User-Perceived Reusability Estimation based on Analysis of Software Repositories

Michail Papamichail, Themistoklis Diamantopoulos, Ilias Chrysovergis, Philippos Samlidis, Andreas Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki, Greece
mpapamic@issel.ee.auth.gr, thdiaman@issel.ee.auth.gr, iliachry@ece.auth.gr, filippas@ece.auth.gr, asymeon@eng.auth.gr

***Abstract*—The popularity of open-source software repositories has led to a new reuse paradigm, where online resources can be thoroughly analyzed to identify reusable software components. Obviously, assessing the quality and specifically the reusability potential of source code residing in open software repositories poses a major challenge for the research community. Although several systems have been designed towards this direction, most of them do not focus on reusability. In this paper, we define and formulate a reusability score by employing information from GitHub stars and forks, which indicate the extent to which software components are adopted/accepted by developers. Our methodology involves applying and assessing different state-of-the-practice machine learning algorithms, in order to construct models for reusability estimation at both class and package levels. Preliminary evaluation of our methodology indicates that our approach can successfully assess reusability, as perceived by developers.**

***Index Terms*—source code quality, reusability, static analysis, user-perceived quality**

I. Introduction

The concept of software reuse is not new; developers have always tried to reuse standalone sections of (own or others') code, by taking advantage of assets that have already been implemented and released, either wrapped within software libraries or even as components of applications. The benefits of this paradigm include the reduction of time and effort required for software development. Up until recently, the quest for high-quality code to reuse was restricted within organizations, or even within groups, which obviously led to suboptimal searches, given the limited number of projects considered.

During the last decade, however, the popularity of the open-source software paradigm has created the critical mass of online software projects needed, and this deluge of source code lying in online repositories has altered the main challenge of software reuse. Finding software components that satisfy certain functional requirements has become easy. Instead, what is now important is to identify components that are *suitable* for reuse. Assessing the reusability of a component before using it is crucial, since components of poor quality are usually hard to integrate and in some cases they even introduce faults.

The concept of reusability is linked to software quality, and assessing the quality of a component is a challenging task. Quality is multi-faceted and evaluated differently by different people, while it also depends on the scope and the requirements of each software project [1]. Should one follow the ISO/IEC 25010:2011 [2] and ISO/IEC 9126 [3] standards, software reusability, i.e. the extent to which a software component is reusable is related to four major quality properties: *Understandability*, *Learnability*, *Operability* and *Attractiveness*. These properties are directly associated with *Usability*, and further affect *Functional Suitability*, *Maintainability* and *Portability*, thus covering all four quality attributes related to reusability [4], [5] (the rest characteristics are *Reliability*, *Performance and Efficiency*, *Security*, and *Compatibility*).

Some of the aforementioned attributes can be effectively defined by using static analysis metrics, such as the known CK metrics [6], which have been extensively used for estimating software quality [7], [8]. However, one should mention that current research efforts largely focus on quality characteristics such as maintainability or security, and assessing reusability has not yet been extensively addressed. Furthermore, most quality models depend on fixed thresholds for the static analysis metrics, usually defined by an expert [9], and are not efficient enough for quality estimation in all cases. On the other hand, adaptable threshold models suffer more or less from the same limitations, as their ground truth is again an expert-defined quality value [10]. To this end, we argue that an interesting alternative involves employing *user-perceived* quality as a measure of the quality of a software component, an approach initially explored in [11].

In this work, we employ the concepts proposed in [11] so that we associate the extent to which a software component is adopted (or preferred) by developers, i.e. its popularity, with the extent to which it is reusable, i.e. its reusability. Typically, popularity can be determined by the number of stars and forks of GitHub repos. Thus, an important challenge involves relating this information to reusability. We argue that addressing this challenge requires constructing a methodology to decompose the influence of various static analysis metrics into the reusability degree of software. Unlike various systems that employ expert-based approaches, our methodology uses GitHub stars and forks as ground truth information towards identifying the static analysis metrics that influence reusability. Upon computing a large set of metrics both at class and package level, we model their behavior to translate their values into a reusability score. Metric behaviors are then used to train a set of reusability estimation models using different state-of-the-practice machine learning algorithms. Those models estimate the reusability degree of components at both class and package level, as perceived by developers.

