, a new visual element, i.e. a set of connecting lines, was added to the code city visualisation, as shown in Figure 6.5. Each connecting line represents a single type of API usage and extends from the Java class or type where the API element is defined to the API code example in which it is used. Other metrics used in this visualisation (also shown in Figure 6.5) are explained as follows:

- **Height:** This metric represents the extent to which a Java type (or class) and all its associated API elements (e.g. methods) are covered in API code examples. A taller building indicates that a larger proportion of the API elements described in the intended API specification under that Java type are exercised in code examples. If a type has 100% coverage, its building height is set to 100, which is the upper limit for all buildings. The lower limit is 0.5, as setting a building's height to 0 would make it invisible to users.
- **Width:** A building's width represents the frequency of API element usage. The more frequently a Java type and its associated API elements appear in code examples, the wider the building. Furthermore, after calculating the width values for all buildings, they are rescaled to a fixed range [1–10] using the Min-Max Normalisation technique (shown in Formula 6.1). This ensures a consistent and relative comparison between all buildings' widths.

$$X' = a + \frac{(X - X_{\min})(b - a)}{X_{\max} - X_{\min}} \quad (6.1)$$

where:

$X$  is the original value

$X_{\min}$  is the minimum value

## 6.1. APIExCov's Features

$X_{\max}$  is the maximum value

$X'$  is the normalised value

$a$  is the lower bound of the new range

$b$  is the upper bound of the new range

- **Depth:** This metric represents the number of API code examples in which a Java type or its API elements appear. If the usages of an API element are spread across multiple API code example Java files rather than being concentrated in just one or a few, the building's depth is higher. Depth values are also rescaled to a fixed range [1–10] using Formula 6.1.
- **Colour:** This metric represents the same information as the height metric but is used to help instantly differentiate between Java types with no coverage and those with minimal coverage. Java types with no coverage are illustrated as short red buildings, while those with minimal coverage appear as short green buildings. Several shades of green are used to indicate the percentage of coverage; the darker the shade (and the taller the building), the higher the coverage. Connecting lines have the same colour as their source building (Java class).

The metrics explained above apply to all buildings except those in the 'examples' district (Java package), which represent API code examples (Java files). For these buildings, height indicates the diversity of API element usage in the code example. In other words, the building is taller if the code example contains usages of many different API elements. In addition, the width and depth of these buildings are set to fixed values, meaning they do not convey any additional information. However, the colour metric remains the same as in the buildings from other districts, as it only reinforces the height metric.

The API coverage visual report also provides additional information, such as the fully qualified names of Java types and other contextual details, includ-

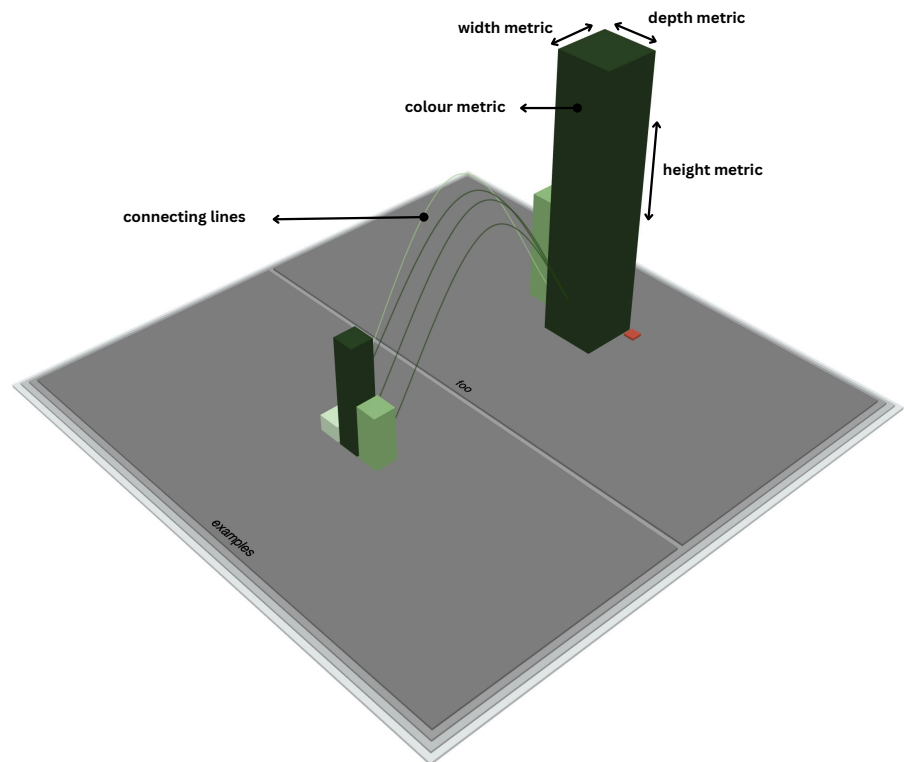


Figure 6.5: Code city metaphor metrics for the API code example coverage visual report.

ing the type of usage and its location (i.e., Java file name and line number). As shown in Figures 6.6 and 6.7, the former appears when API developers hover over buildings, and the latter appears when they hover over connecting lines.

A complete example of an API coverage visual report is shown in Figure 6.8. This report presents the API coverage information for the Jsoup Java library, based on the API code examples from Jsoup’s GitHub repository.<sup>8</sup>

The API coverage visual report was implemented in the JavaScript programming language using the Three.js library.<sup>10</sup> This lightweight and efficient JavaScript library is designed for creating and animating three-

---

<sup>10</sup><https://threejs.org>



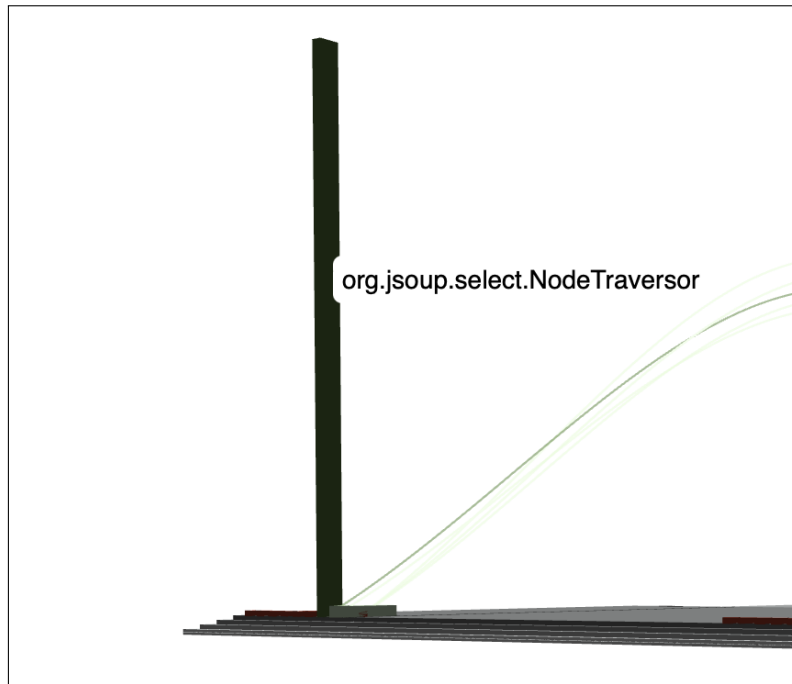


Figure 6.6: Hover information displayed over buildings in the visual API coverage report.

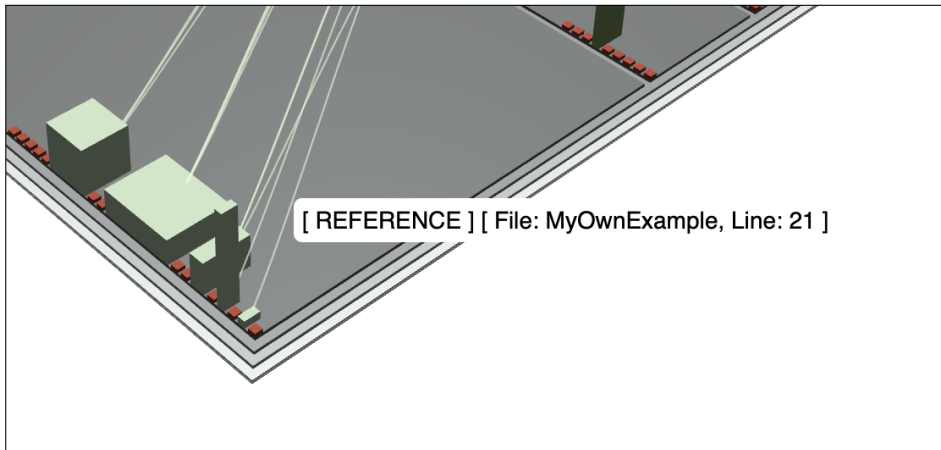


Figure 6.7: Hover information displayed over connecting lines in the visual API coverage report.

dimensional graphics. It uses Web Graphics Library (WebGL), which is a JavaScript API for graphics rendering in a web browser. Moreover, Three.js consists of several core components, including a scene, camera, and renderer.

The scene acts as a container for all the created three-dimensional objects, while the camera functions like human vision, determining which parts of the scene are visible. The renderer, as its name suggests, is responsible for displaying the scene in the web browser from the camera's perspective. Other components, such as lights and materials, define the overall look and feel of the scene and its objects.

In addition, several other modules were used for animation and formatting purposes. These include utilities such as OrbitControls,<sup>11</sup> TextGeometry,<sup>12</sup> and FontLoader.<sup>13</sup>

## 6.2 Constraints

Effort was made to ensure that the grammar of the proposed API description language captures a wide range of method signatures. This includes support for key Java features, such as varargs (...) and generic types (<T>). However, since the proposed language was tested and evaluated against only a small selection of APIs (see Section 6.3) and given the complexity of Java syntax, some corner cases may still not be fully supported.

Future work could extend the evaluation of the API description language by testing it with a more diverse range of APIs to identify edge cases and assess its constraints.

## 6.3 Evaluation

During the evaluation phase of this work, the primary goal was to assess the benefits of the features and mechanisms provided by the proposed API description language. It was essential to determine how this language is particularly useful in simplifying API specifications and making them concise.

---

<sup>11</sup><https://threejs.org/docs/examples/en/controls/OrbitControls>

<sup>12</sup><https://threejs.org/docs/examples/en/geometries/TextGeometry>

<sup>13</sup><https://threejs.org/docs/examples/en/loaders/FontLoader>

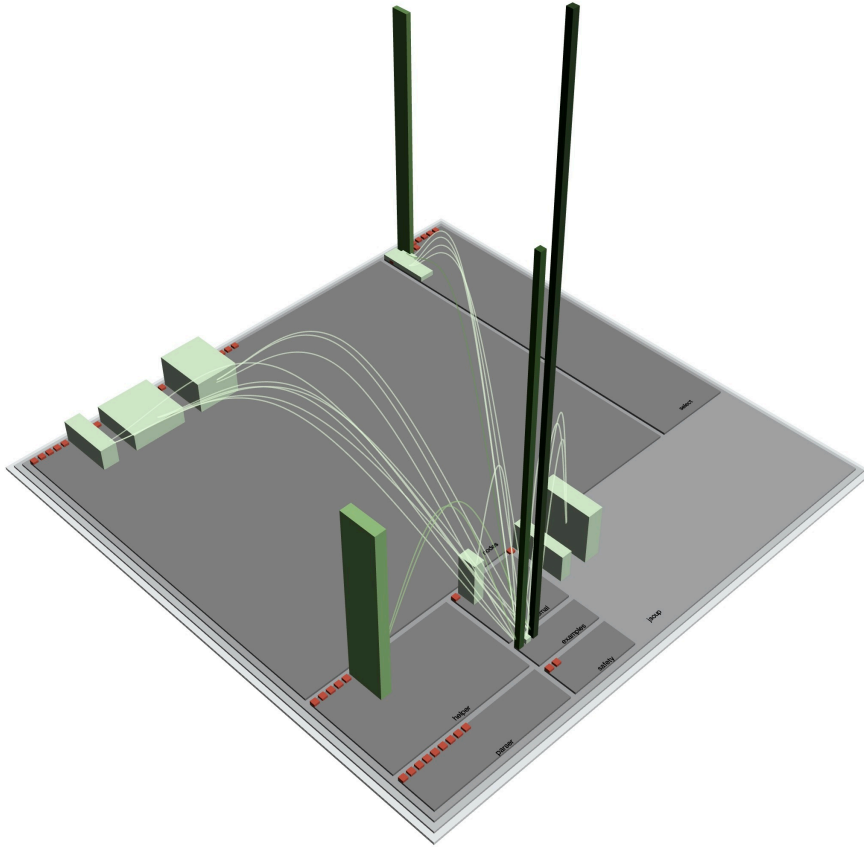


Figure 6.8: A visual report (code city) illustrating the API coverage for the Jsoup Java library. The coverage information is based on the API code examples available in the library’s GitHub repository.

As explained in Section 6.1, the language achieves conciseness through features such as wildcard matching, pattern matching, and inclusion/exclusion filtering (with or without conditions). These constructs allow developers to capture multiple related API elements, such as overloaded methods or constructors, with a single, compact expression rather than enumerating each signature manually. This is particularly beneficial in Java, where constructors and methods are often overloaded and have long, complex parameter lists that are tedious to specify individually.

Specifically, the aim was to answer the following research question (RQ):

**RQ1:** How significant is the reduction in the number of API elements and their character count achieved by applying the features of the proposed API description language?

