Synthesising Linear API Usage Examples for API Documentation

Seham Alharbi*‡, Dimitris Kolovos*, Nicholas Matragkas†

*Department of Computer Science, University of York, York, United Kingdom
†CEA-List, Université Paris-Saclay, Palaiseau, France
‡College of Computer, Qassim University, Buraydah, Saudi Arabia
*{saaa528, dimitris.kolovos}@york.ac.uk
†nikolaos.matragkas@cea.fr

***Abstract*—Code examples are essential resources for learning application programming interfaces (APIs). The shortage of such examples can be a major learning obstacle for API users. Writing and maintaining effective API usage examples can also be an effort-intensive and repetitive process for API developers because API users ideally want such examples to be simple, standalone, and linear. To address these challenges, several approaches have been proposed to automatically extract API code examples from various resources and embed them into official API documents; however, little emphasis has been placed on addressing the underlying issue directly and helping API developers write and maintain API usage examples. In this paper, we present a new approach for automatically synthesising linear code examples from less repetitive versions by in-lining reusable utility methods. The proposed approach aims to benefit API developers in terms of productivity and maintainability, as well as API users in terms of API learnability and comprehension. We have implemented the proposed approach in a prototype for the Java programming language, which we also discuss in this paper.**

***Index Terms*—API, documentation generation, API learnability, code comprehension, linear code, usage examples**

I. Introduction

Reuse-based software development is a major approach in the field of software engineering because it helps developers effectively and accurately build the desired functionalities, facilitate programming tasks, and save time [1], [2]. Software reuse can take many forms, including the use of external software libraries or frameworks through their application programming interfaces (APIs) [3], [4]. However, the effectiveness of using an API is significantly reliant on the availability of sufficient resources for learning it; therefore, APIs come with documentation, such as API reference documentation, user guides, and tutorials [5]. API documentation is a product in itself and thus requires effort to develop and maintain, and its quality cannot be taken for granted [6].

Several studies have shown that one of the main obstacles when learning an API is related to the availability of code examples [4], [6], [7]. Moreover, 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 when learning a new API [5]. However, for API code examples to be effective, API users usually expect them to be simple, standalone, and (when possible) linear. This is because non-linear code examples (i.e. examples consisting of multiple interdependent functions/classes) tend to be more complex, and require more effort to adapt by the user in their own codebase. Furthermore, the creation of API documentation is typically an effort-intensive process [8], and writing efficient and linear API usage examples is a particularly non-trivial task.

To compensate for the shortage of API code examples in API documentation, many researchers have proposed approaches to automatically extract code examples from several resources and embed them into official API documents [9], [10]; however, little attention has been paid to directly addressing the underlying challenge of writing API code examples and helping API developers produce more usage examples for their APIs. The main question that underpins this research is ‘What kind of tool support could help API developers write and maintain knowledge bases of simple, standalone, and linear API code examples from which users can learn effectively?’.

In response to this question, this paper presents an approach that contributes to tackling the challenge of writing and maintaining usage examples for Java APIs. With this approach, linear code examples are automatically synthesised from less repetitive variants by in-lining reusable utility methods. Our assumption is that API users would ideally want the API usage examples to be linear because linear code tends to be less complex and therefore easier to follow and comprehend (see Section III). On the flip side, such examples can be extremely repetitive and tedious to manually write and maintain for API developers, particularly when documenting large APIs.

Our main envisioned contribution is a synthesis prototype that can alleviate the API developers’ burden of writing repetitive and lengthy API code examples for API documentation, thus empowering them to produce more API code examples that can promote API learnability.

The work presented herein aims to answer the following **research questions (RQs)**:

RQ1. What mechanisms and techniques can be used to enable the automated synthesis of simple, standalone, and linear API usage examples (preferred by API users) from less repetitive and more maintainable examples (preferred by API developers)?

RQ2. How much more effective (in terms of API learnability and comprehension) are the generated examples compared to the less repetitive and more maintainable examples from which they were synthesised?

II. Motivating Example

To motivate our proposed approach, we consider the Java code snippets shown in Listings 1 and 2. The code snippet in Listing 1 demonstrates how different layout managers[1] (i.e. *BorderLayout* and *BoxLayout*) operate in the Java Swing API[2]. Although the two methods in this code snippet (lines 1 and 10) show the usage of two different layout managers, they contain some duplicated code (i.e. lines 2, 4, and 6 in *docBorderLayout()* and lines 11, 13, and 17 in *docBoxLayout()*). The manual writing and maintenance of such repetitive examples can be tedious for API developers, particularly for large APIs. Arguably, it would be more efficient if this repetitive code could be moved to a set of reusable methods and invoked whenever needed (methods shown at lines 1 and 9 in Listing 2).