II. Related Work

ISO/IEC 25010:2011 [2] defines *reusability* as a quality characteristic that refers to the degree to which an asset can be used in more than one system, or for building other assets. There are several approaches that aspire to assess the reusability of source code components using static analysis metrics [12], [13]. This assessment, however, is a non-trivial task, and often requires the aid of quality experts to examine and evaluate the source code. Since, however, the manual examination of source code can be very tedious, a common practice involves creating a benchmarking code repository using representative software projects and then applying machine learning techniques in order to calculate thresholds and define the acceptable metric intervals [14]–[16].

Further advancing the aforementioned systems, some research efforts attempt to derive reusability by setting thresholds for quality metrics. Diamantopoulos et al. [4] proposed a metric-based reusability scorer that calculates the degree of reusability of software components based on the values of eight static analysis metrics. The assessment depends on whether the values of the metrics exceed certain thresholds, as defined in current literature. When several thresholds are exceeded, the returned reusability score is lower.

Since using predefined metric thresholds is also limited by expert knowledge and may not be applicable on the specifics of different software projects (e.g. the scope of an application), several approaches have been proposed to overcome the necessity of using thresholds [17]–[20]. These approaches involve quantifying reusability through reuse-related information such as reuse frequency [17]. Then, machine learning techniques are employed in order to train reusability evaluation models using as input the values of static analysis metrics [18]–[20].

Although the aforementioned approaches can be effective for certain cases, their applicability in real-world scenarios is limited. At first, using predefined metrics thresholds [4] leads to the creation of models unable to incorporate the different characteristics of software projects. Automated reusability evaluation systems seem to overcome this issue [18]–[20], however they are still confined by the ground truth knowledge of quality experts for evaluating the source code and determining whether the degree of reuse is acceptable. Apart from its effect on both time and resources, this process may also lead to subjective evaluation, as each expert may prioritize differently the importance of each quality characteristic [21].

In this work, we build a reusability estimation system to provide a single score for every class and every package of a given software component. The estimation is based on the values of a large set of static analysis metrics and measures the extent to which the component is preferred by developers. We adopt the paradigm proposed in [11] and further extend it in the context of reusability. The authors in [11] employ the information of GitHub stars and forks, and through expert knowledge formulate and subsequently estimate software quality as perceived by developers. Instead, we initially evaluate the impact of each metric into the reusability degree of software components individually, and then aggregate the outcome of our analysis in order to construct a final reusability score. Finally, we quantify the reusability degree of software components both at class and package level by training reusability estimation models that effectively estimate the degree to which a component is reusable as perceived by software developers.

III. Reusability Modelling

In this section, we present an analysis of reusability from a quality attributes perspective and design a reusability score using GitHub information and static analysis metrics.

A. GitHub Popularity as Reusability Indicator

We associate reusability with the following major properties described in ISO/IEC 25010 [2] and 9126 [3]): *Understandability*, *Learnability*, *Operability* and *Attractiveness*. According to research performed by ARiSA [22], various static analysis metrics are highly related to these properties. Table I summarizes the relations between the six main categories of static analysis metrics and the aforementioned reusability-related quality properties. "P" is used to show proportional relation and "IP" implies inverse-proportional relation.

TABLE I
Categories of Static Analysis Metrics related to Reusability

Source Code Properties	**Reusability Related Quality Properties**			
	Forks related			**Stars related**
	Understandability	Learnability	Operability	Attractiveness
Complexity	IP	IP	IP	P
Coupling	IP	IP	IP	P
Cohesion	P	P	P	P
Documentation	P	P	–	–
Inheritance	IP	IP	–	P
Size	IP	IP	IP	P

P: Proportional Relation
IP: Inverse-Proportional Relation

As already mentioned, we employ GitHub stars and forks in order to quantify reusability and subsequently associate it with the aforementioned properties. As forks measure how many times the software repository has been cloned, they can be associated with *Understandability*, *Learnability* and *Operability*, as those properties formulate the degree to which a software component is [re]usable. Stars, on the other hand, reflect the number of developers that found the repository interesting, thus we may use them a measure of its *Attractiveness*.

B. Benchmark Dataset

We created a dataset that includes the values of the static analysis metrics shown in Table II, for the 100 most starred and 100 most forked GitHub Java projects (137 in total). These projects amount to more than 12M lines of code spread in almost 15K packages and 150K classes. All metrics were extracted at class and package level using SourceMeter[1].