To answer this research question, the evaluation required a set of API descriptions for well-known Java libraries. Therefore, it utilised three subject libraries selected by Monce et al. [1], based on shared selection criteria, and extracted a fourth library from the same dataset, i.e. the Duets dataset [142].<sup>14</sup> This dataset was created to help software engineers and researchers gain deeper insights into APIs and their usage. It comprises 395 Java libraries and 2,874 clients, all of which are compilable and extracted from GitHub repositories with at least five stars.

The subject libraries used in this evaluation were: Jsoup, a Java library for working with HTML; Spark, a lightweight and expressive web framework for Java; Commons-CLI, which provides a simple API for parsing command-line arguments; and Javassist, a library for bytecode manipulation. An overview of key statistics for these libraries is presented in Table 6.2.

Table 6.2: Overview of key statistics for the subject Java libraries, retrieved from GitHub on October 30, 2023 by Monce et al.[1] and December 9, 2024.

	<b>Commons-CLI*</b>	<b>Jsoup*</b>	<b>Spark*</b>	<b>Javassist</b>
<b>Last release date</b>	2021/10/29	2023/04/29	2020/10/08	2023/12/24
<b>Version</b>	1.5.0	1.16.1	2.9.3	3.30.2-GA
<b>Since</b>	2002	2010	2011	2003
<b>Stars</b>	309	10.4k	9.5k	4.1k
<b>Commits</b>	1,374	1,889	1,067	1,036
<b>Size (LoC)</b>	6,307	27,817	11,298	88,985
<b>Contributors</b>	46	97	100	43
<b>Clients</b>	49k	131k	30k	87k
<b>Source</b>	<a href="#">GitHub</a>	<a href="#">GitHub</a>	<a href="#">GitHub</a>	<a href="#">GitHub</a>

\* As retrieved and analysed by Monce et al. [1]

<sup>14</sup><https://github.com/ASSERT-KTH/Duets/blob/master/dataset/dataset-info.json>

### 6.3. Evaluation

To ensure that the subject libraries were representative, they were selected from various domains. In addition, steps were taken to ensure they met specific criteria [1], such as the availability of sufficient client code and API code examples. The client code of these libraries was particularly important for the work of Monce et al. [1], as it enabled the generation of syntactic usage footprint (SUF) models, as shown in Figure 6.9.

As explained in Section 2.3.2, a SUF model captures all actual usages of an API extracted from the source code of collected third-party clients. Moreover, these SUFs were used in the evaluation of our proposed API description language to generate what we called ‘full API descriptions’, which served as a baseline for assessing the usability of the language.

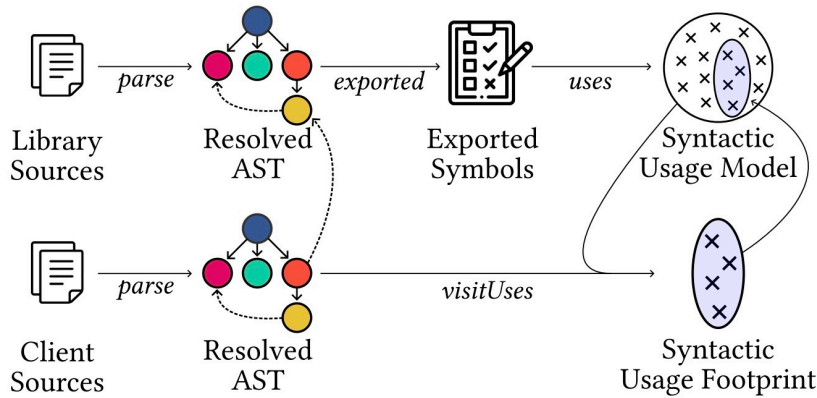


Figure 6.9: Workflow of the approach proposed by Monce et al. [1]. It illustrates the analysis of client sources to generate a syntactic usage footprint (SUF) of a library.

Since the dataset used, i.e. Duets [142], contains only a handful of client projects (15 clients) for the fourth subject library that we selected (Javassist), additional clients were collected following an approach similar to that used by Monce et al. [1] for collecting extra client projects for the other three subject libraries. This step was crucial to ensure that all four subject libraries were relatively comparable in terms of the number of clients, selection criteria, and client collection methodology.

The additional clients for Javassist were extracted from GitHub based on their explicit declaration of a dependency on Javassist, e.g. in `pom.xml` or `build.gradle` files. This extraction yielded a total of 17,088 repositories, which was reduced using the Cochran formula<sup>15</sup> with a conservative proportion of  $p = 0.5$ , a confidence level of  $c = 95\%$ , and a margin of error of  $e = 5\%$ . This resulted in a sample size of 375, which was randomly selected from the extracted Javassist client repositories.

Furthermore, steps were taken to ensure that the selected repositories were retrievable and active, i.e. having at least two contributors and more than ten recent commits [127]. These Javassist clients, along with those provided by Monce et al. [1] for the other three Java libraries (372 for Jsoup, 372 for Commons-CLI, and 373 for Spark), were used to generate the syntactic usage footprints (SUFs) for all four subject libraries. The SUFs were automatically generated using the tool proposed by Monce et al. [1], as shown in Figure 6.9.

As mentioned above, the auto-generated syntactic usage footprints (SUFs) of each subject Java library were used to create the evaluation baseline, i.e. ‘full API descriptions’. These API descriptions are YAML-based and contain all the API elements found in the SUFs. They are structured and described according to the proposed API description language format, where types are listed under packages, and other API elements, such as methods and fields, are listed under their corresponding Java types. However, at this stage, the API description language’s features and patterns are not yet applied. Later, each full API description for each Java library was analysed and compared against the library’s source code. The goal of this analysis was to manually apply the proposed language’s shortening features where applicable and save the modifications in a new version of the API description, referred to as the ‘shortened/compact API description’.

---

<sup>15</sup>A formula for computing the required sample size for a desired level of precision.

### 6.3. Evaluation

For example, if the analysis of the library’s source code found that all methods in a specific Java type were used by clients and were therefore listed under that type in the full API description YAML file, then the list of methods was replaced with an asterisk (\*). An illustrated example is provided in Listings 6.7 and 6.8, which compare the full and shortened versions of the same subset of an API description.

Finally, after applying the features and patterns of the proposed API specification language to shorten the full API descriptions of the four subject Java libraries, the two versions (full and shortened) of each library were compared and evaluated based on the reduction in the **number of API elements** and their overall **character count**.

```
1 packages:
2   - name: org.jsoup
3     types:
4       - name: HttpStatusException
5         methods:
6           - getStatusCode()
7           - getUrl()
8       - name: Jsoup
9         methods:
10          - clean(java.lang.String, java.lang.String, org.jsoup.safety.
              Safelist, org.jsoup.nodes.Document$OutputSettings)
11          - clean(java.lang.String, java.lang.String, org.jsoup.safety.
              Safelist)
12          - clean(java.lang.String, org.jsoup.safety.Safelist)
13          - connect(java.lang.String)
14          - isValid(java.lang.String, org.jsoup.safety.Safelist)
15          - newSession()
16          - parse(java.io.File, java.lang.String, java.lang.String, org.
              jsoup.parser.Parser)
17          - parse(java.io.File, java.lang.String, java.lang.String)
18          - parse(java.io.File, java.lang.String)
19          - parse(java.io.File)
20          - parse(java.io.InputStream, java.lang.String, java.lang.
              String, org.jsoup.parser.Parser)
21          - parse(java.io.InputStream, java.lang.String, java.lang.
              String)
```

## Chapter 6. Coverage of API Code Examples

```
22         - parse(java.lang.String, java.lang.String, org.jsoup.parser.  
           Parser)  
23         - parse(java.lang.String, java.lang.String)  
24         - parse(java.lang.String, org.jsoup.parser.Parser)  
25         - parse(java.lang.String)  
26         - parse(java.net.URL, int)  
27         - parseBodyFragment(java.lang.String, java.lang.String)  
28         - parseBodyFragment(java.lang.String)
```

Listing 6.7: A subset of the full API description for the Jsoup Java library.

```
1 packages:  
2   - name: org.jsoup  
3   types:  
4     - name: HttpStatusException  
5       methods:  
6         - get*  
7     - name: Jsoup  
8       methods:  
9         - connect  
10        - isValid  
11        - newSession  
12        - parseBodyFragment*  
13        - clean*  
14        - parse*
```

Listing 6.8: A subset of the shortened API description for the Jsoup Java library.

## 6.4 Results and Discussion

As explained in Section 6.3, the evaluation primarily focused on assessing the benefits of the proposed language in making API descriptions more concise and easier for API developers to write. As shown in Table 6.3 and Figure 6.10, the results of this evaluation demonstrated that applying the features and patterns of the proposed API specification language reduced the number of API elements in the shortened versions of the API descriptions. However, the extent of this reduction varied across the evaluated Java projects.



## 6.4. Results and Discussion

The percentage decrease in the number of API elements ranged from 16% (Javassist) to 77% (Commons-CLI). Specifically, the number of API elements was reduced from 553 to 174 elements in Jsoup (a 69% decrease); from 184 to 42 elements in Commons-CLI (77% decrease); from 271 to 139 elements in Spark (49% decrease); and from 285 to 238 elements in Javassist (16% decrease).

This variation in reduction could be attributed to differences in naming conventions and coding styles across the four evaluated APIs. Moreover, APIs with consistent and repetitive API element names are more likely to benefit from the proposed API description language, whereas APIs with inherently concise and unique API element names, such as Javassist, retain most of their original full names.

Similarly, the proposed language significantly reduces the character count of the described API elements, as illustrated in Table 6.3 and Figure 6.11. The percentage reduction in character count ranged from 64% in Javassist to 91% in Commons-CLI. Once again, this variation in reduction suggests that Java projects, such as Jsoup and Commons-CLI, may contain longer, more verbose API element names, resulting in greater character reduction when applying the features of the proposed API description language.

While Table 6.3 presents the percentage reduction achieved by writing API descriptions using the proposed language, Figures 6.10 and 6.11 depict the proportion (size) of the shortened API descriptions relative to the full versions for each evaluated Java project. Both the shortened and full versions of the API descriptions for the Jsoup Java library are presented in Appendix B. API descriptions for the other evaluated Java libraries are available in the replication package.

**Answer to RQ1:** The proposed API specification language can reduce the number of API elements by up to 77% and their character count by up to 91% in shortened API descriptions compared to their full versions.

Table 6.3: Comparison of API element counts, character totals, and percentage decreases in full and shortened API descriptions for the evaluated Java libraries and frameworks.

Java Project	# of API Elements*		%	# of Characters**		%
	<i>Full</i>	<i>Shortened</i>		<i>Full</i>	<i>Shortened</i>	
<b>Jsoup</b>	553	174	69%	13247	1508	89%
<b>Commons CLI</b>	184	42	77%	5562	484	91%
<b>Spark</b>	271	139	49%	8615	1152	87%
<b>Javassist</b>	285	238	16%	8483	3065	64%

\* The fields, constructors, methods, and enum constants of the described Java types.

\*\* The characters of the described API elements.

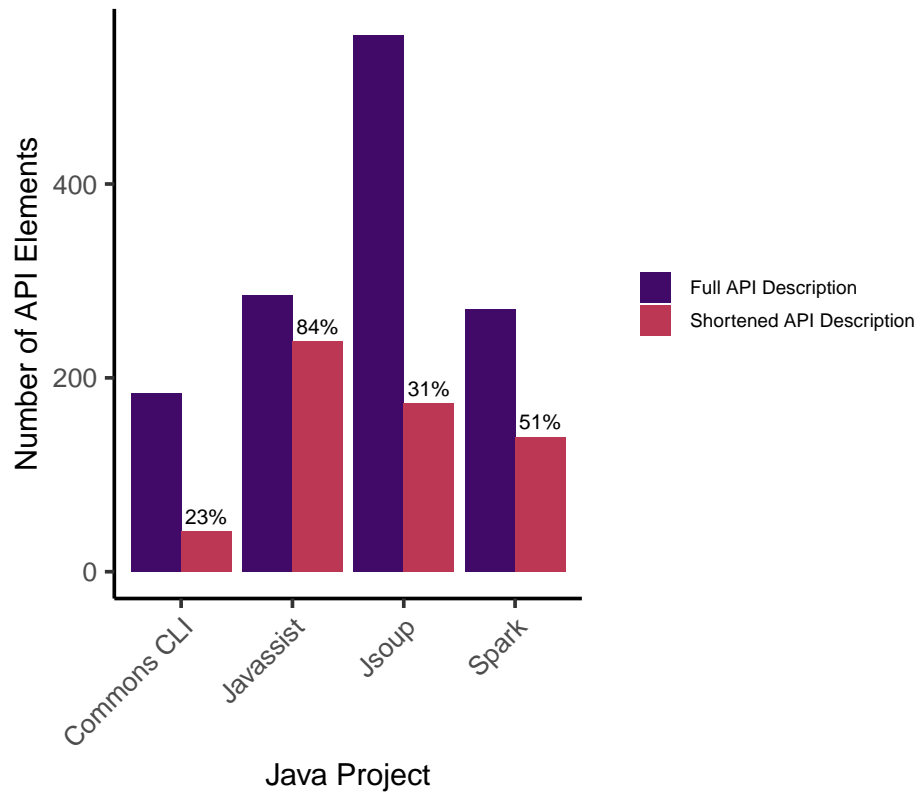


Figure 6.10: Comparison of API elements in full and shortened API descriptions for the evaluated Java libraries and frameworks. Percentages represent the proportion of shortened descriptions relative to full descriptions for each Java project.

