Published in Empirical Software Engineering, 2019 September, Pages 1-35
https://doi.org/10.1007/s10664-019-09771-0

Why reinventing the wheels? An empirical study on library reuse and re-implementation

Bowen Xu[1] · Le An[2] · Ferdian Thung[1] · Foutse Khomh[2] · David Lo[1]

Abstract

Nowadays, with the rapid growth of open source software (OSS), library reuse becomes more and more popular since a large amount of third- party libraries are available to download and reuse. A deeper understanding on *why* developers reuse a library (i.e., replacing self-implemented code with an external library) or re-implement a library (i.e., replacing an imported external library with self-implemented code) could help researchers better understand the factors that developers are concerned with when reusing code. This understanding can then be used to improve existing libraries and API recommendation tools for researchers and practitioners by using the developers concerns identified in this study as design criteria. In this work, we investigated the reasons behind library reuse and re-implementation. To achieve this goal, we first crawled data from two popular sources, F-Droid and GitHub. Then, potential instances of library reuse and re-implementation were found automatically based on certain heuristics. Next, for each instance, we further manually identified whether it is valid or not. For library re-implementation, we obtained 82 instances which are distributed in 75 repositories. We then conducted two types of surveys (i.e., individual survey to corresponding developers of the validated instances and another open survey) for library reuse and re-implementation. For library reuse individual survey, we received 36 responses out of 139 contacted developers. For re-implementation individual survey, we received 13 responses out of 71 contacted developers. In addition, we received 56 responses from the open survey. Finally, we perform qualitative and quantitative analysis on the survey responses and commit logs of the validated instances. The results suggest that library reuse occurs mainly because developers were initially unaware of the library or the library had not been introduced. Re-implementation occurs mainly because the used library method is only a small part of the library, the library dependencies are too complicated, or the library method is deprecated. Finally, based on all findings obtained from analyzing the surveys and commit messages, we provided a few suggestions to improve the current library recommendation systems: tailored recommendation according to users' preferences, detection of external code that is similar to a part of the users' code (to avoid duplication or

Communicated by: Maurizio Morisio

Bowen Xu and Le An are contributed equally.

✉ Bowen Xu
bowenxu.2017@phdis.smu.edu.sg

re-implementation), grouping similar recommendations for developers to compare and select the one they prefer, and disrecommendation of poor-quality libraries.

Keywords Code reuse · Code re-implementation · Library recommendation systems

1 Introduction

Library reuse has been researched in the 1990s, researchers at that time claimed that while many companies were developing proprietary software libraries, library reuse was not yet a major force in most corporate software development (Krueger 1992; Griss 1993). However, nowadays, with the rapid development of open source software (OSS), library reuse has became a very common practice as more and more third-party libraries are available to be downloaded and reused (Abdalkareem et al. 2017; Heinemann et al. 2011; Ruiz et al. 2012). For example, a recent work concluded that in the world of open-source Java development, high reuse rate is not a theoretical possibility but rather a practical reality (Heinemann et al. 2011). Moreover, the availability of reusable functionality, which is a necessary prerequisite for library reuse to occur, is well-established in Java platform. In addition, the costs of developing and maintaining reusable libraries were considered as an investment during the software development in the 1990s (Kim and Stohr 1998). Today, many well-maintained library repositories, which target to different programming languages, have been built to help developers easily reuse code. For example, *NPM*,[1] *Maven*,[2] *RubyGems*,[3] *Packagist*,[4] *PyPI*[5] are respectively library managers/hosts for JavaScript, Java, Ruby, PHP, and Python. We observed that these repositories are growing rapidly. For example, in 2010, Sonatype reported that Maven Central contained over 260,000 Maven libraries.[6] By the end of 2018, the number of unique Maven libraries has reached 3,356,473, which is 12 times larger than it was in 2010. The growth of these open source libraries indicates that developers are more willing to share code. Since such bountiful supply of libraries is not likely to happen without sufficient demand from developers, this tendency suggests a growing demand for code reuse with libraries as well.

In recent literatures, some empirical studies have investigated code reuse in third-party libraries. For example, Mojica et al. (2014) conducted a large-scale empirical study based on more than 200,000 free Android apps across all 30 app categories in Google Play. They found that while library reuse is prevalent in mobile apps, those apps also inherit the disadvantages of reuse, such as increased dependencies. They suggested that more research is needed to analyze this negative impact. Zaimi et al. (2015) investigated the reuse of third-party libraries in five well-known open-source software projects: i.e., *dr Java*, *Findbugs*, *ArgoUML*, *jFreeChart* and *Mogwai*. The results of their study suggest that OSS projects heavily reuse third-party libraries. However, reuse decisions are not frequently revisited, and there is no clear evidence that reuse decisions are quality-driven. Although the above studies have provided insights into third-party library reuse, the reasons *why* developers reuse third-party libraries are still unclear.

[1] Nodejs, https://www.npmjs.com
[2] Maven, https://maven.apache.org
[3] RubyGems, https://rubygems.org
[4] Packagist, https://packagist.org
[5] PyPI, https://pypi.python.org/pypi
[6] Statistics for the Maven Repository, https://search.maven.org/stats

Moreover, some researchers have noticed the opposite phenomenon, i.e., developers re-implement the behavior of an existing library (Kawrykow and Robillard 2009; Sun et al. 2011). Kawrykow and Robillard (2009) proposed a code similarity detection approach that identifies cases of code re-implementation in software projects. To improve the accuracy of Kawrykow et al.'s approach, Sun et al. (2011) proposed a graph-based approach to detect code re-implementations. However, the reasons *why* developers re-implement code instead of using third-party library have not been investigated in the literature.

To fill the gaps left by the above-mentioned lines of work and deepen our understanding of the reasons behind the phenomena of library reuse and re-implementation, we conducted this empirical study with the aims to help software researchers and practitioners better understand the factors that developers are concerned with when reusing code. This understanding can then be used to improve existing library and API recommendation tools (e.g., Thung et al. (2013a), Rahman et al. (2016), Nguyen et al. (2016), and Gu et al. (2016)) by putting developers concerns as design criteria. Moreover, library developers can benefit from understanding key concerns that library users voiced when choosing between library reuse or re-implementation. This understanding helps to further improve the quality of the library.

In this work, we focus on two scenarios of library reuse and re-implementation, (1) replacing self-implemented code with an external library, (2) replacing an imported external library with self-implemented code. Our study investigates the following research questions:

RQ1 Why do developers replace their self-implemented method with an external library method?

RQ2 Why do developers replace an external library method with their self-implemented code?

RQ3 Under what circumstances do developers prefer to reuse or re-implement code?

To answer the above research questions, we conducted two types of surveys and performed a manual qualitative analysis on commit logs:

Individual Survey We surveyed developers who have experienced either of the following scenarios to get insights on their rationales: (1) A developer who replaced a self-implemented method by calling a method from a third-party library (library reuse); (2) A developer who replaced a method call to a third-party library method with a self-implemented method (library re-implementation). To identify real-world instances of the above scenarios, we analyzed commits in Java and Python repositories from multiple sources (e.g., F-Droid[7] and GitHub[8]). We wrote a script to automatically identify likely cases of library reuse and re-implementation. From these cases, we manually examined their correctness. Finally, for library reuse, we obtained 183 instances across 133 repositories. For code re-implementation, we obtained 82 instances across 75 repositories.

We built a customized survey for each of the identified true instances and sent it to the corresponding developer (who made the commit) to ask for the reasons behind the code reuse or code re-implementation. Finally, we received 36 responses out of 139 contacted developers (i.e., response rate: 25.9%) for the individual survey of library reuse, and 13

[7]F-Droid, https://f-droid.org

[8]Github, https://github.com

responses out of 71 contacted developers (i.e., response rate: 18.3%) for the individual survey of code re-implementation.

Open Survey We also conducted an open survey to get inputs from other developers. In the open survey, we first collected demographic of our respondents, i.e., educational attainment, preferred programming language, role in project and software development experience. Next, we asked them questions about library reuse and code re-implementation. We disseminated this open survey in several online communities through Reddit. We also sent the open survey to some of our colleagues, who work as software engineering researchers or developers. Finally, we received 56 responses from the open survey. For more details, please refer to Section 2.4.

Commit Log Analysis During our manual validation on the code reuse and re-implementation candidates, we noticed that some developers mentioned the rationales why they performed such operations. Thus, we also considered these commit messages as supplementary information.

This work makes the following contributions:

- We empirically analyzed a large number of concrete cases in which developers replaced their own implementation with an external library method or vice versa. The manual analysis took months to complete and we released our manually curated dataset publicly to benefit other researchers: https://github.com/swatlab/reuse_reimpl.
- We qualitatively investigated the reasons behind the library reuse and re-implementation phenomena. We found that developers prefer to reuse well-maintained, tested, and easy-to-use code. However, they may switch from code reuse to self-implementations if the reused code is only a small part of the third-party library, deprecated, or involves complicated dependencies.
- We made suggestions for improving the current code recommendation techniques, which should be tailored according to users' preferences, detect external code that is similar to a part of users' code (to avoid duplicate/re-implementation), group similar recommendations for developers to compare and select the one they prefer, as well as avoid recommending code from poor quality libraries.

The remainder of this paper is structured as follows. In Section 2, we describe the design of our empirical study. In Section 3, we present the results of our study. In Section 4.2, we discuss the threats to validity of our study. In Section 5, we discuss related works. In Section 6, we conclude this paper and discuss about future works.

2 Case Study Design

The main goal of this study is to understand why developers switch from their self-implemented code to an external library with the same functionality and the other way around. In practice, developers often reuse a whole library method and/or re-implement an existing method. Thus, in this paper, we detect library reuses and re-implementations at the method level. In the following subsections, we elaborate the design of our case study, including the data collection process, the detection approaches for the above phenomena, and the design of our surveys, which are used to address our research questions. Figure 1 shows an overview of our data collection and analysis approaches.

