How Can I Use This Method?

Laura Moreno*, Gabriele Bavota†, Massimiliano Di Penta‡, Rocco Oliveto§ and Andrian Marcus*
*The University of Texas at Dallas, USA; †Free University of Bozen-Bolzano, Italy;
‡University of Sannio, Italy; §University of Molise, Italy

***Abstract*—Code examples are small source code fragments whose purpose is to illustrate how a programming language construct, an API, or a specific function/method works. Since code examples are not always available in the software documentation, researchers have proposed techniques to automatically extract them from existing software or to mine them from developer discussions. In this paper we propose** MUSE **(Method USage Examples), an approach for mining and ranking actual code examples that show how to use a specific method.** MUSE **combines static slicing (to simplify examples) with clone detection (to group similar examples), and uses heuristics to select and rank the best examples in terms of reusability, understandability, and popularity.** MUSE **has been empirically evaluated using examples mined from six libraries, by performing three studies involving a total of 140 developers to: (i) evaluate the selection and ranking heuristics, (ii) provide their perception on the usefulness of the selected examples, and (iii) perform specific programming tasks using the** MUSE **examples. The results indicate that** MUSE **selects and ranks examples close to how humans do, most of the code examples (82%) are perceived as useful, and they actually help when performing programming tasks.**

I. Introduction

Developers frequently need to use methods they are not familiar with or they do not remember how to use. To learn about such methods, developers usually resort to reference manuals (*e.g.,* Javadoc), Questions and Answers (Q&A) forums (*e.g.,* Stack Overflow), or other information sources. Often, these resources provide little more than generic explanations of the method usage syntax, or focus on the method's technical details. In such cases, developers could benefit from short code fragments, *i.e., code examples*, presenting actual, practical uses of the method. The goal of our work is to automatically generate such code examples, given a specific method.

Our work adds to and complements existing approaches that have been developed to obtain code examples of one sort or another. Some of them focus on matching a query onto an existing code base with the aim of identifying generic code examples [1], [2], [3], [4]. Other approaches try to match user queries or developer's code context onto discussions (containing examples) in Q&A forums [5]. Recently, a few approaches have focused on providing *abstract code examples* for a given API, either as-a-whole [6] or focusing on specific methods [7], [8], [9]. *Abstract code examples* are fragments of code showing usage patterns of a code element (*e.g.,* an API or one of its methods). Even when there are several ways of using a code element, an abstract example presents a synthesized and generic use of a code element. Conversely, *concrete code examples* are working fragments of code showing actual, existing usage scenarios of a code element.

Our conjecture is that providing developers with several concrete method usages would augment abstract code examples and result in better understanding of the method usage. For this reason, we focus on the still open problem of mining relevant concrete code examples for a given method. Specifically, we aim at answering the following question: "*Given a specific method needed to perform a task, what are the necessary steps to use it?*" For instance, once a developer has understood the purpose of an API and has gained an idea of what the various methods do (*e.g.,* through a reference manual), she wants to know what are the typical invocation scenarios for a given method, say `copyInputStreamToFile`. To this aim, she needs to find one or more examples that have the necessary steps to invoke this method, such as, invoking other methods of the API or manipulating the method's parameters. Such a method usage example (see Fig. 1) shows that in order to use the desired method (line 12) two arguments are required (*e.g.,* `zip.getInputStream(entry)` and `file`). The inline comments (lines 8-11) provide information about each argument: the former is an `InputStream` instance whose content will be copied, and the latter is a non-directory `File` instance where the content will be written to. The comments also inform that neither of them should be null. In addition, the usage example shows how all elements involved in the invocation are obtained (lines 1-7). For example, the `InputStream` argument is obtained from a `ZipFile` object instantiated in line 3 and given a `ZipEntry` object instantiated in line 6.

We propose MUSE (**M**ethod **US**age **E**xamples), an approach that automatically finds, extracts, and documents concrete usage examples of a given method (like the one in Fig. 1). MUSE employs state-of-the-art static analysis techniques, which ensure its precision, usability, and set it apart from existing work. It parses existing applications to collect method usages and computes a static backward slice for each usage statement to detect the sequence of relevant steps to invoke the method and prune out code that is less relevant. Since different applications might use the same method in a similar way, MUSE identifies similar examples through clone detection. The detected clones are used as a measure of

```
01 File source;
02 File target;
03 ZipFile zip=new ZipFile(source);
04 Enumeration<? extends ZipEntry> entries=zip.entries();
05 while(entries.hasMoreElements()) {
06  ZipEntry entry=entries.nextElement();
07  File file=new File(target,entry.getName());
08  //zip.getInputStream(entry)->the InputStream to copy bytes from,
09  //must not be null
10  //file->the non-directory File to write bytes to (possibly
11  //overwriting), must not be null
12  FileUtils.copyInputStreamToFile(zip.getInputStream(entry),file);
13 }
```

Fig. 1. Example generated by MUSE.

popularity of the code examples, which is in turn used to rank the groups of similar examples (*i.e.,* code clones). Finally, MUSE selects one representative example for each group based on its *readability* and estimated *reusability effort* (*i.e.,* how much code from the client applications needs to be imported to reuse the example).

We implemented MUSE and evaluated it through three empirical studies. The first study was aimed at assessing the selection and ranking heuristics used by MUSE and involved nine industrial developers. The results indicate that MUSE selects and ranks examples much like human developers do. In the second study, ten code examples were evaluated by 119 surveyed developers, out of which 15 were contributors of the projects for which the examples were produced and the remaining ones worked on projects relying on those libraries. To the best of our knowledge, this is the largest study evaluating the perceived usefulness of automatically generated code examples (1,190 human judgments). The results indicate that 82% of the evaluated code examples are perceived as useful by the study participants.

We also performed an extrinsic evaluation aimed at investigating whether MUSE supports developers in their implementation tasks. This is another notable contribution of this paper, as we involved twelve industrial developers to assess the usefulness of automatically generated code examples during software development. We considered a control group relying on the Web to retrieve documentation and/or examples of the methods to be used. The experiment revealed that the developers using MUSE's examples achieved significantly more complete implementations.

All the material used to run our three studies as well as the raw results are publicly available in a comprehensive replication package [10] (all URLs last verified on 02/09/2015).

