QualBoa: Reusability-aware Recommendations of Source Code Components

Themistoklis Diamantopoulos
thdiaman@issel.ee.auth.gr

Klearchos Thomopoulos
kleathom@ece.auth.gr

Andreas Symeonidis
asymeon@eng.auth.gr

Electrical and Computer Engineering Dept.
Aristotle University of Thessaloniki
Thessaloniki, Greece

ABSTRACT

Contemporary software development processes involve finding reusable software components from online repositories and integrating them to the source code, both to reduce development time and to ensure that the final software project is of high quality. Although several systems have been designed to automate this procedure by recommending components that cover the desired functionality, the reusability of these components is usually not assessed by these systems. In this work, we present QualBoa, a recommendation system for source code components that covers both the functional and the quality aspects of software component reuse. Upon retrieving components, QualBoa provides a ranking that involves not only functional matching to the query, but also a reusability score based on configurable thresholds of source code metrics. The evaluation of QualBoa indicates that it can be effective for recommending reusable source code.

Keywords

Recommendation Systems in Software Engineering; Source Code Metrics; Code Reuse

1. INTRODUCTION

Lately, the rise of the open source community and the introduction of online source code repositories have provided numerous exploitation possibilities in the context of software reuse. Developers often rely on finding reusable source code components, both to reduce the time spent to develop them and to ensure that the resulting software is of high quality (in terms of reliability and functionality delivered). Online source code repositories and question answering communities, such as GitHub or Stack Overflow, have facilitated the task of finding suitable (with respect to a developer query) source code. Furthermore, several specialized *Code Search Engines (CSEs)* and indexing systems, such as Boa [1], offer advanced syntax-aware capabilities as well as other types of information about the source code, including e.g. documentation, license details, etc.

However, exploiting this information effectively is not trivial; the developer usually has to create an appropriate query for a component, manually refine the returned source code, and possibly adapt the component to the system being developed. The idea of automating these tasks has led to the introduction of source code recommendation systems, lying in the area of *Recommendation Systems in Software Engineering (RSSEs)* [2,3]. Most of these efforts have focused on the functional aspects of component reuse, applying matchmaking mechanisms to provide functionally adequate software components to the developer. Certain systems have also employed test cases to ensure that the desired functionality is fulfilled [2], while others have even implemented simple source code transformations in order to integrate the components directly into the source code of the developer [3].

Although these systems cover the functional criteria posed by the developer, they do not offer any assurance concerning the *reusability* of the source code. Reusing retrieved code can be a risky practice, considering no quality expert has assessed it. A possible solution is to exploit the power of the developers corpus, as performed by Bing Developer Assistant [3], which promotes components that have been chosen by other developers. However, crowdsourcing solutions are not always accurate, considering that developers may require specialized components with specific quality criteria. Primal efforts towards this direction can be traced in Code Conjurer [2], which selects the less complex out of all functionally equivalent components, determined by the lines of code. However, defining the criteria for the complexity and generally for the reusability of a component, is not trivial.

In this work, we present an RSSE that covers both the functional and the quality aspects of component reuse. Our system is called QualBoa since it harnesses the power of Boa [1], a sophisticated indexing service, to locate useful software components and compute quality metrics. Upon downloading the source code components from GitHub, QualBoa generates for each component both a functional score and a reusability index based on quality metrics. Furthermore, the generated reusability index is configurable to allow the involvement of quality experts.

The rest of this paper is organized as follows. Section 2 presents the architecture of QualBoa and illustrates its functionality. Section 3 evaluates our system in a component reuse context. Finally, Section 4 summarizes our contributions and provides insight for further research.

MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2903492

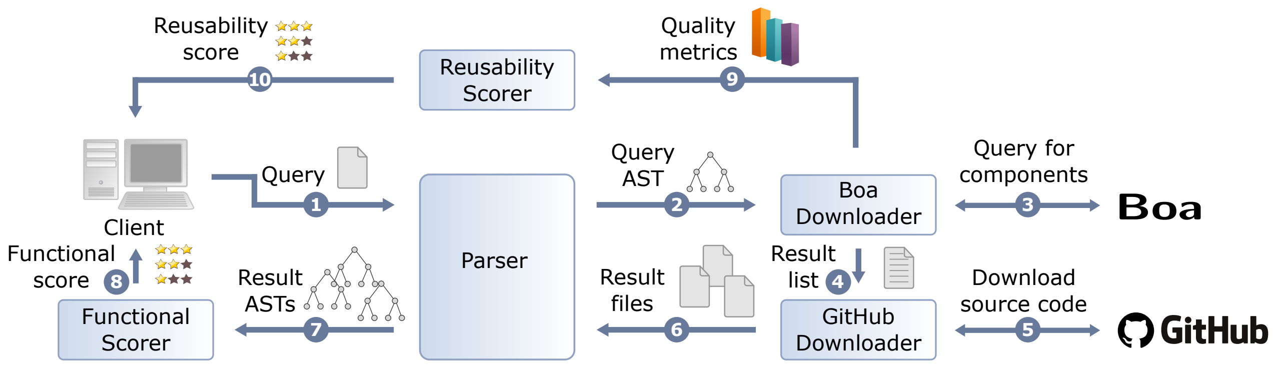

Figure 1: The architecture of QualBoa

2. QUALITY CODE RECOMMENDATIONS

2.1 High-Level Overview of QualBoa

The overall architecture of QualBoa is shown in Figure 1. This architecture is agnostic, since it can support any language and integrate with various code hosting services. In this work, we employ Boa and GitHub as the code hosting services, and implement the language-specific components for the Java programming language. Initially, the developer provides a query in the form of a *signature*. A signature is similar to a Java interface which comprises all methods of the desired class component. An example signature for a "Stack" component is shown in Figure 2.

```
public class Stack{
    public void push(Object element);
    public Object pop();
}
```

Figure 2: Example signature for a class "Stack" with two methods, "push" and "pop"

The signature is given as input to the *Parser*. The Parser, which is implemented using Eclipse JDT[1], extracts the *Abstract Syntax Tree (AST)* of any given Java file. Upon extracting the AST for the query, the *Boa Downloader* parses it and constructs a Boa query for relevant components, including also calculations for quality metrics. The query is submitted to the Boa engine [1], and the response is a list of paths to Java files and the values of quality metrics for these files. The result list is given to the *GitHub Downloader* which downloads the files from GitHub. The files are then given to the Parser so that their ASTs are extracted.

The ASTs of the downloaded files are further processed by the *Functional Scorer*, which creates the *functional score* for each result. The Functional Score of a result file denotes whether the file fulfills the functionality posed by the query of the developer. The *Reusability Scorer* receives as input the metrics for the result files and generates a *reusability score*, denoting the extent to which each file is reusable. Finally, the developer (*Client*) is provided with the scores for all files, and the results are given ranked according to the Functional Score for each file. The procedures of downloading components and constructing their functional and quality scores are described in the following subsections.

[1]http://www.eclipse.org/jdt/

2.2 Downloading Source Code and Metrics

Upon extracting the elements of the query, including class name, method names and types, QualBoa finds useful software components and computes their metrics using Boa [1]. A simplified version of the query is shown in Figure 3.

```
visit(p, visitor {
    ...
    before node: Declaration -> {
        if (match(class_name, node.name)) {
            foreach (i: int; node.methods[i])
                visit(node.methods[i]);
        }
    }
    before node: Method -> {
        for (i := 0; i < len(method_names); i++) {
            if (match(method_names[i], node.name)) {
                match_names[i] = true;
                if (method_types[i] == node.return_type.name)
                    match_types[i] = true;
            }
        }
    }
    ...
    after node: Declaration -> {
        foreach (i: int; def(node.fields[i])) {
            foreach (j: int; def(node.fields[i].modifiers[j])) {
                if (node.fields[i].modifiers[j].visibility ==
                        Visibility.PUBLIC)
                    num_public_fields++;
            }
        }
        ...
    }
});
```

Figure 3: Boa query for components and metrics

The query follows the visitor pattern to traverse the ASTs of the Java files in Boa. At first, the type declarations of all files are visited to determine whether they match the given class name (class_name). The methods of matched files are further visited to check whether their names and types match those of the query (method_names and method_types). The number of matched elements is aggregated to rank the results and a list with the first 150 results is given to GitHub Downloader, which retrieves the files from the GitHub API.

Table 1: The reusability model of QualBoa

Quality Metrics	Extreme Values	Relevant Quality Characteristics: Modularity	Maintainability	Usability	Understandability	Reusability
Average Lines of Code per Method	> 30		×		×	×
Average Cyclomatic Complexity	> 8		×		×	×
Coupling Between Objects	> 20	×		×		×
Lack of Cohesion in Methods	> 20	×				×
Average Block Depth	> 3				×	×
Efferent Couplings	> 20	×		×		×
Number of Public Fields	> 10		×	×		×
Number of Public Methods	> 30		×	×		×
#Metrics per Quality Characteristic:		3	4	4	3	8

The second part of the query involves computing source code metrics for the retrieved files. Figure 3 depicts the code for computing the number of public fields. We extend the query to compute the metrics of Table 1 (see Section 2.4).