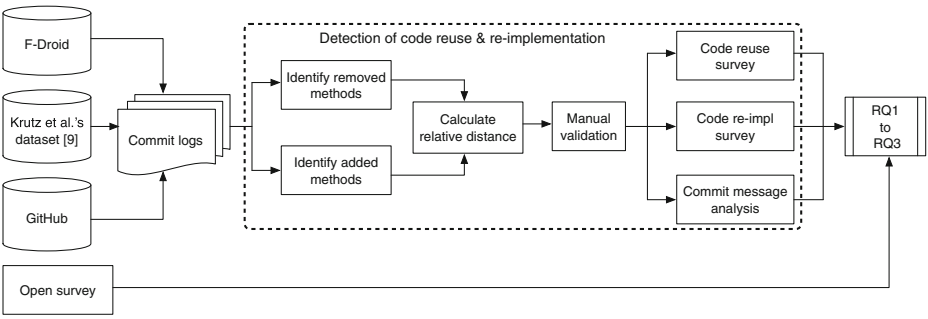


Fig. 1 Overview of our data collection and analysis approaches

2.1 Data Collection

We chose two representative programming languages: Java and Python. Java is a representative for statically-typed language, while Python is a representative for dynamically-typed language. Both languages are extensively used for software development and possess a large developer base. We believe that developers of libraries written in either of these languages can provide us insights in understanding the phenomena of library reuse and re-implementation. To study Java applications, we collect data from Android repositories that were maintained on F-Droid[9] as of August 2017 and that were used by Krutz et al. (2015). In this work, we only considered repositories that are version controlled by Git. After removing redundant repositories in both F-Droid and Krutz et al.'s datasets, we obtained 1,732 unique Android repositories. To study Python applications, we wrote a crawler to collect repositories from GitHub using the GitHub API.[10] To avoid toy projects (i.e., projects that are self-developed, unoriginal, or have a very short history), we referred to Abdalkareem et al.'s work (2017) to filter our subject repositories based on the following criteria: selected repositories must be mainly written in Python, were not forked, contain at least 20 commits, and were developed by at least two developers. In the end, we obtained 4,461 unique Python repository.

2.2 Detection of Library Reuse

We assume that if a pair of removed and added method invocations is located "close" to each other (i.e., there are zero or only a few lines between the two methods in a patch), it is likely to be the case that a developer replaced her own method with an external library method. We refer to this case as *library reuse*. In the rest of this section, we elaborate more on our detection steps.

2.2.1 Identification of Removed Methods

For each studied application, we cloned its Git repository. Then, we used the `git show` command to extract the patch of each commit, from which we identified whether there is any method implementation that was removed and the invocation of the method was also

[9]F-Droid, https://f-droid.org/
[10]Github API, https://developer.github.com/v3

removed. To detect a removed method, we first used the following regular expressions to look for any method declarations in the removed lines (i.e., lines starting with "-") in the patch. In case of Java, we used the following regular expression:

```
(?:(?:public|private|protected|static|final|native|
synchronized|abstract|transient)+\s)+(?:[\$_\w\<\>\[\]]*)\s+
([\$_\w]+)\(([^\)]*)\)?\s*\{?[^\}]*\}?
```

In case of Python, we used the following regular expression.

```
def\s+?(\w+?)\s*\((.+)
```

Python developers may write a "method" without a class, which is called a function. A developer can reuse or re-implement code either in a method or a function. In this paper, we do not specifically distinguish between methods or functions in Python because they have the same effect and will not affect our case study results. When we mention a "method" in Python, it may also mean a function. Libraries in Python are called modules. To simplify our expressions, we refer to Python modules as "libraries" in the rest of the paper.

For Android applications, if a removed method declaration is located at line L_{dcl} (and we assume the method ends at line $L_{dcl}+N$ where $N > 1$), to decide whether the whole method implementation was removed as well, we matched curly bracket pairs (i.e., {}) in L_{dcl} and its subsequent removed lines (i.e., $[L_{dcl}+1, L_{dcl}+N] \mid N \geq 1$). Once each of the left curly brackets (from L_{dcl} to $L_{dcl}+N$) can be matched to a corresponding right curly bracket, we consider that the whole method implementation is removed. If the right counterparts of some left curly brackets have not been found and we meet a non removed line (context line i.e., line starting with a white space, or added line i.e., line starting with "+"), we consider that this method is not fully removed. For Python applications, if a removed method declaration is located at line L_{dcl}, we look whether the consecutive removed lines followed by L_{dcl} (i.e., $[L_{dcl}+1, L_{dcl}+N] \mid N > 1$) have more indentation than L_{dcl}. If yes, and if $L_{dcl}+N+1$ is not an added line, we consider that the method is fully removed. If $L_{dcl}+N+1$ is an added line, and if it has less indentation than L_{dcl}, we also consider that the method is fully removed.

For the fully removed methods, we then examined whether their corresponding invocations (method invocations with the same method name and number of parameters) were also removed. If so, we save the line numbers of the removed invocations into a set Set_{del}. We will later manually validate whether each of these removed invocations corresponds to a completely removed method, which will be described in Section 2.2.4.

Example 1 In the commit `9d0ca05` of the *FBReader* project,[11] a method (`getString()` of Class `HtmlToStringReader`) was fully deleted from lines 124 to 126 from the old revision of the file *HtmlToString.java*[12] (as shown in Fig. 2). Also, the invocation of this method was removed at line 69 (as shown in Fig. 3).

2.2.2 Identification of Imported Methods

For Android apps, we identified newly imported classes by looking for this pattern (`import external.library.class;`) from the added lines (i.e., lines starting with

[11] https://github.com/geometer/FBReaderJ/commit/9d0ca05

[12] https://github.com/geometer/FBReaderJ/commit/9d0ca05#diff-111c3f193c58d04aed7c19db835db11b

```
124     -       public String getString() {
125     -           return new String(myBuffer.toString().trim.toCharArrya());
126     -       }
```

Fig. 2 The removed implementation of the method `getString()` in the commit `9d0ca05` of FBReader

"+") in the patch. For Python apps, we considered all module importing patterns mentioned in PythonModule (2018) to identify newly imported classes from the added lines, i.e.,

```
import (.+?) as (.+)
import (.+?)
from (.+?) import .+ as (.+)
from .+? import (.+?)
```

Then, for each imported class, we sought for the invocation of the class's static method (method directly invoked by the class) and its instance method (method invoked by an instantiated object of the class). We saved the line number of the added method invocations into a set Set_{add}.

Example 2 In the same file of Example 1, an external class (`android.text.Html`) was imported at line 22 in the new revision. A static method of the class (`result = Html.fromHtml(new String(contentArray)).toString();`) was invoked at line 62 (as shown in Fig. 3).

2.2.3 Calculation of Relative Distance

To decide whether a pair of removed and added methods is reasonably close to each other, we calculate its "relative distance" as follows. For each unique pair of L_{del} ($\in Set_{del}$) and L_{add} ($\in Set_{add}$), we designed the following heuristic to calculate their *relative distance*.

In the *unified diff format* (GNU 2017) (which is the default output of the `git show` command), if one line is replaced by another line, the patch will output a removed line followed by an added line.

```
- // the old method invocation
+ // the new method invocation
```

Given a pair of removed (L_{del}) and added (L_{add}) lines in commit C, we first calculated the position of L_{del} in its block of consecutive deleted lines ($Block_{del}$) as well as the position of L_{add} in its block of consecutive added lines ($Block_{add}$). If L_{del} is the i^{th} line in

```
 22  +       import android.text.Html;
     ...
     ...
69   -       result = myHtmlToStringReader.getString();
 62  +       result = Html.fromHtml(new String(contentArray)).toString();
```

Fig. 3 The removed invocation of the method `getString()` and the added invocation of the method `toString()` in the commit `9d0ca05` of FBReader

$Block_{del}$ and L_{add} is the j^{th} line in $Block_{add}$, we calculated the relative distance between L_{del} and L_{add} as:

$$Dist_{relative} = j - i + Lines_{inbetween}$$

Where $Lines_{inbetween}$ denotes the number of lines that are between $Block_{del}$ and $Block_{add}$ but do not belong to the two code blocks. Ideally, developers should replace their own method invocation with an external one at the same place. However, they may sometimes remove comments, white space, or log printing lines after L_{del} or add these kinds of lines before L_{add}. Thus, even in case that L_{add} cannot be perfectly matched to L_{del}'s position (i.e., $Dist_{relative} = 0$), they can still be a pair of the replacement from a self-implemented method to a library method. There is a trade-off between the detection's precision and recall performance when choosing different values of the threshold $Dist_{relative}$. Larger relative distance can yield more candidates, but the precision will be relatively lower (which will also increase the difficulty of our manual validation, see Section 2.2.4); while smaller relative distance can achieve a higher precision, but may miss certain good candidates. To evaluate the sensitivity of the relative distance value, we set $Dist_{relative}$ as 5, 10, and 15, respectively. We found that when this value is 10 or 15, not much new results were detected, but the number of false positives increases significantly. Thus, we choose to set the threshold for the relative distance $Dist_{relative}$ to 5. In case that developers removed a white space or comment line (L_{white}) prior to the deleted method invocation, and added the replacement library method before L_{white}, $Dist_{relative}$ would be negative. To successfully detect these cases, we adjust our criterion of the relative distance as $|Dist_{relative}| < 5$.

In this heuristic, we did not directly compare the line number of L_{del} in C's parent ($C\hat{}$) with the number of L_{add} in C because the code above these lines may be heavily changed, which can result in a large offset between L_{del} in $C\hat{}$ and L_{add} in C.

Example 3 In Examples 1 and 2, the removed invocation is the second line in a consecutive deletion block (i.e., $i = 2$), while the added invocation is the first line in a consecutive addition block (i.e., $j = 1$). Since there is no line between the deletion and addition blocks ($Lines_{inbetween} = 0$), we calculate the relative distance as: $Dist_{relative} = 1 - 2 + 0 = -1$.

2.2.4 Manual Validation

Following the above steps, we detected a total of 19,221 pairs of Android library reuse candidates and 40,927 pairs of Python library reuse candidates. For each candidate, we outputted its commit ID, line numbers of the pair of deleted and added method invocations, as well as the fully qualified class to which the added method belongs to. Then, we manually removed the candidates where the class of the added method belongs to the current application (app). We could not automate this step because the name of the app may not be contained in its class package names. For example, in the commit `5159070`[13] of the *chanu* app, the `FileUtils.copyStream` method was replaced by the `IoUtils.copyStream` method, which belongs to `com.nostra13.universalimageloader.utils.IoUtils`. Although the new class's package name does not contain "chanu", we found that it was also implemented by chanu's developers. After this preliminary filtering, we retained 391 pairs of Android candidates and 167 pairs of Python candidates.

[13] https://github.com/grzegorznittner/chanu/commit/5159070#diff-015d116ababf2863b74874b6ba078cfeR365

To further filter out false positives, two of the authors manually examined the remaining candidates separately. For a given candidate, the two authors (1) read the commit message and checked whether the committer mentioned that their own method was replaced by an external method (this is not a necessary condition but it can help us confirm the correctness of a candidate); (2) verified whether the removed method was implemented by one of the project developers and whether the added method was implemented by external developers; (3) semantically compared the functionalities between the removed and added methods (we only include the results where the removed added methods have an identical functionality). To identify the ownership (i.e., who wrote a specific piece of code) of a removed or added method, the best way would be to directly ask the developers themselves. However, this is not feasible because there are too many subject methods and only a few developers may answer this question. Instead, in Step 2, for a given removed method in a commit, we read the commit message and checked whether developers mentioned that the method is self-implemented or taken from an external source. If the ownership cannot be determined, we searched for the commit in which the method was introduced for the first time. We read the commit message and checked whether developers mentioned that the method was copied from another project. We also read the source code to find organization information and searched the method on the Internet, checking whether the method is similar to code written in past projects. For example, if the namespace (or naming pattern) of the method is different from the whole project, we will investigate where the method was originally from. If we could not find any evidence showing that the method was taken from an external source, we considered the method to be self-implemented. For an added method, besides the aforementioned checks, we also checked the name of the package the method belongs to. If the name does not follow the naming style of other packages in the project, we performed an online search and checked whether the package is taken from another project. Example 4 shows how we performed our manual validation.