Invoking a set of reusable/utility methods will make the code be more maintainable for API developers; however, API users, particularly newcomers to the Java Swing API, will have to jump between the definitions of different utility methods (Listing 2) to fully comprehend how to create different components (e.g. buttons) and place them within a container, as well as how to create and display a GUI.

```
1  public void docBorderLayout() {
2    JPanel panel = new JPanel();
3    panel.setLayout(new BorderLayout());
4    JButton button = new JButton("Button");
5    panel.add(button, BorderLayout.NORTH);
6    JFrame frame = new JFrame("BorderLayoutDemo");
7    // ... adjust frame size and show GUI
8  }
9
10 public void docBoxLayout() {
11   JPanel panel = new JPanel();
12   panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS
       ));
13   JButton button = new JButton("Button");
14   button.setForeground(this.foregroundColor);
15   button.setAlignmentX(Component.CENTER_ALIGNMENT);
16   panel.add(button);
17   JFrame frame = new JFrame("BoxLayoutDemo");
18   // ... adjust frame size and show GUI
19 }
```

Listing 1. Java Swing Layout Managers Usage Examples.

Our assumption is that the fewer method calls (fewer jumps) the code contains, the less complex and easier to follow and comprehend it will be for API users. This is because the code complexity increases with the number of independent paths (e.g. method calls), which can impact the users' comprehension of code (see Section III).

```
1  public JButton createButton(String name, boolean
       withColor) {
2    JButton button = new JButton(name);
3    if (withColor) {
4      button.setForeground(this.foregroundColor);
5    }
6    return button;
7  }
8
9  public void showGUI(String frameTitle, JPanel panel) {
10   JFrame frame = new JFrame(frameTitle);
11   Toolkit tk = Toolkit.getDefaultToolkit();
12   // ... position and set frame size
13   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14   frame.setVisible(true);
15 }
```

Listing 2. Utility Methods.

[1]https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

[2]https://docs.oracle.com/javase/7/docs/api/index.html

The aspiration of this work is to enable API developers to reduce repetition in API usage examples by encapsulating common behaviour in reusable utility functions, while keeping examples simple and linear for API users by automatically inlining such utility functions.

III. Background

To predict various aspects such as software quality, defects, and understandability, several code complexity metrics have been proposed in the field of software engineering. One of the well-known metrics for calculating code complexity is Cyclomatic Complexity [11] which is a quantitative measure of linearly independent paths in the source code. Fewer and less complex paths result in a lower cyclomatic complexity. The cyclomatic complexity is calculated using a control flow graph and the following formula, where E is the total number of edges and N is the total number of nodes.

$$V(G) = E - N + 2 \tag{1}$$

Several studies have analysed the correlations between existing code complexity metrics and code understandability [12], [13]; therefore, new control-flow complexity metrics, such as the Cognitive Complexity have been introduced to provide fairer relative evaluations of code comprehension and overcome the shortcomings of cyclomatic complexity [14].

IV. Related Work

A recent systematic mapping study on API documentation generation approaches [15] has found that the most popular approaches are those that generate API usage examples or templates for using APIs; therefore only those existing approaches that can fall under this category are discussed below.

In a similar line to our work, Buse and Weimer [16] 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 documents, other approaches benefit from publicly available API unit tests [17]–[19]. Using unit tests as API usage examples has been proven to be efficient for supplementing formal API documentation, particularly for newly released APIs that lack the availability of client code.

Some success has been made in filling in the gap between reference and example-based API documentation. This was achieved by using various API code examples found on some online sites, such as Stack Overflow and GitHub Gists, and embedding them into formal API documents [20], [21]. Such 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 documents.

Other approaches have focused on helping API users construct their code. For example, Jadeite [22] allowed API users to share their aggregate experience and collaboratively add placeholders to API documents to indicate the expected classes and methods. In addition, Docio [23] is a system that helps API users understand the actual/dynamic input and output values of API functions. Furthermore, the recommendation of source code examples by submitting queries against API calls was also proposed to help both API developers [24] and users [25], [26].

Unlike these existing approaches, our approach does not mine usage examples from code that was not originally written to document APIs (i.e. unit tests and online sites), but provides a methodology for API developers to write code examples in a structured way; therefore, the input of our approach is handcrafted code that has certain annotations.

V. Approach

Our proposed approach comes in the form of an Eclipse plugin to facilitate its use. It takes as input all the non-linear API code examples that the API developers write to document certain API usages. As shown in Figure 1, each code example (i.e. single Java source file) passes through three main stages, i.e. (1) analysis, (2) transformation, and (3) generation stages. To illustrate how each stage works, we use the motivating example described in Section II and the approach overview shown in Figure 1.