2.3 Mining Source Code Components

The Functional Scorer computes the similarity between each of the ASTs of the results with the AST of the query, in order to rank the results according to the functional desiderata of the developer. Initially, both the query and each examined result file is represented as a list. The list of elements for a result file is defined in the following equation:

$$result = [name, method_1, method_2, \ldots, method_n] \quad (1)$$

where *name* is the name of the class and $method_i$ is a sublist of the i-th method of the class, out of n methods in total. The sublist of a method is defined as:

$$method = [name, type, param_1, param_2, \ldots, param_m] \quad (2)$$

where *name* is the name of the method, *type* is its return type, and $param_j$ is the type of its j-th, out of m parameters in total. Using equations (1) and (2), we can represent each result, as well as the query, as a nested list structure.

Comparing a query to a result requires computing a similarity score between the two corresponding lists, which in turn implies computing the score between each pair of methods. Note that computing the maximum score between two lists of methods requires finding the score of each pair of methods and selecting the highest scoring pairs. Following the approach of the *stable marriage* problem, this is accomplished by ordering the pairs according to their similarity and selecting them one-by-one off the top, noticing whether a method has already been matched. The same process is also used for matching the parameters between method lists.

Class names, method names and types, as well as parameter types are matched using a token set similarity approach. Since in Java identifiers follow the camelCase convention, we first split each string into tokens and compute the Jaccard index for the two token sets. Given two sets, the Jaccard index is defined as the size of their intersection divided by the size of their union. Finally, the similarity between two lists or vectors $\vec{A}$ and $\vec{B}$ (either in method or in class/result level) is computed using the Tanimoto coefficient of the vectors $\vec{A} \cdot \vec{B}/(|\vec{A}|^2 + |\vec{B}|^2 - \vec{A} \cdot \vec{B})$, where $|\vec{A}|$ and $|\vec{B}|$ are the sizes of vectors $\vec{A}$ and $\vec{B}$, and $\vec{A} \cdot \vec{B}$ is their inner product.

For instance, given the component lists $[Stack, [push, void, Object], [pop, Object]]$ and $[IntStack, [pushObject, void, int], [popObject, int]]$, the similarity between the method pairs for the "push" functionality is computed as follows:

$$\begin{aligned} score_{PUSH} = {} & Tanimoto([1,1,1], \\ & [Jaccard(\{push\}, \{push, object\}), \\ & \ Jaccard(\{void\}, \{void\}) \\ & \ Jaccard(\{object\}, \{int\})]\ ) \\ = {} & Tanimoto([1,1,1], [0.5, 1, 0]) \simeq 0.545 \end{aligned} \quad (3)$$

where the query list is always a list with all elements set to 1, since it constitutes a perfect match. Similarly, the score for the "pop" functionality is approximately 0.286 and, thus, the total score for the class is approximately 0.579.

2.4 Recommending Quality Code

Upon constructing a functional score for the components, QualBoa checks whether each component is suitable for reuse using source code metrics. Although the problem of measuring the reusability of a component using source code metrics has been studied extensively [4–7], the choice of metrics is not trivial; research efforts include employing the C&K metrics [5], coupling and cohesion metrics [6], and several other metrics referring to volume and complexity [4]. However, the main quality axes that measure whether a component is reusable are common. In accordance with the definitions of different quality characteristics [8] and the current state-of-the-art [5, 7, 9], reusability spans across the *modularity*, *usability*, *maintainability*, and *understandability* concepts.

Our reusability model, shown in Table 1, includes 8 metrics that refer to one or more of these characteristics. These metrics cover several aspects of a component, including volume, complexity, coupling, and cohesion. The model is simple, marking the value of each metric as *normal* or *extreme* according to the thresholds shown in the second column of Table 1. Each normal value contributes as one quality point in the relevant characteristics. Reusability includes all 8 values. Thus, given e.g. a component with extreme values for 2 out of 8 metrics, the reusability score would be 6 out of 8.

Determining appropriate thresholds for quality metrics is non-trivial, since different types of software may have to reach specific quality objectives. Thus, QualBoa offers the ability to configure these thresholds, while their default values are set according to the current state-of-the-practice, as defined by current research [10] and widely used static code analysis tools, such as PMD[2] and CodePro AnalytiX[3].

