Don’t Reinvent the Wheel: Towards Automatic Replacement of Custom Implementations with APIs

Rosalia Tufano, Emad Aghajani, Gabriele Bavota
SEART @ Software Institute, Università della Svizzera italiana, Lugano, Switzerland
{rosalia.tufano, emad.aghajani, gabriele.bavota}@usi.ch

Abstract—Reusing code is a common practice in software development: It helps developers speedup the implementation task while also reducing the chances of introducing bugs, given the assumption that the reused code has been tested, possibly in production. Despite these benefits, opportunities for reuse are not always in plain sight and, thus, developers may miss them. We present our preliminary steps in building RETIWA, a recommender able to automatically identify custom implementations in a given project that are good candidates to be replaced by open source APIs. RETIWA relies on a “knowledge base” consisting of real examples of custom implementation-to-API replacements. In this work, we present the mining strategy we tailored to automatically and reliably extract replacements of custom implementations with APIs from open source projects. This is the first step towards building the envisioned recommender.

Index Terms—Recommender Systems, APIs

I. INTRODUCTION

Code reuse is a well-known practice aimed at improving both developers productivity and code quality [7]. There is evidence about the benefits of systematically reusing code, especially for what concerns a lower likelihood of having bugs in reused code [6], [11], [8], [12]. Such benefits arise when the reused code is not outdated [15] and follows good patterns of code reuse [10]. In such a context, Application Programming Interfaces (APIs) provide reusable functionalities that can be exploited by developers to (i) speedup the implementation of new features, and (ii) rely on well-tested implementations that have been possibly deployed in hundreds of client projects. Despite these benefits, developers may not be aware of the availability of a specific feature implementation as an API offered by a third-party library, thus missing the opportunity of reusing it. To reduce such a risk, researchers proposed techniques aimed at recommending APIs given the coding context of the developer (i.e., the code they are currently writing) [9], [14], [13]. While these tools could reduce the risk of reimplementing features offered in well-known libraries, such a scenario is still likely to happen, as also demonstrated by the results we present.

We describe our vision for RETIWA (REplacing cusTom Implementations With Apis), a recommender to identify custom implementations (i.e., code implemented from scratch) that can be replaced by third-party APIs. RETIWA is complementary to existing API recommenders [9], [14], [13]: The latter can recommend APIs while the developer is implementing. Differently, RETIWA comes into play once the custom code has been already implemented, automatically identifying it as a “clone” of a feature offered by well-known third-party APIs. This implies that RETIWA cannot save implementation effort, but can still avoid the maintenance of custom code that can be replaced with well-known and likely well-tested APIs, thus boosting code quality.

We present the overall idea behind RETIWA and the first steps we made to build it. We started building a “knowledge base” featuring real replacements of custom implementations with APIs performed by developers in open source projects. We show the challenges behind such a task and the strategies we defined to address it. Such a knowledge base allows to identify common “replacement patterns” that can be used to trigger custom implementation-to-API replacement recommendations.

II. ENVISIONED TOOL

Fig. 1 shows the main steps behind RETIWA. First, based on our preliminary tests, it is not realistic to reliably identify commits in which a custom implementation is replaced with an API without having a starting set of known APIs to support. Thus, our approach starts by mining software libraries with the goal of collecting a set of APIs (Step 1 in Fig. 1) through the Libraries miner component. The libraries to support can be defined by the user. The set of mined APIs is then provided as input to the Client projects analyzer (Step 2). The latter, given a collection of GitHub repositories, clones and analyzes the change history of the projects to identify candidate replacements where a custom implementation is replaced with one of the known APIs (Step 3). We indicate these custom implementation-to-API replacements as a → API, with a representing the lefthand side (LHS) and API the righthand side (RHS). A single commit may contain multiple replacements, i.e., several custom implementations are replaced with different APIs.