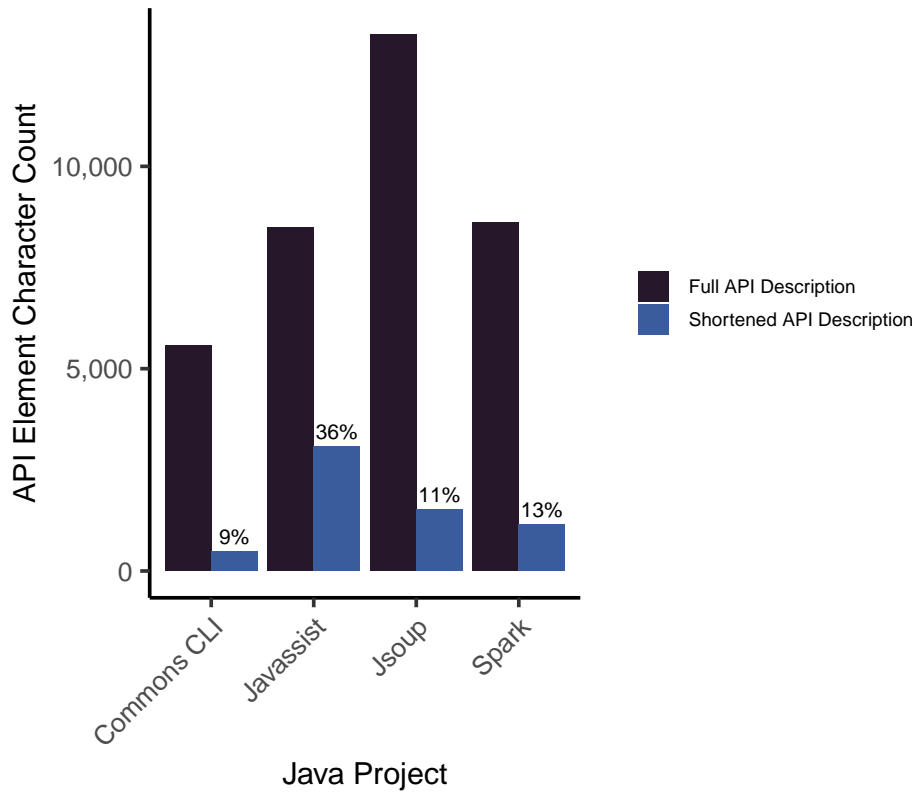


Figure 6.11: Comparison of API element character counts in full and shortened API descriptions for the evaluated Java libraries and frameworks. Percentages indicate the proportion of shortened descriptions relative to full descriptions for each Java project.

#### 6.4.1 Qualitative Comparison of APIExCov and Code Coverage Tools

This part of the discussion is interested in qualitatively comparing between the proposed API code example coverage tool (APIExCov) and some widely-used Java code coverage tools, such as JaCoCo (Java Code Coverage Library),<sup>16</sup> Cobertura,<sup>17</sup> and OpenClover.<sup>18</sup> The purpose of this evaluation is not treating APIExCov as a direct competitor but rather as a complementary tool to these existing code coverage tools. This is because the purpose

<sup>16</sup><https://www.eclemma.org/jacoco/>

<sup>17</sup><http://cobertura.github.io/cobertura/>

<sup>18</sup><https://openclover.org>

## Chapter 6. Coverage of API Code Examples

and scope of these code coverage tools differ significantly from APIExCov, although both play a crucial role in the overall software quality, i.e. the quality of its tests and the comprehensiveness of its documentation code examples.

Code coverage tools, such as those mentioned above, are used to assess the thoroughness of a test suite. They help determine whether **all** code has been satisfactorily tested, which can contribute to deciding whether the software being built is ready for release. In contrast, APIExCov evaluates whether **specific pre-specified API elements** are included and exercised in API code examples. Furthermore, unlike traditional code coverage tools, which work at the execution level to ensure that source code is executed during tests, APIExCov performs static analysis on API source code and documentation examples.

Another key difference between the proposed API coverage tool and test coverage tools is coverage granularity. APIExCov reports coverage at the level of individual API element usage, e.g. method calls, whereas traditional test coverage tools measure coverage across multiple granularity levels, such as code branches, lines, and methods. This distinction is inherent because, in software testing, ideally, all code should be fully tested, which means coverage must be reported for the entire codebase. In contrast, API code examples are typically expected to cover only the parts of the code intended for external use.

In addition, APIExCov generates two types of coverage reports (textual and visual) both in HTML format, which is a common approach among many widely used test coverage tools. However, tools such as JaCoCo also produce reports in other formats, such as XML and CSV, which are well-suited for integration with various software development tools.

Finally, although the two types of coverage tools discussed above serve distinct purposes, they ultimately target the same audience, i.e. software

developers.

## 6.5 Limitations and Observations

Although the proposed intended API specification language and its accompanying example coverage tool are practical and beneficial to both API developers and users, they do have some limitations. For instance, the proposed language requires API developers to manually define their intended APIs in a YAML-based API description. While this approach is effective for relatively small APIs, it could become burdensome for large libraries and frameworks that contain thousands of API elements. Therefore, reducing this effort by automating the generation of such an API description would be more efficient. This could be achieved by developing automated tools to generate an initial API description based on some source code annotations or comments. API developers could then manually edit and refine this initial version of the API description as needed.

In addition, the proposed API description language does not currently account for API versions, which could be problematic as APIs evolve over time. This limitation means that API developers must manually update the API description file whenever new features are introduced (e.g. adding new methods) or when certain API elements are removed in a new version. Tracking API versions and ensuring that API code examples remain valid across all versions would provide significant benefits. Moreover, incorporating metadata and enabling the description of the API behaviour, such as method call preconditions and recommended usage patterns, would enhance the expressiveness of the language.

Another aspect that requires further investigation and research is the portability of the proposed API description language to other general-purpose programming languages beyond Java. The clear, hierarchical structure of the proposed API specification language makes it adaptable for de-

scribing APIs in other programming languages. However, its portability depends on various factors, such as the target language’s paradigms and API design conventions. For example, C# is a programming language that is similar to Java, sharing very similar syntax, static typing, and object-oriented features, which makes the proposed API description language highly portable to it. Adaptation for C# may only require incorporating C#-specific features, such as the use of properties. In contrast, adapting the proposed API description language for programming languages that differ significantly from Java would require more extensions and modifications to its grammar to support the target language-specific features. For instance, porting the proposed API specification language to Python would require changes, such as support for module-based APIs and dynamic typing.

## 6.6 Threats to Validity

Several threats to validity that may have affected the accuracy or generalisability of the obtained results have been identified. The following sections discuss these threats in more detail and explain the steps taken to mitigate them.

### 6.6.1 Construct Validity

The results reported in this chapter were based on a direct comparison between ‘full API descriptions’, which were auto-generated and acted as a baseline, and ‘shortened API descriptions’, which showed how the proposed API specification language’s features can reduce verbosity when describing API elements. For example, instead of using fully qualified names for methods’ and constructors’ parameter types, users can use an underscore symbol, simple type names, or even the API element name alone if it is enough to identify the API element. However, this may have affected the results re-

## 6.6. Threats to Validity

trieved using the character count reduction metric since all the parameter types of methods and constructors in the full API descriptions were presented with fully qualified names. To mitigate this, an additional metric, i.e. element count reduction, was used, as it is not impacted by the omission of fully qualified names of parameters' types. Furthermore, to ensure a fair comparison, the content structure remained consistent across both versions of the API descriptions. However, an additional evaluation, e.g. a usability study or qualitative feedback from API developers, could give further insights into the benefits of the language's features.

### 6.6.2 Internal Validity

The internal validity threats in this study may arise from various factors, such as API size and the naming conventions used for API elements. This is particularly relevant since the metrics used, i.e. the number of API elements and character count, in this evaluation are highly sensitive to factors like code style and formatting. These types of threats were mitigated by applying strict and repeatable shortening techniques and rules across both versions of the API description (full and shortened). For example, effort was made to ensure that both the full and shortened API descriptions have the same structure, the same number of listed Java packages and types, and that the number of listed methods and constructors remains the same before the shortening rules were applied. Furthermore, these rules also helped reduce subjectivity that could arise from manually shortening API descriptions. Also, selecting four subject Java libraries from the same dataset using clear extraction criteria has reduced potential selection bias.

### 6.6.3 External Validity

The external validity threats may stem from the fact that only four Java libraries were selected as subject candidates in this evaluation. This could

limit the generalisability of the achieved results to libraries from other domains or programming languages. However, to mitigate such threats, steps were taken to ensure that the dataset from which the evaluated libraries were retrieved is representative and includes a diverse collection of Java libraries. Furthermore, the four selected Java libraries were from different domains and are, to some extent, comparable in terms of the number of clients and size. These selection criteria helped enhance the reliability of the results in providing a better understanding of the feasibility of the proposed API specification language and its accompanying example coverage tool. Nonetheless, these proposed approaches would benefit from further investigation and evaluation with a larger set of libraries.

### 6.7 Summary

This chapter discussed the key features and evaluation of a proposed API code example coverage tool. The main purpose of this tool is to analyse API code examples and report coverage information for a set of API elements that are given by API developers. These API elements are described in advance by API developers using a proposed intended API description language. The functionality of the proposed coverage tool, as well as its architecture, was explained in Section 6.1. A detailed discussion of the grammar and use of the proposed API specification language was given in Section 6.1.1. This discussion was supported with an example of an API description to show the key features that the language offers. Later, the implementation of the coverage tool was explained, starting with the analysis and parsing processes in Section 6.1.2. Then, an overview of how the described API elements are validated was presented in Section 6.1.3. The way the tool analyses and extracts coverage information from API code examples was discussed in Section 6.1.4.

The proposed API code example coverage tool produces two main for-



## 6.7. Summary

mats of coverage reports, textual and visual. The reasons for choosing these formats, along with the technologies used to implement them, were explained in Sections 6.1.5 and 6.1.6. The constraints that the proposed coverage tool imposes are outlined in Section 6.2. Subsequently, a detailed explanation of the steps that were followed to evaluate the tool was presented in Section 6.3. This was followed by a discussion on the evaluation results in Section 6.4. Finally, the chapter concludes with key observations on the usability of the proposed tool and highlights some of its limitations in Section 6.5.

## Chapter 7

# Conclusions and Future Work

This thesis has proposed novel approaches for assisting API developers in producing effective API code examples. The overarching goal of these approaches is to improve the availability and maintainability of code examples in API documentation. The thesis has examined the two research hypotheses defined in Section 3.4.

**H<sub>1</sub>:** *API users prefer code examples to be linear. Conversely, linear API code examples contain significant code repetition which makes them challenging for API developers to write and maintain. This gap can be bridged by a code synthesis tool that automatically transforms non-linear example code into linear code.*

**H<sub>2</sub>:** *A domain-specific language (DSL) for API specification can enable API developers to concisely describe the intended surface of their APIs. An API description written using this DSL can serve as the foundation for an API code example coverage tool that provides useful insights specifically tailored to API code example coverage.*

The remainder of this chapter is structured as follows. Section 7.1 summarises the content of each chapter of the thesis. Section 7.2 outlines its

contributions to the field of software engineering. Section 7.3 presents the key results, while Section 7.4 suggests potential directions for future research.

## 7.1 Summary

In this thesis, we presented a set of approaches aimed at mitigating the problem of the shortage of API code examples in official API documentation. This aim was achieved by first conducting an extensive literature review to investigate the possible underlying causes of such shortage, followed by defining the research objectives, hypotheses, and questions that shaped the direction of this research. **Chapter 2** provided a comprehensive background for this research. It discussed the importance of APIs and API documentation in software development. It also presented the capabilities and limitations of several state-of-the-art tools used for API code example generation. **Chapter 3** provided an analysis of the literature review presented in Chapter 2 and outlined the research framework, including the research objectives, questions, hypotheses, and scope. **Chapter 4** presented the methodology and results of a user study conducted to evaluate the impact of source code linearity and length on the comprehensibility and reusability of API code examples. **Chapter 5** presented the architecture, implementation, and evaluation of the proposed linear code synthesis approach. It also discussed its constraints, current limitations, and several potential optimisation techniques. **Chapter 6** presented the grammar and features of a DSL for intended API specification. It also discussed the evaluation results, limitations, and constraints of the DSL. Moreover, it introduced the implementation of a static code analysis tool for measuring API code example coverage.

## 7.2 Thesis Contributions

The key contributions of this thesis are as follows:

- **A controlled code comprehension user study** to investigate how different structural aspects of source code; specifically, the degree of linearity and length, affect API users' comprehension of API code examples. The aim of this user experiment was to provide API developers with insights into the effective structuring of API code examples.
- **A linear code synthesis approach** for the Java programming language designed to reduce the burden on API developers of writing lengthy, repetitive, and costly-to-maintain API code examples. This, in turn, enables them to produce more API code examples, thus, enhancing API learnability for API users.
- **A domain-specific language for intended API specification**, which enables API developers to formally and precisely define the elements that comprise their intended APIs. This DSL includes a set of features designed to facilitate the API description process, making it more efficient.
- **An API code example coverage tool** built on top of the proposed API description DSL. This tool measures the extent to which a given set of code examples covers an intended API. Coverage information is generated in both textual and visual formats.