[1]https://www.sourcemeter.com/

TABLE II
OVERVIEW OF STATIC METRICS AND THEIR APPLICABILITY ON DIFFERENT LEVELS

Static Analysis Metrics			Compute Levels	
Type	Name	Description	Class	Package
Complexity	NL{•,E}	Nesting Level {Else-If}	×	
	WMC	Weighted Methods per Class	×	
Coupling	CBO{•,I}	Coupling Between Object classes {Inverse}	×	
	N{I,O}I	Number of {Incoming, Outgoing} Invocations	×	
	RFC	Response set For Class	×	
Cohesion	LCOM5	Lack of Cohesion in Methods 5	×	
Documentation	AD	API Documentation	×	
	{•,T}CD	{Total} Comment Density	×	×
	{•,T}CLOC	{Total} Comment Lines of Code	×	×
	DLOC	Documentation Lines of Code	×	
	P{D,U}A	Public {Documented, Undocumented} API	×	×
	TAD	Total API Documentation		×
	TP{D,U}A	Total Public {Documented, Undocumented} API		×
Inheritance	DIT	Depth of Inheritance Tree	×	
	NO{A,C,D,P}	Number of {Ancestors, Children, Descendants, Parents}	×	
Size	{•,T}{•,L}LOC	{Total} {Logical} Lines of Code	×	×
	N{A,G,M,S}	Number of {Attributes, Getters, Methods, Setters}	×	×
	N{CL,EN,IN,P}	Number of {Classes, Enums, Interfaces, Packages}		×
	NL{A,G,M,S}	Number of Local {Attributes, Getters, Methods, Setters}	×	
	NLP{A,M}	Number of Local Public {Attributes, Methods}	×	
	NP{A,M}	Number of Public {Attributes, Methods}	×	×
	NOS	Number of Statements	×	
	TNP{CL,EN,IN}	Total Number of Public {Classes, Enums, Interfaces}		×
	TN{CL,DI,EN,FI}	Total Number of {Classes, Directories, Enums, Files}		×

C. *Evaluation of Metrics' Influence on Reusability*

As GitHub stars and forks refer to repository level, they are not adequate on their own for estimating the reusability of class level and package level components. Thus, we estimate reusability using static analysis metrics. For each metric, we first perform distribution-based binning and then relate its values to those of the stars/forks to incorporate reusability information. The final reusability estimation is computed by aggregating over the estimations derived by each metric.

1) Distribution-based binning: As the values of each metric are distributed differently among the repositories of our dataset, we first define a set of intervals (bins), unified across repositories, that approximate the actual distribution of the values. Thus, we use values from all packages (or classes) of our dataset to formulate a generic distribution, and then determine the optimal bin size that results in the minimum information loss. We use the *Doane* formula [23] for selecting the bin size in order to account for the skewness of the data. Figure 1 depicts the histogram of the Comments Density (CD) metric at package level, which will be used as an example throughout this section. Following our binning strategy, 20 bins are produced. CD values appear to have positive skewness, while their highest frequency is in the interval [0.13, 0.39]. After having selected the appropriate bins for each metric, we construct the histograms for each of the 137 repositories. An example for the histograms of two repositories is shown in Figure 2. The two distributions differ, which is expected since each repository has it own characteristics (different scope, contributors, etc.).

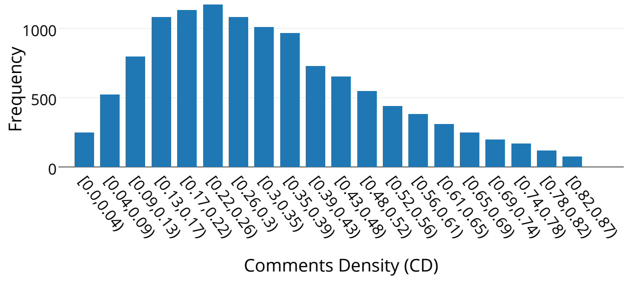


Fig. 1. Package-level Distribution of Comments Density for all Repositories.

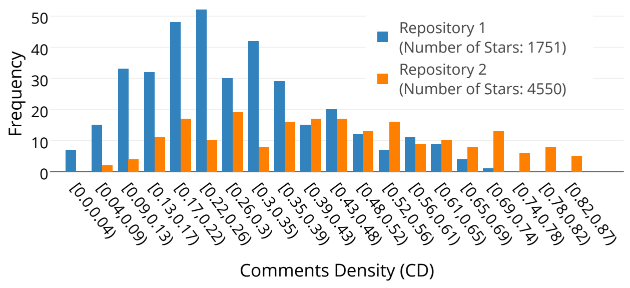


Fig. 2. Package-level Distribution of Comments Density for two Repositories.