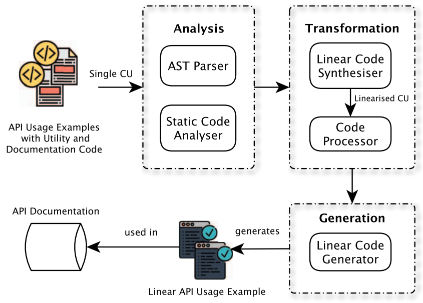

Fig. 1. Approach Overview.

A. Source Code Analysis

To use our prototype, the developers of API libraries should add two Java annotations, i.e. *@Documentation* to the API documentation methods (line 2 in Listing 3), and *@Utility* to reusable methods (Listing 2). These annotations help our analyser to differentiate between the pieces of code (methods) that are written to document a certain API usage, and pieces of code that are reusable across multiple API usage examples. As a first step, each annotated API usage example (i.e. a single Java source file) is parsed to generate its type-resolved abstract syntax tree (AST), which is then traversed and analysed by a visitor to locate all utility method invocations and their bindings found in the bodies of both documentation and utility methods.

```
1 @Documentation
2 public void docBorderLayout() {
3   JPanel panel = new JPanel();
4   panel.setLayout(new BorderLayout());
5   JButton button =  createButton("Button", false);
6   panel.add(button, BorderLayout.NORTH);
7   showGUI("BorderLayoutDemo", panel);
8 }
```

Listing 3. A Non-linear Documentation Method.

To parse and analyse Java source code, we used Eclipse Java Development Tools (JDT)[3], which is an Eclipse project that provides a set of plugins that support Java code parsing, analysis, validation, and manipulation (e.g. refactoring). Moreover, these JDT plugins contain a set of APIs to facilitate their extension and reuse [27].

B. Linear Code Synthesis

The prototype automatically in-lines all utility invocations found in documentation methods to synthesise and generate linear code examples. However, successfully in-lining the utility method invocations requires ensuring that the resulting synthesised code is syntactically correct. For example, both the documentation and utility methods must not contain any variables with the same name that will clash with each other after the in-lining process is complete; therefore, to ensure the correctness of the synthesised code examples, our prototype reuses the refactoring capabilities of Eclipse JDT to in-line utility method invocations. This refactoring operates by replacing a reference (i.e. a method call) with the method's implementation, and also ensures that no syntax errors will result (e.g. renames a variable if another variable with the same name is present).

After parsing and analysing each code example, a code synthesiser (transformation stage in Figure 1) takes as input all the documentation methods found in the example source file and processes them individually to locate all utility invocations within their bodies. After processing the selections of each method invocation (i.e. its position as an AST node), and prior to completing the in-lining process, the prototype also processes the body of the invoked utility method to ensure that it is linear and that all the other utility method invocations within its body are in-lined first. It also ensures and warns the user if the utility method declaration contains circular calls to utility methods, as this is a restriction placed by the JDT's inline feature.

C. Detection and Elimination of Dead Code

Dead code is source code that is never executed [28]. Furthermore, dead code elimination is usually applied for the purpose of optimisation (e.g. reducing the program size) or refactoring [29]. Since in-lining method invocations can result

[3]https://projects.eclipse.org/projects/eclipse.jdt

in some pieces of unreachable code, it is therefore crucial for our approach to consider the removal of dead code. For example, in-lining the method call at line 5 in Listing 3 will result in the body of the if-statement in the *createButton* method (lines 3-5 in Listing 2) being dead because the passed boolean value is 'false' (see lines 2-4 in Listing 4).

```
1 JButton button = new JButton("Button");
2   if (false) {
3     button.setForeground(this.foregroundColor);
4   }
```

Listing 4. Dead Code.

Because dead code detection and elimination is a common compiler optimisation process, our prototype reuses the capabilities of JDT in detecting and eliminating unresolved compilation errors and warnings. Moreover, JDT Core[4] defines a rich description of all standard Java problems as detected by the compiler.

Once all documentation methods are in-lined, the entire compilation unit is passed to a code processor that detects and removes dead code (transformation stage in Figure 1). Next, the compilation unit is sent to a code generator (generation stage in Figure 1) that generates linear and dead-code-free API examples and stores them in a separate package under the source folder of the same project. All the generated Java source files are copies of the original files, which remain unchanged to allow any future modifications.

VI. Plan for Evaluation

It is evident that source code with fewer lines tends to be more maintainable; therefore it is prudent for software developers to structure their code into reasonably sized files; however, to evaluate the usefulness of our proposed synthesis approach in helping API developers with writing less-repetitive code examples, we started to collect API documentation code examples from open-source projects, factor out duplicated code in a set of utility methods, and assess the benefits of our approach in terms of code conciseness. We plan to collect more examples to gain more insights into the strengths and limitations of our proposed approach and better understand the mechanisms that can be used to extend our approach. The collected examples will firstly be rewritten using the proposed approach, and then a set of software maintainability, complexity, and conciseness metrics will be used to directly compare the original and rewritten code examples. By doing so, we aim to answer **RQ1**.