### 7.3 Research Results

The results presented in this thesis were achieved by first breaking down the overall research goal into more specific objectives, as outlined below:

- (RO<sub>1</sub>) **Conduct a controlled user-based experiment to assess the impact of different source code structures on API users' performance and comprehension of API code examples.**

This research objective was achieved by conducting a user study with

61 Java programmers to evaluate the impact of source code linearity and length on API users' comprehension of code examples. The results of this user experiment are detailed in Chapter 4.

- (RO<sub>2</sub>) Create a dataset of API code examples extracted from multiple open-source Java projects to help understand and specify the extent and nature of code repetition in API code examples.**

This research objective was achieved by collecting API code examples from nine popular open-source Java libraries and analysing them for code repetition. This process is described in detail in Sections 5.3 and 5.4.

- (RO<sub>3</sub>) Develop a code synthesis prototype for the Java programming language that enables API developers to write less repetitive, more maintainable, and comprehensible API code examples for API documentation.**

This was accomplished by contributing an approach for linear source code synthesis, which is explained and evaluated in Chapter 5.

- (RO<sub>4</sub>) Develop a Domain-Specific Language (DSL) that allows API developers to concisely describe API elements intended for external use.**

To achieve this research objective, a DSL was developed to provide formal and concise specifications of intended APIs. Further details on this language and its evaluation are provided in Sections 6.1.1, 6.3, and 6.4.

- (RO<sub>5</sub>) Develop a static code analysis tool, built on top of the proposed DSL, to help API developers assess the coverage of their API code examples.**

This research objective was achieved by contributing an API code ex-

ample coverage tool that provides insights into how a described intended API is covered by code examples. More details on the implementation of this static code analysis tool are provided in Chapter 6.

The research questions outlined below have been formulated to achieve the above-stated research objectives.

- (RQ<sub>1</sub>) How does the linearity of an API code example impact a programmer’s performance in terms of correctness and time spent in tasks that require code comprehension?**

API users generally spent less time comprehending and reusing linear code examples; however, the impact on correctness was not substantial.

- (RQ<sub>2</sub>) What effects does the length of a linear API code example have on its comprehensibility and reusability?**

Study results revealed a significant difference in both comprehension and reusability. Linear code is easier to comprehend and reuse, particularly when it is short.

- (RQ<sub>3</sub>) Does the degree of linearity in a non-linear API example affect its comprehensibility and reusability?**

Yes, participants spent less time reusing the non-linear code examples when the linearity value was lower. However, the differences in participants’ performance here may be attributed to the variations in the implemented API functionality and the tasks required. Further details are in Section 4.2.

- (RQ<sub>4</sub>) How often do API code examples contain duplicate or near-duplicate code that can be eliminated using the proposed linear code synthesis approach?**

The percentages of the API code examples that contained duplicate code ranged between 18% and 56% in the selected Java libraries.

**(RQ<sub>5</sub>) How often are duplicate code fragments repeated across different API code examples?**

A substantial number of duplicate code fragments were recurrent in many of the API code examples subsets. Further details are in Section 5.4.

**(RQ<sub>6</sub>) How much reduction of duplicate code is achieved by the proposed linear code synthesis approach?**

The proposed linear code synthesis approach brought more than a 30% decrease in LOC in five of the nine evaluated Java projects.

**(RQ<sub>7</sub>) How can an intended API be specified in a formal and concise manner?**

The proposed API specification language can reduce the number of API elements by up to 77% and their character count by up to 91% in shortened API descriptions compared to their full versions. Further details on how this reduction is achieved are provided in Section 6.4.

**(RQ<sub>8</sub>) What insights can be extracted by using a formal intended API specification and the proposed API code example coverage tool?**

APIExCov reports coverage information and highlights areas of the API that would benefit from additional illustrative code examples. Further extracted insights are detailed in Sections 6.1.5, 6.1.6, and 6.4.1.

All proposed tools, techniques, and study results in this thesis have confirmed the two main research hypotheses defined in Section 3.2.

## 7.4 Future Work

As with any piece of work, there is always room for optimisation and the addition of more features. The following aspects have been identified as part of the future work for this research, as outlined below:

- **Template-based linear code synthesis:** As discussed in Section 5.5.1, the current version of the proposed linear code synthesiser does not eliminate all types of code repetition found in API code examples due to constraints imposed by Java syntax. This limitation can be addressed by incorporating a template language such as Apache Velocity<sup>1</sup> to create code templates that serve as skeletons for the unsupported repetitive code patterns. In these templates, the values of the elements that differ will be put as placeholders. Subsequently, the linear code examples will be synthesised based on the invoked template. However, while this extension could further improve the flexibility and usability of the proposed linear code synthesis approach, it may also introduce additional complexity and readability issues into the synthesis workflow.
- **Interactive generation of linear API code examples:** Another extension of the current version of the proposed linear code synthesis approach is to enable API users to interactively specify the values with which the linear API code examples are generated. This can be achieved by introducing a new Java annotation (i.e. `@DocGen`), through which API users would be prompted to enter some values as command-line arguments. These values would then be used to expand/inline the documentation methods provided by API developers beforehand. The concept behind this approach is similar to the concept of JUnit

---

<sup>1</sup><https://velocity.apache.org>



Theories,<sup>2</sup> and those command line values passed by API users can be equivalent to the values of the data points in JUnit Theories.

- **Expansion of the code comprehension user study:** As presented in Chapter 4, the controlled experiment used to evaluate the hypothesis in this thesis was conducted online with a small number of API code examples from a single Java library, i.e. Joda-Time. We recommend expanding this study by incorporating a larger set of code examples from more diverse APIs and increasing the number of participants. Also, conducting the experiment in a controlled physical setting could enhance experimental validity. Furthermore, it would be interesting to investigate whether the participants' activities, as they work with code examples of different linearity levels, align with those reported in existing studies on the COIL<sup>3</sup> model [114, 143]. In addition, further insights can be gained by involving Large Language Models (LLMs) to examine whether they show similar comprehension and reusability results for both linear and non-linear API code examples, compared to those obtained from human participants.
- **Optimisation of the API specification language:** The proposed DSL for intended API descriptions can be further enhanced to include features that provide more customised coverage information. For instance, it would be more practical if API developers could define, in advance within the API description YAML file, the usage type of interest, e.g. `INHERITANCE` for an API element of type `Type`, to be only reported in the coverage results. This feature would ensure that the reported coverage information is better aligned with the API developers' specific goals of creating API code examples.

---

<sup>2</sup><https://junit.org/junit4/javadoc/4.12/org/junit/experimental/theories/Theories.html>

<sup>3</sup>Collection and Organization of Information for Learning.

- **Automatic generation of intended API specifications:** Currently, the API description language requires API developers to manually define their intended APIs, which can be tedious, particularly for large APIs. Enabling the automatic generation of such API descriptions (as discussed in Section 6.5) would further enhance the usability of the proposed DSL.

# Appendices

## Appendix A

# Code Reuse Task Scoring Rubrics

Category	Letter	Numerical Value (%)	Description
Correct	A	100%	Answers that are <b>correct</b> completely and accurately address the task.
Almost Correct	B	70%	Answers that are <b>largely correct</b> but have <b>minor errors</b> or omissions.
Partially Correct	C	40%	Answers that address <b>only half</b> of the task.
Incorrect	D	0%	Answers that are largely <b>incorrect</b> or don't address the task's requirements.
No answer	F	0%	<b>No response</b> to the task

# Criteria - part 1 of the task

	Correct	Almost Correct	Partially Correct	Incorrect	No answer
<b>ChronologyExample</b>	Integrates an additional chronology alongside the Buddhist chronology. Displays the year, month, and day from both chronologies. no errors or issues in the code.	Same as <b>(Correct)</b> but some errors or issues in the code are present but do not significantly affect functionality.	Display the year, month, and day from one chronology correctly but not the other. Display some of the elements but not the rest (e.g. month but not year).	Attempts to provide a solution but is fundamentally incorrect or incomplete. The code does not work with major errors.	Response provides no code.
<b>DateExample</b>	The code successfully retrieves future and past dates. It accurately skips weekend days and Mondays. There are no errors or issues.	Same as <b>(Correct)</b> but some errors or issues in the code are present but do not significantly affect functionality.	Display one date accurately but not the other. It skips weekend days but not Mondays and vice versa.	Attempts to provide a solution but is fundamentally incorrect or incomplete. The code does not work with major errors.	Response provides no code.
<b>DurationExample</b>	The code successfully excludes the specified holiday (Summer Bank Holiday) from the date calculations. accurately updates the relevant variables and output to reflect the modified calculation. There are no errors.	Same as <b>(Correct)</b> but some errors or issues in the code are present but do not significantly affect functionality.	Excludes the specified holiday but might have issues with the update of variables or output. Defines a day for the bank holiday but does not correctly use it. There are notable errors or issues in the code, but some elements are functional.	Attempts to provide a solution but is fundamentally incorrect or incomplete. The code does not work with major errors.	Response provides no code.
<b>IntervalExample</b>	The code successfully prevents	Same as <b>(Correct)</b> but some errors or	The code partially adds a feature to	Attempts to provide a solution but is	Response provides no code.

	<p>meeting scheduling if a clash with a previously scheduled meeting is detected <b>using the overlap() API method</b>. It accurately checks for clashes and cancels or reschedules meetings accordingly. There are no errors.</p>	<p>issues in the code are present but do not significantly affect functionality.</p>	<p>prevent meeting clashes but might have issues with the implementation. Checks for clashes but does not use them to prevent meeting scheduling.</p>	<p>fundamentally incorrect or incomplete. The code does not work with major errors.</p>	
--	--	--	---	---	--

# Criteria - part 2 of the task

	Correct
<b>ChronologyExample</b>	Removed <b>LOC</b> : 40, 51-71 - code in Util.java is not counted.
<b>DateExample</b>	Removed <b>LOC</b> : 70-75 - code in Util.java is not counted.
<b>DurationExample</b>	Removed <b>LOC</b> : 73-86 - code in Util.java is not counted.
<b>IntervalExample</b>	Removed <b>LOC</b> : 82-89, 97-104 - code in Util.java is not counted.



## Appendix B

```
1 packages :
2   - name : org.jsoup
3     types :
4       - name : HttpStatusException
5         methods :
6           - getStatusCode()
7           - getUrl()
8       - name : Jsoup
9         methods :
10          - clean(java.lang.String, java.lang.String, org.jsoup.safety.
              Safelist, org.jsoup.nodes.Document$OutputSettings)
11          - clean(java.lang.String, java.lang.String, org.jsoup.safety.
              Safelist)
12          - clean(java.lang.String, org.jsoup.safety.Safelist)
13          - connect(java.lang.String)
14          - isValid(java.lang.String, org.jsoup.safety.Safelist)
15          - newSession()
16          - parse(java.io.File, java.lang.String, java.lang.String, org.
              jsoup.parser.Parser)
17          - parse(java.io.File, java.lang.String, java.lang.String)
18          - parse(java.io.File, java.lang.String)
19          - parse(java.io.File)
20          - parse(java.io.InputStream, java.lang.String, java.lang.
              String, org.jsoup.parser.Parser)
21          - parse(java.io.InputStream, java.lang.String, java.lang.
              String)
22          - parse(java.lang.String, java.lang.String, org.jsoup.parser.
              Parser)
23          - parse(java.lang.String, java.lang.String)
24          - parse(java.lang.String, org.jsoup.parser.Parser)
```

## Appendix B.

```
25         - parse(java.lang.String)
26         - parse(java.net.URL,int)
27         - parseBodyFragment(java.lang.String,java.lang.String)
28         - parseBodyFragment(java.lang.String)
29     - name: UncheckedIOException
30       constructors:
31         - UncheckedIOException(IOException)
32       methods:
33         - getMimeType()
34         - getUrl()
35         - toString()
36     - name: Connection
37       methods:
38         - cookie(java.lang.String,java.lang.String)
39         - cookies(java.util.Map)
40         - data(java.lang.String,java.lang.String,java.io.InputStream
41             ,java.lang.String)
42         - data(java.lang.String,java.lang.String,java.io.InputStream
43             )
44         - data(java.lang.String,java.lang.String)
45         - data(java.lang.String[])
46         - data(java.util.Map)
47         - execute()
48         - followRedirects(boolean)
49         - get()
50         - header(java.lang.String,java.lang.String)
51         - headers(java.util.Map)
52         - ignoreContentType(boolean)
53         - ignoreHttpErrors(boolean)
54         - maxBodySize(int)
55         - method(org.jsoup.Connection$Method)
56         - newRequest()
57         - parser(org.jsoup.parser.Parser)
58         - post()
59         - postDataCharset(java.lang.String)
60         - proxy(java.lang.String,int)
61         - proxy(java.net.Proxy)
62         - referrer(java.lang.String)
63         - request()
64         - requestBody(java.lang.String)
65         - response()
```

```

65         - timeout(int)
66         - url(java.lang.String)
67         - userAgent(java.lang.String)
68     - name: Base
69         methods:
70             - addHeader(java.lang.String, java.lang.String)
71             - cookie(java.lang.String, java.lang.String)
72             - cookie(java.lang.String)
73             - cookies()
74             - hasCookie(java.lang.String)
75             - hasHeader(java.lang.String)
76             - hasHeaderWithValue(java.lang.String, java.lang.String)
77             - header(java.lang.String, java.lang.String)
78             - header(java.lang.String)
79             - headers()
80             - headers(java.lang.String)
81             - method()
82             - method(org.jsoup.Connection$Method)
83             - multiHeaders()
84             - removeCookie(java.lang.String)
85             - removeHeader(java.lang.String)
86             - url()
87             - url(java.net.URL)
88     - name: Request
89         methods:
90             - data()
91             - parser()
92             - proxy()
93             - proxy(java.net.Proxy)
94             - requestBody()
95             - timeout()
96     - name: KeyVal
97         methods:
98             - hasInputStream()
99             - inputStream(java.io.InputStream)
100             - key()
101             - value()
102             - value(java.lang.String)
103     - name: Response
104         methods:
105             - body()
106             - bodyAsBytes()