After labeling cases that satisfy the conditions 2 and 3 above, the authors compared their results. They discussed on each discrepancy until a consensus was achieved. They also removed all duplicate cases in a commit (e.g., in the commit `6752f2d` of the *Twidere-Android* app, we found 36 identical cases where a customized method used to convert string to digit was replaced by an external method for the same purpose). Finally, for Android, we obtained 128 cases of library reuse, which were performed by 79 developers and are distributed in 71 apps; for Python, we obtained 65 cases of library reuse, which were performed by 60 developers and are distributed in 62 apps.

Example 4 In the commit message of Example 3, the committer mentioned: "*android.text.Html instead of own html parser*". The removed and added methods are both used to parse an HTML string. In addition, the class of the added method is from an Android official API,[14] which was not implemented by developers of the *FBReader* app.

2.3 Detection of Re-implementations

Similar to the heuristic we used in Section 2.2, we assume that in an app, if there is a pair of removed and added method invocations located "close" to each other, where the removed method was imported from elsewhere and the added method is implemented by a developer

[14] https://developer.android.com/reference/android/text/Html.html

of the app, it is likely to be the case that the developer replaced an external library method by her own implementation. We refer to this case as *library re-implementation*.

We used the same approach as described in Section 2.2 to identify such pairs of removed and added methods. In a commit, for any newly implemented method, we saved its line number into the set Set_{add}. For any removed method invocation, if the library the method belongs to is also removed, we saved the line number of the invocation and the library the method belongs to, into the set Set_{del}. From the detected results, we calculated the relative distance (see Section 2.2.3) between each unique pair of deleted and added methods, i.e., the relative distance between any L_{del} ($\in Set_{del}$) and any L_{add} ($\in Set_{add}$). We used the same threshold to filter candidates: $|Dist_{relative}| < 5$.

We detected a total of 2,835 pairs of Android candidates and 43,823 pairs of Python candidates on library re-implementation. We performed a manual validation on these candidates. For each candidate, we outputted its commit ID, line numbers of the pair of deleted and added method invocations, as well as the fully qualified class to which the removed method belongs. We first manually removed the candidates where the class of the removed method belongs to the current app. As a result, 83 pairs of Android candidates and 73 pairs of Python candidates remained. As we have mentioned in Section 2.2.4, not all incorrect candidates can be eliminated automatically because some self-implemented classes cannot be simply identified from their names, e.g., *isoparser* can either be a library on PyPI,[15] or a self-implemented class. For each of the remaining candidates, two of the authors performed a manual inspection separately with the following steps: (1) read the commit message and check whether the committer mentioned that they removed an external method and implemented an equivalent one themselves; (2) verify whether the added method was implemented by one of the project authors and whether the removed method was implemented by external developers; (3) semantically compare the functionalities between the added and removed methods (to understand the functionality of the removed method, we may perform an online search). They resolved any discrepancies through an in-person discussion on each pair and then removed duplicate cases. Finally, for Android, we found 34 cases of library re-implementation, which were performed by 32 developers and are distributed in 30 apps; for Python, we found 48 cases of library re-implementation, which were performed by 47 developers and are distributed in 45 apps.

Example 5 Figure 4 shows an example in the commit `5d1e7e8` of the *Impeller* project.[16] The invocation of the method `setUrlDrawable`, which belongs to the class `com.koushikdutta.urlimageviewhelper.UrlImageViewHelper`, was removed from the file `src/eu/e43/impeller/ActivityAdapter.java`. This method was invoked at line 145 in the old revision. In the same file, a method `setImage` was invoked instead at line 143 in the new revision. This method was newly implemented in the file `src/eu/e43/impeller/ImageLoader.java`. Through a manual inspection, we found that both `setUrlDrawable` and `setImage` are used for loading an image into an object. The removed class belongs to another Android library,[17] which was not implemented by developers of *Impeller*. We also learnt the motivation of

[15] https://pypi.python.org/pypi/isoparser
[16] https://github.com/erincandescent/Impeller/commit/5d1e7e8
[17] https://github.com/koush/UrlImageViewHelper

```
33    -   import com.koushikdutta.urlimageviewhelper.UrlImageViewHelper;
      ...
      ...
145   -   UrlImageViewHelper.setUrlDrawable(image,
              getImage(json.getJSONObject("actor")));
143   +   m_ctx.getImageLoader().setImage(image,
              getImage(json.getJSONObject("actor")));
```

Fig. 4 The removal of a package import and `setUrlDrawable()` method invocation as well as the addition of `setImage()` method invocation in commit `5d1e7e8` of Impeller

this change from the commit message: *"change from UrlImageViewHelper to a custom implementation ..."*. Thus, we believe that this is a valid case of library re-implementation.

2.4 Survey

We now provide detailed information about the surveys that were conducted.

2.4.1 Survey on Library Reuse

To understand why developers used an external library to replace their self-implemented methods, we designed a survey on Google Forms and distributed it (via emails) to the developers who performed the library reuse detected in Section 2.2. We encouraged these developers to answer the questions in a free-form text (except for Questions 3 and 7 presented below). At the same time, we also provided the surveyed developers with a few answer options for some questions (where they could make multiple choices). To mitigate biases, we randomly generated the order of the options for each multiple choice question. Thus, our participants may not receive the options with the same order as shown below. Before asking questions, we showed each participant the code snippet(s) where she replaced her own implemented method by an external library method. The questions asked in our survey are as follows.

1. *What is (are) the reason (s) why you did not use the library method in the first place?* This is a required question, for which, we provided the following options to our participants:

   – I did not know how to use this library method (or I found that the library method was hard to use).
   – I was not aware of this library when I implemented the code.
   – The required library method had not been introduced yet at the moment of my implementation.
   – Other.

   This question along with Question 2 can provide us direct reasons why developers switched from their own implementation to an external library. If library reuse is developers' ultimate purpose, the answers can provide us with ideas that could help prevent from such a "switch", which can save developers' time and efforts. We selected these options because Kawrykow and Robillard (2009) and Sun et al. (2011) indicated that developers may not reuse existing libraries because they are not aware of them. Sun et al. (2011) argued that the lack of familiarity with relevant libraries would also lead

developers to re-implement existing code. We encouraged participants to provide other possible reasons in a free-form text.

2. *Why did you replace your code with this library method?*
   This is a required question, for which, we provided the following options:

   – Because I want to have a more efficient implementation.
   – Because the library method is more reliable.
   – I want my code to be more easily tested.
   – I want to maintain my code more easily.
   – Other.

   These options are inspired by the results of Abdalkareem et al.'s study (2017) on the code reuse of JavaScript packages. They found that developers tend to believe that open source libraries are well implemented, tested, reliable, and easy to maintain. We encouraged participants to provide other possible reasons in a free-form text.
3. *Do you actively search for library reuse opportunities (i.e., code that can be replaced by library methods)?*
   This is a required question, for which we only allow a binary answer (i.e., Yes or No). If the answer is "Yes", we then ask the participant Question 4; otherwise, we jump to Question 6. This question, along with Questions 4 to 6, can let us know whether developers performed a search for library reuse at the early stage of their development, by which means they did such a search, otherwise, why they did not actively search for library reuse.
4. *When do you start looking for library reuse opportunities?*
   Participants can answer this question in a free-form text.
5. *How do you perform such search?*
   Participants can answer this question in a free-form text. Then, we ask them Question 7.
6. *Why don't you search such opportunities?*
   Participants can answer this question in a free-form text, after which we ask them Question 7.
7. *Do you find it challenging to look for library reuse opportunities?*
   This is a required question, for which we only allow a binary answer (i.e., Yes or No). The answer to this question can help us understand whether a better code recommendation approach is needed.
8. *Which criteria do you consider when replacing a piece of your own code with a corresponding library implementation?*
   Participants can answer this question with free-form text. The answer to this question can help us understand developers' requirements when reusing code, thereby help in improving the current library recommendation strategies.

2.4.2 Survey on Library Re-implementation

To investigate the reasons why developers gave up on an existing external library and choose to implement their own method, we designed another survey targeted to the developers who performed such operations. We first showed our participants the code snippets of the library re-implementation that they made. We then asked them the following questions. As in the survey of library reuse, we encouraged our participants to answer the questions in a free-form text (except for Question 3, which requires a binary answer). For some questions, we provided our participants with multiple-choice options, which are mostly inspired from

previous studies. For this survey, we also randomly generated the order of the options for each multiple choice question in order to mitigate any potential biases.

1. *What is(are) the reason(s) why you use the library method in the first place?*
   This is a required question, for which we provided these options to our participants:
   Because I thought that

   – this library was easy to use.
   – this library was well tested.
   – this library was well maintained.
   – this library had a good performance.
   – using this library can increase our productivity.
   – the license of this library was compatible with my project.
   – other.

   Although we expect that some developers prefer reusing code, this question alongside Question 2 can provide us with the reasons why developers did the opposite. The answers may point us to the weaknesses of the current libraries and provide ideas to improve the current code recommendation systems. We select these options based on two previous studies. Piccioni et al. (2013) found that usability is an important factor that developers consider when choosing a library, e.g., accurate and complete documentation. In addition, Abdalkareem et al. (2017) observed that developers prefer libraries that have good testability, maintainability, performance, and license compatibility. We encouraged participants to provide other possible reasons in a free-form text.
2. *Why did you replace this library method with the self-implemented method?*
   This is a required question, for which we provided these options:
   Because I need to:

   – increase the security level.
   – improve performance.
   – replace this deprecated library.
   – fix incompatibilities induced by this library method during the evolution of my project.
   – reduce the size of my project (or a simpler solution).
   – reduce the dependency overhead involved by this library method.
   – make my code more flexible.
   – avoid license issues.
   – other.

   These options are mostly inspired by the study of Abdalkareem et al. (2017), who found that developers are often worried about some potential weaknesses in their imported libraries, such as security, performance, dependency overhead, and license issues. Moreover, Kawrykow and Robillard (2009) indicated that "APIs sometimes evolve in a backward-compatible fashion, without any element being annotated as deprecated". Also, to reduce the size and increase the flexibility of a project, developers may choose to give up on an existing library method and implement an equivalent method themselves (Blog of Jos de Jong 2017). We encouraged participants to provide other possible reasons in a free-form text.
3. *Did the above self-implemented code meet your expectation?*
   This is a required question, which only accepts a binary answer (Yes/No). We assume that some developers may want to avoid problems, such as complex dependencies

(when they choose to reimplement a library method). Once there is a new library that better fit their requirements, they might perform library reuse again.

