A Novel Approach to Detect Unintentional Re-implementations

Veronika Bauer Tobias Völke
Technische Universität München, Germany
{bauerv, voelke}@in.tum.de

Elmar Jürgens
CQSE GmbH, Germany
juergens@cqse.eu

***Abstract*—Unintentional re-implementation of existing functionality is an issue frequently reported in practice and causes increased efforts for development and maintenance. However, instances are hard to find with existing approaches. For practitioners, this increases maintenance risks, such as inconsistent bug fixing, and hinders quality improvement efforts. For researchers, this hinders a reliable quantification of the issue.**

We propose a pragmatic approach combining identifier-based concept location with static analysis to detect candidate re-implementations between two sets of source code. We present initial results from applying the approach to detect re-implementations of utility functionality present in libraries within a sample of Java projects.

***Index Terms*—software reuse, API, library, software maintenance, Java, missed reuse opportunities, re-implementation**

I. Introduction

An abundance of valuable software assets is present in companies' code repositories, via Open Source libraries, and commercial component markets. Nevertheless, developers tend to re-implement existing functionality [18], [3], missing out on the benefits of reuse opportunities. Furthermore, this can result in the creation of "Simions" [17], independent re-implementations of existing functionality that do not share a common origin in terms of code. Simions have long term negative effects in the form of increased development and maintenance efforts.

Re-implementations can happen easily for various reasons: (1) the scale of development in large projects makes staying up to date with reusable entities challenging. This entails a certain amount of parallel implementation efforts. Despite the higher probability of a required functionality being available, the increasing effort for searching, understanding and adapting a reusable incites implementing functionality from scratch [3], [18]. (2) The use of established protocols might impose specific implementation steps that are duplicated [1]. (3) To achieve business goals, duplicate implementations might be necessary at times. (4) Evolution of libraries might make parts of the code obsolete [18]. (5) Low API usability prevents users from finding the implementations realizing a specific concept [26].

Discovering re-implementations is difficult in theory and practice: first, semantic equivalence checking is well studied (e.g. [7]) and a generally undecidable problem. Approaches to detect re-implementations therefore are constrained to resort to approximations. Second, previous work [9], [17] concludes that existing approaches, such as clone detection or random testing approaches [16], do not provide satisfactory results to detect re-implementations in practice.

Consequently, research so far is unable to realistically quantify the size of the problem. Practitioners, on the other hand, miss an approach providing support to avoid new and discover existing re-implementations [3]. Therefore, we need a new approach to discover missed reuse opportunities in the form of (unintentional) re-implementations.

Problem statement: Re-implementations of existing functionality happen in practice and entail negative effects, such as increased costs for development and maintenance. However, we are lacking a comprehensive approach to discover them. As a result, the extent of the phenomenon remains unclear. Furthermore, practitioners lack support to address the issue.

Contributions: This paper presents a novel approach to discover re-implementations between software libraries and a system's source code. To this end, we establish a broader definition of similarity, based on the concepts embodied in the identifiers. We implement our approach for Java systems and provide a calibrated set of parameters for it. We report on a proof of concept evaluation, detecting re-implementation of library functionality in three Java systems.

II. Related Work

Prior work has addressed cases of semantic code duplication. Our notion of re-implementations is related as follows:

Semantic clones [12] are code fragments with isomorphic program dependence graphs, and therefore structurally similar. Their behaviour can, but does not need to, be functionally similar. **Accidential clones** [1] are code fragments of different origin that are syntactically similar due to the adherence to a specific protocol. This does, however, not imply behavioural similarity. **Type-4 clones** [28], **"wide miss" clones** [20], and **Simions** [17] refer to the same phenomenon: behaviourally similar code fragments that have no common origin. Unlike cloned code, these fragments are likely to differ greatly in their structure [17].

In the following, we present approaches that aim to detect or avoid untintentional re-implementations.

