A Method Based on Naming Similarity to Identify Reuse Opportunities

Johnatan Oliveira, Eduardo Fernandes, Maurício Souza, Eduardo Figueiredo
Software Engineering Laboratory (LabSoft), Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
{johnatan.si, eduardofernandes, mrasouza, figueiredo}@dcc.ufmg.br

Abstract. *Software reuse is a development strategy in which existing software components are used to implement new software systems. There are many advantages of applying software reuse, such as minimization of development efforts and improvement of software quality. A few previous work propose methods for recommendation of reuse opportunities. In this paper, we propose a method for identification and recommendation of reuse opportunities based on the similarity of the names of classes. Our method, called JReuse, computes a similarity function to identify similarly named classes from a set of systems from a specific domain. The identified classes compose a repository with reuse opportunities. We also present a prototype tool to support the proposed method. We applied our method, through the tool, to 72 systems, collected from GitHub, of four different domains: accounting, restaurant, hospital, and e-commerce. In total, these systems have 1,567,337 lines of code and 12,598 classes. As a result, we observe that JReuse is able to identify and recommend the main, most frequent classes per domain.*

1. Introduction

Software reuse is a development strategy in which existing software components, called reusable assets, are used to implement new software systems [Krueger 1992]. Previous work study and indicate this strategy as an alternative to the traditional development, since reuse provides an increase of the software quality and a decrease of the development efforts by using previously developed, and sometimes already tested, software component [Mohagheghi and Conradi 2007, Morisio et al. 2002, Ravichandran and Rothenberger 2003].

The extraction of reusable assets is essential to support the software reuse activity by building repositories of reuse opportunities [Guo and Luqi 2000]. These methods may apply to different contexts related with software reuse, including the support of feature extraction for a software product line [Lee et al. 2004]. Many methods have been proposed in the literature to support the extraction of reuse opportunities from software systems [Caldiera and Basili 1991, Kawaguchi et al. 2006, Kuhn et al. 2007, Maarek et al. 1991, Ye and Fischer 2005].

There are different approaches used by the proposed methods to identify reuse opportunities, such as natural-language processing [Maarek et al. 1991], formal specifications [Caldiera and Basili 1991], machine learning [Kawaguchi et al. 2006], and other Information Retrieval (IR) approaches [Kuhn et al. 2007, Ye and Fischer 2005]. However, to the best of our knowledge, we did not find a method for extraction of reuse opportunities and reuse recommendation considering the most frequent source code elements such as classes from systems of the same domain.

This paper is an extension of previous work [Oliveira, J., et.al 2016] that proposes a method for extraction of reuse opportunities, called JReuse. Considering a set of software systems, JReuse aims to identify classes with similar names through a similarly analysis from different systems. Then, we are able to identify classes to recommend as reuse opportunities. We also present a prototype tool that applies the proposed method.

Additionally to our earlier contributions, we conduct an evaluation of our method through an experiment with 72 Java systems that belong to four different domains: accounting, restaurant, hospital, and e-commerce. We collected all these systems from GitHub[1]. As a result, we observe that JReuse is able to identify reuse opportunities using naming similarity analysis. That is, our method can provide meaningful classes for the analyzed domains, and these classes may represent reuse opportunities to developers of new systems from the respective domain.

The remainder of this paper is organized as follows. Section 2 presents background to support the study comprehension, in addition to related work. Section 3 proposes the JReuse method for reuse opportunities extraction, as a prototype tool that supports the proposed method. Section 4 presents an evaluation of the method. Section 5 describes the results obtained through the evaluation and discusses lessons learned. Section 6 presents threats to the study validity. Finally, Section 7 concludes the paper with suggestions for future work.

2. Background and Related Work

This section presents background information to support the comprehension of this study. In addition, it discusses related work. Section 2.1 overviews software reuse and its supporting techniques. Section 2.2 discusses related work that propose methods for identification of reuse opportunities from software systems.

2.1. Software Reuse

In software reuse, developers use previously implemented software components to develop new software systems [Krueger 1992]. The main goal of reuse is the improvement of software quality aspects followed by an increase of the development efficiency [Ravichandran and Rothenberger 2003]. There are many approaches to support reuse in software development. As an example, Krueger (1992) presents an extensive study regarding definitions, approaches, and application of software reuse.