To validate our core assumption (i.e. linear code is easier for API users to comprehend and follow), we will conduct a controlled user-based experiment. Through this code comprehension experiment, we intend to assess the impact of both linear and non-linear code on the performance of API users. Also, the code examples will be of different sizes to assess whether the length of a linear example has an impact on its understandability. In addition, to ensure the representativeness and diversity of the experimental subjects, we will strive to target both professional Java developers and university students. The participants will be divided into two groups, each of which will be working with a different representation of the API examples (i.e. linear and non-linear). All participants will be requested to perform the same set of code comprehension tasks. Through these tasks, we will carry out a quantitative comparison between the two groups of participants. This comparison will be based on three main variables: a) the accuracy of answers, b) the response time of accurate answers, and c) the response time of inaccurate answers. By conducting this user-oriented experiment, we aim to answer **RQ2**.

VII. Current Status and Future Work

Thus far, an initial version of the code synthesis prototype has been implemented, and a set of API code examples from an open-source project[5] was extracted to validate the usefulness of our proposed approach. The current results are encouraging; however, we plan on extracting more usage examples from other popular open-source projects and conducting the user-based experiment discussed in Section VI to ensure that our prototype can benefit both API developers and users in terms of programming productivity and API learnability. Moreover, we will investigate the ability of our prototype to go beyond what JDT's inline feature allows (i.e. in-lining is only possible on simple methods with a single return statement, or methods used in assignments). In addition, we intend to implement two extensions, i.e. a) an interactive linear example generator that allows a user's interaction and specification of some properties of the generated API examples, and b) a feature that allows the API developers to know what APIs need to be demonstrated in code examples by reporting documentation coverage metrics (similar to test coverage). We would also like to investigate whether the proposed approach could be applicable to other strongly-typed languages.

VIII. Conclusion

The shortage of API code examples in API documentation has been shown to be a significant API learning obstacle. On the other hand, writing and maintaining such API usage examples can be an expensive process for API developers. In this paper, we presented a PhD project aimed at addressing this problem by proposing a linear code synthesis approach that facilitates the process of linearising and generating code examples for API documentation. Our assumption is that, on the one hand, many simple and standalone API code examples are preferable from the users' perspective, while on the other hand, such examples are repetitive and tedious for developers to write and maintain. We also argued that linear code examples can increase the productivity of API users, and we plan to empirically validate this by carrying out a user-based code comprehension experiment (Section VI). The results obtained thus far are promising, and further research will be conducted to extend the features of the proposed prototype (Section VII).

[4]https://www.eclipse.org/jdt/core

[5]https://www.eclipse.org/epsilon/

References

[1] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, vol. 145, pp. 164–179, 2018.

[2] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta, "Recommending api function calls and code snippets to support software development," *IEEE Transactions on Software Engineering*, 2021.

[3] 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 (FormaliSE)*. IEEE, 2021, pp. 86–96.

[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. 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.

[6] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE software*, vol. 32, no. 4, pp. 68–75, 2015.

[7] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[8] 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, 2020.

[9] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, "Adding examples into java documents," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 540–544.

[10] L. Wang, L. Fang, L. Wang, G. Li, B. Xie, and F. Yang, "Api example: An effective web search based usage example recommendation system for java apis," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 592–595.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308–320, 1976.

[12] 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.

[13] 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.

[14] A. Campbell, "Cognitive complexity: A new way of measuring understandability," SonarSource S.A., Tech. Rep., 2018.

[15] K. Nybom, A. Ashraf, and I. Porres, "A systematic mapping study on API documentation generation approaches," in *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 462–469.

[16] R. P. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings - International Conference on Software Engineering*, 2012, pp. 782–792.

[17] S. M. Nasehi and F. Maurer, "Unit tests as api usage examples," in *26th IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[18] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining API usage examples from test code," in *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*. Institute of Electrical and Electronics Engineers Inc., dec 2014, pp. 301–310.

[19] D. Hoffman and P. Strooper, "API documentation with executable examples," *Journal of Systems and Software*, vol. 66, no. 2, pp. 143–156, 2003.

[20] 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, 2013.

[21] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th international conference on software engineering*. IEEE Computer Society, 2014, pp. 643–652.

[22] 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.

[23] 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, 2017, pp. 364–367.

[24] 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. [Online]. Available: http://www.koders.com/

[25] 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.

[26] 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.

[27] "Eclipse java development tools (jdt)." [Online]. Available: https://www.eclipse.org/jdt/

[28] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[29] S. Romano and G. Scanniello, "Exploring the use of rapid type analysis for detecting the dead method smell in java code," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 167–174.