The parsing performed to identify such replacements may result in false positives. For this reason, the Replacements selector applies a set of heuristics to exclude from the dataset instances likely to represent false positives (Step 5). The instances surviving such a filter are provided as input to the Replacements clustering component (Step 6), which is in charge of grouping replacement instances characterized by the same RHS. These are code changes, possibly coming from different repositories, in which developers replaced a variety of custom implementations with the same API. The output of this step are the actual replacement rules C → API, with C being a set of custom implementations (Step 7). These rules can be used to recommend to developers the replacement of custom implementations with suitable APIs (Steps 8 and 9).
In particular, a clone detector can be used to identify, either in a git repository or directly in the IDE, a component \( p \) being similar to one of the clusters of custom implementations \( (\mathcal{C}) \).

Assuming the reliability of such a clone detector, the corresponding RHS \( (i.e., \text{the associated API}) \) can then be recommended to the developer as a replacement for \( p \).

We implemented the steps 1-6 shown in Fig. 1 by instantiating \textit{RETIWA} to the Java language and by working at method-level granularity: We look for replacement instances in the form \( m \rightarrow API \), where \( m \) is a Java method. In Section III we analyze the meaningfulness of the method-to-API replacements that \textit{RETIWA} was able to automatically identify, leaving the finalization of the recommender system (dashed part in Fig. 1) as future work. In the following, we provide details on the components we implemented and the design decisions we took.

\textbf{A. Libraries Miner}

The first component is in charge of building a database of potential APIs that our approach may suggest as replacement for an equivalent custom implementation. Libraries \textit{miner} currently supports the collection of APIs from Maven libraries.

Given a list of libraries of interest, Libraries \textit{miner} retrieves their source code by downloading and uncompressing the *-source.jar for their latest release hosted on the Maven Central Repository [2]. The set of public methods (APIs) in each library is then extracted using the Eclipse Java Parser [1]. For each identified API we store the following information: package name, the path of the file from which it has been extracted, and its method signature. Libraries \textit{miner} also extracts all packages defined in the source code of each library. Such information will be used to identify if import statements in client projects refer to any of the target libraries.

\textbf{B. Client Projects Analyzer}

This component is responsible for cloning a set of given GitHub repositories and analyzing their change history with the goal of identifying potential replacements of a custom method with one of the APIs of interest. The analysis of each client project starts by creating a linear history of commits of the repository’s default branch using the \texttt{git log <default-branch> --first-parent} command. Then, we iterate through all commits and perform the following steps on every two consecutive snapshots \( s_i \) and \( s_{i+1} \), where \( s_i \) is the system’s snapshot before the changes introduced by commit \( c_i \) and \( s_{i+1} \) is the snapshot after the changes introduced by \( c_i \).

We start by extracting all method declarations and method invocations in the \( s_i \) and \( s_{i+1} \) snapshots. The method declarations represent all internal methods \( (i.e., \text{methods actually implemented in the system under analysis, excluding those imported from external libraries}) \) existing in the system at a given snapshot. We indicate them with \( D_s \) and \( D_{s+1} \). We use instead \( I_s \) and \( I_{s+1} \) to indicate all method invocations existing in the two snapshots. These lists are extracted by checking out the corresponding snapshot \( \langle i.e., \text{git checkout <commit>} \rangle \) and parsing the obtained Java files using SrcML [4]. The main idea behind extracting these lists is to check in the following steps if the commit \( c_i \) (i) replaced all method invocations to an internal method \( m \) with invocations to a non-internal API; and (ii) deleted the implementation of the internal method \( m \), replaced by the usage of API.

