
Proceedings Paper:
Alharbi, Seham, Kolovos, Dimitris and Matragkas, Nicholas (2024) Towards Generating Maintainable and Comprehensible API Code Examples. In: Proceedings - 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024. 31st IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, 12-15 Mar 2024 Proceedings - 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024. Institute of Electrical and Electronics Engineers Inc., FIN, pp. 830-834.

https://doi.org/10.1109/SANER60148.2024.00090

Towards Generating Maintainable and Comprehensible API Code Examples

Seham Alharbi *
Department of Computer Science
University of York
York, United Kingdom
saaa528@york.ac.uk

Dimitris Kolovos
Department of Computer Science
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

Nicholas Matragkas
CEA-List
Université Paris-Saclay
Palaiseau, France
nikolaos.matragkas@cea.fr

***Abstract*—One of the most effective resources for learning application programming interfaces (APIs) is code examples. The shortage of such examples can pose a significant learning obstacle for API users. API users desire simple, understandable, self-contained examples that are easy to reuse in their applications. However, writing and maintaining code examples that meet the preferences of API users can be a tedious and repetitive activity for API developers. To address this issue, we present a new approach that aims to ease the writing and maintenance of code examples for API developers, while also improving learnability and comprehension for API users. The approach automatically synthesises linear and more comprehensible API code examples from less repetitive and more maintainable versions by inlining reusable utility methods. We implement this approach in a prototype for the Java programming language. We also evaluate its usefulness in terms of conciseness on a dataset of 600 API code examples extracted from nine open-source Java libraries. The results are encouraging and show that the proposed approach can reduce code repetition and bring a decrease of up to 37% in the lines of code of the evaluated API code examples.**

***Index Terms*—APIs, code examples, software maintainability**

I. Introduction

The limited availability of high-quality code examples in Application Programming Interface (API) documentation hinders API learnability, as users often rely on code examples for initial API understanding [1], [2]. Efforts to tackle this issue involved proposing systems for mining various resources, including online sites and test code, to extract API usage examples [3], [4]. However, only a few approaches have directly addressed the challenges faced by API developers and documentation writers in creating and maintaining code examples [5]. We therefore propose an approach that aims to tackle the problem of the shortage of code examples from a different perspective. It intends to assist API developers in creating and annotating knowledge bases of concise, self-contained and more maintainable API code examples. These examples serve as input for generating longer, more linear[1] and easier-to-follow versions of the same examples. Specifically, our goal is to reduce duplication in code examples, making it easier for API developers to write and maintain such examples. This, in turn, will ultimately increase the number of examples available to API users and enhance API learnability. Such an approach is needed since existing research shows that API developers have remarkably little tool support for effectively documenting their APIs [6].

Our main assumptions are that:

1) In addition to the known characteristics of effective code examples [7], API users may prefer linear code examples (e.g., examples in Listings 1 and 2) because linear code can eliminate the need for frequent jumps between method definitions, making it easier to read [8], comprehend[2] and adapt in the users' codebases.
2) On the flip side, writing such examples, particularly for large APIs, can be tedious and require substantial effort from API developers, as they may contain a significant amount of repetitive code that can pose challenges during their maintenance and evolution [9]. Therefore, developers may prefer writing less linear but more maintainable examples (e.g., Listing 4).

Thus, our approach provides a middle-ground solution for automatically synthesising linear API code examples from less repetitive and more maintainable versions by refactoring reusable methods.

To assess our synthesis approach, we evaluated 600 API code examples from nine popular Java projects. Our approach reduced repetition and example size by over 30% in a good portion of the evaluated API code examples.

The main novel contributions of this work can be summarised as follows:

- A synthesis prototype for the Java programming language that can alleviate API developers' burden of writing repetitive and costly-to-maintain API code examples, thus enabling them to produce more examples for API users, which can enhance API learnability.

*Seham Alharbi is also affiliated with the College of Computer, Qassim University, Buraydah, Saudi Arabia.

[1]Code that does not consist of multiple interdependent methods or classes.

[2]The linear structure of source code could minimise programmers' eye movements, which can enhance the comprehension process [8].

- A dataset[3] of real-world API code examples extracted from widely-used open-source Java projects, along with identified duplicate code found among them.
- An evaluation of the proposed synthesis approach that shows that the approach can bring a fair amount of reduction in code repetition.
- Some observations that suggest future extensions and evaluation.