A. Detecting similar implementations

Closest to our approach is the work by Marcus and Maletic [20]: they aim to interactively detect *high-level con-*

cept clones by computing the similarity of source code documents (that can be of the granularity of files or methods) and clustering of the results. The similarity is computed with Latent Semantic Indexing, LSI [8]. The clustering can be enhanced by using structural information. Determining relevant high-level concepts is done by the user. In a case study, the authors uncover simions of a list within one system. Our approach differs in scope and techniques: We aim to find simions between a corpus of libraries and one or more systems. Since the libraries determine the relevant concepts for the analysis, we need a pragmatic way to extract their key concepts from their source code. For this, we choose TF-IDF which is robust accross systems and does not require extensive tuning. Furthermore, we take use the program structure to restrict the vocabulary used by the analysis.

Al-Ekram et al. [1] report empirical findings on *accidential cloning* across software systems. Their approach detects structurally similar code fragments caused by usage patterns required by specific technologies. However, the authors state that their approach is likely to miss re-implementations that are fundamentally different in structure.

Jiang and Su [16] propose random testing to automatically mine functionally equivalent code fragments. The source code is randomly cut in chunks. Two chunks are considered equivalent if they produce the same output for the same random input data. The study reports promising results for the test systems, namely a Linux Kernel and a sorting benchmark. These systems are written in C and, due to their functionality, do not require functionality, such as string processing, which is prevalent in average systems. Deissenboeck et al. [9] found that reproducing Jiang and Su's experiment on Java code yielded insatisfactory results. Apart from challenges induced by the different requirements of the technical platform, they consider their definition of equivalence problematic: first, it does not account for side effects. This causes code fragments to be pronounced equivalent that a programmer would deem fundamentally different. Second, independently of the given input, most code chunks did not produce any output or yielded exceptions. By their definition, these chunks are equivalent. In practice, this information is of little value.

Kawrykow and Robillard [18] propose an approach to mine Java systems for methods "imitating" library methods available to these systems. Their goal is to replace functionality implemented in the client code by calls provided by the library. They abstract method bodies to program elements and perform a matching between the available library methods and the client methods. Whilst they cater to the important use case of replacing obsolete client methods by library methods, the notion of equivalence on the method level is still too restrictive for our task: the re-implementations we are looking for might be present in different code structures and would therefore be missed by the approach.

B. Detecting similar applications

Using API calls to find relevant code has been proposed and applied before, e.g. in the context of code search [6], [22], [23] and rapid prototyping [24]. However, we do not know of this technique being used to track and quantify simions in existing software systems. Despite the differing context of the work, the successful use of API calls as indicators for relevant code encourages us to exploit this idea for our goal.

Teyton et al. [30] support the process of library migration by mining function mappings from projects that have already completed the transition between two given libraries. This approach pragmatically overcomes the challenges of establishing a notion of "similarity" in terms of the program constructs themselves and is, therefore, immune against differences in structure and vocabulary. However, in our context, historical data is not applicable.

C. Preventing re-implementations

Thung et al. [31] address the proplem of duplicate implementation in a constructive way: by analyzing repositories to determine which APIs are used together, they provide recommendations of potentially useful APIs for a given project. Their goal is to inform developers of existing APIs before they re-implement the respective functionality. Our work complements this approach by discovering already existing re-implementations that could be replaced by libraries.

Code recommenders, proposed by [11], [14], [15], [21], [32], address re-implementations by recommending code snippets or applicable library methods depending on the current development context. Similar to code recommenders, enhanced code completion [4], [27] aims to ease discovery of existing functionality to the developer. These approaches do not support detection of already existing re-implementations. However, the techniques employed to generate recommendations include code structure and identifier analyses.