4. *Under what circumstances would you choose an external library method rather than implement one by yourself?*
   We ask this and the subsequent question because it can help to improve the current library recommendation systems if we understand the circumstances when developers switch from an external method to their own implementation and the other way around. Participants can answer this question in a free-form text.
5. *Under what circumstances would you choose to implement a method by yourself rather than seek an external library?*
   Participants can answer this question in a free-form text.

2.4.3 Open Survey

Our library reuse and re-implementations surveys target 207 Android and Python developers (after removing duplicate ones). According to Singer et al.'s study (2008), the response rate in questionnaire-based software engineering surveys is rather low, i.e., around 5%. To obtain more opinions from the development communities, we also designed an open survey, which is based on the surveys described in Sections 2.4.1 and 2.4.2 with some additional questions as shown below:

- **Background questions (all questions are required)**:
  1. Educational attainment
  2. Preferred programming language
  3. Role in project
  4. Software development experience (time period in years)
- **Preliminary questions on library reuse:**
  1. Do you think that replacing a self-implemented code with a library method is a common phenomenon in development?
  2. Have you ever replaced a self-implemented code with a library method?
- **Library reuse questions:**
  If participants answer yes to the preliminary question #2, we will ask them the following questions:
  1. Why didn't you use the library method in the first place? (participants can answer this in a free-form text)
  2. Did any of the following factors play a role in your decision to not use the library method? (we provided the same options as Question 1 in Section 2.4.1)
  3. Why did you replace your code with this library method? (participants can answer this in a free-form text)
  4. Did any of the following factors play a role in your decision to replace your code with this library method? (we provided the same options as Question 2 in Section 2.4.1)
- **Preliminary questions on library re-implementation (all questions are required):**
  1. Do you think that replacing an external library method with your own code is a common phenomenon in development?
  2. Have you ever replaced an external library method with your own code?

– **Library re-implementation questions:**
  If participants answer yes to the above preliminary question #2, we will ask them the following questions:

  1. What is(are) the reason(s) why you used the library method in the first place? (participants can answer this in a free-form text)
  2. Did any of the following factors play a role in your decision to use the library in the first place? (we provided the same options as Question 1 in Section 2.4.2)
  3. Why did you replace this library method with the self-implemented method? (participants can answer this in a free-form text)
  4. Did any of the following factors play a role in your decision to replace this library method with the self-implemented method? (we provided the same options as Question 2 in Section 2.4.2)

– **General questions (not required):**
  We ask the same questions as Questions 4 and 5 in Section 2.4.2.

We published the open survey on some development online communities, such as Python community at Reddit,[18] Android community at Reddit,[19] Developer community at Reddit.[20] We also invited some of our colleagues, who work as software engineering researchers or developers, to participate in this survey.

Since the library reuse and re-implementation surveys will be sent to individual developers, to distinguish them from the open survey, in the rest of the paper, we will refer to each of them respectively as "individual survey" and "open survey".

2.4.4 Analysis on Survey Responses

In general, there are two types of questions in our surveys, multiple-choice questions and open-ended questions. For answering the multiple-choice question, we analyzed the distribution of answers. For open-ended questions, we applied card sorting to interpret the answers. In detail, two of the authors independently aggregated similar answers, and then extracted key sentences from them. Next, we discussed together to condense the answers into key findings. Finally, we summarized all findings and discuss their implications for practitioners and researchers. For the latter, we compare our findings with the capabilities of the state-of-the-art tools (e.g., library recommendation tools) and recommend desired features for future work.

2.5 Analysis on Commit Messages

In our manual validation on the library reuse and re-implementation candidates, from some commit messages, we read the motivations why the code authors performed these operations. These commit messages can be used as supplementary information for our analysis since not all developers will answer our survey questions.

We extracted commit messages from each of the validated library reuse and re-implementation cases. Two of the authors independently classified the motivations extracted from the commits. One commit may contain more than one motivation, such as to improve

[18] https://www.reddit.com/r/Python
[19] https://www.reddit.com/r/Android
[20] https://www.reddit.com/r/developer

reliability and performance. For the commits where we cannot extract any useful information related to this study, we put them in the category "unknown".

We then compared our classification results. We discussed on each discrepancy until reaching an agreement.

3 Case Study Results

From the library reuse surveys, we received 15 responses out of the 79 contacted Android developers (i.e., response rate: 19%); and 21 responses out of the 60 contacted Python developers (i.e., response rate: 35%). From the code re-implementation surveys, we received 4 responses out of the 31 contacted Android developers (i.e., response rate: 12.9%); and 9 responses out of the 40 contacted Python developers (i.e., response rate: 22.5%). Most of our participants answered all the non-required questions in free-form texts. For library reuse, only 3 out of 15 Android developers (20%) and 2 out of 21 Python developers (9.5%) did not answer these questions. For code re-implementation, only 1 out of 4 Android developers (25%) and 1 out of 9 Python developers (11.1%) did not answer these questions.

In addition, we received 56 responses from the open survey. Table 1 shows the background information of our survey participants. Based on the responses, most of our participants (83.9%) have received higher education. Python (42.9%), Java (14.3%), PHP (10.7%), and JavaScript (8.9%) are their most favorite programming languages. Most participants are working as developers (73.2%) and a few of them are working as project managers, architects, and algorithm engineers (17.9%). Regarding the development experience, most participants have more than 3 years of experience, 32.1% of them have worked for more than 5 years, 12.5% have worked for 4-5 years, and 17.9% have worked for 3-4 years.

Figure 5 shows answers of the preliminary questions of the open survey. 69.6% of the participants think that replacing a self-implemented code with a library method is a common phenomenon, and 83.9% of them acknowledge this phenomenon in practice. Regarding the phenomenon where developers replace an external library method with their self-implemented code, only 39.3% of the participants think that this is common in development. However, 76.8% of them acknowledge that they have performed this in practice.

In the rest of this section, we will show the other results obtained from the individual and open surveys, and discuss their implications in addressing our three research questions. As aforementioned, since not all contacted developers participated in our surveys, we also use the extracted commit messages as additional source of information for our analysis. Since there are some overlapped questions between the two kinds of surveys, in the rest of this section, we will combine the results for the identical questions.

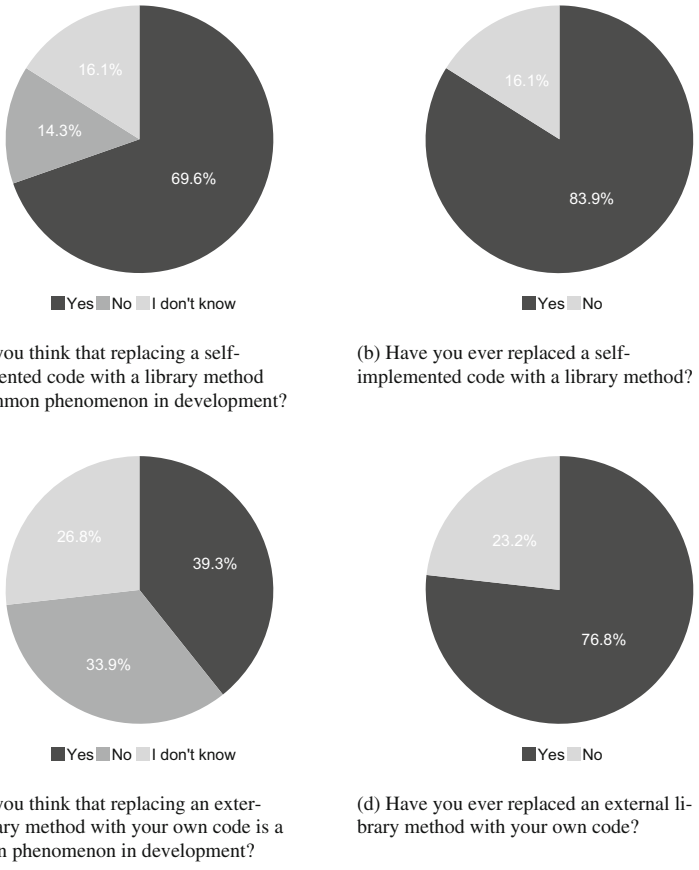
3.1 (RQ1) Why do Developers Replace Their Self-Implemented Method with an External Library Method?

Table 2 shows the options chosen by our participants for the first three questions in the library reuse survey (refer to Section 2.4.1). Our first research question investigates the reason why developers did not use a library method in the first place but use it later to replace their own code. 46% of the participants vote that they were not aware of that method; implying that developers would not reinvent the wheel if they know a library that serves their purpose. This reason is particularly voted by Android (47%) and open (50%) survey participants, while only 30% Python participants vote for this reason. Although we expect

Table 1 Answers to background questions of the open survey

(a) Education attainment		
Education	Number	Percentage
Bachelor	2	3.6%
Master	4	7.1%
Doctorate	20	35.7%
University without degree	12	21.4%
Secondary school	11	19.6%
Professional degree	3	5.4%
Prefer not to answer	4	7.1%
(b) Preferred programming language		
Language	Number	Percentage
Python	24	42.9%
Java	8	14.3%
PHP	6	10.7%
JavaScript	5	8.9%
C#	6	10.7%
C	1	1.8%
C++	1	1.8%
Swift	2	3.6%
Other	3	5.4%
(c) Role in project		
Role	Number	Percentage
Developer	41	73.2%
Project manager	3	5.4%
Architect	3	5.4%
Algorithm engineer	4	7.1%
Other	5	8.9%
(d) Development experience		
Experience	Number	Percentage
Less than a year	3	5.4%
1-2 years	8	14.3%
2-3 years	10	17.9%
3-4 years	10	17.9%
4-5 years	7	12.5%
More than 5 years	18	32.1%

that Android developers often program with an IDE (which may come with a code recommendation system), they still have difficulties to find an appropriate library. Many of the current code recommendation techniques, such as Thung et al. (2013a), make recommendations based on the relationships of existing libraries in a project. Few of these techniques can semantically understand developers' need about their ongoing code and none of them can actively seek for appropriate libraries online (i.e., in order to minimize the chance of missing any useful libraries). Designers and researchers of future code recommendation systems should realize these problems and improve their techniques along these directions. In addition, 14% developers acknowledge that they did not know how to use the library in



(a) Do you think that replacing a self-implemented code with a library method is a common phenomenon in development?

(b) Have you ever replaced a self-implemented code with a library method?

(c) Do you think that replacing an external library method with your own code is a common phenomenon in development?

(d) Have you ever replaced an external library method with your own code?