```

## Appendix B.

```
107         - bodyStream()
108         - bufferUp()
109         - charset()
110         - charset(java.lang.String)
111         - contentType()
112         - parse()
113         - statusCode()
114         - statusMessage()
115     enums:
116         - name: Method
117 - name: org.jsoup.helper
118     types:
119         - name: DataUtil
120         methods:
121             - readToByteBuffer(java.io.InputStream,int)
122 - name: HttpConnection
123         constructors:
124             - HttpConnection()
125         methods:
126             - connect(java.lang.String)
127             - connect(java.net.URL)
128             - execute()
129             - request()
130 - name: HttpConnection$KeyVal
131         methods:
132             - create(java.lang.String,java.lang.String,java.io.
                InputStream)
133             - create(java.lang.String,java.lang.String)
134 - name: HttpConnection$Request
135 - name: HttpConnection$Response
136 - name: Validate
137         methods:
138             - isTrue(boolean,java.lang.String)
139             - notNull(java.lang.Object)
140             - notNullParam(java.lang.Object,java.lang.String)
141 - name: W3CDom
142         constructors:
143             - W3CDom()
144         methods:
145             - asString(org.w3c.dom.Document,java.util.Map)
146             - asString(org.w3c.dom.Document)
147             - contextNode(org.w3c.dom.Document)
```

```

148         - convert(org.jsoup.nodes.Document)
149         - fromJsoup(org.jsoup.nodes.Document)
150         - fromJsoup(org.jsoup.nodes.Element)
151         - namespaceAware()
152         - namespaceAware(boolean)
153         - OutputHtml()
154         - OutputXml()
155     - name: org.jsoup.internal
156     types:
157         - name: ConstrainableInputStream
158           methods:
159             - readToByteBuffer(int)
160             - reset()
161         - name: StringUtil
162           methods:
163             - in(java.lang.String, java.lang.String[])
164             - inSorted(java.lang.String, java.lang.String[])
165             - isAscii(java.lang.String)
166             - isBlank(java.lang.String)
167             - isNumeric(java.lang.String)
168             - isWhitespace(int)
169             - join(java.lang.String[], java.lang.String)
170             - join(java.util.Collection, java.lang.String)
171             - normaliseWhitespace(java.lang.String)
172             - padding(int, int)
173             - padding(int)
174             - resolve(java.lang.String, java.lang.String)
175     - name: org.jsoup.nodes
176     types:
177         - name: Attribute
178           constructors:
179             - Attribute()
180             - Attribute(java.lang.String, java.lang.String)
181           methods:
182             - getKey()
183             - getValue()
184             - hasDeclaredValue()
185             - html()
186             - isBooleanAttribute(java.lang.String)
187             - setKey(java.lang.String)
188             - setValue(java.lang.String)
189             - toString()

```

## Appendix B.

```
190         - name: Attributes
191           methods:
192             - add(java.lang.String, java.lang.String)
193             - asList()
194             - clone()
195             - dataset()
196             - get(java.lang.String)
197             - getIgnoreCase(java.lang.String)
198             - hasDeclaredValueForKey(java.lang.String)
199             - hasDeclaredValueForKeyIgnoreCase(java.lang.String)
200             - hasKey(java.lang.String)
201             - hasKeyIgnoreCase(java.lang.String)
202             - html()
203             - iterator()
204             - put(java.lang.String, java.lang.String)
205             - put(org.jsoup.nodes.Attribute)
206             - remove(java.lang.String)
207             - size()
208             - toString()
209         - name: CDataNode
210           methods:
211             - text()
212         - name: Comment
213           methods:
214             - asXmlDeclaration()
215             - clone()
216             - getData()
217             - isXmlDeclaration()
218             - nodeName()
219             - setData(java.lang.String)
220             - toString()
221         - name: DataNode
222           constructors:
223             - DataNode(java.lang.String)
224           methods:
225             - getWholeData()
226             - setWholeData(java.lang.String)
227             - toString()
228         - name: Document
229           constructors:
230             - Document(java.lang.String)
231           methods:
```

```

232         - body()
233         - charset()
234         - charset(java.nio.charset.Charset)
235         - clone()
236         - connection()
237         - createElement(java.lang.String)
238         - createShell(java.lang.String)
239         - documentType()
240         - expectForm(java.lang.String)
241         - forms()
242         - head()
243         - location()
244         - normalise()
245         - outerHtml()
246         - outputSettings()
247         - outputSettings(org.jsoup.nodes.Document$OutputSettings)
248         - parser()
249         - text(java.lang.String)
250         - title()
251         - title(java.lang.String)
252         - updateMetaCharsetElement()
253         - updateMetaCharsetElement(boolean)
254     - name: DocumentType
255         constructors:
256             - DocumentType(java.lang.String, java.lang.String, java.lang.
                String)
257         methods:
258             - publicId()
259             - systemId()
260     - name: Document$OutputSettings
261         constructors:
262             - OutputSettings()
263         methods:
264             - charset()
265             - charset(java.lang.String)
266             - charset(java.nio.charset.Charset)
267             - escapeMode()
268             - escapeMode(org.jsoup.nodes.Entities$EscapeMode)
269             - indentAmount(int)
270             - maxPaddingWidth()
271             - maxPaddingWidth(int)
272             - outline(boolean)

```

## Appendix B.

```
273         - prettyPrint()
274         - prettyPrint(boolean)
275         - syntax()
276         - syntax(org.jsoup.nodes.Document$OutputSettings$Syntax)
277     - name: Element
278         constructors:
279             - Element(java.lang.String)
280             - Element(org.jsoup.parser.Tag, java.lang.String)
281             - Element(org.jsoup.parser.Tag, java.lang.String, org.jsoup.
                nodes.Attributes)
282             - Elements(org.jsoup.nodes.Element[])
283             - Elements()
284         methods:
285             - addClass(java.lang.String)
286             - after(java.lang.String)
287             - after(org.jsoup.nodes.Node)
288             - append(java.lang.String)
289             - appendChild(org.jsoup.nodes.Node)
290             - appendChildren(java.util.Collection)
291             - appendElement(java.lang.String)
292             - appendText(java.lang.String)
293             - appendTo(org.jsoup.nodes.Element)
294             - attr(java.lang.String, boolean)
295             - attr(java.lang.String, java.lang.String)
296             - attributes()
297             - baseUrl()
298             - before(java.lang.String)
299             - before(org.jsoup.nodes.Node)
300             - child(int)
301             - childNodeSize()
302             - children()
303             - childrenSize()
304             - className()
305             - classNames()
306             - classNames(java.util.Set)
307             - clearAttributes()
308             - clone()
309             - closest(java.lang.String)
310             - closest(org.jsoup.select.Evaluator)
311             - cssSelector()
312             - data()
313             - dataNodes()
```



```

314         - dataset()
315         - doSetBaseUri(java.lang.String)
316         - elementSiblingIndex()
317         - empty()
318         - endSourceRange()
319         - expectFirst(java.lang.String)
320         - filter(org.jsoup.select.NodeFilter)
321         - firstElementChild()
322         - firstElementSibling()
323         - forEach(org.jsoup.helper.Consumer)
324         - getAllElements()
325         - getElementById(java.lang.String)
326         - getElementsByAttribute(java.lang.String)
327         - getElementsByAttributeStarting(java.lang.String)
328         - getElementsByAttributeValue(java.lang.String, java.lang.
           String)
329         - getElementsByAttributeValueContaining(java.lang.String,
           java.lang.String)
330         - getElementsByAttributeValueEnding(java.lang.String, java.
           lang.String)
331         - getElementsByAttributeValueMatching(java.lang.String, java.
           lang.String)
332         - getElementsByAttributeValueMatching(java.lang.String, java.
           util.regex.Pattern)
333         - getElementsByAttributeValueNot(java.lang.String, java.lang.
           String)
334         - getElementsByAttributeValueStarting(java.lang.String, java.
           lang.String)
335         - getElementsByClass(java.lang.String)
336         - getElementsByIndexEquals(int)
337         - getElementsByIndexGreaterThan(int)
338         - getElementsByIndexLessThan(int)
339         - getElementsByTag(java.lang.String)
340         - getElementsContainingOwnText(java.lang.String)
341         - getElementsContainingText(java.lang.String)
342         - getElementsMatchingOwnText(java.lang.String)
343         - getElementsMatchingOwnText(java.util.regex.Pattern)
344         - getElementsMatchingText(java.lang.String)
345         - hasAttributes()
346         - hasChildNodes()
347         - hasClass(java.lang.String)
348         - hasText()

```

## Appendix B.

```
349         - html()
350         - html(java.lang.Appendable)
351         - html(java.lang.String)
352         - id()
353         - id(java.lang.String)
354         - insertChildren(int, java.util.Collection)
355         - insertChildren(int, org.jsoup.nodes.Node[])
356         - is(java.lang.String)
357         - is(org.jsoup.select.Evaluator)
358         - isBlock()
359         - lastElementChild()
360         - lastElementSibling()
361         - nextElementSibling()
362         - nextElementSiblings()
363         - nodeName()
364         - normalName()
365         - ownText()
366         - parent()
367         - parents()
368         - prepend(java.lang.String)
369         - prependChild(org.jsoup.nodes.Node)
370         - prependChildren(java.util.Collection)
371         - prependElement(java.lang.String)
372         - prependText(java.lang.String)
373         - previousElementSibling()
374         - previousElementSiblings()
375         - removeAttr(java.lang.String)
376         - removeClass(java.lang.String)
377         - root()
378         - select(java.lang.String)
379         - select(org.jsoup.select.Evaluator)
380         - selectFirst(java.lang.String)
381         - selectFirst(org.jsoup.select.Evaluator)
382         - selectXpath(java.lang.String, java.lang.Class)
383         - selectXpath(java.lang.String)
384         - shallowClone()
385         - siblingElements()
386         - tag()
387         - tagName()
388         - tagName(java.lang.String)
389         - text()
390         - text(java.lang.String)
```

```

391         - textNodes()
392         - toggleClass(java.lang.String)
393         - traverse(org.jsoup.select.NodeVisitor)
394         - val()
395         - val(java.lang.String)
396         - wholeOwnText()
397         - wholeText()
398         - wrap(java.lang.String)
399     - name: Entities
400       methods:
401         - escape(java.lang.String,org.jsoup.nodes.
              Document$OutputSettings)
402         - escape(java.lang.String)
403         - getByName(java.lang.String)
404         - unescape(java.lang.String)
405     - name: FormElement
406       methods:
407         - elements()
408         - formData()
409         - submit()
410     - name: Node
411       methods:
412         - absUrl(java.lang.String)
413         - addChildren(int,org.jsoup.nodes.Node[])
414         - addChildren(org.jsoup.nodes.Node[])
415         - after(java.lang.String)
416         - after(org.jsoup.nodes.Node)
417         - attr(java.lang.String,java.lang.String)
418         - attr(java.lang.String)
419         - attributes()
420         - attributesSize()
421         - baseUrl()
422         - before(java.lang.String)
423         - before(org.jsoup.nodes.Node)
424         - childNode(int)
425         - childNodes()
426         - childNodesCopy()
427         - childNodeSize()
428         - clearAttributes()
429         - clone()
430         - doSetBaseUrl(java.lang.String)
431         - empty()

```

## Appendix B.

```
432         - ensureChildNodes()
433         - equals(java.lang.Object)
434         - filter(org.jsoup.select.NodeFilter)
435         - firstChild()
436         - forEachNode(org.jsoup.helper.Consumer)
437         - hasAttr(java.lang.String)
438         - hasAttributes()
439         - hashCode()
440         - hasParent()
441         - hasSameValue(java.lang.Object)
442         - html(java.lang.Appendable)
443         - lastChild()
444         - nextSibling()
445         - nodeName()
446         - normalName()
447         - outerHtml()
448         - outerHtml(java.lang.Appendable)
449         - ownerDocument()
450         - parent()
451         - parentNode()
452         - previousSibling()
453         - remove()
454         - removeAttr(java.lang.String)
455         - replaceWith(org.jsoup.nodes.Node)
456         - root()
457         - setBaseUri(java.lang.String)
458         - shallowClone()
459         - siblingIndex()
460         - siblingNodes()
461         - sourceRange()
462         - toString()
463         - traverse(org.jsoup.select.NodeVisitor)
464         - unwrap()
465         - wrap(java.lang.String)
466     - name: CDataNode
467       constructors:
468         - CDataNode(java.lang.String)
469       methods:
470         - text
471     - name: Range
472       methods:
473         - end()
```

```

474         - isTracked()
475         - start()
476         - toString()
477     - name: Range$Position
478       methods:
479         - columnNumber()
480         - isTracked()
481         - lineNumber()
482         - pos()
483         - toString()
484     - name: TextNode
485       constructors:
486         - TextNode(java.lang.String)
487       methods:
488         - clone()
489         - createFromEncoded(java.lang.String)
490         - getWholeText()
491         - isBlank()
492         - nodeName()
493         - splitText(int)
494         - text()
495         - text(java.lang.String)
496         - toString()
497     - name: XmlDeclaration
498       constructors:
499         - XmlDeclaration(java.lang.String,boolean)
500       methods:
501         - getWholeDeclaration()
502         - name()
503 - name: org.jsoup.parser
504   types:
505     - name: CharacterReader
506       constructors:
507         - CharacterReader(java.lang.String)
508         - CharacterReader(java.io.Reader,int)
509         - CharacterReader(java.io.Reader)
510       methods:
511         - advance()
512         - columnNumber()
513         - consumeTo(char)
514         - consumeToAny(char[])
515         - current()