To perform this check, we start by running the command \texttt{git diff s_s+1 --word-diff --unified=0 --ignore-all-space --F.java for each file F.java modified or renamed in c_i}. This version of \texttt{git diff} outputs pairs of \([-\text{oldcode-}] [+\text{newcode+}] \) snippets, with the diff algorithm trying to match newly added code fragments \( (i.e., \text{newcode}) \) with deleted code ones \( (i.e., \text{oldcode}) \) when possible. We use such a command to identify invocations to \( m \) replaced with invocations to API \( m \) in commit \( c_i \). This is done by parsing the diff output using SrcML to match replaced method invocations. A few clarifications are needed on this step. First, we only focus on files modified or renamed in \( c_i \) since we look for method invocation replacements which cannot happen in added or deleted files. Second, \texttt{git diff} relies on heuristics to match deleted and added code fragments when possible, thus being a source of imprecisions in our approach. Third, similarly, SrcML is applied to parse fragments of code in the diff output, thus again resulting in imprecisions when matching method invocations.

To exclude pairs \( m \rightarrow API \) which are not of our interest \( (e.g., API \text{ is not an actual external API}) \), we make sure that:

1. \( D_s \) contains \( m \). This ensures that the method \( m \) was an internal method declared in snapshot \( s_i \).

2. The \( m \) implementation in \( s_i \) is non-empty \( (i.e., \text{it has a body}) \) and does not invoke any method having the same exact name as \( API \) (even with a different signature). This filtering step ensures that \( m \) actually implemented something and, thus, could be a candidate to be replaced with an external API.
Also, it excludes cases in which the developers were already using the API (or a slightly different version of it taking different parameters), with \( m \) acting as a wrapper for API. We are not interested in these cases since our goal is to recommend replacements of custom methods with APIs. However, if the developers are already aware of the existence of API, there is no reason for recommending it and probably they had a reason for not using it in the first place.

(3) \( I_{s_{i+1}} \) contains API (i.e., the set of invocations in snapshot \( s_{i+1} \) must contain the invocation to the candidate API), otherwise SrcML failed to recognize a true method invocation from the newcode code fragment.

(4) There is no \( m \) declaration in \( D_{s_{i+1}} \) nor \( m \) invocation in \( I_{s_{i+1}} \). This indicates that (i) the implementation of the candidate custom method \( m \) has been deleted from the system (\( m \not\in D_{s_{i+1}} \)); and (ii) no invocations to it exist anymore (\( m \not\in I_{s_{i+1}} \)).

(5) There is no method declarations matching API in \( D_{s_{i+1}} \), otherwise API is a not an external API method.

The outcome of the aforementioned steps on commit \( c_i \) is a set of candidate method-to-API replacements in form of \( m \rightarrow \text{API} \) for which (i) \( m \) is a custom method that has been deleted in \( c_i \); and (ii) API is an external method that has been added in \( c_i \) to replace all invocations to \( m \). The Client projects analyzer also makes sure that the API is part of the libraries of interest provided as input to the Libraries miner. This is done by verifying whether at least one of the added import statements in the files in which a \( m \rightarrow \text{API} \) replacement happened matches with the list of packages extracted by the Libraries miner for the libraries of interest. If this is not the case, the corresponding candidate replacement pair is removed.

The final set of candidate replacement instances for each commit contains: (i) the GitHub repository owner/name; (ii) the commit sha; (iii) custom method information, including its signature and full implementation; (iv) information about the added API, including signature and potential libraries it belongs to based on the import statements analysis; (v) file paths in which the replacements occurred; and (vi) the total number invocation replacements occurred in that commit (i.e., the number of times an invocation to \( m \) is replaced with API).

C. Replacements Selector

This component is in charge of further filter out from the set of candidate replacement those instances likely to be false positives. Indeed, while implementing our approach, we noticed that two simple heuristics could be used to remove instances unlikely to be relevant for our goal. First, we exclude all replacement instances in which the custom method \( m \) is either a getter, a setter, or a main method. We do not see interesting scenarios in which it could make sense to recommend the replacement of these types of methods with APIs.

Second, we conjecture that the number of \( m \rightarrow \text{API} \) invocation replacements performed in a commit can be an indicator of how “reliable” is the method-to-API replacement we identified.