Fig. 5 Answers of the preliminary questions of the open survey

the first place. Some participants further explain that *"the library was badly documented"*. This result suggests that library vendors should improve the readability of their documentation. Furthermore, many current library recommendation techniques, such as Thung (2016) and Thung et al. (2013b), rely on text analysis. These techniques cannot work well with badly or non-documented libraries. Better approaches, such as semantic source code analysis, need to be proposed. In addition, 28% participants said that the required library method has not been introduced at that time. As discussed above, if a code recommendation system can actively look for appropriate libraries online, once such libraries are available, the system can recommend them to the developers; allowing them to switch early from using their own implementations to reusing code before their project becoming overly complex. From comments of the participants, we learned other reasons as follows: evolution of the project (*"quick prototyping"*, *"the required complexity climbed"*, *"required functionality was simple in the first phase of development"*, *"there was no need when the original method was created"*) and work transfer from one developer to another (*"(the) original code was implemented before I joined the project"*, *"this code was already introduced when I initially started working on the project ... I realized it (the self-implemented code) could be removed with a function provided by the Android APIs instead"*).

Table 2 Answers to Questions 1-3 of the library reuse survey (for each question, we calculated the percentage of the answers to a specific option over the total number of answers)

Reason	Android	Python	Open
(a) What is(are) the reason(s) why you did not use the library method in the first place?			
I did not know how to use it	0	1 (1%)	15 (13%)
I was not aware of it	8 (7%)	7 (6%)	38 (33%)
It had not been introduced	6 (5%)	8 (7%)	18 (16%)
Other	3 (3%)	7 (6%)	5 (4%)
(b) Why did you replace your code with this library method?			
Efficiency	4 (3%)	4 (3%)	25 (18%)
Reliability	2 (1%)	10 (7%)	23 (16%)
Testability	4 (3%)	5 (4%)	21 (15%)
Maintainability	12 (9%)	15 (11%)	0
Other	3 (2%)	5 (4%)	7 (5%)
(c) Do you actively search for library reuse opportunities?			
Yes	11 (13%)	17 (20%)	37 (45%)
No	4 (5%)	4 (5%)	10 (12%)

Regarding why developers replaced their own implementation with a library method, our provided options, improving reliability, development efficiency, testability, and maintainability, received equally important votes (i.e., 25%, 24%, 22%, and 20% votes respectively). This result is inline with the finding of Abdalkareem et al. (2017). Although nobody directly voted for "having a better maintainability" in the open survey, some participants left comments in-line with this reason: *"more elegant code"*, *"my implementation is hard to maintain"*, *"... better class readability and less code to maintain"*, etc. By analyzing participants' comments, we observed other reasons: improving security (*"for security consideration"*), performance (*"see if I could achieve better performance"*), obtaining additional features (*"library method sometimes does more"*, *"it (the library) was more robust and feature complete"*), permission or license issues (*"the new method doesn't require RECORD_AUDIO permission, and the need for that had a frequent complaint from users"*). In addition, some developers trust external libraries more than their own code: *"I think the developer who can publish (this) library must be more senior than me"*, *"my code lacks verification"*, *"... (the library) is peer reviewed"*, *"the library is of higher quality"*, *"(it) depends, if the one is a common (library) and from a well known organization (I) will use it"*. This result suggests that library recommendation systems should extract and show reviews, quality assessment, and organization information to developers, and prioritize code written by well-reputed organizations.

From the result of both individual and open surveys, we find that most participants have actively sought for library reuse opportunities. Some participants look for such opportunities more proactively (*"at the beginning of a new feature"*, *"all the time"*), others wait until their code is too complex (*"the pure implementation starts getting hard to manage"*, *"whenever it feels like what I'm doing could be part of a separate project"*), when they face problems (*"when bugs appear"*, *"when my implementation is becoming a mess"*), or when they realize that someone else has implemented the same functionality (*"when the implementation feels like somebody should already have written that"*). Developers may

also conditionally seek for library reuse, when *"... (having had) a clear picture in mind how and in which direction the project will evolve"* or *"... (doing) complex repetitive, boring tasks"*. Moreover, we notice some interesting reasons why developers do not actively seek for library reuse. Some developers only want to have a challenge (*"I think I can do it"*). Some developers do not want to increase dependency complexity (*"depending on third parties is more work for simple things"*). Some developers do not need to seek library reuse for a small-scale project (*"the project is a spare-time project, and I don't have spare time to do such code-maintenance activities unless essential to immediate progress."*, *"if it ain't broke, don't fix it? This is a hobby project in minimal maintenance mode"*). Some others have confidence in themselves and/or do not prioritize the practice of library reuse (*"I didn't think about it in the first place"*, *"I know when I need a library and in this case I will look for one. I will not scan my code thinking about which part can be replaced by library code."*). However, all of the developers eventually replaced their own code with a library method; indicating that library recommendation would be helpful even if developers did not think so initially.

21 out of 36 participants, who answered Question 6 of the individual survey, do not think that searching for library reuse opportunities is challenging. Regarding the way of searching for a library reuse opportunity, using general-purpose search engines (especially Google) is the first choice for 19 out of 22 participants who answered Question 5 of the individual survey. 11 participants searched from general code bases or forums (including GitHub, GitLab, StackOverflow, and Hack News[21]). 4 participants searched from language specific websites (including Android Arsenal[22] and PyPI). 2 participants searched from the documents of Android and Python standard library. Nobody has mentioned the use of any library or code recommendation tool. Only one participant used her newsletters to find library reuse opportunities. Although library recommendation techniques have been proposed and improved for many years, in this work, we do not find an empirical evidence to support the fact that developers have used these techniques to successfully find libraries they need. There is still a gap between the practices of code search and recommendation.

In the last question of the library reuse survey, we ignored some vague answers such as "good code quality", and learned the criteria that developers use to replace their own code with an external library code. 22 out of 34 participants look whether the library is well maintained and tested (*"well maintained"*, *"well tested"*, *"whether it is actively developed"*, *"update cadence e.g., how many commit in the last 6 months"*). 9 participants look for the reputation of the library (*"popular library"*, *"respected developers"*, *"widely adopted"*, *"used by other projects"*, *"exposure on Stack Overflow"*). 4 participants look for the readability of code and documentation (*"documentation"*, *"readability"*, *"clearness of code"*). 4 participants look for the stability (*"does it have a stable API?"*). 3 look for the size or complexity of the library (*"complexity of code"*, *"conciseness"*). 3 look for license compatibility (*"license clauses of library"*, *"forkability"*). 2 look for the difficulty to integrate the library into their project (*"it needs to fit in to my existing API and be close to a net zero code change"*, *"does it require only few changes to be integrated in my project?"*).

From the above results, we learned that, when taking decisions to replace a self-implemented method with an equivalent library method, criteria may vary according to circumstances. A tutorial video, which is recommended by a participant, on "designing and evaluating reusable components" (YouTube video 2004) can be useful when taking such

[21] https://news.ycombinator.com
[22] https://android-arsenal.com

decisions. To summarize this, we would like to cite a participant's comment: *"Is out implementation, out of all potential implementations out there, worth keeping, and will getting rid of ours for a more maintained/supported version be worth it. In other words, if we get rid of our implementation, but the cost is adding 3-4+ dependencies to use a different one, it may not be worth it; it needs to be evaluated on a case-by-case basis."*

Figure 6 shows the motivations that we identify from the commit messages where library reuse occurred. We ignored 132 out of the 170 messages, in which we cannot identify any useful information regarding the motivation of switching to library reuse. All of the identified motivations are either expected when we designed the survey or mentioned by survey participants as well. In general, this result is consistent with our observations from the survey result.

> *In general, developers replace their self-implemented method with an external library method because they were initially not aware of the library or the library had not been introduced. After realizing that there is a well maintained and tested library that meets their requirement, they later used the library method to replace their own implementation.*

3.2 (RQ2) Why do Developers Replace an External Library Method with Their Self-Implemented Code?

Table 3 shows the options chosen by our participants for the first two questions in the library re-implementation survey (refer to Section 2.4.2). As we only receive 4 responses from the Android survey, in the following analyses, we will not discuss the reason why Android developers do not choose some of the options.

Our participants used an external library method in the first place because they think the library is easy to use (25%), can increase development productivity (19%), and is well maintained (18%). Although the library method was eventually discarded, this result implies that an easy-to-use library can attract developers to adopt it; if a library is hard to use, developers may find it easier to implement the library's functionalities. Therefore, if library vendors expect their code to be well-adopted, ease-of-use would be an important criterion to take into account. From participants' comments, we learned other reasons, i.e., *"it is the recommended method for Android"*, *"I was mistakenly under the impression it was part of the Python standard library"*).

Regarding why developers replaced the first adopted library method and switch to implement their own method, reducing dependency (21%), improving flexibility (19%), and having a simpler solution (18%), are voted as the three most popular reasons. Some participants further explained: *"FYI super old project, we wanted to reduce dependencies when*

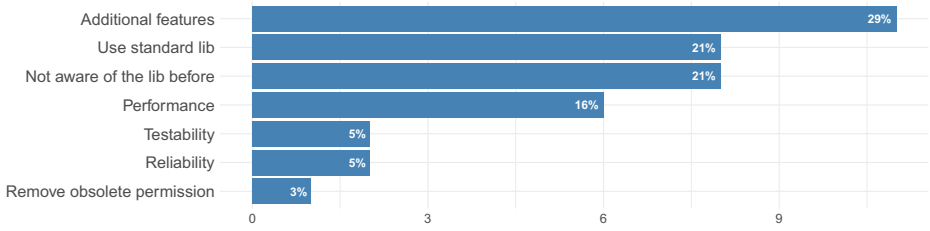


Fig. 6 Motivations identified from the messages of the commits where library reuse occurred

Table 3 Answers to Questions 1 and 2 of the library re-implementation survey (for each question, we calculated the percentage of the answers to a specific option over the total number of answers)

Reason	Android	Python	Open
(a) What is(are) the reason(s) why you use the library method in the first place?			
Testability	1 (1%)	4 (2%)	19 (12%)
License compatibility	0	2 (1%)	15 (9%)
Usability	1 (1%)	5 (3%)	34 (21%)
Performance	0	1 (1%)	18 (11%)
Productivity	0	3 (2%)	28 (17%)
Maintainability	1 (1%)	3 (2%)	24 (15%)
Other	2 (1%)	2 (1%)	1 (1%)
(b) Why did you replace this library method with a self-implemented method?			
Better performance	0	3 (2%)	16 (11%)
Reduce dependency	0	4 (3%)	26 (18%)
Better security	0	0	5 (3%)
Fix incompatibility	0	1 (1%)	12 (8%)
Simplicity	0	3 (2%)	23 (16%)
Avoid license issues	0	0	8 (5%)
Replace deprecated lib.	0	1 (1%)	12 (8%)
Flexibility	2 (1%)	1 (1%)	25 (17%)
Other	2 (1%)	2 (1%)	0

possible", *"(I) didn't need to wrestle a 800 pound gorilla to do a simple few things"*, *"I only needed one function so I didn't want to have a full library"*, *"down the road, the client needed more specific features of which the library did not provide nor expose"*, *"(I) can't easily refactor across library boundaries"*. Library vendors should also make their products more flexible and easy to modify without introducing too much complexity in the configuration process. Moreover, given two libraries that provide a similar functionality, many developers are likely to prefer the light-weighted one. Thus, library vendors should take this into account when making and maintaining their products. Moreover, from the comments, we realize that *"bug in library"* can also make developers switch to their own implementations.