2) Relate bins values with GitHub stars and forks: In this step, we construct a set of data instances that relate each metric bin value to a GitHub stars (or forks) value. To that end, we use the produced histograms (one for each repository using the bins calculated in the previous step) in order to construct a set of data instances that relate each metric bin value (here the CD) to a GitHub stars (or forks) value. So, we aggregate these values for all the bins of each metric, i.e. for metric X and bin 1 we gather all stars (or forks) values that correspond to packages (or classes) for which the metric value lies in bin 1 and aggregate them using an averaging method. This process is repeated for every metric and every bin.

Practically, for each bin value of the metric, we gather all relevant data instances and calculate the weighted average of their stars (or forks) count, which represents the stars-based (or forks-based) reusability value for the specific bin. The reusability scores are defined using the following equations:

$$RS_{Metric}(i) = \sum_{repo=1}^{N} freq_{p.u.}(i) \cdot \log(S(repo)) \quad (1)$$

$$RF_{Metric}(i) = \sum_{repo=1}^{N} freq_{p.u.}(i) \cdot \log(F(repo)) \quad (2)$$

where $RS_{Metric}(i)$ and $RF_{Metric}(i)$ refer to the stars and the forks-based reusability score of the i-th bin for the metric under evaluation, respectively. $S(repo)$ and $F(repo)$ refer to the number of stars and the number of forks of the repository, respectively, while the use of logarithm acts as a smoothing factor between the big differences in the number of stars and forks among the repositories. Finally, the term $freq_{p.u.}(i)$ is the normalized/relative frequency of the metric value of the i-th bin, defined as $F_i / \sum_{i=1}^{N}(F_i)$, where F_i is the absolute frequency (i.e. count) of the values lying in the i-th bin. For example, if a repository had 3 bins with 5 CD values in bin 1, 8 values in bin 2, and 7 values in bin 3, then the normalized frequency for bin 1 would be $5/(5+8+7) = 0.25$, for bin 2 it would be $8/(5+8+7) = 0.4$, and for bin 2 it would be $7/(5+8+7) = 0.35$. The use of normalized frequency was chosen to eliminate any biases caused by the high variance in the number of packages among the different repositories.

Figure 3 illustrates the results of applying (1), i.e. the star-based reusability, to the CD metric values at package level. As shown in this Figure, the reusability score based on CD is maximum for CD values in the interval $[0.3, 0.4]$.

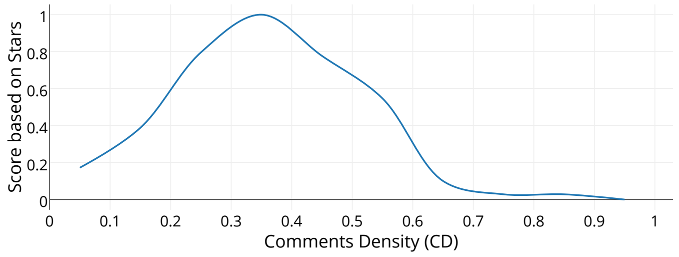


Fig. 3. Comments Density versus Star-based Reusability Score.

Finally, based on the fact that we compute a forks-based and a stars-based reusability score for each metric, the final reusability score for each source code component (class or package) is given by the following equations:

$$RS_{Final} = \frac{\sum_{j=1}^{k} RS_{Metric}(j) \cdot corr(metric_j, stars)}{\sum_{j=1}^{k} corr(metric_j, stars)} \quad (3)$$

$$RF_{Final} = \frac{\sum_{j=1}^{k} RF_{Metric}(j) \cdot corr(metric_j, forks)}{\sum_{j=1}^{k} corr(metric_j, forks)} \quad (4)$$

$$ReusabilityScore = \frac{3 \cdot RF_{Final} + RS_{Final}}{4} \quad (5)$$

where k is the number of metrics at each level. RS_{Final} and RF_{Final} correspond to the final stars-based and forks-based scores. $RS_{metric}(j)$ and $RF_{metric}(j)$ refer to the scores for the $j - th\ metric$ as given by equations (1) and (2), while $corr(metric_j, stars)$ and $corr(metric_j, forks)$ correspond to the Pearson correlation coefficient between the values of $j - th\ metric$ and the stars and forks. Finally, $ReusabilityScore$ refers to the final reusability score and is the weighted average of the final stars-based and forks-based scores. More weight (3 vs 1) is given in the forks-based score as it is associated with more reusability-related quality attributes (see Table I).