II. Approach Overview

To illustrate our approach, let us consider two real-world code examples (Listings 1 and 2) from the open-source Vonage Voice API [10]. Listing 1 demonstrates sending dual-tone multi-frequency tones to an active call, while Listing 2 shows playing 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 1. Vonage API 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
8      final String UUID = call.getUuid();
9      final String TEXT = "Hello ... ";
10     client.getVoiceClient() .startTalk(UUID, TEXT, var);
```

Listing 2. Vonage API Example (2) - SendTalkToCall.java.

While the above code snippets illustrate two distinct API usages, they are very similar in structure and contain duplicated code. The only code statements that differ are those highlighted in the same colour in both snippets. Moving repetitive code to a reusable/utility method (e.g., `createCallEvent` in Listing 3) would make the two API code examples much more modular and maintainable. However, following and reusing the resulting non-linear code examples (e.g., Listing 4) might not be straightforward for API users because users may have to jump between method definitions and copy and paste multiple methods instead of a single, linear, self-contained example.

```
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            Thread.sleep(threadMillis);
8
9            final String UUID = call.getUuid();
10           final String STRING = string;
11
12           if (isTalk) {
13               client.getVoiceClient().startTalk(UUID, STRING,
                     TextToSpeechLanguage.AMERICAN_ENGLISH);
14           } else {
15               client.getVoiceClient().sendDtmf(UUID, STRING);
16           }
17 }
```

Listing 3. Utility Method.

[3]All implementation code, data, and R scripts are available in our replication package at: https://figshare.com/s/ac8128c17420fa9c5d2e

The objective of our 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 reusable utility methods and then automatically refactor and inline the calls to these utility methods to produce simple and linear API code examples.

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

Listing 4. Documentation Method - Vonage API - SendDtmfToCall.java.

The proposed linear code synthesiser works by automatically refactoring and inlining the calls to utility methods to produce simple, linear, correct and dead-code-free API code examples. It comes in the form of an Eclipse plug-in to facilitate its use. It takes as input a single non-linear API code example (i.e., a Java source code file)[4], which then passes through four main stages: (1) code analysis, (2) code transformation, (3) code processing and (4) code generation. The following sections explain each of these stages in more detail.

A. Code Analysis

The proposed synthesiser relies on static analysis of annotated source code and abstract syntax tree (AST) parsing. API developers must use two main Java annotations: `@Documentation` for methods illustrating API usage and containing non-linear code, and `@Utility` for methods encapsulating reusable code. A visitor then analyses, marks annotated methods, and locates calls to utility methods, along with their bindings found in the definitions of the marked methods for later inlining. The Java source code parsing and analysis are done using Eclipse JDT.

B. Code Transformation

A `MethodDeclarationTransformer` takes each documentation method (e.g., the main method in Listing 4) found in the under-processing Java source code file and automatically inlines all the calls to utility methods found in its definition (e.g., the method call at line 5 in Listing 4). This transformer works recursively, meaning that it programmatically performs the following steps: (1) it traverses the documentation method, (2) inlines the first encountered instance of a utility method call and all other nested calls within the invoked method, (3) it updates the AST node of the documentation method once the inlining is complete, (4) it updates the entire example source code in preparation for the next inlining of utility method invocations (if any exist), and (5) it repeats all previous steps for the next encountered utility method call, this time on the updated version of the example. It also manages potential errors that developers might make, such as annotating a method with both annotations or calling a documentation method within a utility method. In such cases, the synthesiser does not inline and generate code.

[4]Select a Java source code file → right-click → synthesise code.

The transformer utilises the refactoring capabilities of Eclipse JDT (i.e., `InlineMethodRefactoring`) to inline utility method calls. The rationale for choosing JDT is its ability to prevent syntax errors during code transformation, thus generating correct and executable code. In addition, the transformer complies with several JDT constraints, such as the prevention of inlining calls to recursive methods, multi-return methods and methods used as parameters for other methods. Another reason for utilising Eclipse JDT, and not using a template language, is because we wanted the more condensed examples (i.e., non-linear examples) used in our approach to be valid Java code so that API developers can benefit from error checking, code completion, and other similar features. However, each of these two approaches has its strengths and weaknesses.

C. Code Processor and Generator

The code processor refines the resulting linear code by detecting and removing redundant elements such as temporary variables and dead code that could be generated during the inlining process done by the code transformer. Once all documentation methods are inlined and refined, including the removal of unnecessary import statements and annotations, the entire example is sent to a linear code generator. This generator generates and stores the linear API code example in a separate package within the source folder of the active Java project.