II. MUSE Overview

When a user needs method usage examples from a project release of interest (P_i), she provides P_i's source code and Javadoc (if available), and a list of client projects using P_i as library. With no other input required, MUSE proceeds as follows:

- The *Clients Downloader* downloads the source code of the client projects.
- The *Example Extractor* parses P_i's source code to extract its public methods and analyzes each of the client projects looking for calls to such methods. From each of these calls, MUSE computes a backward slice, each one representing a "raw" usage example.
- The *Example Evaluator* ranks the "raw" usage examples and selects the most representative ones.
- The *Example Injector* adds informative comments to each example. Also, for each method covered by at least one code example, the *Example Injector* creates an HTML page containing all the selected examples, as ranked by the *Example Evaluator*, and injects such a page in the official HTML Javadoc documentation.

A. Clients Download

For each client project of P_i to consider, the user provides a link to a compressed file containing its source code. There is no minimum number of clients required by MUSE. Clearly, the more clients are given, the higher the likelihood of identifying method usage examples. The generation of the list of possible clients can be automated. In our studies, for example, we used a script to automatically download the client projects using the library of interest (*i.e.,* P_i) from its Maven page. A project P_i indexed in Maven includes in its Web page a list of client projects using it (*e.g.,* see the Maven page of `Apache commons-io 2.4` at http://tinyurl.com/lmgd9m3). We did not integrate such a script in MUSE, since we are working on a more general Web crawler able to collect client projects without the Maven infrastructure.

Starting from the compressed file, the *Clients Downloader* automatically retrieves the source code of each client and builds it by using the Eclipse Java Development Tools (JDT), in conjunction with the Maven Eclipse Plugin, when appropriate (*i.e.,* when the code of a client includes a `pom.xml` file). Note that building the client projects is required, since the static slicer used by MUSE during the examples extraction can only be used on compilable code. The downloaded clients that cannot be successfully built are discarded. Thus, the output of this component is the set of built client projects.

B. Example Extraction, Selection, and Ranking

The *Example Extractor* parses the source code of the set of built client projects and searches for invocations of P_i's public methods. Then, the *Example Extractor* uses the JDeodorant Java static slicer [11] to compute an *intra-procedural, backward slice* for each identified method invocation.

At this point, every method in P_i will have a number of slices, each one representing a "raw" code example. If none of the clients uses a given method, then no examples will be extracted for it. There are cases, however, when the *Example Extractor* finds many examples for a given method. On one hand, having a large number of usage examples for a method is desirable, as different examples might show alternative usages of the method and some of them might be easier to adapt than others to the developer's task. On the other hand, providing developers with a large set of code examples, which might contain several similar elements, would result in information overload. MUSE tries to avoid such a situation by identifying groups of similar examples and reporting just one *representative* example for each group. Also, MUSE provides a *ranking* of the selected examples, indicating to the developer those that are more likely to be useful. We describe in more details how the *ranking* and *selection* are performed.

1) Ranking: The *Example Evaluator* relies on type-2 clone detection. Given a set of slices (*i.e.,* "raw" examples) $E = \{ex_1, ex_2, \ldots, ex_n\}$ extracted for a method $m_j \in P_i$, the *Example Evaluator* uses the Simian clone detector [12] to identify examples in E that are type-2 clones (*i.e.,* identical code fragments except for variations in comments, identifiers, literals, types, and whitespace). We chose to use Simian since

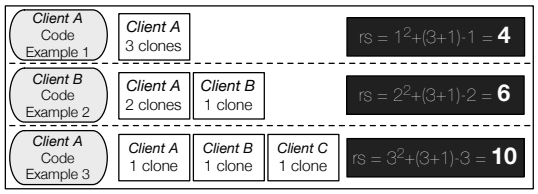

Fig. 2. rs assigned to three code examples all having three clones.

(i) it is able to detect type-1 and type-2 clones [13], (ii) it is very efficient in terms of CPU and memory usage, and (iii) it is freely available for research purposes [12]. While there are techniques able to detect also type-3 (*i.e.,* clones with added, removed, and/or modified statements) [14] and type-4 (*i.e.,* clones with similar semantics but different implementation) clones [13], we are not interested in capturing them, since they would likely represent different usage examples.

The output of Simian consists of all pairs of code examples $(ex_k, ex_l) \in E \times E$ identified as type-2 clones. Based on this information, the *Example Evaluator* collects the clones of each code example $ex_i \in E$ and uses them to compute a ranking score based on their "popularity" across the mined clients. The conjecture, inspired by previous work by Keivanloo *et al.* [1], is that code examples found several times in the clients (*i.e.,* having a high number of type-2 clones) represent good usage patterns of the analyzed method. Thus, the more type-2 clones a code example has, the higher the ranking score assigned by MUSE to the code example is. However, the *Example Evaluator* makes a distinction between clones found in the *same* client and clones spread across *different* clients, considering the latter as more important. Clones present within the same client could be method usages implemented by the same developer, possibly as result of copied and pasted source code. Thus, they are considered less indicative of "popularity" of a code example, compared to clones present in different clients. Formalizing, the *ranking score* (rs) for a code example ex_i is computed as follows:

$$rs(ex_i) = \#clients^2 + \#clones + 1 - \#clients \quad (1)$$

where $\#clones$ represents the total number of ex_i's clones identified across the clients, the added unit represents ex_i itself, and $\#clients$ represents the number of different client projects in which ex_i and its clones were found. To better understand how the ranking works, Fig. 2 shows the rs (see the black rectangles) computed for three code examples, each one having three clones in the clients. As it can be noticed, the most important factor in the ranking is the $\#clients$.

Based on their ranking score, the code examples are listed in descending order to form a ranked list, where each item consists of a group of code examples G_i, represented by a code example and followed by its clones. Referring back to the scenario depicted in Fig. 2, Example 3 (and its clones) would be ranked in the first position, Example 2 in the second position, and Example 1 in the third position. This ranking is used by MUSE to present the code examples of a method to the developer: the higher the ranking is, the higher the estimated usefulness of a code example is. Note that the ranking depends on the choice of clients. Some potentially useful examples might be ranked low if the client applications rarely use such examples and, *vice versa*, a given example may not be particularly useful outside the set of chosen examples.

2) Selection: As mentioned before, each item in the ranked list consists of a group G_i of similar code examples (*i.e.,* type-2 code clones). We conjecture that showing to the developer a set of type-2 clone examples would result in information overload. Hence, MUSE shows the developer just one representative example for each group, *i.e.,* the code example in G_i easier to *read* and *reuse*.