IV. Reusability Estimation

In this section, we devise a methodology that receives as input the values of static analysis metrics and estimates software reusability at class and package level. In specific, we use the calculated bins (see Section III) to train one model for each individual metric applied at each level. The output of each model provides a reusability score that originates from the value of the class (or package) under evaluation for the corresponding metric. All the scores are then aggregated to a final reusability score that represents the degree to which the class (or package) is adopted by developers.

We evaluate three techniques to select the optimal for fitting the metrics' behavior: *Support Vector Regression (SVR)* with radial basis function (RBF) kernel, Random Forest using bagging ensemble, and *Polynomial Regression* where the optimal degree is determined by applying the elbow method on the square sum of residuals. To account for cases where the number of bins, and consequently the number of training data points, is low, we used linear interpolation up to the point where the dataset for each model contained 60 instances.

Figure 4 illustrates the fitting procedure for the case of the forks-based reusability score based on the values of the API Documentation (AD) metric at package level. The figure depicts four lines, one that corresponds to the actual behavior of the metric, and three more lines, one for each model. It is obvious that the Random Forest outperforms the other two models (SVR and Polynomial Regression). This fact is also reflected in the values of the Mean Absolute Error for the AD metric, which are 0.0688, 0.1201 and 0.1228 respectively.

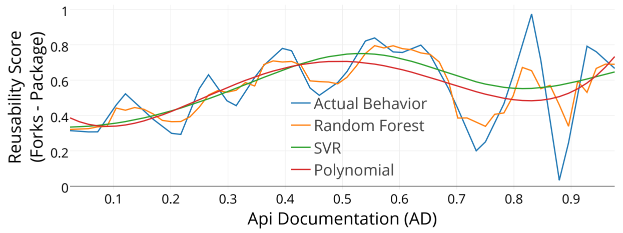


Fig. 4. Fitting procedure for the API Documentation metric at Package level.

We further compare the three models using the *Normalized Root Mean Squared Error (NRMSE)* metric. Given the actual scores $y_1, y_2, \dots, y_n$, the predicted scores $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$, and the mean actual score $\bar{y}$, the NRMSE is calculated as follows:

$$NRMSE = \sqrt{\frac{1}{N} \cdot \frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{N}(\bar{y} - y_i)^2}} \quad (6)$$

where N is the number of samples in the dataset. NRMSE was selected as it does not only take into account the average difference between the actual and the predicted values, but also provides a comprehensible result in a certain scale.

Figure 5 presents the mean NRMSE for the three algorithms regarding the reusability scores (forks/stars based) at both class and package levels. The Random Forest clearly outperforms the other two algorithms in all four categories.

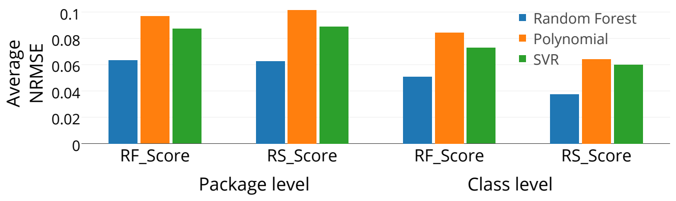
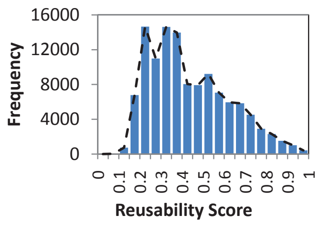
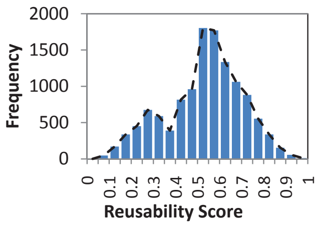


Fig. 5. Average NRMSE for all three machine learning algorithms.

Figure 6 depicts the distribution of the reusability score at class and package levels. As expected, the score in both cases follows a distribution similar to normal and the majority of instances are accumulated evenly around 0.5. For the score at class level, we observe a left-sided skewness. After manual inspection of the classes with scores in $[0.2, 0.25]$, they appear to contain little valuable information (e.g. most of them have LOC < 10) and thus are given low score.