D. Concept location

The use of identifiers is a common strategy for concept location [19]. Methods from text retrieval, TR, (such as Term Frequency-Inverse Document Frequency, TF-IDF [29], or Latent Semantic Indexing, LSI [8]) are used to extract concepts given by e.g. use cases from source code. Recently, these approaches have been enhanced by adding static and/or dynamic program information. A study by Basset and Kraft [2] further suggests that structural term weighting can improve TR based concept location. To the best of our knowledge, the mentioned techniques have not been applied to our case.

III. General Approach

The following section presents our approach. At this stage of our work, we focus on discovering re-implementations of well established concepts available in open source libraries.

Our simion detection proceeds as follows (see Figure 1): it takes as input the so-called "concept library", a curated collection of libraries from which the concepts are mined, as well as the "study object", consisting of one ore more software systems in which we look for simions. The identifiers of concept library and study object are extracted and analyzed in a preprocessing phase to learn the specific concepts present

in their source code. The preprocessed identifier information is then used in the matching phase to compute the likelihood of two code entities implementing equivalent functionality.

By resorting to identifiers, we overcome the problem of restricting similarity to a syntactic level. As studies have shown, identifiers are valuable sources for capturing programmers' intent [5], [10]. Therefore, we assume that two code fragments that contain identifiers belonging to the same concept might provide the same functionality and could, therefore, be potential re-implementations. This assumption is strengthened by Haiduc and Marcus [13].

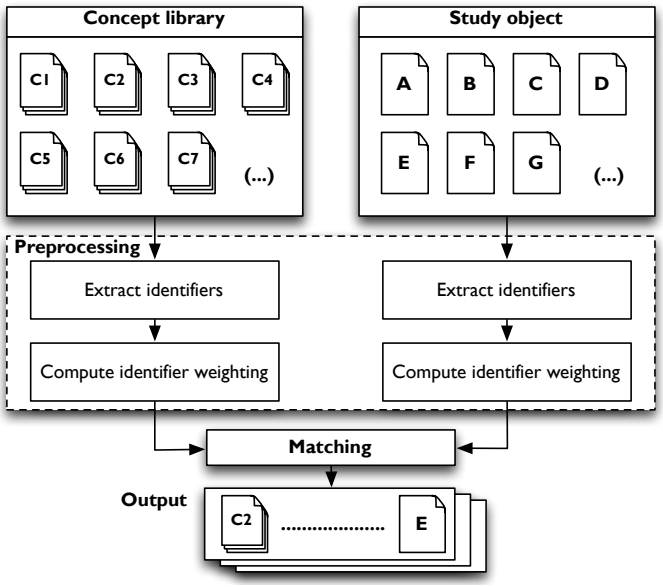

Fig. 1. This figure illustrates our approach: we extract relevant identifiers for each concept and compute the best matches within the study object. In the preprocessing step, different approaches can be taken to select and prepare the identifiers that are subsequently used.

Whilst the intuition behind this approach is quite simple, relying solely on identifiers risks to clutter the results with false positives: the same identifiers occur when defining a specific functionality as well as when using it. To mitigate this, we only consider identifiers present in *declarations* of methods, fields, and classes.

Our approach abstracts functionality provided by identifiers on a per-file basis. We opt for this granularity to capture concepts spread over several methods. During the preprocessing phase, we assign a set of "significant" identifiers to each source code file. We deem those identifiers as significant that best[1] capture the concepts of the respective file. Based on this information, we compute a similarity score for all files within the study object and the files of the concept library[2]. The similarity score is computed as follows: $\frac{\sum_{i \in I_{b \cap s}} v(i)}{\sum_{i \in I_b} v(i)}$, with I_b denoting the relevant identifiers of a concept file and $I_{b \cap s}$ denoting the overlapping relevant identifiers of a concept and a study object file. $v(i)$ denotes the weight assigned to the given identifier i.

We implemented our prototype on top of ConQAT[3], an Open Source software quality analysis tool.

IV. Calibration of the approach

The quality of the obtained results depends significantly on the processing (and the quality) of the identifiers. In this section, we describe the variation points and the steps of calibrating the parameters of our approach.