MUSE uses the *readability* metric proposed by Buse and Weimer [15] to evaluate the readability of a given example. This metric combines a set of low-level code features (*e.g.,* identifiers length, number of loops, *etc.*) and has been shown to be 80% effective in predicting developers' readability judgments. We used the authors' implementation of such a metric, which is available at http://tinyurl.com/kzw43n6. Given a source code fragment (a code example in our case), the readability metric takes values between 0 (lowest readability) and 1 (maximum readability).

Regarding the ease of *reuse*, given a code example ex_i, we define and measure its reusability as follows:

$$reuse(ex_i) = \begin{cases} \frac{\#JavaObjectTypes}{\#ObjectTypes} & \text{if } \#ObjectTypes > 0 \\ 1.0 & \text{otherwise} \end{cases}$$

where $\#JavaObjectTypes$ is the number of object types used by ex_i and belonging to the Java framework, and $\#ObjectTypes$ is the total number of different object types used by ex_i. The *reuse* metric is in the range $[0, 1]$, where 0 indicates that all object types in the code example are "custom objects" (low *reusability*), and 1 indicates that all object types in the code example belong to the Java framework or that no object types are present in the code example (high *reusability*). The intuition behind this metric is that reusing a code example that makes use of custom object types requires importing those objects into the code under development, which implies an extra task during reuse. Thus, the higher the number of custom object types used by a code example is, the lower its *reuse* value is, hence it is more difficult to reuse the example.

Since the readability and reuse metrics are defined in the range $[0, 1]$, we linearly combine them to obtain the following *selection score* (*ss*):

$$ss(ex_i) = 0.5 \times readability(ex_i) + 0.5 \times reuse(ex_i) \quad (2)$$

indicating the overall score of the code example ex_i. Note that we are assigning the same "importance" to both metrics when computing the overall selection score, as we do not prefer *readability* over *easiness of reuse* or *vice versa*. Clearly, the weights can vary, if needed.

MUSE selects from each group G_i of type-2 clone examples in the ranked list the one having the highest ss value. Thus, after the *selection* process, each method in P_i is associated with a list of ranked and diverse code examples.

C. Example Documentation and Injection

The ranked list of representative code examples for each method is provided to the *Example Injector*. This component extracts information from the Javadoc documentation of each method $m_j \in P_i$ and includes it in the code examples. Currently, MUSE extracts the textual descriptions of m_j's parameters (identified with the tag `@param`) and includes them as inline comments to explain the arguments passed to m_j invocation right where it occurs. The generated examples look like the one shown in Fig. 1 (at page 1) for the `copyInputStreamToFile` method of the `Apache commons-io` project. Finally, every ranked list of representative code examples is presented as an HTML page, which is injected into the official Javadoc documentation (if provided by the user). An example of a Javadoc page augmented by MUSE with the generated examples can be found at http://tinyurl.com/msvvusw (check the orange links).

III. Study I: Validation of Ranking and Selection

Study I is a survey with software developers, with the *goal* of evaluating MUSE's example ranking and selection.

A. Research Questions and Context Selection

This study aims at providing a human assessment of the *ranking* and *selection* heuristics. Thus, we formulate the following research questions:

> **RQ$_1$:** *Does* MUSE*'s example ranking reflect developers' judgment of code examples representing groups of clones?*
>
> **RQ$_2$:** *Does* MUSE*'s example selection reflect developers' judgment of similar code examples?*

The study was conducted through an online survey. We invited 20 industrial developers from our professional networks. Each participant completed a pre-questionnaire, and then performed the evaluation of both the ranking and the selection of code examples for six randomly selected methods (*i.e.,* one from each object system).

The pre-questionnaire aims at assessing the participants' experience and expertise. It consists of the following questions:

- *For how many years have you developed software?* Any numeric value equal or higher than zero was accepted.
- *Do you have any industrial experience as software developer? If yes, how long?* Any numeric value equal or higher than zero was accepted.
- *How often do you make use of third party libraries in your software projects?* Possible answers fall in a four-point Likert scale [16]: *Never*; *Rarely* – I use third party libraries in less than 33% of my projects; *Occasionally* – I use third party libraries in more than 33% but less than 66% of my projects; and *Frequently* – I use third party libraries in more than 66% of my projects.
- *How often do you use code examples found on the Internet to check how to use the API of a library?* Possible answers adopt a four-point Likert scale similar to previous one: *Never*; *Rarely* – I use code examples for less than 33% of the APIs I use; *Occasionally* – I use code examples for more than 33% but less than 66% of the APIs I use; and *Frequently* – I use code examples for more than 66% of the APIs I use.

TABLE I
APACHE LIBRARIES USED IN THE EMPIRICAL STUDIES.

Library (version)	#Public Methods	#Clients	#Methods with Examples	Mean #Examples per method
commons-io (2.4)	979	87	94	4.96
commons-lang3 (3.1)	1,900	54	86	3.87
httpclient (4.1.2)	1,369	14	82	4.84
poi (3.9)	18,239	22	293	4.67
tika (1.4)	1,610	12	63	3.35
xerces2-j (2.9.1)	6,645	10	50	8.28

The objects of this study are: (i) the libraries for which MUSE mines examples, and (ii) the clients from which examples are extracted. In terms of libraries, we considered six open-source Apache projects, including two generic, widely-adopted libraries (`commons-io` and `commons-lang3`) and four libraries that serve to more specific tasks (`httpclient`, `poi`, `tika`, and `xerces2-j`). The characteristics of the projects are reported in Table I, including the number of clients MUSE used in each case, the number of methods for which MUSE found examples, and the mean number of examples found for each method.

B. Study Design and Analysis Method

In order to evaluate the ranking and the selection of code examples, we adopted the following generic procedure. Suppose that we have extracted the code examples $e1, \ldots, e_7$, and that they were grouped in the following clone groups: $G_1 = \{e_1, e_2\}$, $G_2 = \{e_3, e_4\}$, and $G_3 = \{e_5, e_6, e_7\}$. Assume also that, for each clone group, MUSE selected (based on their readability and reusability) e_1, e_4, and e_6, respectively, as the representative examples. Finally, assume that MUSE has ranked these examples as follows: (1) e_6, (2) e_1, and (3) e_4.

We asked the participants to perform two tasks to assess both ranking and selection. First, we showed each participant the representative examples (in our case e_6, e_1, and e_4) ordered randomly. For each example, the participants were asked to assign a score using a Likert scale: 1=*Not useful at all*; 2=*Slightly useful* 3=*Useful*; and 4=*Very useful*. The expectation here is that the higher the representative example is in the rank, the greater the score provided by the developer to that example should be.