(a)



(b)

Fig. 6. Reusability score distribution at (a) Class and (b) Package level.

V. Evaluation

To evaluate our approach, we used the 5 projects shown in Table III, out of which 3 were retrieved from GitHub and 2 were automatically generated using the tools of S-CASE[2]. Human-generated projects are expected to exhibit high deviations in their reusability score, as they include variable sets of components. By contrast, auto-generated projects are RESTful services and, given their automated structure generation, are expected to have high reusability scores and low deviations.

[2]http://s-case.github.io/

TABLE III
Dataset Statistics

#	Project Name	Type	# Packages	# Classes
1	realm-java	Human-generated	137	3859
2	liferay-portal	Human-generated	1670	3941
3	spring-security	Human-generated	543	7099
4	Webmarks	Auto-generated	9	27
5	WSAT	Auto-generated	20	127

In the following subsections, we provide an analysis for the reusability estimation of these projects as well as an indicative estimation example for certain classes and packages.

A. Reusability Estimation Evaluation

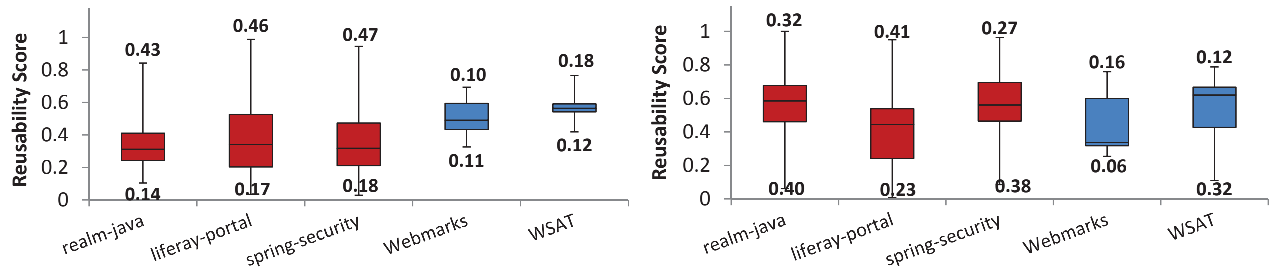
Figures 7a and 7b depict the distributions of the reusability score for all projects at class level and package level, respectively. The boxplots in blue refer to the auto-generated projects, while the ones in red refer to the human-generated ones. At first, it is obvious that the variance of the reusability scores is higher in the human-generated projects than in the auto-generated ones. This is expected since the projects that were generated automatically have proper architecture and high abstraction levels. These two projects also have similar reusability values, which is due to the fact that projects generated from the same tool ought to share similar characteristics that are reflected in the values of the static analysis metrics.

The high variance for the human-generated projects indicates that our methodology is capable of distinguishing components with both high and low degrees of reusability. Finally, given the reusability score distribution for all projects, we conclude that the results are consistent regardless of the project size. Despite the fact that the number of classes and the number of packages vary from very low values (e.g. only 9 packages and 27 classes) to very high values (e.g. 1670 packages and 7099 classes), the reusability score is not affected.

B. Example Reusability Estimation

Further assessing the validity of the reusability scores, we manually examined the static analysis metrics of sample classes and packages in order to check whether they align with the estimation. Table IV provides an overview of a subset of the computed metrics for representative examples of classes and packages with different reusability scores. The table contains static analysis metrics for two classes and two packages that received both high and low reusability scores.

Concerning the components that received high reusability score (Class 1 and Package 1), they appear to be well documented (the value of the Comments Density (CD) is 20.31% and 41.5%, respectively), which indicates that they are suitable for reuse. In addition, their values for the Lines of Code (LOC) metric, which are highly correlated with understandability and thus reusability, are typical. Thus, the scores for those components are rational. On the other hand, the class that received low score (Class 2) appears to have low cohesion (the LCOM5 metric value is high) and high coupling (the



(a) (b)

Fig. 7. Boxplots depicting Reusability Distributions for 3 Human-generated (🟥) and 2 Auto-generated (🟦) projects, (a) at Class level and (b) at Package level.

RFC metric value is high), while the package with low score (Package 2) appears to have little valuable information (only 38 LOC). Those code properties are crucial for the reusability-related quality attributes and thus the low scores are expected.