There are two steps in the process that allow for variation: (1) identifier extraction and (2) identifier ranking. For both steps, we present the potential options. Then, we describe the setup of the calibration process, the tested parameter configurations, and the resulting set of parameters used for our proof-of-concept evaluation.

Identifier extraction: The first step of the preprocessing phase is the extraction of the identifiers. We assume that preselection of identifiers guided by the program structure would improve the precision of our findings. To quantify the effect of this step we included this decision in our calibration process. Second, we tested the impact of splitting[4] and partitioning the identifiers on our results.

Identifier ranking: The second step of the preprocessing phase assigns a weight to the identifiers extracted for each file. Pragmatically, one could count the absolute frequencies to identify the most relevant concepts in a file. However, this approach risks to overshadow important concepts. For this reason, we compare the effect of ranking concepts according to their absolute identifier frequency to using the TF-IDF metric. Furthermore, we test the impact of several threshold values for TF-IDF.

A. Calibration setup

To calibrate our approach, we considered the specific use case of discovering potential re-implementations of well known "Collections" concepts in the Qualitas Corpus. We buildt up our concept library by curating these concepts from well known Open Source libraries, such as Apache Commons, Trove[5] and Guava[6]. By manually assessing the library implementations, we found that indeed the vocabulary used to represent the concepts in the identifiers was very similar.

Study Object: To test the suitability of a configuration, we need a way to measure the quality of the result we obtain. Since we can not manually establish the number of re-implementations of a given concept within the 112 systems

[1] "Best" is determined by the characteristic identifiers contained in the corpus of libraries.

[2] Depending on the context, not all concepts need to be searched for. Instead, the analysis can be run for specific concepts present in the library, such as "Collections", "I/O" etc.

[3] www.conqat.org

[4] We used CamelCase as well as non alpha-numeric characters as indicators for splitting. Furthermore, we applied an English word stemmer and removed trailing digits.

[5] http://trove.starlight-systems.com/

[6] https://code.google.com/p/guava-libraries/

present in the Qualitas Corpus[7], we injected deliberate re-implementations into the corpus by inserting the files of the Guava Collections into the Qualitas Corpus[8]. In this way, we obtain a known set of expected hits for our approach. Nevertheless, determining the quality of the result remains challenging. Manually validating the presence of all expected Guava files in the results is infeasible. Furthermore, results yielding the same number but different files can not be differentiated in quality. To address these challenges, we set up the experiment as shown in Figure 2: for each configuration, we run our analysis once. Then, we randomly sample 10 files from the Guava Collections and probe the result set to find out 1) whether they are included and 2) in which position of the result set they occur. This probing step is repeated 100 times, each time with a different random sample.

Configurations: We account for the mentioned variation points in the following way: The selection of the identifiers is done either by extracting all identifiers present in the current file or extracting only identifiers present in declarations of classes, methods, and variables.

Partitioning of the identifiers refers to employing splitting techniques and providing, in addition, substring representations of the identifiers. Take as example the identifier *arrayStackItem*. The ordered substring representations would yield: {*array*, *stack*, *item*, *arrayStack*, *stackItem*, *arrayStackItem*}. This variation is either on or off. The configuration for TF-IDF varies from counting the identifier frequency to using TF-IDF with the threshold values of 5, 10, and 15.

To compare the results for each configuration, we computed the following metrics per probing step: the *average hit count*, establishing how many of the files in our probing set are present in the results, and the *position*, denoting the rank of the files in the result set. The final performance of a configuration is rated by averaging the average hit count and the position values for all the probing steps.