After that, for one (randomly selected) clone group among G_1, G_2, and G_3, say G_3, we showed the examples belonging to that group (in this case e_5, e_6, and e_7) in a random order. Just as before, we asked the participants to evaluate each example using the 1–4 Likert scale. In this case, the expectation is that the score provided by the developer to the example selected as representative should be greater than the score provided to the other examples in the clone group.

To address **RQ$_1$**, we compute and report the Spearman's rank correlation between the participants' evaluation (in Likert scale) and the example ranking (according to equation 1) for each evaluated example. Our expectation is that, if our ranking heuristic is adequate, then the evaluation should be negatively correlated with the ranking. We report descriptive

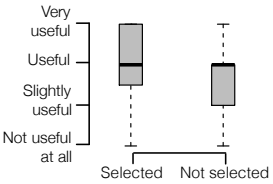

Fig. 3. Evaluation of *selected* and *not selected* examples.

statistics showing the evaluation distribution for examples having different ranking. To address $\mathbf{RQ}_2$, we compare the rating of each example we selected (*e.g.,* e_5 for clone group G_3) against the rating provided to all other examples using the Mann-Whitney test [17]. Our expectation is that, if the selection heuristic is adequate, then the evaluation of the example selected by the heuristic (according to equation 2) should be significantly higher than of all the other examples. The results are considered as statistically significant at $\alpha = 0.05$.

C. Analysis of the Results

Nine participants completed the survey. They declared, on average, 6.9 years of development experience (median=7) and 4.1 years of industrial experience (median=4). All claimed to *frequently* using third-party libraries in their projects, and eight of them (89%) reported also *frequently* using code examples found on the Web to check how to use APIs.

By analyzing the ranking results ($\mathbf{RQ}_1$), we obtained a (statistically significant, p-value < 0.001) Spearman rank correlation with $\rho = -0.24$. As expected, the correlation is negative, although small. The congruence between the participants' assessment and MUSE's ranking is, however, confirmed by the ratings assigned by participants to the code examples having different rankings. In particular, the top-1 code examples received a 3.1 score, on average, followed by those in 2nd and 3rd position (2.5), and to those in position 4th, 5th, and 6th (2.3). Note that none of the methods of this evaluation had more than six different code examples. In summary, we can answer $\mathbf{RQ}_1$ by stating that the MUSE's ranking heuristic properly reflects the developers' evaluation.

Turning to $\mathbf{RQ}_2$, the results of the Mann-Whitney test did not show a statistically significant difference between the ratings assigned by participants to the examples selected as representative and those assigned to all other examples (p-value=0.26). Looking into the data we noticed that, for some of the methods of this study, the code examples showed to the participants were extremely similar, thus pushing participants to assign flat ratings to the presented examples (*e.g.,* assign *useful* to all clones). However, looking at the ratings distributions shown in Fig. 3 for *selected* and *not selected* code examples, it is clear that the *selected* examples were generally preferred by participants, thus supporting our selection heuristic. We can therefore answer $\mathbf{RQ}_2$ by saying that the *selected* examples are considered more useful by the developers than the *not selected* ones.

IV. Study II: Perceived Usefulness of Examples

Study II is a survey with software developers, with the *goal* of investigating their perceived usefulness of the code examples generated by MUSE. The *context* consists of (i) *objects*, *i.e.,* code examples extracted by MUSE for methods of six open source systems, and (ii) *participants*, *i.e.,* 119 open source and professional developers providing their opinions on MUSE's code examples.

A. Research Question and Context

We aim at investigating whether developers consider the generated code examples useful for understanding the usage of methods. Thus, we formulate the following research question:

> **$\mathbf{RQ}_3$:** *Are* MUSE*'s usage examples considered useful by developers?*

We answer this research question by asking open-source and professional developers to assess, through an online questionnaire, their perceived usefulness of MUSE's usage examples. The objects (*i.e.,* code examples) of our study are generated by MUSE for the same six systems adopted in Study I. In particular, we randomly selected ten methods from each system for which MUSE generated at least one usage example. We asked the study participants to evaluate the usefulness of the top ranked examples (*i.e.,* the best ones as selected by MUSE's *Examples Evaluator*) for each of the selected methods via an online survey described below.

As participants of this study, we targeted (i) developers of the six object systems (referred to as *library developers*) and (ii) developers of open-source projects using any of the object systems as libraries (referred to as *client developers*). The list of *library developers* was extracted from the official Apache committers' page (http://tinyurl.com/4bg8yw4), which reports the committers for each Apache project. While such a page only reports the committers' id (a sort of nickname), their email can be obtained by adding `@apache.org` to the committer's id. In total, we identified 100 *library developers*. The process for extracting the *client developers*' emails required building an ad-hoc crawler for visiting the Maven page of each identified client project and looking for a "Developers" table (see *e.g.,* http://tinyurl.com/lmgd9m3). Using such a crawler, we identified 609 *client developers*. Our choice of participants was driven by the goal of evaluating the usefulness of the method usage examples from the perspective of developers familiar with the libraries. We consider the *library developers* and the *client developers* uniquely qualified to assess the difficulties encountered in understanding the usage of the methods, hence also able to assess whether MUSE's examples can help in such a process.

Each developer received an email with: (i) instructions on how to participate in our study, and (ii) a link to the survey for the specific object system for which the participant is a *library* or *client* developer.

B. Study Design and Analysis Method

The survey is composed of two parts. The first one aims at gathering information about the developers' background, and contains the same questions as the *Study I* pre-questionnaire (see Section III-B), with the addition of the following question: *Have you contributed to P_i's implementation?*, a yes/no

System	#Developers	#Industrial Experience	#Library Developers
commons-io	57	50	6
commons-lang	31	24	1
http-client	8	6	3
poi	7	6	3
tika	5	3	0
xerces-j	11	6	2
total	119	95	15

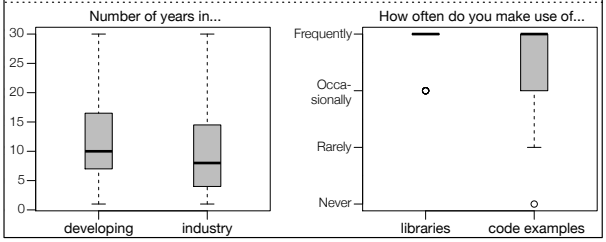

Fig. 4. Study II: Participants data

question aimed at verifying if the developer was a contributor (*i.e., library developer*) to P_i.