There are two main approaches of software reuse: *ad hoc* and systematic reuse [Mohagheghi and Conradi 2007]. In the *ad hoc* approach, software reuse is applied in an opportunistic way, without planning. An example of *ad hoc* reuse is the use of random software code snippets extracted from the Web [Sojer and Henkel 2011]. In turn, the

[1] https://github.com

systematic reuse follows specific protocols and processes to provide the use of existing software components when developing new systems [Mohagheghi and Conradi 2007]. Moreover, there are two way to identify reuse opportunities: forward identification, in which software reuse is planned before the development of software systems; and reverse identification, in which reuse opportunities are identified from a set of existing software systems [Wang et al. 2005].

Previous work investigate advantages and drawbacks of systematic software reuse [Mohagheghi and Conradi 2007, Mohagheghi et al. 2004]. Mohagheghi et al. (2004) study the impacts of reuse on the software quality through an empirical study on large-scale software components. They conclude that reuse contributes positively in the software quality, since it provides software components with lower defect-density and higher stability when compared with non-reused components. Mohagheghi and Conradi (2007) provide a literature review on the impact of software reuse in the industrial development context. They list decrease of flaws, reduction of development efforts, and increasing productivity as the main advantages provided by software reuse.

Several studies in the literature propose supporting techniques for identification of reuse opportunities. For instance, natural language processing relies on lexical inspection of source code elements [Maarek et al. 1991]. In turn, formal specifications consists of conducting the analysis of software models and metrics [Caldiera and Basili 1991]. Finally, architectural style [Monroe and Garlan 1996] is a technique supported by the analysis of high-level component interaction, generally applied to software design and modeling; and machine learning that gathers different types of analysis, such as semantic categorization of software components [Kawaguchi et al. 2006].

2.2. Identification of Reuse Opportunities

Previous work investigate the identification of reuse opportunities from software systems [Inoue et al. 2005, Koziolek et al. 2013, Li et al. 2005, Mende et al. 2009, Michail and Notkin 1999, Oliveira et al. 2007, Ye and Fischer 2005]. As an example, Inoue et al. (2005) propose a graph-based technique to support the extraction of frequently used components in a given software component repository. The proposed technique relies on ranking components based on their usage by other components from the repository. The authors also present a supporting tool called SPARS-J, for analysis of Java classes and identification of reuse opportunities.

In turn, Koziolek et al. (2013) present a technique for identification of reuse opportunities based on domain analysis. The proposed technique aims to support the assessment of potential Software Product Line implementation by organizations. This technique encompasses feature modeling of the domain, comparison of systems in architectural level, and the extraction of reusable components. However, unlike JReuse, their technique does not compute similarity between names of classes with aim the identify reuse opportunities.

Li et al. (2005) present an approach for identification of reusable components from legacy systems. The proposed approach aims to support reengineering tasks, i.e. the implementation of new systems based on existing source code. For this purpose, the authors propose the generation of the Abstract Syntax Tree (AST) for analysis and extraction of modules and components as candidate for reuse. As a drawback, this approach lacks a prioritization of the identified reuse opportunities, i.e. the propose technique does not compute the relevance of the opportunities for recommendation to the user. On the other hand, JReuse computes a prioritization score for the identified reuse opportunities, based on the occurrence of the classes among the analyzed systems.

Mende et al. (2009) propose a tool to support software evolution and maintenance. For this purpose, the tool identifies similar methods along the source code and recommends them to the developer by merging the identified methods. The proposed tool computes code clones in method-level and uses the Levenshtein's algorithm for textual comparison of methods. As well as the mentioned technique, JReuse uses the Levenshtein's algorithm for textual comparison, but in the context of similarity computation in the level of classes.

Michail and Notkin (1999) propose CodeWeb, a tool to support the comparison of software libraries in terms of components, i.e. classes and methods, provided by these libraries. For this purpose, the tool performs naming similarity computation to identify similar classes and methods from a set of libraries. On the other hand, JReuse is able to identify reuse opportunities in both libraries and traditional software systems implemented in Java.