B. Calibrated configuration

The calibration procedure yielded the following configuration as the most suitable: *preselecting* the identifiers according to the program structure, *partitioning* the identifiers, and using *TF-IDF with a threshold value of 5*. Consequently, we run our proof of concept evaluation with these settings.[9]

C. Taxonomy of the results

From the calibration results, we built up a taxonomy of findings by manually assessing the first 250 results. The code fragments matched the following categories:

A code fragment is a *potential re-implementation* if it implements or extends functionality contained by or equal to the concept we searched for. Clearly, our approach can not prove the fragment's functional equivalence. However, it can point developers to candidate points in order to establish whether they really present missed reuse opportunities. Potential re-implementations can have a varying degree of similarity to the concept implementation in the library. We therefore differentiate between *perfect match*, where study object and concept library implement the same functionality, *similar match*, where study object and concept library implement similar aspects of the same concept, and *concept match*, where study object and concept library implement different aspects of the same concept.

If a code fragment merely wraps library calls for a specific concept, our approach will still include it in the result. However, we consider this case as a *concept application* and do not classify it as missed reuse opportunity.

False positives unrelated to the considered concept are seen as *bad matches*. For situations where we can not fit a result in any of these categories, we label them as *undefined*.

Result classification: The distribution of the calibration findings is as follows: 195/250 potential re-implementations (out of which 11/195 perfect matches, 86/195 similar matches and 98/195 concept matches), 30/250 concept applications, 18/250 bad matches, and 7/250 undefined. The ranking of the results intuitively presented the potential re-implementations with higher values than the other categories.

We, furthermore, assessed the 18 bad matches found within the 250 results. The majority (16/18) of bad matches occurred due to similarities in the vocabulary employed by different concepts, in our case string manipulation and iterations over collections. The remaining bad matches were applications of the concept contained e.g. in implementations wrapping the usage of a library.

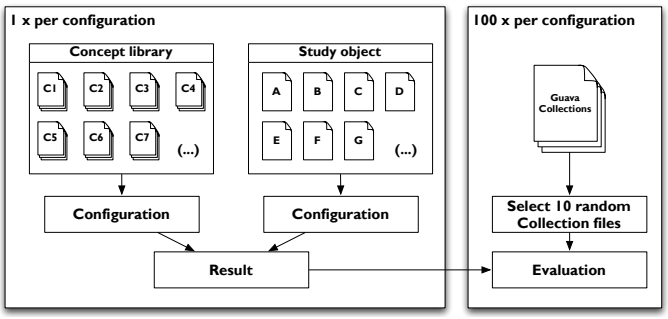

Fig. 2. This figure visualizes our evaluation process for detecting re-implementations of Collection functionality. To establish a base-line of known duplicates, we injected the Collection implementation of Guava into the Qualitas Corpus and measured the detection rates for these known "re-implementations".

V. Proof of concept evaluation

Our proof of concept evaluation provides a first answer to the following question: Which re-implementations do we find within our study objects?

To answer this question, we select three Open Source Java projects, analyze them and manually examine the source code indicated as re-implementation by our approach. We restrict the search for re-implementations again to the "Collections"

[7]We used the Qualitas Corpus version 20130901r and the JRE 1.6.0.

[8]For this setup, we removed the Guava Collections from the concept library.

[9]Comparing our calibrated approach to [20] would be interesting. However, it is unclear if their system is available.

concept. Furthermore, we limit the assessment to the first 30 results for each system.

Setup: The concept library used for this evaluation contains the Apache, Trove and Guava collections. Our study objects are the Apache projects "MyFaces" and "Tomcat", present in the Qualitas Corpus, and the "Spring IO" framework[25]. The systems provide functionality related to web applications with Java. Due to this specialization, we expect them to use given collection implementations. Therefore, re-implementations of this concept would be missed reuse opportunities.

Results: The inspection of the analysis results yielded the following re-implementations: a perfect match of *IteratorEnumeration* and a concept match for *MapEntries* in MyFaces, a *NullComparator* perfect match and a similar match for a *UnmodifiableMap* in the Spring framework, and a perfect match for the *ArrayStack* implementation, a perfect match for the *Entry* implementation and a similar match for a *HashMap* implementation in Tomcat. For all three systems, the perfect matches ranked within the first five positions of the findings.