The second part of the survey consists of the evaluation of ten code examples covering ten different API methods of each object library. For each code example to evaluate, we showed to the participants the full path of the method that the code example refers to, *i.e.,*`package.class.method-(parameters)` and the best code example generated by MUSE (as selected by the *Example Evaluator*). Then, we asked the participants *"How useful is the code example in understanding the usage of the aforementioned method?"* This question was answered by providing a score on a four-point Likert scale: *Not useful at all*; *Slightly useful*; *Useful*, and *Very useful*. Also, the participants had the opportunity to justify their score in a free-text form shown for each evaluated code example, and they were encouraged to leave any additional comment about the evaluated code examples.

The survey of each object system was hosted on a Web application (http://www.qualtrics.com/) that allows participants to complete the questionnaire in multiple rounds (*e.g.,* answering a few questions on one day, and the others later). The developers had 15 days to respond. At the end of this period, we collected 119 complete questionnaires distributed among the object systems as reported in the top part of Fig. 4.

We analyze the results using box plots. Then, we rely on developers' comments to qualitatively discuss MUSE's strengths and weaknesses highlighted by the evaluation.

C. Analysis of the Results

We first analyze the developers' background, summarized in Fig. 4. The 119 developers involved in our study have, on average, 13 years of developing experience (median=10). The 95 (80%) developers with industrial experience spent, on average, 9 years in industry (median=8)—see box plots in the left-bottom corner of Fig. 4. Almost all developers (113 out of 119—95%) *frequently* use third-party libraries in their projects, while 76 (64%) *frequently* use code examples to check how to use the API of a library, and just four developers (3%) *rarely* (3) or *never* (1) use them—see box plots in the right-bottom corner of Fig. 4. Fifteen of 119 developers (13%) have contributed to the development of the object systems.

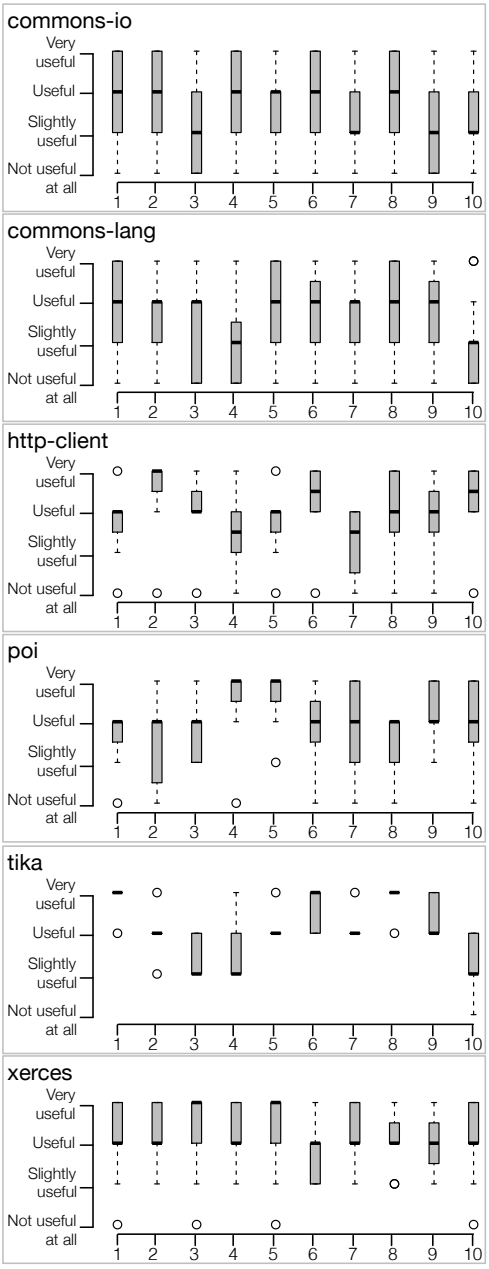

Fig. 5. Study II: Answers to the question *"How useful is the code example in understanding the usage of the aforementioned method?"*

To answer $\mathbf{RQ}_3$, Fig. 5 aggregates the answers provided by developers when evaluating the usefulness of the ten code examples (numbered with IDs from 1 to 10) showed for each system. A global view across the 60 assessed examples shows that 82% of them (49 examples) have been appreciated by developers as *very useful* (13%) or *useful* (69%) (median scores). Only 11 (18%) of the code examples generated by MUSE and evaluated by participants were found *slightly useful* (median score). None of the 60 examples were considered as *not useful at all* (median). Also, the perceived usefulness of MUSE's code examples seems to be independent from (i) the system on which it is applied, and (ii) the number of client projects parsed to extract such examples (see Table I for details

about the number of analyzed clients for each system).

We also analyzed the participants' rationale justifying their scores. Positive scores were seldom justified by developers and mostly included exclamations like "*Nice!*", "*This is a very good one!*", or *"Excellent example!"* We also received positive feedback about the feature that generates documentation and inserts it in the code examples, *e.g.,* "*You have an immediate example of how to use the API and a description of the parameters: this way you don't have to endlessly switch between code and docs.*"

The most frequent reason for negative scores was the *unnecessary complexity* of some of the examples. This happens when the examples generated by MUSE include code statements that, even if needed to invoke the method and thus included in the slicing step, do not directly relate to the method usage. An instance of such a case is the third code example in the `commons-io` survey, which received several low scores (median=*slightly useful*) . This code example refers to the `copyDirectory(File,File,FileFilter)` method in the `FileUtils` class. Such a method copies the contents of an existing directory into a target directory by applying an optional filtering (*e.g.,* only copying files having a specific extension). The generated code example consists of 44 lines of code, out of which the first 38 are aimed at instantiating objects used later to create the two `Files` representing the existing and the target directories. Thus, just the last six lines of the example were strictly related to the method invocation. This aspect has been highlighted in several developers' comments, *e.g.,* "*The example should contain only the lines after 38; otherwise the code will seem too complex (event if it is not; like in this case)*."

Developers also pointed out that they would have expected a *description of the returned value* in some of the examples. For instance, several developers pointed out this issue for the code example related to the `getLevenshteinDistance(CharSequence, CharSequence)` method of the `commons-lang`'s `StringUtils` class, *e.g.,* "*It would be fantastic to also report the output of the API invocation when possible*." This feature is not currently implemented by MUSE, but we plan to integrate it in the future.