12 out of 13 participants who answered Question 3 of the library re-implementation survey thought that their self-implementations meet their expectation. However, in their comments, some participants also discussed potential drawbacks of their self-implementations, including: additional implementation efforts (*"it took longer (time) to write it myself"*), additional maintenance efforts ("*if using a library, no need to maintain it*"), lower reliability (*"(the self-implementation has) the potential for introducing bugs"*), and lower performance (*"I might not be an expert on how to do thing properly. Specialized libraries will surely do better."*).

Figure 7 shows the library re-implementations' motivations that we identified from the commit messages where the re-implementations' occurred. We ignored 47 out of 81 commit messages where no useful information can be extracted about the motivation of library re-implementations. Among the identified motivations, removing dependency is mentioned most of the time. Particularly, two committers said that they removed the dependency

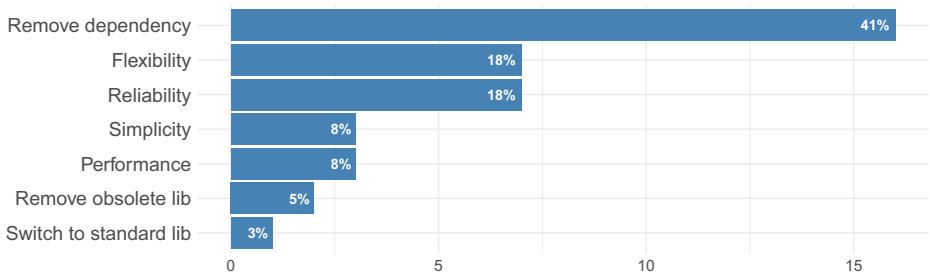


Fig. 7 Motivations identified from the messages of the commits where library re-implementations occurred

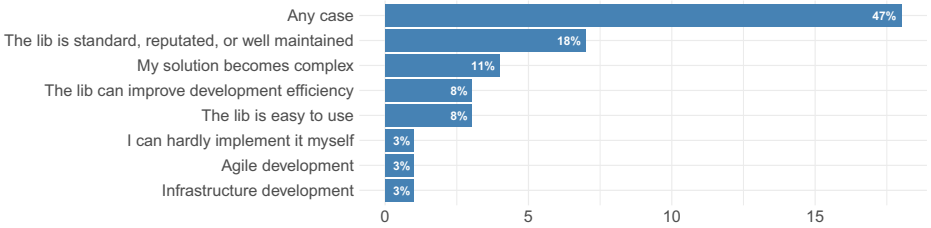
because it was *"only used once"*. Other motivations, such as making the code more flexible, more reliable, simpler, and more performant, are also mentioned in the analyzed commit messages. Similar to the survey participants' comments, removing obsolete libraries as well as implementing their own solution based on standard libraries could also be the reasons why developers discarded external libraries. In general, this result is consistent with what we obtained from the surveys.

Developers replace an external library with their own implementation because they tend to choose an easy-to-use library method in the first place. Once they realize that the used library method is only a small part of the library, and the library dependencies are too complicated, or the library method becomes deprecated, they may switch to replace the library with their own code. Library vendors should make their code flexible and lightweight.

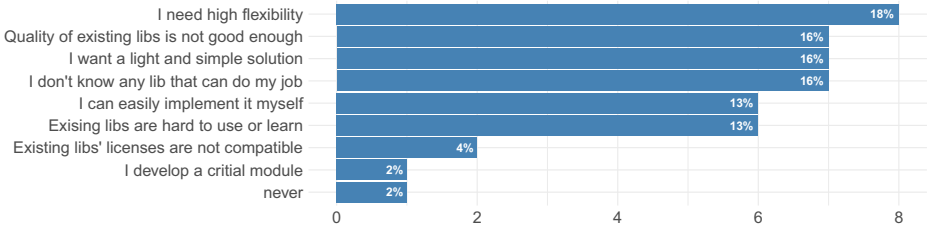
3.3 (RQ3) Under what Circumstances do Developers Prefer to Reuse or Re-implement Code?

In **RQ3**, we want to investigate developers' preference towards library reuse and library re-implementation in a more general context.

We collected and card sorted the answers of the last two questions in the library re-implementation and open surveys (refer to Sections 2.4.2 and 2.4.4). Figure 8 depicts the circumstances under which developers prefer to reuse code. Nearly half of the participants (47%) prefer to reuse an existing code in any case. A participant's answer can explain this well: *"this (reusing code) is my first choice. I don't usually self-implement something unless I'm confident that a pre-existing solution doesn't exist or a pre-existing solution doesn't suit my needs"*. For the developers who actively seek for library reuse opportunities, some of them analyze the quality (whether it is well maintained, reputable, or a part of the standard library) of a library before adopting it; some others judge whether a library is easy to use or can improve development efficiency. Even though previous studies advocated that library reuse can reduce the cost in development and maintenance, improve development efficiency and product performance (Heinemann et al. 2011; Mohagheghi et al. 2004; Basili et al. 1996; Abdalkareem et al. 2017), some developers do not consider it until after their code becomes too complex or when they can hardly implement what they want. This result suggests that not all developers seek for library reuse opportunities at all times. A library recommendation system, which suggests a qualified library satisfying developers' requirement, can help them improve development efficiency and



(a) Under which circumstances do you prefer to reuse code?



(b) Under which circumstances do you prefer to implement code yourself?

Fig. 8 Answers to the general questions

avoid reinventing the wheel. From the answers of this question, we also learned that some developers prefer library reuse, especially in the case of agile development or infrastructure development.

Regarding the circumstances under which developers prefer to implement code themselves, "requiring higher flexibility of code" is mentioned most of the time. Indeed, publicly available libraries are designed for general developers. If a developer expects to fully customize a functionality, she may have to implement it herself. Another strong reason that pushes developers to implement their own code is when they cannot find any library that satisfies their requirement. If a functionality is easy to implement, such as a *"quick and dirty work"* mentioned by a participant, developers do not need to reuse code. Under these circumstances, library reuse does not have much advantage. Moreover, we also observed that developers prefer implementing code themselves because they find that existing libraries are not easy-to-use and/or understand, not simple or lightweight enough, or have incompatible license with their project. To deal with these problems, code recommendation systems can recommend more than one library, showing their characteristics (e.g., size, dependencies, user rating, license, team information), and allow developers to choose the most appropriate one.

Half of our survey participants prefer to reuse code at all times. However, developers may also want to make their own implementations if they need a higher code flexibility.

4 Discussion

The following subsections summarize our findings and their implications to practitioners and researchers, and acknowledge some threats to validity.

4.1 Implications

In general, we observed that developers replaced their self-implemented method with an external library method because they were initially unaware of the library or the library had not been introduced. After realizing that there is a well-maintained and tested library that meets their requirement, they later used the library method to replace their self-implemented method. Moreover, developers replaced an external library method with their self-implemented method because they tend to choose an easy-to-use library method in the first place. Once they realize that the used library method is only a small part of the library, the library dependencies are too complicated, or the library method is deprecated, they may replace the library method with their self-implemented method.

According to our results, if library vendors want their product to be more widely used, we suggest them to improve library documentation, make the library easy-to-use, and reduce the size and complexity of the library. An IDE with library recommendation systems can help developers to seek library reuse opportunities, thereby preventing them from re-inventing the wheel. For any industrial project, developers should always actively seek library reuse opportunities, especially if their self-implemented code are becoming overly complex to maintain. Another suggestion to developers is that either decision (reusing code or implementing it themselves) should be carefully considered at the beginning of project development. Once a piece of code is deeply integrated and interacted with other parts of the project, the cost of replacing it would be tremendous.

Moreover, code recommendation systems can also help developers find and adopt code or library they need. A number of code recommendation techniques have been proposed in the literature (see Section 5.1 for details). The current code recommendation techniques often make recommendations based on the relationships of existing libraries in a project, library usage history, and some semantic features. Comparing our findings with capabilities of these tools we find that such tools do not help in the following circumstances:

1. In our library reuse survey, nearly half of our participants acknowledged that they were not aware of the library they eventually adopted at the time of their implementations. In practice, recommendation tools require the capability to search for solutions in the Internet. Such capability would minimize the chance of missing any libraries that might be useful for developers.
2. Developers may want to reuse a library they have used or reused in another project, which needs a recommendation tool that can record and analyze developers code usage, reuse or programming preference.
3. According to our survey results, before applying an external library, developers often want to know whether the library is produced by well-reputed team, whether it is well documented, easy-to-use, and flexible enough. In addition, developers do not explicitly receive the characteristics of the library from the recommendation results. However, most current code recommendation systems do not group similar recommendations for developers to compare and select the one they prefer.
4. The current recommendation systems only yield “positive” results but developers may also want to be aware of “negative” results, i.e., the libraries they should not use.
5. The current recommendation systems do not consider open source license compatibilities, which cannot help developers to avoid license violations.

Based on the above weaknesses, we suggest that the current library recommendation techniques could be improved from the following aspects (which we believe to be interesting directions for future work):

- *Tailored recommendation*: Since developers may have their own preference of reusing libraries, the system can study and collect users' preference before giving them suggestions. For example, the system can analyze all external libraries used by a developer in her current and past projects, use machine-learning algorithms to classify these libraries according to the domain or requirement of this developer, and use these information to make better recommendations.
- *Detection of similar solution*: The system can search for a piece of code that has similar functionality to a part of the project (e.g., a method, class, module). An early suggestion of library reuse potential can prevent developers from reinventing the wheel. Earlier approaches (e.g., Kawrykow and Robillard (2009) and Sun et al. (2011)) have already been capable of detecting re-implementation of a piece of library code if the library has already been used in the project. However, developers might also want to prevent re-implementations of the code from unused libraries. In such case, semantic analysis and clone detection techniques can help to search similar code snippets. A deep-learning based framework introduced by Wei and Li (2017) can potentially be leveraged to achieve this.
- *Grouped recommendations*: The system can group similar recommendations for developers to compare and select the library they prefer. In such a group, the system can further rank the the recommendations based on their number of users, reviews, and documentation quality.
- *Display of libraries' characteristics*: This can help to quickly assess the quality of a library. As aforementioned, when there are multiple candidates, the system can also use this information to rank the recommendations. For example, when a developer wants to install a plugin from Eclipse Marketplace, a summary of the plugin will be provided. Likewise, when a developer imports a new external library, the recommendation system can popup a summary window, showing characteristics of the library.
- *Disrecommendation*: As we have learned, replacing a deprecated or inactive library is one of the reasons why developers switched from library reuse to reimplementation. If an imported library is deprecated, obsolete, badly rated, or inactively maintained, a library recommendation tool may want to suggest developers to replace the library with an alternative library. For example, a disrecommendation system can scan all imported external libraries and connect to the libraries' website, checking whether any library is deprecated or out of maintenance. If so, the system will warn developers to avoid this library and provide detailed reasons on why they should do so. A future recommendation systems may also predict deprecation or future issues (more generally) with some libraries and pro-actively recommend alternatives.
- *License compatibility suggestion*: To help software organizations avoid license violations, library recommendations tools can also detect the license of the recommended library, comparing it with the license of developers' home project, checking whether the recommended library can be legally imported.