```

## Appendix B.

```
516         - isEmpty()
517         - isTrackNewlines()
518         - lineNumber()
519         - pos()
520         - trackNewlines(boolean)
521     - name: ParseError
522       methods:
523         - getErrorMessage()
524         - getPosition()
525         - toString()
526     - name: Parser
527       constructors:
528         - Parser(org.jsoup.parser.TreeBuilder)
529       methods:
530         - getErrors()
531         - getTreeBuilder()
532         - htmlParser()
533         - isTrackPosition()
534         - parseBodyFragment(java.lang.String, java.lang.String)
535         - parseFragment(java.lang.String, org.jsoup.nodes.Element,
536                           java.lang.String, org.jsoup.parser.ParseErrorList)
537         - parseFragment(java.lang.String, org.jsoup.nodes.Element,
538                           java.lang.String)
539         - parseFragmentInput(java.lang.String, org.jsoup.nodes.
540                               Element, java.lang.String)
541         - parseInput(java.lang.String, java.lang.String)
542         - parseXmlFragment(java.lang.String, java.lang.String)
543         - settings(org.jsoup.parser.ParseSettings)
544         - setTrackErrors(int)
545         - setTrackPosition(boolean)
546         - unescapeEntities(java.lang.String, boolean)
547         - xmlParser()
548     - name: ParseSettings
549       constructors:
550         - ParseSettings(boolean, boolean)
551       methods:
552         - normalizeAttribute(java.lang.String)
553         - normalizeTag(java.lang.String)
554     - name: Tag
555       methods:
556         - formatAsBlock()
557         - getName()
```

```

555         - isBlock()
556         - isEmpty()
557         - isInline()
558         - isKnownTag(java.lang.String)
559         - isSelfClosing()
560         - toString()
561         - valueOf(java.lang.String,org.jsoup.parser.ParseSettings)
562         - valueOf(java.lang.String)
563     - name: TokenQueue
564       constructors:
565         - TokenQueue(java.lang.String)
566       methods:
567         - addFirst(java.lang.String)
568         - chompBalanced(char, char)
569         - chompToIgnoreCase(java.lang.String)
570         - consumeCssIdentifier()
571         - consumeElementSelector()
572         - consumeTo(java.lang.String)
573         - consumeWhitespace()
574         - consumeWord()
575         - escapeCssIdentifier(java.lang.String)
576         - isEmpty()
577         - remainder()
578         - unescape(java.lang.String)
579     - name: XmlTreeBuilder
580     - name: HtmlTreeBuilder
581     - name: ParseErrorList
582       methods:
583         - tracking
584   - name: org.jsoup.safety
585     types:
586       - name: Cleaner
587         constructors:
588           - Cleaner(org.jsoup.safety.Safelist)
589         methods:
590           - clean(org.jsoup.nodes.Document)
591           - isValid(org.jsoup.nodes.Document)
592     - name: Safelist
593       constructors:
594         - Safelist()
595         - Safelist(org.jsoup.safety.Safelist)
596       methods:

```

## Appendix B.

```
597         - addAttributes(java.lang.String, java.lang.String[])
598         - addEnforcedAttribute(java.lang.String, java.lang.String,
          java.lang.String)
599         - addProtocols(java.lang.String, java.lang.String, java.lang.
          String[])
600         - addTags(java.lang.String[])
601         - basic()
602         - basicWithImages()
603         - isSafeAttribute(java.lang.String, org.jsoup.nodes.Element,
          org.jsoup.nodes.Attribute)
604         - isSafeTag(java.lang.String)
605         - none()
606         - preserveRelativeLinks(boolean)
607         - relaxed()
608         - removeAttributes(java.lang.String, java.lang.String[])
609         - removeEnforcedAttribute(java.lang.String, java.lang.String)
610         - removeProtocols(java.lang.String, java.lang.String, java.
          lang.String[])
611         - removeTags(java.lang.String[])
612         - simpleText()
613     - name: org.jsoup.select
614       types:
615         - name: Elements
616           constructors:
617             - Elements(org.jsoup.nodes.Element[])
618             - Elements()
619           methods:
620             - addClass(java.lang.String)
621             - after(java.lang.String)
622             - append(java.lang.String)
623             - attr(java.lang.String, java.lang.String)
624             - attr(java.lang.String)
625             - before(java.lang.String)
626             - clone()
627             - comments()
628             - dataNodes()
629             - eachAttr(java.lang.String)
630             - eachText()
631             - empty()
632             - eq(int)
633             - first()
634             - forms()
```



```

635         - hasAttr(java.lang.String)
636         - hasClass(java.lang.String)
637         - hasText()
638         - html()
639         - html(java.lang.String)
640         - is(java.lang.String)
641         - last()
642         - next()
643         - next(java.lang.String)
644         - nextAll()
645         - nextAll(java.lang.String)
646         - not(java.lang.String)
647         - outerHtml()
648         - parents()
649         - prepend(java.lang.String)
650         - prev()
651         - prev(java.lang.String)
652         - prevAll()
653         - prevAll(java.lang.String)
654         - remove()
655         - removeAttr(java.lang.String)
656         - removeClass(java.lang.String)
657         - select(java.lang.String)
658         - tagName(java.lang.String)
659         - text()
660         - textNodes()
661         - toggleClass(java.lang.String)
662         - toString()
663         - traverse(org.jsoup.select.NodeVisitor)
664         - unwrap()
665         - val()
666         - val(java.lang.String)
667         - wrap(java.lang.String)
668     - name: NodeTraversor
669     methods:
670         - filter(org.jsoup.select.NodeFilter,org.jsoup.select.
            Elements)
671         - traverse(org.jsoup.select.NodeVisitor,org.jsoup.nodes.Node
            )
672     - name: QueryParser
673     methods:
674         - parse(java.lang.String)

```

## Appendix B.

```
675     - name: CombiningEvaluator
676     - name: CombiningEvaluator$And
677       methods:
678         - toString()
679     - name: Or
680     - name: Evaluator
681     - name: Tag
682       constructors:
683         - Tag(java.lang.String)
684     - name: Selector
685     - name: SelectorParseException
686       constructors:
687         - SelectorParseException(java.lang.String)
688     - name: NodeVisitor
689       methods:
690         - head(org.jsoup.nodes.Node,int)
691         - tail(org.jsoup.nodes.Node,int)
692     - name: NodeFilter
693       methods:
694         - head(org.jsoup.nodes.Node,int)
695         - tail(org.jsoup.nodes.Node,int)
696     enums:
697     - name: FilterResult
```

Listing B.1: Full intended API description of the Jsoup Java library.

```
1 packages:
2   - name: org.jsoup
3     types:
4       - name: HttpStatusException
5         methods:
6           - get*
7       - name: Jsoup
8         methods:
9           - connect
10          - isValid
11          - newSession
12          - parseBodyFragment*
13          - clean*
14          - parse*
15       - name: UncheckedIOException
16         constructors:
```

```

17         - UncheckedIOException(IOException)
18     - name: UnsupportedOperationException
19         methods:
20             - getMimeType
21             - getUrl
22             - toString
23     - name: Connection
24         methods:
25             - -url(_)
26             - -sslSocketFactory
27             - -data(Collection<KeyVal>)
28             - -cookieStore
29             - -request(_)
30             - -response(_)
31     - name: Base
32         methods:
33             - "*"
34     - name: Request
35         methods:
36             - -proxy(_,_)
37             - proxy*
38             - parser()
39             - requestBody()
40             - data()
41             - timeout()
42     - name: KeyVal
43         methods:
44             - -key(_)
45             - -inputStream()
46     - name: Response
47         methods:
48             - "*"
49     enums:
50         - name: Method
51     - name: org.jsoup.helper
52     types:
53         - name: DataUtil
54             methods:
55                 - readToByteBuffer
56         - name: HttpConnection
57             constructors:
58                 - HttpConnection()

```

## Appendix B.

```
59         methods :
60             - connect*
61             - execute
62             - request*
63     - name : KeyVal
64         methods :
65             - create*
66     - name : Request
67     - name : Response
68     - name : Validate
69         methods :
70             - notNullParam
71             - isTrue( _, _ )
72             - notNull( _ )
73     - name : ValidationException
74     - name : W3CDom
75         constructors :
76             - "*"
77         methods :
78             - asString
79             - contextNode
80             - Output*
81             - fromJsoup*
82             - namespaceAware*
83             - convert( _ )
84     - name : Consumer
85         methods :
86             - "*"
87 - name : org.jsoup.internal
88 types :
89     - name : ConstrainableInputStream
90         methods :
91             - readToByteBuffer
92             - reset
93     - name : StringUtil
94         methods :
95             - in
96             - inSorted
97             - isAscii
98             - isBlank
99             - isNumeric
100            - isWhitespace
```

```

101         - normaliseWhitespace
102         - padding
103         - join*
104         - resolve*
105         - -join(Iterator<?>,_ )
106         - -resolve(URL,_ )
107 - name: org.jsoup.nodes
108   types:
109     - name: Attribute
110       constructors:
111         - - Attribute(_,_ ,_)
112       methods:
113         - get*
114         - hasDeclaredValue
115         - isBooleanAttribute
116         - set*
117         - toString
118         - -getValidKey
119         - html*
120       conditions:
121         - excMethods:
122           - "name === 'html' && accessModifier === 'protected'"
123 - name: Attributes
124   methods:
125     - add
126     - asList
127     - clone
128     - dataset
129     - remove
130     - hasDeclared*
131     - hasKey*
132     - iterator
133     - size
134     - toString
135     - get*
136     - put*
137     - -put(_ ,boolean)
138     - html
139 - name: CDataNode
140   methods:
141     - text
142 - name: Comment

```

## Appendix B.

```
143         constructors:
144             - "*"
145         methods:
146             - "*"
147     - name: DataNode
148         constructors:
149             - "*"
150         methods:
151             - "*WholeData"
152             - toString
153     - name: Document
154         constructors:
155             - "*"
156         methods:
157             - -shallowClone
158             - -quirksMode*
159             - -parser(_)
160             - -connection(_)
161     - name: DocumentType
162         constructors:
163             - "*"
164         methods:
165             - name
166             - "*Id"
167     - name: OutputSettings
168         constructors:
169             - "*"
170         methods:
171             - -outline()
172             - -indentAmount()
173     - name: Element
174         constructors:
175             - "*"
176         methods:
177             - -getElementsMatchingText(Pattern)
178             - -forEachNode
179             - -doClone
180             - -ensureChildNodes
181     - name: Entities
182         methods:
183             - escape*
184             - getByName
```

```

185         - unescape
186     - name: FormElement
187         methods:
188             - elements
189             - formData
190             - submit
191     - name: Node
192         methods:
193             - "*"
194         conditions:
195             - excMethods:
196                 - "(accessModifier == 'protected') && (name ==
                    'hasAttributes' || name == 'ensureChildNodes' || name ==
                    'addChildren' || name ==
197                     'outerHtml' )"
198     - name: CDataNode
199         constructors:
200             - "*"
201         methods:
202             - text
203     - name: Range
204         methods:
205             - end
206             - isTracked
207             - start
208             - toString
209     - name: Position
210         methods:
211             - -equals
212             - -hashCode
213     - name: TextNode
214         constructors:
215             - "*"
216         methods:
217             - "*"
218     - name: XmlDeclaration
219         constructors:
220             - "*"
221         methods:
222             - getWholeDeclaration
223             - name
224     - name: org.jsoup.parser

```

## Appendix B.

```
225     types:
226         - name: CharacterReader
227           constructors:
228             - "*"
229           methods:
230             - -close
231             - -toString
232         - name: ParseError
233           methods:
234             - -getCursorPos
235         - name: Parser
236           constructors:
237             - "*"
238           methods:
239             - get*
240             - "*Parser"
241             - isTrackPosition
242             - parse*
243             - settings(_)
244             - setTrack*
245             - unescapeEntities
246             - -parseInput(Reader,_)
247             - -parse
248         - name: ParseSettings
249           constructors:
250             - "*"
251           methods:
252             - normalize*
253         - name: Tag
254           methods:
255             - formatAsBlock
256             - getName
257             - is*
258             - toString
259             - valueOf*
260             - -isKnownTag()
261             - -isForm*
262         - name: TokenQueue
263           constructors:
264             - "*"
265           methods:
266             - addFirst
```



```

267         - chompBalanced
268         - chompToIgnoreCase
269         - consume*
270         - escapeCssIdentifier
271         - isEmpty
272         - remainder
273         - unescape
274         - -consume
275         - -consumeToIgnoreCase
276         - -consumeToAny
277     - name: XmlTreeBuilder
278     - name: HtmlTreeBuilder
279     - name: ParseErrorList
280     methods:
281         - tracking
282 - name: org.jsoup.safety
283   types:
284     - name: Cleaner
285       constructors:
286         - "*"
287       methods:
288         - clean
289         - isValid
290     - name: Safelist
291       constructors:
292         - "*"
293       methods:
294         - "*"
295 - name: org.jsoup.select
296   types:
297     - name: Elements
298       constructors:
299         - Elements()
300         - Elements(Element)
301       methods:
302         - -filter
303     - name: NodeTraversor
304       methods:
305         - filter(_,Elements)
306         - traverse(_,Node)
307     - name: QueryParser
308       methods:

```

## Appendix B.

```
309         - parse
310     - name: CombiningEvaluator
311     - name: And
312     methods:
313         - toString
314     - name: Or
315     - name: Evaluator
316     - name: Tag
317     constructors:
318         - "*"
319     - name: Selector
320     - name: SelectorParseException
321     constructors:
322         - SelectorParseException(_)
323     - name: NodeVisitor
324     methods:
325         - "*"
326     - name: NodeFilter
327     methods:
328         - "*"
329     enums:
330         - name: FilterResult
```

Listing B.2: Shortened intended API description of the Jsoup Java library.