Another situation highlighted in some comments was the *lack of alternative usage scenarios* for a given method. Our design choice for this study was to limit the time needed to complete the survey as much as possible, thus favoring a high response rate. For this reason, we asked developers to evaluate only one of the code examples generated by MUSE (the top-ranked one) for each method. Therefore, while we did not show alternative usage scenarios, it is possible that by presenting MUSE's entire ranked list of examples for each method, this issue would have not being reported.

The open comments at the end of the questionnaire confirmed MUSE's usefulness. There were enthusiastic comments, such as, "*The purpose of this work seems to be appealing*", "*Overall good; sometimes too complex; but if they are automatically generated is interesting*", or "*I would say 80% of them is useful. Some contain statements that are not needed to use the method*." There were also constructive suggestions, helpful to plan our future work, *e.g.*: "*The code examples that are most useful at the method level are unit tests. They express the behavior and usage of the code through example inputs and outputs*", "*The comments with argument descriptions are very useful; however API examples need to be focused on use of the API call and not incorporate extra code*."

In conclusion, we answer $\mathbf{RQ}_3$ stating that 82% of MUSE's code examples have been considered either useful (69%) or very useful (13%) by developers with consistent results achieved across the six object systems.

V. Study III: Extrinsic Evaluation

Study III is a controlled experiment with the *goal* of evaluating how useful are MUSE's code examples to developers during a programming task. The *quality focus* is the completeness of the task a developer can perform in a limited time frame, *e.g.,* because of a hard deadline. The *context* consists of 12 industrial developers as *participants* and of development tasks performed by using APIs of specific libraries as *objects*.

A. Research Questions and Context

We aim at addressing the following research question:

> $\mathbf{RQ}_4$: *Do* MUSE*'s examples help developers to complete their programming tasks?*

To answer $\mathbf{RQ}_4$ we asked the twelve developers to perform two programming tasks involving the use of specific libraries, one with the availability of MUSE's code examples, and another one without the examples. The two tasks are:

- T_1: Create a Java program that, given the URL of a PDF document available on the Internet, downloads the PDF and prints in the console all its metadata and textual content. The task consisted of three subtasks: (i) download the PDF using the `httpclient` library; (ii) extract the PDF metadata using the `tika` library; and (iii) extract the PDF textual content using the `tika` library.
- T_2: For this task we created a directory containing (in different subdirectories) files having different extensions, including *.csv*, storing portions of the OpenFlights Airports database[1]. Then, we asked participants to create a Java program that, given the local path of the directory described above: (i) creates an output directory D and an output *.csv* file F; (ii) retrieves all *.csv* files related to the airports database by using the `commons-io` library; (iii) checks the completeness of the data reported in each *.csv* file by verifying that its rows (each one containing data about one airport) contain the correct number of columns by using the `commons-lang3` library; (iv) copies the "correct" files into D by using the `commons-io` library; and (v) prints in F the 100 first rows of each "correct" file by using the `commons-lang` library.

Each task involved at least two libraries for which MUSE generated code examples, as described in the previous studies.

[1] http://openflights.org/data.html

Note that while (part of) the implementation of both tasks is possible using alternate libraries (or none), we required the participants to use specific libraries in each step. Otherwise, it would not have been possible to observe the usefulness of the MUSE's examples when using the libraries.

In order to assess the quality of the implementations and answer $\mathbf{RQ}_4$, we measure the task *completeness* as the dependent variable in our experiment, as done in previous similar studies (see *e.g.,* [5]). For a given task T_i with subtasks $T_{i,j}$, we define the completeness of T_i as the sum of the completeness of its subtasks $T_{i,j}$. The completeness score of the subtasks is proportional to its difficulty and complexity: $T_{1,1} = 50$; $T_{1,2} = 30$; $T_{1,3} = 20$; $T_{2,1} = 5$; $T_{2,2} = 25$; $T_{2,3} = 30$; $T_{2,4} = 15$, $T_{2,5} = 25$. If a subtask $T_{i,j}$ is implemented correctly, then it receives its maximum score, otherwise it receives *0* (zero). Hence, the completeness score for a task T_i ranges from 0 to 100. Since this is difficult to automatically evaluate, we asked two independent developers to serve as evaluators and measure the completeness by performing code reviews on each task implemented by each participant. The evaluators did not know the goal of the study, nor which tasks were performed with or without MUSE's code examples. The evaluators compared the tasks independently and conducted a discussion when their scores diverged. This happened on 7 out of the 24 evaluated tasks (*i.e.,* 2 tasks for each of the 12 participants) and in each case the evaluators reached an agreement quickly, by performing an additional code inspection.

The main factor and independent variable of this study is the availability of the code examples generated by MUSE. Specifically, such a factor has two values: *i.e.,* code examples available (CE) or not (NCE). Participants were allowed to use any resource they want to complete the tasks, including material available on the Internet (note that in both treatments we provided the official Javadoc documentation of the libraries to use, augmented in CE with the code examples generated by MUSE). This was done to simulate a real development context. Also, note that while other techniques have been proposed in the literature to generated code examples (*e.g.,* [7], [8], [6]), we chose to use as control group (*i.e.,* NCE) the current state of the practice. Indeed, the wide availability of Q&A websites (*e.g.,* Stack Overflow), forums, and dedicated websites from which developers can grab code examples, makes the Internet the most natural benchmark for our approach.

Factors that could influence the results are: (i) the (possible) different difficulty of the two tasks; (ii) the different level of fatigue in the two sessions; (iii) the participants' development experience; and (iv) their knowledge of the libraries.

B. Study Design and Analysis Method

The study design—shown in Table II—is a classical paired design for experiments with one factor and two treatments. The design is conceived in such a way that: (i) each participant worked with both CE and NCE; (ii) each participant performed different tasks (T1 and T2) across the two sessions; (iii) different participants worked with CE and NCE in different ordering, as well as on the two different tasks T1 and T2. Overall, this means partitioning participants into four groups, receiving different treatments in the two lab sessions.

TABLE II
STUDY III: DESIGN.