VI. Threats to validity

The preliminary character of our investigation entails a number of threats to validity. Firstly, the concept library and the study objects are currently very specific. It remains to be seen how well the approach performs on a larger and less carefully curated collection of libraries and systems.
Secondly, we calibrated our approach for a well known concept, encompassing a clear vocabulary. This characteristic might not be necessarily given for other concepts. It remains to be seen if our approach can provide helpful results nevertheless. Possibly, it could be enhanced by sourcing domain knowledge from the implementors or including ontologies.
Determining the equivalence of implementations by automatic ranking as well as manual inspection remains challenging. Therefore, neither the weighting function nor the manual assessment for determining the quality of the results can be perfectly reliable.

VII. Conclusion and Future Work

We presented a pragmatic approach to detect re-implementations. Due to a wider notion of similarity, which is based on the concepts contained in the source code, it is able to find potential duplicates that likely would be missed by established approaches such as clone detection. Our preliminary results look promising and encourage us to follow up with extended evaluations. In this way, we aim to quantify the extent of re-implementations in software systems as well as to support practitioners to avoid or remove missed reuse opportunities.

Acknowledgements

Thanks go to Sonia Haiduc, Diego Marmsoler, Lars Heinemann for helpful comments on earlier versions of this work. Parts of this work were funded by the Federal Ministry of Education and Research, Germany (BMBF). Project code Software Campus (TU München), grant number 01IS12057.

References

[1] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. In *ISESE*, 2005.
[2] B. Basset and N. A. Kraft. Structural Information Based Term Weighting in Text Retrieval for Feature Location. In *ICPC'13*.
[3] V. Bauer, J. Eckhardt, B. Hauptmann, and M. Klimek. An exploratory study on reuse at google. In *SER&IP's 2014*.
[4] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, 2009.
[5] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM'00*, 2000.
[6] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *FASE*. 2009.
[7] G. Cousineau and P. Enjalbert. Program equivalence and provability. In *MFCS*, 1979.
[8] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. 1990.
[9] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner. Challenges of the dynamic detection of functionally similar code fragments. In *CSMR'12*.
[10] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.
[11] E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in apis. In *ECOOP*, 2011.
[12] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE' 08*.
[13] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *ICPC'08*.
[14] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-based context-dependent api method recommendation. In *CSMR'12*, 2012.
[15] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.
[16] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA*, 2009.
[17] E. Jürgens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *CSMR*, 2010.
[18] D. Kawrykow and M. P. Robillard. Improving api usage through automatic detection of redundant code. In *ASE' 09*.
[19] A. Marcus and S. Haiduc. Text retrieval approaches for concept location in source code. In *ISSSE'11*.
[20] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *ASE'01*.
[21] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick. Rascal: A recommender agent for agile reuse. *Artif. Intell. Rev.*, 24(3-4):253–276, 2005.
[22] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *ICSE*, pages 364–374, 2012.
[23] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *Software Engineering, IEEE Transactions on*, 38(5):1069–1087, 2012.
[24] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher. Recommending source code for use in rapid software prototypes. In *ICSE*, 2012.
[25] Pivotal Software, Inc. The spring io platform.
[26] D. Ratiu and J. Jürjens. Evaluating the reference and representation of domain concepts in apis. In *ICPC*, pages 242–247, 2008.
[27] R. Robbes and M. Lanza. Improving code completion with program history. *Autom. Softw. Eng.*, 17(2):181–212, 2010.
[28] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, 115, 2007.
[29] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval.
[30] C. Teyton, J.-R. Falleri, and X. Blanc. Automatic discovery of function mappings between similar libraries. In *WCRE*, 2013.
[31] F. Thung, D. Lo, and J. Lawall. Automated library recommendation. In *WCRE*, pages 182–191, 2013.
[32] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand. In *IUI*, pages 159–166, 2002.