[2]https://pmd.github.io/

[3]https://developers.google.com/java-dev-tools/codepro/

3. EVALUATION

The source code of QualBoa and all data required to reproduce our findings are available in the repository:

https://github.com/AuthEceSoftEng/QualBoa

We evaluated QualBoa in the dataset of [2], which contains 7 queries for different types of components, shown in the first column of Table 2. For each query, we examine the first 30 results of QualBoa, and mark each result as useful or not useful, considering a result as useful if integrating it in the developer's code would require minimal or no effort. In other words, useful components are the ones that are understandable and at the same time cover the required functionality. Benchmarking and annotation were performed separately by the first and the second author respectively.

Given these marked results, we check the number of relevant results retrieved for each query, and compute also the average precision for the result rankings. The average precision was considered as the most appropriate metric since it covers not only the relevance of the results but also and more importantly their ranking. We further assess the files using the reusability model of QualBoa and, upon normalizing to percentages, report the average reusability score for the useful/relevant results of each query. Our evaluation is summarized in Table 2.

Table 2: Evaluation results of QualBoa

Query	#Relevant Results	Average Precision	Reusability Score
Calculator	18	59.27%	70.83%
ComplexNumber	15	86.18%	82.76%
Matrix	10	95.88%	88.68%
MortgageCalculator	7	100.00%	87.17%
ShoppingCart	13	100.00%	100.00%
Spreadsheet	2	100.00%	88.54%
Stack	22	77.59%	100.00%
Average	12.43	88.42%	88.28%

Concerning the number of relevant results, QualBoa seems to be quite effective. In specific, our RSSE successfully retrieves at least 10 useful results for 5 out of 7 queries. Additionally, almost all queries have average precision scores higher than 75%, indicating that the results are also ranked correctly. This is particularly obvious in cases where the number of retrieved results is limited, such as the MortgageCalculator or the Spreadsheet component. The perfect average precision scores indicate that the relevant results for these queries are placed on the top of the ranking.

The results of our system are also highly reusable, as indicated by the reusability score for each query. In 5 out of 7 queries, the average reusability score of the relevant results is quite near or higher than 87.5%, indicating that on average the components do not surpass more than 1 of the thresholds defined in Table 1, so they have a reusability score of at least 7 out of 8. The reusability score is also related to the overall complexity of the queried component. In specific, data structure components such as Stack or ShoppingCart are not prone to severe quality issues and thus have perfect scores. On the other hand, complex components such as Calculator or ComplexNumber may contain methods that inter-operate, thus their reusability scores are expected to be lower.

4. CONCLUSION

Although several systems have been developed for finding source code components, the assessment of the reusability of these components has not been adequately addressed. In this work, we presented QualBoa, a system that incorporates functional and quality information to recommend components that are not only functionally equivalent to the query of the developer but also reusable. Our evaluation indicates that QualBoa is effective for retrieving reusable results.

Further work lies in several directions. The reusability model can be refined to comply with the specifics of different component categories. It can also be automatically adapted so that the recommended components have the same quality characteristics with the source code of the developer.

5. ACKNOWLEDGMENTS

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission.

6. REFERENCES

[1] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE 2013, pages 422–431, May 2013.

[2] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Softw.*, 25(5):45–52, September 2008.

[3] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. Building bing developer assistant. Technical Report MSR-TR-2015-36, Microsoft Research, May 2015.

[4] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, February 1991.

[5] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson, and Giancarlo Succi. Does refactoring improve reusability? In *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components*, ICSR'06, pages 287–297, 2006.

[6] Gui Gui and Paul D. Scott. Coupling and cohesion measures for evaluation of component reusability. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 18–21, New York, NY, USA, 2006. ACM.

[7] Jeffrey S. Poulin. Measuring software reusability. In *Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability*, pages 126–138, Nov 1994.

[8] Diomidis Spinellis. *Code Quality: The Open Source Perspective (Effective Software Development Series)*. Addison-Wesley Professional, 2006.

[9] Fatma Dandashi. A method for assessing the reusability of object-oriented code using a validated set of automated measurements. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, SAC '02, pages 997–1003, New York, NY, USA, 2002. ACM.

[10] Kecia A. M. Ferreira, Mariza A. S. Bigonha, Roberto S. Bigonha, Luiz F. O. Mendes, and Heitor C. Almeida. Identifying thresholds for object-oriented software metrics. *J. Syst. Softw.*, 85(2):244–257, 2012.