Session	Group A	Group B	Group C	Group D
1	T1-NCE	T1-CE	T2-NCE	T2-CE
2	T2-CE	T2-NCE	T1-CE	T1-NCE

First, we conducted a pre-experiment briefing where we illustrated in detail the experiment procedure. We made sure not to reveal the study research questions. After that, participants filled-in a pre-questionnaire including all questions present in the pre-questionnaire of *Study I* (see Section III-A), plus a question requiring a self-assessment of their knowledge of the four libraries used in the two tasks on a four-point Likert scale going from 1=*No knowledge* to 4=*High knowledge*. Then, the participants had to perform the study in two sessions of 60 minutes each. In other words, participants had a maximum of 60 minutes to complete each of the required tasks. Each participant received the instructions for the task to perform in the first session. After 60 minutes, each participant provided the implemented code for the required task. A 30-minutes break was given before starting the second session to avoid fatigue effects. During the break participants did not have the chance to exchange information among them. After the break, each participant received the instructions for the second task and, 60 minutes later, they provided the implemented code for the required task. Finally, once the study was completed, we asked participants to also evaluate the usefulness of the code examples by providing (and justifying) a score on a four-point Likert scale: *Not useful at all*, *Slightly useful*, *Useful*, and *Very useful*.

As for the analysis method, we present box plots of the completeness achieved by participants with the two treatments (*i.e.,* CE and NCE). Also, we statistically compare the two distributions of completeness by using the Mann-Whitney test [17]. The results are intended as statistically significant at $\alpha = 0.05$. We also estimate the magnitude of the difference between the two different distributions by using the Cliff's Delta (or d), a non-parametric effect size measure [18] for ordinal data. We followed the guidelines by Grissom and Kim [18] to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Finally, to check the influence of the various co-factors (*i.e.,* task, lab session, development experience, industrial experience, and knowledge of the used libraries) from a statistical standpoint, and their interaction with the main factor treatment, we use the permutation test [19], a non-parametric alternative to Analysis of Variance (ANOVA).

C. Analysis of the Results

The pre-questionnaires provided the following information about the background of the participants involved in our study: on average, they have five years of development experience (median=4.5) out of which two have been spent in industry (median=2). The majority of developers (70%) *often* or *very*

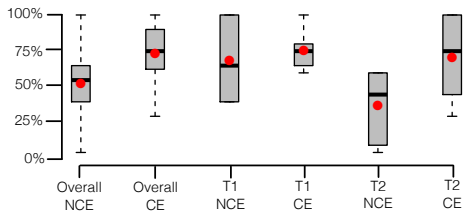

Fig. 6. **RQ**$_4$: Completeness achieved by participants.

often use third-party libraries in their projects and *very often* (83%) use code examples found on the Internet to understand how to use APIs. Their knowledge of the four object libraries required for the two tasks is generally limited (*i.e.,* median=1.5 for `commons-io`, median=1 for `commons-lang`, median=2 for `httpclient`, and median=1 for `tika`).

Fig. 6 reports the box plots of completeness achieved by participants with (CE) and without (NCE) MUSE's code examples. Results are shown when considering both tasks as a single dataset (*overall*), and separately. On average, participants achieved 53% completeness in NCE against the 73% in CE. Such a difference is statistically significant (p-value=0.03) with a medium effect size (d=0.472).

As it can be noticed from Fig. 6, the difference in completeness in favor of the CE treatment is present in both tasks. The difference is 8% in task T_1 (*i.e.,* 76% *vs* 68%), while it grows up to 33% in task T_2 (*i.e.,* 71% *vs* 38%). Only the participants' completeness in task T_2 has a significant difference between the two treatments (p-value=0.02) with a large effect size (d=0.611). Such a difference can be justified by considering that task T_2 required to use more methods from the object libraries than task T_1.

When asking in the post-questionnaire about the usefulness of the code examples, nine participants (75%) rated them as *useful*, two (17%) as *slightly useful*, and one (8%) as *not useful at all*. The latter justified his answer by explaining that *"while the examples can help in understanding the method usage, I still prefer surfing on the Internet for more rich discussions on how to use methods."*

Finally, the statistical analysis of the co-factors' influence has highlighted that only the *knowledge of the used libraries* has a significant effect on the completeness (p-value=0.006), although it does not interact with the main factor. In other words, people with higher knowledge of the used libraries perform better, independently of the availability of the code examples generated by MUSE.

Summarizing, our experiment allows us to answer **RQ**$_4$ by stating that the use of MUSE's code examples increased the developers' work quality (i.e., completeness) by up to +20%.

VI. Threats to Validity

Threats to *construct validity* concern the measurements performed to address our research questions. Both Study I and II rely on a subjective assessment through a Likert scale [16]. Study III aims at overcoming this limitation by assessing the examples' usefulness in an actual development task. Concerning study factors that can influence our results (*internal validity*), we assessed and reported the participants' background in all our studies. Furthermore, in Study III we also analyzed whether this and other confounding factors, *e.g.,* task ordering, could influence the results. Last, but not least, in Study I and II we randomized the ordering in which the examples were shown to the participants, to mitigate any sort of learning or fatigue effect.

As detailed in the analysis method of the three studies, we used appropriate statistical tests, correlation, and effect size measures as needed to support our results and to mitigate threats related to *conclusion validity*. Finally, it is possible that the selected examples used in the studies are not fully representative of the readability/reusability (in Study I) and usefulness of the examples generated by MUSE (in Study II and Study III). This may limit the generality of our results (*external validity*). However, we tried to involve—at least in Study I and II—a relatively large set of examples from six different projects to mitigate this problem. Study III is smaller, but still it involved examples generated for four different projects and, as explained above, it assesses the examples' usefulness better than just a questionnaire.

VII. Related Work

API usage examples are a common output of code search approaches based on text matching. Chatterjee *et al.* [3] and Keivanloo *et al.* [1], for instance, use textual similarity to return a ranked list of abstract examples relevant to a natural language (NL) query formulated by the user and expressing her task at hand. Similar to MUSE, Chatterjee *et al.*'s approach (SNIFF) documents, clusters and ranks the extracted code snippets. The clustering and ranking criteria, however, are based on the textual content and the frequency of the extracted patterns within the data set, respectively. Keivanloo *et al.*'s approach combines textual similarity and clone detection techniques to find relevant code examples, and ranks them according to (i) their textual similarity to the query and (ii) the completeness and popularity of their encoded patterns. In a similar token, but generating a ranked list of concrete API usage examples, the Structural Semantic Indexing (SSI) proposed by Bajracharya *et al.* [2] combines heuristics based on structural and textual aspects of the code, based on the assumption that code entities containing similar API usages are also similar from a functional point of view. Other tools, such as Strathcona [4] and Prompter [5], automatically generate queries from the developer's code context. In Strathcona [4], the code snippets relevant to such queries are identified through six structural heuristics based on inheritance links, method calls, and used types. The resulting examples are ranked according to their frequency in the final set. Prompter [5] matches the generated query with Stack Overflow entries, to automatically push discussions relevant to the developers' task at hand. Unlike MUSE, task-based code search techniques rely heavily on textual analysis to extract, cluster and rank code elements. Our approach builds code snippets based on static program analysis (specifically code slicing). Moreover, task-based API usage examples are not attached to a single API—they show how different APIs can be used for the same

goal (*i.e.,* the task at hand). MUSE's examples are meant to help developers to reuse a particular method, once they know it might help with the task at hand. In other words, we envision MUSE to be used by developers for getting more precise information about the methods they want to use, after using one of the tools mentioned above.