III. Preliminary Evaluation

We aimed at answering the following research questions (RQs):

- **RQ1** How many API code examples contain duplicated code that can be eliminated using the proposed approach?
- **RQ2** How much reduction of duplicated code is achieved by the proposed approach?
- **RQ3** How often are the duplicated code fragments repeated across the API code examples?

To answer these RQs, we followed the five-step evaluation process explained below.

A. Data Collection

To evaluate our approach, we had to obtain a dataset of self-contained and compilable API code examples grouped by their associated Java libraries. These examples need to be explicitly provided or referenced by the API developers to illustrate specific API usage. This was crucial as our evaluation focused on understanding the current state of API code examples and measuring code repetition. Existing datasets like CodeSearchNet[5] and code from search engines like SearchCode[6] were unsuitable for our criteria since they were extracted from various open-source projects or online sites, such as Stack Overflow [11].

To build a suitable dataset, we collected API code examples from popular and open-source Java projects available on GitHub. We based this library selection on the following criteria: (1) the popularity (i.e., 50 starts on GitHub or used by 30 other projects) and activity level (i.e., recent commits and at least five contributors) of the library, (2) the availability of complete and compilable code examples in its repository and (3) its domain.

To extract GitHub projects, we utilised GitHub Search[7]. Examples were sourced from the official library website, the 'samples' or 'examples' folder in the library's GitHub repository, or external tutorials linked in the library's GitHub page description. Seven Java libraries from different domains were randomly selected, along with two standard Java APIs (JDBC and Java Applets) that are frequently used in existing literature [12]. This selection strikes a balance between the generalisability of our findings and the effort required for rigorous manual evaluation of the examples.

B. Data Cleaning

Through this step, we tried to reduce noise in each subset (Java library) of the dataset and focus the analysis on relevant API usage examples only. Thus, we removed 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. These included 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.

C. Similarity Detection

In this step, we individually examined subsets, searching for code examples with near-duplicate code suitable for our proposed approach. By utilising the JPlag code similarity detector[8], we conducted pairwise comparisons of source code files to identify similarities. JPlag is capable of detecting exact and modified code clones at different granularities and offers a web-based interface for result visualisation.

Subsequently, we manually inspected the identified clones, assessing their potential as input for our code synthesiser. This inspection focused on determining whether duplicate code could be moved into a separate utility method and substituting it with a method call for the subsequent automated refactoring by our synthesiser.

D. Examples Rewriting

We took copies of the API code examples that contained applicable similarities (i.e., similarities that can be eliminated using our synthesis approach) and manually rewrote them based on the structure of our proposed synthesis approach. This means that we factored out repetitive code in a set of utility methods, called these methods wherever needed and added the required Java annotations (i.e., `@Documentation` and `@Utility`). We maintained the same coding style used in the original examples.

[5]https://github.com/github/CodeSearchNet
[6]https://searchcode.com
[7]https://seart-ghs.si.usi.ch
[8]https://github.com/jplag/JPlag

TABLE I
STATISTICS OF THE SELECTED JAVA LIBRARIES AND PACKAGES.

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

*Lines of code (LOC).

E. Examples Evaluation

We evaluated the conciseness of the rewritten examples and measured the reduction in code repetition by comparing them to their original versions. This was done by computing the relative percentage decrease (Formula 1) in the non-comment, non-blank lines of code (LOC) in each API code example. LOC is calculated using *cloc*.[9]

$$\text{PercentageDecrease} = \frac{^{LOC}\text{Original} - ^{LOC}\text{Rewritten}}{^{LOC}\text{Original}} \times 100\% \quad (1)$$

IV. RESULTS AND DISCUSSION

As listed in Table II, four out of the nine scanned Java projects contained over 40% API code examples with near-duplicate code that could be factored out and reduced using our approach. The duplicate code manifested several patterns, such as similar interface implementation and type instantiation. The mean percentage of such examples (across all evaluated Java libraries) is 36.33% and the median percentage is 36%.

These results show that the percentage of examples containing repetitive code varied among the evaluated Java libraries, thus making it not possible to conclude which library domain is likely to benefit more from our proposed approach. This is true since the repetitiveness in code examples is highly impacted by the coding style API developers prefer when writing 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.

It is also worth mentioning that four of the selected Java libraries already contained a set of utility methods for some repetitive functionalities, which could indicate API developers' desire to reduce repetitiveness in API code examples.

As shown in Table II and illustrated in Figure 1, the median of 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. Precisely, more than 50% of the evaluated API code examples in three of these Java libraries received more than a 15% decrease in LOC, whereas in JAXB only 33% of the examples obtained such a decrease.