TABLE IV
METRICS FOR CLASSES AND PACKAGES WITH DIFFERENT REUSABILITY

Metrics	Classes		Packages	
	Class 1	Class 2	Package 1	Package 2
WMC	14	12	–	–
CBO	12	3	–	–
LCOM5	**2**	**11**	–	–
CD (%)	**20.31%**	**10.2%**	**41.5%**	**0.0%**
RFC	12	**30**	–	–
LOC	**84**	199	**2435**	**38**
TNCL	–	–	**8**	**2**
Reusability Score	**95.78%**	**10.8%**	**90.59%**	**16.75%**

VI. Conclusion and Future Work

In this work, we proposed a novel software reusability estimation approach based on the rationale that the reusability of a component is associated with the way it is perceived by developers, and thus could be useful for assessing the reusability of a component before integrating it into one's source code. Our evaluation indicates that our approach can be effective for estimating reusability at class and package level.

Concerning future work, an interesting idea would be to investigate and possibly redesign the reusability score in a domain-specific context. Finally, evaluating our system under realistic reuse scenarios, possibly also involving software developers, would be useful to further validate our approach.

References

[1] S. Pfleeger and B. Kitchenham, "Software quality: The elusive target," *IEEE Software*, pp. 12–21, 1996.

[2] "ISO/IEC 25010:2011," https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en, 2011, [Online; accessed October 2017].

[3] "ISO/IEC 9126-1:2001," https://www.iso.org/standard/22749.html, 2001, [Online; accessed October 2017].

[4] T. Diamantopoulos, K. Thomopoulos, and A. Symeonidis, "QualBoa: reusability-aware recommendations of source code components," in *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016*. IEEE, 2016, pp. 488–491.

[5] F. Taibi, "Empirical Analysis of the Reusability of Object-Oriented Program Code in Open-Source Software," *International Journal of Computer, Information, System and Control Engineering*, vol. 8, no. 1, pp. 114–120, 2014.

[6] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[7] C. Le Goues and W. Weimer, "Measuring code quality to improve specification mining," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 175–190, 2012.

[8] H. Washizaki, R. Namiki, T. Fukuoka, Y. Harada, and H. Watanabe, "A framework for measuring and evaluating program source code quality," in *Proceedings of the 8th International Conference on Product-Focused Software Process Improvement*, 2007, pp. 284–299.

[9] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Unsupervised Learning for Expert-Based Software Quality Estimation," in *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering*, ser. HASE'04, Washington, DC, USA, 2004, pp. 149–155.

[10] T. Cai, M. R. Lyu, K.-F. Wong, and M. Wong, "ComPARE: A generic quality assessment environment for component-based software systems," in *Intern. Symposium on Information Systems and Engineering*, 2001.

[11] M. Papamichail, T. Diamantopoulos, and A. Symeonidis, "User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics," in *IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016*. IEEE, 2016, pp. 100–107.

[12] A. P. Singh and P. Tomar, "Estimation of Component Reusability through Reusability Metrics," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 8, no. 11, pp. 1965–1972, 2014.

[13] P. S. Sandhu and H. Singh, "A reusability evaluation model for OO-based software components," *International Journal of Computer Science*, vol. 1, no. 4, pp. 259–264, 2006.

[14] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[15] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[16] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*. IEEE, 2014, pp. 254–263.

[17] T. G. Bay and K. Pauls, "Reuse Frequency as Metric for Component Assessment," ETH, Department of Computer Science, Zurich, Tech. Rep., 2004, technical Reports D-INFK.

[18] A. Kaur, H. Monga, M. Kaur, and P. S. Sandhu, "Identification and performance evaluation of reusable software components based neural network," *International Journal of Research in Engineering and Technology*, vol. 1, no. 2, pp. 100–104, 2012.

[19] S. Manhas, R. Vashisht, P. S. Sandhu, and N. Neeru, "Reusability Evaluation Model for Procedure-Based Software Systems," *International Journal of Computer and Electrical Engineering*, vol. 2, no. 6, 2010.

[20] A. Kumar, "Measuring Software reusability using SVM based classifier approach," *International Journal of Information Technology and Knowledge Management*, vol. 5, no. 1, pp. 205–209, 2012.

[21] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy, "A probabilistic software quality model," in *27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 243–252.

[22] "ARiSA - Reusability related metrics," http://www.arisa.se/compendium/node38.html, [Online; accessed September 2017].

[23] D. P. Doane and L. E. Seward, "Measuring skewness: a forgotten statistic," *Journal of Statistics Education*, vol. 19, no. 2, pp. 1–18, 2011.