More related to our approach is the research on extraction of code examples showing the usage of specific APIs. MAPO, proposed by Xie and Pei [7] and extended by Zhong *et al.* [9], sets the foundations on mining abstract usage examples of a given API method. MAPO analyzes code snippets retrieved by code search engines to extract all the call sequences involving the desired API method. A subset of sequences covering all method calls is identified and then clustered into similar usage scenarios, according to heuristics based on method names, class names, and called API methods composing each sequence. For each cluster, MAPO identifies usage patterns based on frequent call sequences, and, finally, it ranks the patterns based on their similarity with the developer's code context. MUSE builds on the same idea of extracting examples for a specific method but it differs from MAPO in several ways: (i) to find the usage of a method, MUSE considers its complete signature, whereas MAPO uses only the method name, which can lead to mismatch between methods with the same name but different signatures (*i.e.,* polymorphic methods); (ii) MUSE uses slicing for locating the statements associated to the analyzed usage, as opposed to control-flow based heuristics; (iii) for clustering similar examples, MUSE applies widely-used clone detection techniques, while MAPO uses similarity heuristics; (iv) MUSE's ranking is independent of the developers' context and considers the readability and easiness of reuse of the examples, instead of the similarity measures used by MAPO, which depend on the development context; and finally, (v) MUSE's output is a list of concrete usage examples patterns, as opposed to abstract examples that do not explicitly offer control flow and instantiation steps.

UP-Miner [8] is a variation of MAPO that removes the redundancy in the resulting example list. To this end, UP-Miner clusters the extracted method sequences based on n-grams and discovers a pattern for each cluster by applying a frequent sequence mining algorithm. As these patterns might be similar, UP-Miner executes another clustering round on them. The resulting patterns are ranked according to their frequency and presented as probabilistic graphs. Similar to MAPO, but different from MUSE, UP-Miner produces abstract examples. Moreover, MUSE makes use of clone detection techniques to cluster similar code examples, instead of the probabilistic language model used by UP-Miner.

Other approaches focus on generating code examples for an entire API or a set of APIs. Acharya *et al.* [20] detect patterns on the conjunct usage of different APIs by mining frequent partial orders from common API usage scenarios. Given a set of APIs, inter-procedural static traces are produced to construct and cluster partial orders from API sequential patterns, which are in turn mined to extract ordering rules between APIs. Differently from MUSE, this approach considers how a set of APIs is usually combined across different systems.

More recently, Buse and Weimer [6] proposed to generate documented abstract API usages by extracting and synthesizing code examples of a particular API data type. Their approach mines examples by identifying and ordering (i) code instantiations of the given data type, and (ii) the statements relevant to those instantiations as defined by previously extracted path predicates, computed from intra-procedural static traces. The examples are then clustered based on their statement ordering and data type usage. For each cluster, an abstract example (*i.e.,* a usage pattern) is formed by merging its code examples, and finally documented according to predefined heuristics that depend on the kind of statement and most frequent names in the mined code. As with the code search approaches, MUSE could be used to complement these abstract API usage patterns with specifics on the methods of interest.

Also related to our work is the study conducted by Ying and Robillard [21] aimed at investigating how humans produce code examples, summarizing them from existing code. One of their findings is that developers rarely copy the code verbatim, rather they try to abstract the relevant parts. MUSE's main innovation is the use of code slicing, which aims at removing unnecessary code from the examples, emulating the developers' behavior highlighted by Ying and Robillard's study.

VIII. Conclusion and Future Work

MUSE extracts and documents code examples from the code of applications that make use of the specific method. Among related approaches, MUSE is novel as it extracts concrete method usage examples (as opposed to abstract usage patterns) by using static code slicing, and documents them with comments that help to understand the method's parameters. MUSE has been empirically evaluated through three different empirical studies, involving in total 140 open-source and professional developers and conducted on six Java libraries. The first study (a survey) showed that MUSE ranks and selects code examples close to how developers do. The second (also a survey) revealed that 82% of MUSE's examples are perceived as useful or very useful by developers. The third study (a controlled experiment) showed that developers improve the quality of their implementations when using MUSE's examples. Future work will focus on improving: (i) the examples' documentation, *e.g.,* by commenting the output of the method invocation; and (ii) the filtering of less important statements.

Acknowledgments

The authors would like to thank Nikolaos Tsantalis for his help in using the JDeodorant slicer and all the participants of our three studies. Laura Moreno and Andrian Marcus are supported in part by grants from the National Science Foundation (CCF-1017263 and CCF-085706). Massimiliano Di Penta is partially supported by the Markos project, funded by the European Commission under Contract Number FP7-317743.

REFERENCES

[1] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *ICSE*. ACM, 2014, pp. 664–675.

[2] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 157–166.

[3] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A search engine for java using free-form queries," in *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, ser. Lecture Notes in Computer Science. Springer, 2009, pp. 385–400.

[4] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, 2005, pp. 117–125.

[5] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to turn the IDE into a self-confident programming prompter," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM, 2014, pp. 102–111.

[6] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 782–792.

[7] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 54–57.

[8] 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 *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 319–328.

[9] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[10] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. (2015) How can I use this method? The MUSE approach. Replication package. [Online]. Available: http://tinyurl.com/mgwo9le

[11] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[12] S. Harris. (2003) Simian - Similarity Analyser. [Online]. Available: http://www.harukizaemon.com/simian/

[13] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[14] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[15] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[16] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. London: Pinter, 1992.

[17] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[18] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[19] R. D. Baker, "Modern permutation test software," in *Randomization Tests*. Marcel Decker, 1995.

[20] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. ACM, 2007, pp. 25–34.

[21] A. Ying and M. Robillard, "Selection and presentation practices for code example summarization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. ACM, 2014, pp. 460–471.