Based on our findings, we provide several concrete improvements to existing works in Section 5.3. Furthermore, to allow replication and verification of our study, a replication package is publicly available to interested researchers.[23] Moreover, we point out several

[23] Replication package, https://github.com/XBWer/Why-Reinventing-the-Wheel.

directions to extend our study. First, the state-of-the-art clone detection tool can potentially be applied to identify more instances of library reuse and re-implementation. Second, more diverse data sources can be considered, such as GitLab, BitBucket, and SourceForge. Third, more types of program languages can be analyzed, such as JavaScript which is the most commonly used programming language at the time of writing.[24]

4.2 Threats to Validity

There are several threats that may potentially affect the validity of our study. In this section, we discuss the threats to validity of our study by following the guidelines for case study research (Yin 2002).

Threats to construct validity are concerned with the relationship between theory and observation. We designed some heuristics to detect real world cases of code reuse and code re-implementation. However, the heuristics cannot detect all possible cases. For example, a developer could potentially replace her self-implemented code with a method from an already imported library in the project. Unfortunately, she did not realize that the method of the library can fulfill her requirement in the first place. Our heuristics did not cover the above case because it will yield a lot of false positives, which will require a lot of time to validate. Our current heuristics alone costed several months of validation. Moreover, the goal of this study is not to find all possible cases of code reuse and code re-implementation. Instead, we aim to provide empirical evidences on these two phenomena. Although researchers, such as Kawrykow and Robillard (2009) and Sun et al. (2011), have discussed these phenomena in previous works, nobody has shown any real world example. We aim to understand the reasons why developers replaced their self-implemented code with an external library and the other way around by collecting real world examples.

Threats to internal validity are concerned with the factors that may affect a dependent variable and were not considered in the study. In our surveys, we provided options for participants to answer some of the questions. These options are inspired by previous studies, such as Abdalkareem et al. (2017) and Piccioni et al. (2013). However, to mitigate biases led by these pre-defined options, we always encouraged our participants to use their own words to answer the questions. As a result, we obtained some valuable information from the answers in the free-form text, which were not pre-defined within the options. Our surveys received a higher response rate than the average rate in software engineering research surveys (Singer et al. 2008). One of the reasons is that our survey invitation provides some information that are specific to the target survey respondent (including their name, project name, target commit, target lines of code, and how the library reuse/re-implementations were performed). This specific information increases the chance of contacted developers responding to our email compared to emails with only generic contents. Another reason is that we sent a reminder to developers if we did not receive their response after a week, and another one after a month if we still did not receive their response.

Threats to conclusion validity are concerned with the relationship between the treatment and the outcome. This threat mainly derives from our manual validation of code reuse and code re-implementation. During this process, we need to identify whether an added method comes from a third-party library or was implemented by developers themselves.

[24] Stack Overflow Survey, https://insights.stackoverflow.com/survey/2019#technology.

In order to minimize this threat, two of the authors independently validated each of the cases detected by the heuristics. They then compared their results and resolved each of the conflicts. The whole process took several months. Through individual surveys, a portion of our detected cases was confirmed by developers. However, we can hardly guarantee the correctness of other validated cases. For example, a developer may copy code from a library to her project and later replaced the copied code with another library code. Thus, none of the code was implemented by the developer herself. On the other hand, we *card sorted* some textual information, such as the free-form text answers from surveys or commit messages. The card sorting classification results were verified and discussed between the authors. However, as any other taxonomic studies, we cannot guarantee a 100% accuracy on our classification results. We publish our classification result along with the analyzed commit messages online: https://github.com/swatlab/reuse_reimpl. Due to privacy reasons, we cannot publish all the details of our survey answers. Future replications are welcome to validate our work.

Threats to external validity are concerned with the generalizability of our results. In this work, we studied code reuse and re-implementation phenomena in two programming languages: Java and Python. We mined data from 1,732 Android repositories and 4,461 Python repositories. Java and Python are representatives for statically-typed and dynamically-typed languages, respectively. Both languages are popular for software development. Nevertheless, replicating our work for other programming languages (such as C++, C#, and JavaScript) is required to broaden our understanding of the phenomenon. There are 109 developers participating in our survey. This number is as large as many prior studies that also performed surveys to better understand a certain software engineering phenomenon (Abdalkareem et al. 2017; Lethbridge 2000; Iivari 1996). Still, our survey respondents' feedback may not represent the opinions of all developers. We do not view our work as a one-off work, but one of many to fully understand library reuse and re-implementation. We welcome future studies to extend and/or replicate our study with different participants and datasets. In this study, we only considered code reuse and re-implementation from libraries because, according to the rapid growth of OSS libraries in the recent years, we believe that library reuse is one of the main ways in which developers reuse code. Still, the reasons why developers reuse code from other sources, such as frameworks or knowledge sharing platforms (e.g., Stack Overflow), are also worthy for investigation. We encourage researchers to investigate this direction in the future.

5 Related Work

5.1 Library Reuse

Library reuse has been researched since the 90s. In 1992, Krueger (1992) surveyed different approaches to software reuse and provided several insights to library reuse. First, the author claimed that the major challenge to implement large libraries of reusable components is to find concise abstractions. Better abstraction can improve the reuse rate. Although we focus on open source applications in our work, our results show that *removing dependency* is the most important reason behind developers' decision to replace a library with a self-implementation. Second, the author mentioned that library implementor must provide specifications that succinctly describe component behavior. It corroborates our conclusion that *display of libraries' characteristics* can help to quickly assess the quality of a library. Third, another challenge of library reuse is that developer must take time to study and

understand how to use the library. We also found that the *ease to use* is the main reason behind developers' decision to use an external library method in the first place. Different from Krueger's work, Kim and Stohr (1998) surveyed software reuse in practice and provided several technical and non-technical factors that need to be considered. First, the cost of developing and maintaining reusable libraries are considered as an investment during the software development. This issue has been well addressed because of the rapid development of open source libraries nowadays. Our study shows that *no need to develop and maintain* becomes an important factor that motivate library reuse. Second, the nature of program languages are related to library reuse. For example, the advent of Java provides a new and potentially source of reusable software resources making it possible to create distributed object-oriented applications that function independently of particular operating systems or hardware platforms. In this sense, Java can better support the concept of widespread software reusability. Research works in the 90s mainly discussed internal library reuse, i.e., developers need to design, develop and maintain libraries by themselves. Thus, the corresponding costs are considered. However, we focus on the external library reuse, i.e., developers use third-party library and no need to develop and maintain by themselves.

In the recent decade, with the rapid growth of open source software (OSS), many studies have shown that library reuse is a very common practice in many different programming communities (e.g., Java (Heinemann et al. 2011), JavaScript (Abdalkareem et al. 2017), and Android (Ruiz et al. 2012; Mojica et al. 2014)). However, only a few of works paid attention on the reason behind library reuse. Emerging package management platforms, such as Node Package Manager (*NPM*), are introduced to facilitate code sharing. Abdalkareem et al. (2017) analyzed more than 230,000 *NPM* packages and 38,000 JavaScript applications. They observed that trivial package reuse is common and is increasing in popularity in the Node.js community. They conducted a survey with 88 Node.js developers and observed that trivial packages are widely used because developers assumed these packages to be well implemented and tested. To empirically verify this assumption, they validated the most cited reasons and drawbacks on the trivial package reuse. They found that only 45% of the studied trivial packages contain test code, despite the fact that trivial packages were expected to be "deployment tested". Additionally, they found that 12% of the studied trivial packages have more than 20 dependencies. Hence, developers should be careful in choosing to use trivial packages.

Our study complements those studies in several ways. Firstly, all of the above related works either only focused on one programming language or too general while we investigated multiple popular program languages with different naturals (i.e., Java and Python). Second, although a few of the above related works investigated why developers reuse an external library to replace their self-implemented code, none of them collect real world cases and surveyed developers for the reasons behind. In this work, we focused on the instances where developers initially self-implement a piece of code and then replace the code by using a third-party library. We manually identified the cases of library reuse in two different programming languages and utilized a qualitative analysis to understand why developers do not use a library in the first place and what challenges they encounter when choosing to use a library. Additionally, we also conducted an open survey to investigate the main factors that influence developers' decision on whether they should reuse a library or not.

5.2 Code Re-implementation

Nowadays, library reuse is a common practice. However, libraries were not always used by developers. In particular, developers at times re-implement the behavior of an existing library.

Kawrykow and Robillard (2009) assumed some reasons for code re-implementation, e.g., developers are not familiar with the library, they are not aware of all the functionalities, or they got lost in a huge collection of APIs. The authors argued that imitating API code represents an ineffective usage of libraries as such re-implementation is not necessary, and the existence of imitated codes creates maintenance burden. To detect cases of code re-implementation, they developed a technique which extends code similarity detection techniques with new matching relations between abstractions of the code re-implementation and library methods. 405 actual cases of potentially suboptimal API usage are detected within 10 open source Java systems. The overall precision of the approach is 31% and the average per-system precision is 21%. To improve the accuracy of Kawrykow et al.'s approach (Kawrykow and Robillard 2009), Sun et al. (2011) proposed a graph-based approach to detect code re-implementations. They used trace subsumption relation of data dependency graphs to characterize the similarity between self-implemented code and library code. Their approach detected 313 code re-implementation cases with higher average precision, i.e., 82%, for the same dataset.

Above works are based on the assumption that code re-implementation happens because developers did not find suitable library or API to reuse. However, there was no study that has empirically investigated the reasons why developers re-invent the wheel. To fill this gap, we detected cases where developers replace a library code with its equivalent self-implemented code and then surveyed the corresponding developers to understand their reasons in doing so. We find that developers replace an external library with their own implementation because they tend to choose an easy-to-use library method in the first place. Once they realize that the reused code is only a small part of the library, the library's dependencies are too complicated, or the library becomes deprecated, they may switch to replace the library with their own code.

5.3 Library Recommendation

Nowadays, a large amount of code is available to be downloaded and used, e.g., third-party libraries with APIs. However, developers are often unaware of suitable code to be used for their projects and might miss these opportunities. Code recommendation techniques are introduced to alleviate this problem.