# Bibliography

- [1] G. Monce, T. Couturou, Y. Hamdaoui, T. Degueule, and J.-R. Falleri, “Lightweight Syntactic API Usage Analysis with UCov,” in *32nd IEEE/ACM International Conference on Program Comprehension (ICPC ’24)*, 2024, pp. 426–437.
- [2] S. Jiang, A. Armaly, C. McMillan, Q. Zhi, and R. Metoyer, “Docio: Documenting API Input/Output Examples,” in *IEEE International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, 6 2017, pp. 364–367.
- [3] M. Nassif, Z. Horlacher, and M. P. Robillard, “Casdoc: Unobtrusive Explanations in Code Examples,” in *30th International Conference on Program Comprehension (ICPC ’22)*. Virtual Event, USA. ACM, 2022, pp. 631–635.
- [4] M. P. Robillard, “What makes APIs hard to learn? answers from developers,” *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.
- [5] M. P. Robillard and R. Deline, “A field study of API learning obstacles,” *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 12 2011.
- [6] M. Zibran, F. Eishita, and C. Roy, “Useful, but usable? factors affecting the usability of APIs,” in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 151–155.

## Bibliography

- [7] B. A. Myers and J. Stylos, “Human-centered design can make application programming interfaces easier for developers to use,” *COMMUNICATIONS OF THE ACM*, vol. 59, no. 6, 2016.
- [8] M. Meng, S. Steinhardt, and A. Schubert, “Application Programming Interface Documentation: What Do Software Developers Want?” *Journal of Technical Writing and Communication*, vol. 48, no. 3, pp. 295–330, 2018.
- [9] G. Uddin and M. P. Robillard, “How API Documentation Fails,” *IEEE Software*, vol. 32, no. 4, pp. 68–75, 7 2015.
- [10] D. M. Arya, J. L. Guo, and M. P. Robillard, “Information correspondence between types of documentation for APIs,” *Empirical Software Engineering*, vol. 25, no. 5, pp. 4069–4096, 9 2020.
- [11] G. Bondel, A. Cerit, and F. Matthes, “Challenges of API Documentation from a Provider Perspective and Best Practices for Examples in Public Web API Documentation,” in *International Conference on Enterprise Information Systems, ICEIS - Proceedings*, vol. 2. Science and Technology Publications, Lda, 2022, pp. 268–279.
- [12] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Diruscio, and M. Di Penta, “Recommending API Function Calls and Code Snippets to Support Software Development,” *IEEE Transactions on Software Engineering*, pp. 2417–2438, 2021.
- [13] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, “Toxic Code Snippets on Stack Overflow,” *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 560–581, 3 2021.
- [14] K. Nybom, A. Ashraf, and I. Porres, “A systematic mapping study on API documentation generation approaches,” in *44th Euromicro Con-*

- ference on Software Engineering and Advanced Applications, SEAA*, 2018, pp. 462–469.
- [15] R. P. Buse and W. Weimer, “Synthesizing API usage examples,” in *Proceedings - International Conference on Software Engineering*, 2012, pp. 782–792.
  - [16] M. Ghafari, K. Rubinov, and M. M. Pourhashem K, “Mining unit test cases to synthesize API usage examples,” *Journal of Software: Evolution and Process*, vol. 29, no. 12, 12 2017.
  - [17] J. Kim, S. Lee, S. W. Hwang, and S. Kim, “Enriching documents with examples: A corpus mining approach,” *ACM Transactions on Information Systems*, vol. 31, no. 1, 1 2013.
  - [18] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, “Mining API usage examples from test code,” in *30th International Conference on Software Maintenance and Evolution*, 2014, pp. 301–310.
  - [19] S. M. Nasehi and F. Maurer, “Unit tests as API usage examples,” in *26th IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.
  - [20] M. Kajko-Mattsson, “A Survey of Documentation Practice within Corrective Maintenance,” *Empirical Software Engineering*, vol. 10, pp. 31–55, 2005.
  - [21] E. Raelijohn, M. Famelis, and H. Sahraoui, “Checking temporal patterns of API usage without code execution,” in *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering*. IEEE, 2021, pp. 86–96.

## Bibliography

- [22] S. Nielebock, P. Blockhaus, J. Krüger, and F. Ortmeier, “An Experimental Analysis of Graph-Distance Algorithms for Comparing API Usages,” *arxiv.org*, 2021.
- [23] J. Ofoeda, R. Boateng, and J. Effah, “Application programming interface (API) research: A review of the past to inform the future,” pp. 76–95, 7 2019.
- [24] E. Moritz, L.-V. Mario, P. Denys, and G. Mark, “ExPort: Detecting and Visualizing API Usages in Large Source Code Repositories,” in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*., 2013, pp. 646–651.
- [25] W. Liu, B. Chen, X. Peng, Q. Sun, and W. Zhao, “Identifying change patterns of API misuses from code changes,” *Science China Information Sciences*, vol. 64, no. 3, pp. 1–19, 2021.
- [26] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, and J. Karstens, “A case study of API redesign for improved usability,” *Proceedings - 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008*, pp. 189–192, 2008.
- [27] J. Sae Young, X. Yingyu, B. Jack, A. M. Brad, S. Jeff, E. Ralf, K. Jan, E. Arkin, and K. B. Daniela, “Improving documentation for esoa apis through user studies,” in *End-User Development, V. Pipek, M. B. Rosson, B. de Ruyter, and V. Wulf, Eds*, 2009, pp. 86–105.
- [28] W. G. Lutters and C. B. Seaman, “Revealing actual documentation usage in software maintenance through war stories,” *Information and Software Technology*, vol. 49, no. 6, pp. 576–587, 6 2007.
- [29] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, “You Get Where You’re Looking for: The Impact of Information

- Sources on Code Security,” in *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*. Institute of Electrical and Electronics Engineers Inc., 8 2016, pp. 289–305.
- [30] X. Li, J. Jiang, S. Benton, Y. Xiong, and L. Zhang, “A Large-scale Study on API Misuses in the Wild,” *Proceedings - 2021 IEEE 14th International Conference on Software Testing, Verification and Validation, ICST 2021*, pp. 241–252, 4 2021.
- [31] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. Gall, “Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation,” *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 1004–1023, 9 2020.
- [32] Jan, “javadoc,” 2016. [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>
- [33] “What is JavaDoc tool and how to use it?” 2020. [Online]. Available: <https://www.geeksforgeeks.org/what-is-javadoc-tool-and-how-to-use-it/>
- [34] “Sphinx,” 2022. [Online]. Available: <https://www.sphinx-doc.org/en/master/>
- [35] U. Dekel and J. D. Herbsleb, “Improving API Documentation Usability with Knowledge Pushing,” *Institute for Software Research, School of Computer Science*, p. 632, 2009.
- [36] C. Treude and M. P. Robillard, “Augmenting API documentation with insights from stack overflow,” in *International Conference on Software Engineering*, vol. 14-22-May-, 2016, pp. 392–403.

## Bibliography

- [37] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, “On the comprehension of program comprehension,” *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, 2014.
- [38] M. Kim, L. Bergman, T. Lau, and D. Notkin, “An Ethnographic Study of Copy and Paste Programming Practices in OOPL,” in *The International Symposium on Empirical Software Engineering (ISESE’04)*, 2004.
- [39] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code,” *CHI 2009 Software Development*, 2009.
- [40] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, “What makes a good code example?: A study of programming Q&A in StackOverflow,” in *IEEE International Conference on Software Maintenance, ICSM*, 2012, pp. 25–34.
- [41] M. Meng, S. Steinhardt, and A. Schubert, “How developers use API documentation: an observation study,” *Communication Design Quarterly Review*, vol. 2, no. 7, pp. 40–49, 2019.
- [42] A. Hora, “APISonar: Mining API usage examples,” *Software - Practice and Experience*, vol. 51, no. 2, pp. 319–352, 2 2021.
- [43] S. Radevski, H. Hata, and K. Matsumoto, “Towards Building API Usage Example Metrics,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 619–623.
- [44] T. Couturou, “UCov: A Static Analysis Tool for API Usage Coverage Validation,” Ph.D. dissertation, Bordeaux university, LaBRI, Bordeaux, France, 2023.



- [45] J. Singer, “Practices of Software Maintenance,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, pp. 139–145.
- [46] R. Holmes, R. J. Walker, and G. C. Murphy, “Strathcona Example Recommendation Tool,” in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 237–240.
- [47] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, “Mining succinct and high-coverage API usage patterns from source code,” in *IEEE International Working Conference on Mining Software Repositories*, 2013, pp. 319–328.
- [48] E. Aghajani, C. Nagy, O. L. Vega-Marquez, M. Linares-Vasquez, L. Moreno, G. Bavota, and M. Lanza, “Software Documentation Issues Unveiled,” in *IEEE/ACM 41st International Conference on Software Engineering*, vol. 2019-May. IEEE Computer Society, 5 2019, pp. 1199–1210.
- [49] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, “Are code examples on an online Q&A forum reliable?: A study of API misuse on stack overflow,” in *ACM/IEEE International Conference on Software Engineering*, 2018, pp. 886–896.
- [50] D. Hoffman and P. Strooper, “API documentation with executable examples,” *Journal of Systems and Software*, vol. 66, no. 2, pp. 143–156, 5 2003.
- [51] S. Subramanian, L. Inozemtseva, and R. Holmes, “Live API documentation,” in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 5 2014, pp. 643–652.

## Bibliography

- [52] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, “Improving API documentation using API usage information,” in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 2009, pp. 119–126.
- [53] L. Wei Mar, Y.-C. Wu, and H. Christine Jiau, “Recommending Proper API Code Examples for Documentation Purpose,” in *18th Asia-Pacific Software Engineering Conference*, 2011, pp. –338.
- [54] C. McMillan, D. Poshyvanyk, and M. Grechanik, “Recommending Source Code Examples via API Call Usages and Documentation,” in *The 2nd International Workshop on Recommendation Systems for Software Engineering*, 2010, p. 83.
- [55] J. Jiang, J. Koskinen, A. Ruokonen, and T. Systä, “Constructing Usage Scenarios for API Redocumentation,” in *15th IEEE International Conference on Program Comperhension(ICPC’07)*, 2007.
- [56] M. Ward, “Program Comprehension,” Software Technology Research Lab, De Montfort University., Tech. Rep.
- [57] A. Fekete and Z. Porkoláb, “A comprehensive review on software comprehension models,” *Annales Mathematicae et Informaticae*, vol. 51, pp. 103–111, 2020.
- [58] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola, “Code Park: A New 3D Code Visualization Tool,” in *2017 IEEE Working Conference on Software Visualization*, vol. 2017-October. Institute of Electrical and Electronics Engineers Inc., 10 2017, pp. 43–53.
- [59] A. L. Mattila, P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen, and H. Väättäjä, “Software visualization today - Systematic literature review,” in *The 20th International Academic Mindtrek Conference*. Association for Computing Machinery, Inc, 10 2016, pp. 262–271.

- [60] L. Bedu, O. Tinh, and F. Petrillo, “A tertiary systematic literature review on software visualization,” in *7th IEEE Working Conference on Software Visualization*. Institute of Electrical and Electronics Engineers Inc., 9 2019, pp. 33–44.
- [61] R. Wettel and M. Lanza, “Visualizing Software Systems as Cities,” in *4th IEEE International workshop on visualizing software for understanding and analysis*, 2007, pp. 92–99.
- [62] W. Richard and L. Michele, “CodeCity 3D Visualization of Large-Scale Software,” in *Companion of the 30th international conference on Software engineering*. ACM Digital Library, 2013, pp. 921–922.
- [63] R. Wettel and M. Lanza, “Program Comprehension through Software Habitability,” in *15th IEEE International Conference on Program Comprehension (ICPC’07)*. IEEE, 2007, pp. 231–240.
- [64] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza, “The City Metaphor in Software Visualization: Feelings, Emotions, and Thinking,” *Multimedia Tools and Applications*, vol. 23, no. 78, 2019.
- [65] R. Cates, N. Yunik, and D. G. Feitelson, “Does Code Structure Affect Comprehension? On Using and Naming Intermediate Variables,” in *IEEE International Conference on Program Comprehension*, vol. 2021-May. IEEE Computer Society, 5 2021, pp. 118–126.
- [66] S. Ajami, Y. Woodbridge, and D. G. Feitelson, “Syntax, predicates, idioms — what really affects code complexity?” *Empirical Software Engineering*, vol. 24, no. 1, pp. 287–328, 2 2019.
- [67] E. Avidan and D. G. Feitelson, “Effects of Variable Names on Comprehension: An Empirical Study,” in *IEEE 25th International Conference on Program Comprehension*. IEEE Computer Society, 6 2017, pp. 55–65.