[9]https://github.com/AlDanial/cloc

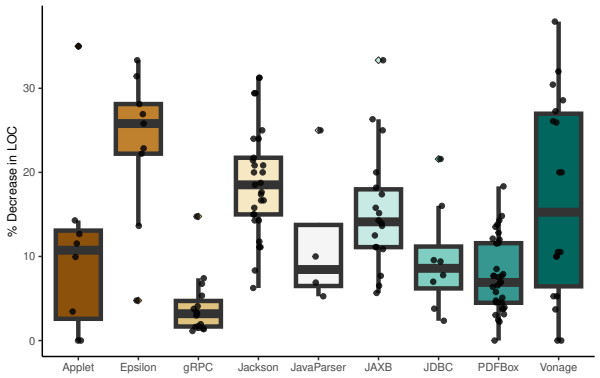

Fig. 1. Distributions of the Evaluated API Code Examples and the Percentage Decrease in their Lines of Code (LOC). Each Boxplot Aggregates the API Examples of each Java Library.

TABLE II
SUMMARY OF THE EVALUATION RESULTS.

Library	% Applicable	% Decrease (in LOC)	# of Utility Methods	# of Utility 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

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.

The significance of this evaluation lies not only in reducing duplicate code within a single example but also in assessing the frequency of duplicate code across the entire subset (column four in Table II). For instance, in the JDBC subset, while eliminating some duplicate code fragments may not significantly reduce individual example sizes, these fragments are recurrent throughout the entire subset. Therefore, for RQ3, we examine how often a repetitive code fragment appears across other API examples within the same Java library. This

analysis could indicate the API developers' efforts in writing repetitive code examples.

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

V. Limitations and Observations for Further Extensions

Template-based code synthesis: As discussed in Section II, for a practical reason (i.e., Java syntax), the current version of our prototype could not accommodate all the detected patterns of code similarity. Therefore, an extension that allows a template-based linear example code synthesis is required to address this limitation.

Interactive example generation: An extension involves allowing API users to specify values interactively for the API code example generator using a new annotation (`@DocGen`). This prompts users to input values as command-line arguments, enhancing the search for relevant examples, and thus promoting API learnability.

VI. Threats to Validity

To enhance **construct validity**, we manually reviewed JPlag's detection results to identify additional instances of undetected similarity. Also, we used an automated tool for calculating LOC in API code examples to minimize subjective bias and human error.

To mitigate **internal validity** threats, we maintained consistent coding and formatting styles across API code example versions (original and rewritten). This is crucial as the program size metric used is highly sensitive to code formatting.

To reduce **external validity** threats, we selected code examples based on specific criteria, spanning diverse domains in Java. However, our approach would benefit from further evaluation with a larger and more diverse dataset.

VII. Related Work

To our knowledge, prior studies have not empirically explored the prevalence of duplicated code in code examples. Nevertheless, van Bladel and Demeyer [13] showed that test code, which is somewhat similar to code examples, contains more than double the redundancy found in production code. Such duplication has been proven to negatively impact program maintainability and comprehensibility [14].

Various methods have been proposed to compensate for the shortage of code examples, including extracting code from online sources [3], using publicly available unit tests [4], [5] and improving code example format and maintainability [15]. Unlike these approaches, our method enables API developers to write structured, maintainable code examples. Its input is manually crafted code with specific annotations, rather than code that was not originally written to document APIs. This makes it particularly beneficial for newly released APIs with no existing client code to mine.

VIII. Conclusion and Future Work

We proposed an approach to address the issue of the repetitiveness of API code examples. This synthesiser 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. Our evaluation with real-world API examples showed substantial reductions in code repetition and example size.

For future work, we plan to validate our assumptions through user studies, confirming that (1) linear code enhances comprehensibility and reusability for API users, and (2) API developers would prefer to use our proposed approach to minimise duplicated code. Additionally, we aim to implement the extensions discussed in Section V and make our approach applicable to other strongly typed programming languages.

References

[1] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[2] 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.

[3] 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.

[4] 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.

[5] S. M. Nasehi and F. Maurer, "Unit tests as API usage examples," in *26th IEEE International Conference on Sofware Maintenance in Timisoara, Romania*, 2010.

[6] K. Nybom, A. Ashraf, and I. Porres, "A systematic mapping study on API documentation generation approaches," in *44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA*, 2018, pp. 462–469.

[7] 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.

[8] 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.

[9] 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.

[10] "Vonage Quickstart Examples for Java," 2023. [Online]. Available: https://github.com/Vonage/vonage-java-code-snippets

[11] 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.

[12] 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.

[13] B. van Bladel and S. Demeyer, "A comparative study of test code clones and production code clones," *Journal of Systems and Software*, vol. 176, 6 2021.

[14] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 56–65.

[15] M. Nassif, Z. Horlacher, and M. P. Robillard, "Casdoc: Unobtrusive Explanations in Code Examples," in *30th International Conferenceon Program Comprehension (ICPC '22)*. Virtual Event, USA. ACM, 2022.