Indeed, if we observe that in a given instance the internal method \( m \) is removed and the several invocations to it that were present in the system are replaced with invocations to API, this (i) supports the idea that \( m \) was a sort of “utility” method invoked in different parts of the code; and (ii) reduces the chances that the replacement is the result of a parsing error in the diff. We study how applying a threshold \( t \) on the minimum number of call replacements impacts the reliability of the identified replacement instances (Section III).

III. PRELIMINARY STUDY

The goal of this study is to assess the feasibility of RETIWA in terms of the possibility to automatically identify changes in which custom implementations are replaced with APIs. We aim at answering RQ₁: To what extent is it possible to automatically identify method-to-API replacements? The automatic identification of \( m \rightarrow \text{API} \) replacements is a prerequisite for building RETIWA. RQ₁ looks at the precision of the approaches described in Section II.

A. Study Design

The Libraries miner collected 38 Apache commons libraries by leveraging the MVN repository website [2]. These libraries can be identified as those having org.apache.commons as maven group-id and have been downloaded and parsed as described in Section II-A. We used the SEART GitHub Search [5] to collect all client projects non-fork Java GitHub repositories having at least 500 commits and 10 stars. These filters have been set in an attempt to exclude toy/personal projects. The Client projects analyzer obtained 9,788 repositories as result of this search. However, only a fraction of these repositories (1,856) declared a dependency towards one of the considered libraries during their change history. These are the only projects from which we can expect useful data points for our study.

The analysis of these repositories, performed as described in Section II, resulted in 337 candidate replacements which we manually analyzed to answer RQ₁. In particular, each \( m \rightarrow \text{API} \) candidate replacement was inspected independently by two authors with the goal of classifying it as a true or false positive. To come up with such a classification, the inspector looked at (i) the diff of the commit on GitHub, (ii) a summary we created featuring all the invocations to \( m \) that were replaced with an invocation to \( API \), and (iii) the commit note. Conflicts, that arisen for 34 instances (10%), have been solved by a third author not involved in the original classification. We report the precision of the identified instances (i.e., the percentage of true positives among the candidate replacements) and discuss how it can be improved by increasing the minimum number \( t \) of call replacements (see Section II-C).

B. Results Discussion

Table I reports the results achieved as output of our manual validation. The first column shows the threshold applied as additional filtering criterion to remove replacement commits not featuring at least \( t \) call replacements.
The first row (i.e., \( t \geq 1 \)) represents the scenario in which such a filter is not applied, since all instances RETIWA identified will have by construction at least one call replacement. As it can be seen, without any additional filtering the precision of the 337 identified replacements is limited to 48.9\%. By increasing the value of the \( t \) threshold, the precision quickly increases, with a 83.8\% already achieved with \( t \geq 2 \) (i.e., at least two invocations to the custom method \( m \) have been replaced with the API in the commit). Clearly, such an increase in precision has in cost in terms of true positive replacements that are excluded as having \( t < 2 \) (165-67=98 true positives are excluded). This is the usual recall vs precision tradeoff that should be assessed based on how the envisioned recommender system will be built on top of this data. One option is also to provide developers with the possibility to decide the value of \( t \): Higher values will result in less recommendations likely to be of high quality, while lower values, especially 1, will trigger more recommendations including, however, a higher percentage of false positives. In the following we discuss three concrete examples of replacements we found, while the whole dataset is available in our replication package [3].

| \( t \) | \# Instances | \# True Positives | Precision (%) |
|---|---|---|---|
| \( \geq 1 \) | 337 | 165 | 48.9% |
| 2 | 80 | 67 | 83.8% |
| 3 | 46 | 39 | 84.8% |
| 4 | 33 | 28 | 84.8% |
| 5 | 25 | 23 | 92.0% |

This method is used in the system to check for the existence of the given String in the array as it can be seen from the replaced method call, in which it is used to return true in case indexOf returns a value != -1. The replacing API contains already returns a boolean thus also simplifying the locations in the code in which it is used instead of indexOf.