## Bibliography

- [68] B. Sharif and J. I. Maletic, “An eye tracking study on camelcase and under-score identifier styles,” in *IEEE 18th International Conference on Program Comprehension*, 2010, pp. 196–205.
- [69] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, “The impact of identifier style on effort and comprehension,” *Empirical Software Engineering*, vol. 18, no. 2, pp. 219–276, 4 2013.
- [70] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “Effective identifier names for comprehension and memory,” *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 12 2007.
- [71] T. Vieira Ribeiro and G. Horta Travassos, “Attributes Influencing the Reading and Comprehension of Source Code-Discussing Contradictory Evidence,” *CLEI Electronic Journal*, vol. 21, no. 1, 2018.
- [72] J. C. Hofmeister, J. Siegmund, and D. V. Holt, “Shorter identifier names take longer to comprehend,” *Empirical Software Engineering*, vol. 24, no. 1, pp. 417–443, 2 2019.
- [73] R. J. Miara, E. Ct, J. A. Musselman, J. A. Musse, J. A. Navarro, and B. Shneiderman, “PROGRAM INDENTATION AND COMPREHENSIBILITY,” *Communications of the ACM*, vol. 26, no. 11, 1983.
- [74] Y. Geffen and S. Maoz, “On method ordering,” in *IEEE International Conference on Program Comprehension*, vol. 2016-July. IEEE Computer Society, 7 2016, pp. 1–10.
- [75] A. Jbara and D. G. Feitelson, “On the effect of code regularity on comprehension,” in *22nd International Conference on Program Comprehension (ICPC 2014)*. Association for Computing Machinery, 6 2014, pp. 189–200.

- [76] N. Peitek, J. Siegmund, and S. Apel, “What drives the reading order of programmers? an eye tracking study,” in *28th International Conference on Program Comprehension (ICPC ’20)*. ACM, 10 2020, pp. 342–353.
- [77] B. El-Haik and A. Shaout, *SOFTWARE DESIGN FOR SIX SIGMA A Roadmap for Excellence*. Hoboken, New Jersey: A JOHN WILEY & SONS, INC., 2010.
- [78] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, “A SLOC Counting Standard,” in *The 22nd Annual Forum on COCOMO and Systems/Software Cost Modeling*. Center for Systems and Software Engineering, University of Southern California., 2007, pp. 1–16.
- [79] S. K. Chang, *Handbook of software engineering and knowledge engineering*. World Scientific, 2001, vol. 1.
- [80] D. Kafura and G. R. Reddy, “The Use of Software Complexity Metrics in Software Maintenance,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 13, no. 3, 1987.
- [81] J. Borstler and B. Paech, “The Role of Method Chains and Comments in Software Readability and Comprehension-An Experiment,” *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 886–898, 9 2016.
- [82] S. A. Chowdhury, G. Uddin, and R. Holmes, “An Empirical Study on Maintainable Method Size in Java,” in *2022 Mining Software Repositories Conference, MSR 2022*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 252–264.
- [83] N. Kukreja, “Measuring Software Maintainability,” Quandary Peak Research, Los Angeles, California, Tech. Rep., 2

## Bibliography

2015. [Online]. Available: <https://quandarypeak.com/2015/02/measuring-software-maintainability/>
- [84] P. Oman and J. Hagemester, “Construction and testing of polynomials predicting software maintainability,” *Journal of Systems and Software*, vol. 24, no. 3, pp. 251–266, 1994.
- [85] T. J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.
- [86] “Essential complexity,” 3 2021. [Online]. Available: <https://www.ibm.com/docs/en/raa/6.1?topic=metrics-essential-complexity>
- [87] M. H. Halstead, “Natural laws controlling algorithm structure?” *ACM Sigplan Notices*, vol. 7, no. 2, pp. 19–26, 1979.
- [88] Kurt Heckman, “Halstead Software Complexity,” 12 2020. [Online]. Available: <https://www.vcalc.com/wiki/halstead%20software%20complexity>
- [89] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, “Program Comprehension and Code Complexity Metrics: An fMRI Study,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 5 2021, pp. 524–536.
- [90] S. Scalabrino, G. Bavota, V. Christopher, L.-V. Mario, P. Denys, and O. Rocco, “Automatically Assessing Code Understandability:How Far Are We?” in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 417–427.
- [91] A. Campbell, “Cognitive Complexity: A New Way of Measuring Understandability,” SonarSource S.A., Tech. Rep., 2018.
- [92] D. Beyer, A. F. . I. t. I. Conference, and u. 2010, “A simple and effective measure for complex low-level dependencies,” in *18th IEEE In-*

- ternational Conference on Program Comprehension.* IEEE, 2010, pp. 80–83.
- [93] Martin Fowler, *Refactoring: Improving the Design of Existing Code*. Berkeley, CA, USA: Addison-Wesley Professional, 1999.
- [94] A. O’connor, M. Shonle, and W. Griswold, “Star Diagram with Automated Refactorings for Eclipse,” *eclipse’05*, San Diego, CA, Tech. Rep., 10 2005.
- [95] N. Lambaria and T. Cerny, “A Data Analysis Study of Code Smells within Java Repositories,” in *17th Conference on Computer Science and Intelligence Systems*, 2022, pp. 313–318.
- [96] “Eclipse Java development tools (JDT),” 2022. [Online]. Available: <https://www.eclipse.org/jdt/>
- [97] J. Al Dallal and A. Abdin, “Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review,” *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2018.
- [98] F. Tomassetti, “How to create pragmatic, lightweight languages Learn the process to create DSLs and GPLs,” Tech. Rep., 2016. [Online]. Available: [http://leanpub.com/create\\_languages](http://leanpub.com/create_languages)
- [99] M. Mernik, J. H. Cwi, A. M. Sloane, J. H. Nl, and . A. M. Sloane, “When and How to Develop Domain-Specific Languages When and How to Develop Domain-Specific Languages 317,” Tech. Rep. 4, 2005.
- [100] M. Fowler, *Domain-specific languages*. Addison-Wesley, 2011.
- [101] R. D. Kelker, *Clojure for domain-specific languages*. Packt Publishing, 2013.

## Bibliography

- [102] D. Kolovos, “The eclipse modeling framework (emf),” 2021.
- [103] R. F. Paige, D. S. Kolovos, and F. A. Polack, “A tutorial on metamodelling for grammar researchers,” in *Science of Computer Programming*, vol. 96, no. P4. Elsevier, 12 2014, pp. 396–416.
- [104] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes, “Engineering a DSL for software traceability,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5452, pp. 151–167, 2009.
- [105] K. Sledziewski, B. Bordbar, and R. Anane, “A DSL-based approach to software development and deployment on cloud,” in *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, 2010, pp. 414–421.
- [106] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, “What should developers be aware of? An empirical study on the directives of API documentation,” *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 12 2012.
- [107] E. Aghajani, C. Nagy, M. Linares-Vasquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, “Software documentation: The practitioners’ perspective,” in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 6 2020, pp. 590–601.
- [108] “Gorilla Experiment Builder,” 2023. [Online]. Available: <https://gorilla.sc>
- [109] A. Danilova, A. Naiakshina, S. Horstmann, and M. Smith, “Do you really code? Designing and evaluating screening questions for online surveys with programmers,” in *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 5 2021, pp. 537–548.



- [110] D. Russo, “Recruiting Software Engineers on Prolific,” in *the1st International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES 2022)*, vol. 5, no. CSCW1. Association for Computing Machinery, 4 2022.
- [111] B. Reid, M. Wagner, M. d’Amorim, and C. Treude, “Software Engineering User Study Recruitment on Prolific: An Experience Report,” in *International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES’22)*, vol. 5, no. CSCW1. Association for Computing Machinery, 4 2022.
- [112] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, “Measuring Programming Experience,” in *2012 20th IEEE International Conference on Program Comprehension*. IEEE, 2012, pp. 73–82.
- [113] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, “Measuring and modeling programming experience,” *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299–1334, 2014.
- [114] G. Gao, F. Vpichick, M. Ichinco, and C. Kelleher, “Exploring Programmers’ API Learning Processes: Collecting Web Resources as External Memory,” in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2020, pp. 1–10.
- [115] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, “Mental models and software maintenance,” *Journal of Systems and Software*, vol. 7, no. 4, pp. 341–355, 1987.
- [116] A. Von Mayrhauser and A. M. Vans, “Program Understanding Behavior During Adaptation of Large Scale Software,” in *the 6th International Workshop on Program Comprehension (IWPC’98)*, 1998, pp. 164–172.

## Bibliography

- [117] D. G. Feitelson, “Considerations and Pitfalls in Controlled Experiments on Code Comprehension,” in *IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 3 2021, pp. 106–117.
- [118] S. Shapiro and M. Wilk, “An analysis of variance test for normality (complete samples),” *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.
- [119] L. Howard, “Robust tests for equality of variances,” *Contributions to probability and statistics*, pp. 278–292, 1960.
- [120] “Vonage Quickstart Examples for Java,” 2023. [Online]. Available: <https://github.com/Vonage/vonage-java-code-snippets>
- [121] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 5 2009.
- [122] B. Hu, Y. Wu, X. Peng, J. Sun, N. Zhan, and J. Wu, “Assessing Code Clone Harmfulness: Indicators, Factors, and Counter Measures,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 225–236.
- [123] H. Husain, H.-H. Wu, T. Gazit, G. Miltiadis, and A. M. Brockschmidt, “CodeSearchNet Challenge Evaluating the State of Semantic Code Search,” *preprint arXiv:1909.09436*, 2019.
- [124] T. Zhang, D. Yang, C. Lopes, and M. Kim, “Analyzing and Supporting Adaptation of Online Code Examples,” in *Proceedings - International Conference on Software Engineering*, vol. 2019-May. IEEE Computer Society, 5 2019, pp. 316–327.
- [125] “SearchCode,” 2022. [Online]. Available: <https://searchcode.com>
- [126] “GitHub,” 2023. [Online]. Available: <https://www.github.com>

- [127] E. Kalliamvakou, G. Gousios, t. Kelly Blincoe, L. Singer, D. M. German, and D. Damian, “The Promises and Perils of Mining GitHub,” in *In Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.
- [128] “GitHub REST API,” 2023. [Online]. Available: <https://docs.github.com/en/rest?apiVersion=2022-11-28>
- [129] O. Dabic, E. Aghajani, and G. Bavota, “Sampling Projects in GitHub for MSR Studies,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021*. Institute of Electrical and Electronics Engineers Inc., 5 2021, pp. 560–564.
- [130] D. Eisenberg, J. Stylos, and B. Myers, “Apatite: A New Interface for Exploring APIs,” *CHI 2010: Interaction Techniques*, pp. 1331–1334, 2010.
- [131] L. Prechelt, G. Malpohl, and M. Philippsen, “Finding Plagiarisms among a Set of Programs with JPlag,” *Journal of Universal Computer Science*, vol. 8, no. 11, 2002.
- [132] T. Sağlam, S. Hahner, J. W. Wittler, and T. Kühn, “Token-based plagiarism detection for metamodels,” in *Proceedings - ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022: Companion Proceedings*. Association for Computing Machinery, Inc, 10 2022, pp. 138–141.
- [133] P. Terence, “ANTLR (ANother Tool for Language Recognition),” 2025. [Online]. Available: <https://www.antlr.org/>
- [134] D. Moody, “The physics of notations: Toward a scientific basis for constructing visual notations in software engineering,” *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.

## Bibliography

- [135] D. Avison and G. Fitzgerald, *Information Systems Development: Methodologies, Techniques and Tools*, 3rd ed. Blackwell Scientific, 2003.
- [136] J. Bertin, *Semiology of Graphics: Diagrams, Networks, Maps*. Univ. of Wisconsin Press, 1983.
- [137] J. H. Larkin and H. A. Simon, “Why a Diagram is (Sometimes) Worth Ten Thousand Words,” *Cognitive Science*, vol. 11, no. 1, pp. 65–100, 1987.
- [138] W. Lidwell, K. Holden, and J. Butler, *Universal Principles of Design: A Cross-Disciplinary Reference*. Rockport Publishers, 2003.
- [139] P. Goolkasian, “Pictures, Words, and Sounds: From Which Format Are We Best Able to Reason?” *The Journal of General Psychology*, vol. 127, no. 4, 2000.
- [140] A. Bacchelli, F. Rigotti, L. Hattori, and M. Lanza, “Manhattan-3D City Visualizations in Eclipse,” in *ECLIPSE IT*, 2011, pp. 307–310.
- [141] M. Weninger, L. Makor, and H. Mossenbock, “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor,” in *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*. Institute of Electrical and Electronics Engineers Inc., 9 2020, pp. 110–121.
- [142] T. Durieux, C. Soto-Valero, and B. Baudry, “Duets: A dataset of reproducible pairs of java library-clients,” in *Proceedings - 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021*. Institute of Electrical and Electronics Engineers Inc., 5 2021, pp. 545–549.

## Bibliography

- [143] S. Sparman and C. Schulte, “Analysing the API learning process through the use of eye tracking,” in *the 2023 Symposium on Eye Tracking Research and Applications (ETRA)*, 2023, pp. 1–6.