Recommend a Code for a Given Project Thung et al. (2013a) proposed an approach *LibRec* to automatically recommends libraries to developers for a particular project. *LibRec* takes as input a set of libraries that a project currently uses, and recommends other libraries that are likely to be relevant. *LibRec* combines association rule mining and collaborative filtering. The association rule mining component extracts libraries that are commonly used together and then rates each of the libraries based on their likelihood to appear together with the currently used libraries. The collaborative filtering component works on the assumption that similar projects are likely to share similar third-party libraries and then rates each of the libraries based on how many of the top-N most similar projects use it. Based on our findings, *LibRec* can be improved in several ways. First, recommending libraries by simply considering used libraries may not be sufficient. This is especially true at the beginning of a project. At that time, developers may not be aware of many usable libraries for their project. Thus, only a limited number of libraries are likely to be used. Corresponding to our second suggestion (*detection of similar solution*), we should not only consider libraries that has been used in the project, but we should also consider existing self-implementation to better prevent developers from reinventing the wheel.

Different than *LibRec*, Ouni et al. (2017) proposed a search-based approach *LibFinder* to recommend potentially useful libraries. They consider the library recommendation problem as a multi-objective optimization problem. A multi-objective search-based algorithm is applied to find a trade-off among three objectives : 1) maximizing co-usage between a candidate library and the actual libraries used by a given system, 2) maximizing the semantic similarity between a candidate library and the source code of the system, and 3) minimizing the number of recommended libraries. It is worthwhile to mention that *LibFinder* achieves a better performance by detecting the semantic similarity between a library and the code of the system, which is consistent with our second suggestion (*detection of similar solution*). However, *LibFinder* can still be improved by performing a deeper analysis on the development preference i.e., corresponding to our first suggestion (*tailored recommendation*). For example, if most of the libraries used in the project are developed for large data processing, it indicates that the project need to handle large data. Thus, the recommended libraries should also be equipped with such capability.

Recommend a Code for a Given Query Rahman et al. (2016) proposed an API recommendation approach *RACK* that recommends a list of relevant APIs for a given natural language query by leveraging the crowdsourced knowledge in Stack Overflow. They found that Stack Overflow might be a potential source for code search keywords and APIs. At least two APIs are used in each of the accepted answers in Stack Overflow, and about 65% of the API classes from the core packages are used in those answers. Also, titles from Stack Overflow's questions are a major source for code search keywords. Based on above findings, they proposed a two-step approach: (a) construct token-API mapping database, and (b) recommend relevant APIs for a search query. In step (a), they extracted tokens in a question's title and map the APIs in the corresponding accepted answer. In step (b), they employed two heuristics (i.e., Keyword-API Co-occurrence and Keyword-Keyword Coherence) to collect candidate APIs given a query and then used two metrics (i.e., API Likelihood and API Coherence) to estimate the relevance of the candidate APIs for the given query. Lastly, a ranked list of the candidates are obtained and top-K APIs from the list are returned for recommendation. Different from Rahman et al.'s work, Gu et al. (2016) proposed a deep learning based approach *DeepAPI* to recommend API usage sequences for a given natural language query. *DeepAPI* adapts a neural language model named RNN Encoder-Decoder. It encodes a word sequence (i.e., user query) into a fixed-length context vector and generates an API sequence based on the context vector. They also augmented the RNN Encoder-Decoder by considering the importance of individual APIs. The advantages of *DeepAPI* is that it does not rely on information retrieval techniques, which makes it different from other code search techniques (e.g., McMillan et al. (2011) and Lv et al. (2015)). Based on our findings, both *RACK* and *DeepAPI* can be improved by profiling library, i.e., corresponding to our fourth suggestion (*display of APIs' characteristics*). *RACK* considers all APIs mentioned in 172,043 Stack Overflow questions and *DeepAPI* collects APIs from 442,928 Java projects from GitHub without any further filtering based on characteristics. However, the developers' opinion towards APIs can be collected from Stack Overflow or other API review boards, e.g., Uddin and Khomh (2017). We believe that a display of APIs' characteristics can help to quickly assess the quality of APIs. Gao et al. (2015) studied the problem of recommending suitable APIs that satisfy users' need for mashup creation. They proposed a manifold ranking framework for API recommendation. First, they categorized existing mashups into functionally similar clusters. Then, they recommended APIs for each mashup cluster using manifold ranking algorithm. Three factors are taken into consideration: (1) APIs that are in functionally similar mashups, (2) popularity of APIs, and (3) similarity

between APIs. Different than *RACK* and *DeepAPI*, *APIs' popularity* is considered. To some extent, it supports our suggestion that analysis of libraries' (or APIs') characteristics can improve reuse rate. In summary, our suggestions outline five potential directions to further improve existing code recommendation systems.

6 Conclusion

In this work, we explored the reasons behind two opposite developer behaviors, i.e., library reuse and code re-implementation. To achieve this goal, we identified real world instances from multiple sources and then performed two types of surveys, i.e., individual survey and open survey. Moreover, we also performed a manual qualitative analysis on commit logs as a supplement. Our experiment results show that, the reason why developer replace their self-implemented method with an external library method is mainly because they were initially not aware of the library or the library had not been introduced. Once they find a well maintained and tested library that meets their requirement, they reuse it. The reasons why developer re-implement code by themselves are mainly because the used library method is only a small part of the library, the library dependencies are too complicated, or the library method becomes deprecated. Besides, we also provided five aspects that could be helpful to improve the current code recommendation systems. In the future, we plan to further improve existing library recommendation approaches by taking into consideration the multiple factors that we discovered in this work, e.g., usability, complexity of the external code. In addition, we will also investigate whether the state-of-the-art clone detection tools are able to detect similarity between self-implemented code and external code.

References

Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? an empirical case study on npm. In: 11th joint meeting on foundations of software engineering. ACM, pp 385–395

Basili VR, Briand LC, Melo WL (1996) How reuse influences productivity in object-oriented systems. Commun ACM 39(10):104–116

Blog of Jos de Jong (2017) The art of creating simple but flexible APIs. http://josdejong.com/blog/2014/10/18/the-art-of-creating-simple-but-flexible-apis/, online. Accessed 14 Nov 2017

Gao W, Chen L, Wu J, Gao H (2015) Manifold-learning based api recommendation for mashup creation. In: 22nd IEEE international conference on web services. IEEE, pp 432–439

GNU (2017) Unified diff format. http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html, online. Accessed 14 Sept 2017

Griss ML (1993) Software reuse: From library to factory. IBM Syst J 32(4):548–566

Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: 24th international symposium on foundations of software engineering. ACM, pp 631–642

Heinemann L, Deissenboeck F, Gleirscher M, Hummel B, Irlbeck M (2011) On the extent and nature of software reuse in open source java projects. In: 13th international conference on software reuse. Springer, pp 207–222

Iivari J (1996) Why are case tools not used? Commun ACM 39(10):94–103

Kawrykow D, Robillard MP (2009) Improving api usage through automatic detection of redundant code. In: 24th international conference on automated software engineering. IEEE, pp 111–122

Kim Y, Stohr EA (1998) Software reuse: survey and research directions. J Manag Inf Syst 14(4):113–147
Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183
Krutz DE, Mirakhorli M, Malachowsky SA, Ruiz A, Peterson J, Filipski A, Smith J (2015) A dataset of open-source android applications. In: 12th working conference on mining software repositories. IEEE, pp 522–525
Lethbridge TC (2000) Priorities for the education and training of software engineers. J Syst Softw 53(1): 53–71
Lv F, Zhang H, Lou JQ, Wang S, Zhang D, Zhao J (2015) Codehow: Effective code search based on api understanding and extended boolean model (e). In: 30th international conference on automated software engineering. IEEE, pp 260–270
McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: finding relevant functions and their usage. In: 33rd international conference on software engineering. ACM, pp 111–120
Mohagheghi P, Conradi R, Killi OM, Schwarz H (2004) An empirical study of software reuse vs. defect-density and stability. In: 26th international conference on software engineering. IEEE Computer Society, pp 282–292
Mojica IJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE (2014) A large-scale empirical study on software reuse in mobile apps. IEEE Softw 31(2):78–86
Nguyen AT, Hilton M, Codoban M, Nguyen HA, Mast L, Rademacher E, Nguyen TN, Dig D (2016) Api code recommendation using statistical learning from fine-grained changes. In: 24th international symposium on foundations of software engineering. ACM, pp 511–522
Ouni A, Kula RG, Kessentini M, Ishio T, German DM, Inoue K (2017) Search-based software library recommendation using multi-objective optimization. Inf Softw Technol 83:55–75
Piccioni M, Furia CA, Meyer B (2013) An empirical study of api usability. In: 7th international symposium on empirical software engineering and measurement. IEEE, pp 5–14
PythonModule (2018) Python official documentation on modules. https://docs.python.org/2/tutorial/modules.html, online. Accessed 29 March 2018
Rahman MM, Roy CK, Lo D (2016) Rack: Automatic api recommendation using crowdsourced knowledge. In: 23rd international conference on software analysis, evolution, and reengineering, vol 1. IEEE, pp 349–359
Ruiz IJM, Nagappan M, Adams B, Hassan AE (2012) Understanding reuse in the android market. In: 20th international conference on program comprehension. IEEE, pp 113–122
Singer J, Sim SE, Lethbridge TC (2008) Software engineering data collection for field studies. In: Guide to advanced empirical software engineering. Springer, pp 9–34
Sun C, Khoo SC, Zhang SJ (2011) Graph-based detection of library api imitations. In: 27th IEEE international conference on software maintenance. IEEE, pp 183–192
Thung F (2016) Api recommendation system for software development. In: 31st international conference on automated software engineering, pp 896–899
Thung F, Lo D, Lawall J (2013a) Automated library recommendation. In: 20th working conference on reverse engineering. IEEE, pp 182–191
Thung F, Wang S, Lo D, Lawall J (2013b) Automatic recommendation of api methods from feature requests. In: 28th international conference on automated software engineering. IEEE Press, pp 290–300
Uddin G, Khomh F (2017) Automatic summarization of api reviews. In: 2017 32nd IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 159–170
Wei H, Li M (2017) Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: 26th international joint conference on artificial intelligence, pp 3034–3040
Yin RK (2002) Case study research: design and methods - Third Edition, 3rd edn. SAGE Publications
YouTube video (2004) Designing and evaluating reusable components. https://www.youtube.com/watch?v=ZQ5_u8Lgvyk, online. Accessed 29 March 2018
Zaimi A, Ampatzoglou A, Triantafyllidou N, Chatzigeorgiou A, Mavridis A, Chaikalis T, Deligiannis I, Sfetsos P, Stamelos I (2015) An empirical study on the reuse of third-party libraries in open-source software development. In: 7th Balkan conference on informatics conference. ACM, p 4