Fig. 2. Replacing the custom indexOf method with contains API

In Fig. 2 the name of the GitHub repository and the commit we refer to are shown at the top. Following in italic is the commit message used by the developer who explicitly indicates the aim of the commit of removing “home-grown routines” in favor of APIs implemented in Apache commons. In the reported example the custom method indexOf returns the index of a given String in the array provided as parameter or -1 if the array does not contain the String.

This method is used in the system to check for the existence of the given String in the array as it can be seen from the replaced method call, in which it is used to return true in case indexOf returns a value != -1. The replacing API contains already returns a boolean thus also simplifying the locations in the code in which it is used instead of indexOf.

Answer to RQ1. The automatic identification of changes replacing custom implementations with APIs is challenging but feasible. Indeed, the approach we presented was able to identify 337 of these commits with a precision of 48.9\% and, using specific filtering heuristics (e.g., \( t \geq 2 \)), the precision level can be substantially boosted to > 80\%. More research is needed to optimize the recall vs precision tradeoff.
Towards this goal, our future work will span three main directions. First, we plan to explore alternatives for a more reliable code parsing. The obvious one is to perform a full parsing of the snapshots before and after each commit to identify custom code replaced with APIs. Such a process, while precise, is extremely expensive when applied on thousands of systems (i.e., hundreds of thousands of commits to analyze).

Second, the combination of heuristics we adopt to identify replacement commits may be suboptimal and exclude several true positives. For example, RETIWA only selects as candidate replacement commits those in which a known import statement (i.e., an import statement coming from one of the parsed libraries) is added in the commit. This is based on the assumption that the API usage implies the addition of the import statement.

Third, once the identification of replacements is crystallized, we will focus on implementing the two remaining steps of RETIWA, namely the clustering and the triggering of recommendations, as described in Section II.

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 851720).

REFERENCES

[1] “Eclipse JDT Core,” https://www.eclipse.org/jdt/core/, 2022.
[2] “Maven Central Repository,” https://mvnrepository.com/, 2022.
[3] “Replication Package,” https://github.com/APIreplacement/api_replacement, 2022.
[4] “SrcML,” https://www.srcml.org/, 2022.
[5] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in github for MSR studies,” in 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021. IEEE, 2021, pp. 560–564.
[6] W. Frakes and C. Fox, “Quality improvement using a software reuse failure modes model,” IEEE Transactions on Software Engineering, vol. 22, no. 4, pp. 274–279, 1996.
[7] J. Gaffney and R. Cruickshank, “A general economics model of software reuse,” in International Conference on Software Engineering, 1992, pp. 327–337.
[8] S. Haefliger, G. von Krogh, and S. Spaeth, “Code reuse in open source software,” Management Science, vol. 54, no. 1, pp. 180–193, 2008.
[9] L. Heinemann and B. Hummel, “Recommending api methods based on identifier contexts,” in Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, ser. SUITE ’11, 2011, p. 174.
[10] C. Kasper and M. W. Godfrey, “‘cloning considered harmful’ considered harmful: patterns of cloning in software,” Empir. Softw. Eng., vol. 13, no. 6, pp. 645–692, 2008.
[11] J. Li, A. Gupta, J. Arvid, B. Borretzen, and R. Conradi, “The empirical studies on quality benefits of reusing software components,” in 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), vol. 2, 2007, pp. 399–402.
[12] P. Mohagheghi and R. Conradi, “An empirical investigation of software reuse benefits in a large telecom product,” ACM Trans. Softw. Eng. Methodol., vol. 17, no. 3, 2008.
[13] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta, “Recommending api function calls and code snippets to support software development,” IEEE Transactions on Software Engineering, 2021.
[14] F. Thung, “Api recommendation system for software development,” in 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 896–899.
[15] P. Xia, M. Matsushita, N. Yoshida, and K. Inoue, “Studying reuse of out-dated third-party code in open source projects,” Information and Media Technologies, vol. 9, no. 2, pp. 155–161, 2014.