Oliveira et al. (2007) propose a method and a supporting tool for recommendation of reusable software components. The tool applies a technique called Automatic Identification of Software Components to identify candidate components for reuse. The tool, called Digital Assets Discoverer, performs static code analysis for identification of reuse opportunities. The tool, called Digital Assets Discoverer, performs *static code analysis* for identification of reuse opportunities. In this type of source code analysis, there is no requirement for the source code to run [Ramler et al. 2016]. Therefore, the static analysis is the opposite of the *dynamic analysis*, in which source code has to compile and run to be analyzed [Cornelissen et al. 2009]. In addition, the proposed tool provides an interactive graphic interface and data export. As a differential to this previous work, JReuse prioritizes the identified reuse opportunities.

Finally, Ye and Fischer (2005) present CodeBroker, a tool to support runtime identification of reusable software components. The proposed tool relies on information retrieval techniques. CodeBroker relies on search engines and Javadoc artifacts for code analysis. Our method, JReuse, performs static analysis of the source code and, therefore, does not provide code analysis in runtime. We decided to propose a method based on static analysis since we aim to analyze several systems at the same time and, therefore, the runtime analysis could be a significant limitation of our method. However, as aforementioned, JReuse provides the prioritization of reuse opportunities.

In this paper, we propose a method and a supporting tool, both called JReuse, to identify classes as candidates for reuse in systems from a domain. For this purpose, we apply lexical code analysis. Unlike related work, our method applies to two scenarios. First, to support the identification of reuse opportunities in software systems. Second, to guide users regarding the partial design of software systems under development, by recommending the most frequent entities that may compose the new system. Our method also ranks software entities identified as reuse opportunities by their frequency of appearance in different systems from the domain. We expect to support reuse by suggesting classes that are the most used in systems from a specific domain.

3. Proposed Method

This section explains in detail the proposed method for identification of reuse opportunities. Section 3.1 describes the similarity-based process applied by our method to identify reuse opportunities. Section 3.2 proposes our method and its steps. Finally, Section 3.3 presents a tool that implements our method.

3.1. Identifying Similarity

Previous work investigate the use of textual similarity in the context of source code analysis [Tian et al. 2014, Zhen et al. 2008]. There are many applications for similarity analysis in software systems, such as comparison of dialects, spell check, and plagiarism detection [Liu and Lu 2008]. Our propose method, JReuse, takes advantage of source code and similarity analysis for identification of reuse opportunities. As discussed in Section 2.2, our method uses the static source code analysis [Cornelissen et al. 2009] for identification of reuse opportunities based on the similarity between names of classes.

We conducted an *ad hoc* literature review in order to select algorithms that compute similarity between strings to be used by our method. For this purpose, we searched for the most popular similarity computation algorithms to find the one that fits our study purpose. After the literature review, we selected the Levenshtein's algorithm [Yujian and Bo 2007]. This algorithm is a similarity function used by our method to compute lexical similarity between names of classes from different systems. In short terms, given two strings *A* and *B*, the algorithm computes the number of changes required to turn *A* into *B*.

To identify similarly named classes, we adopted a threshold of 75% for the minimum similarity between two names of entities. The authors of this study derived empirically such threshold because some well-known naming conventions for classes may lead to similarly named entities that clearly represent different purposes. As an example, we obtain a similarity of 72% for the class names `Costumer` and `CostumerDAO`, observed by the authors as frequent names of classes in e-commerce systems. However, we intuitively expect that two classes with these names implement different functions, since DAO classes implement database persistence.

Table 1 presents some examples of class names and the respective similarity rate using the chosen algorithm. We checked each class name in the table to identify typos. As a result, we observed that they correspond to the exact terms identified by JReuse. In Table 1, we present eight matches between names of classes from two software systems: System *A* and System *B*. Each match has at least 75% of similarity rate between names of classes, in accordance to our empirical threshold. Note that our threshold covers, for instance, names of classes that vary from singular to plural (e.g., `Client` and `Clients`).

3.2. Proposed Method and Its Steps

A software domain is a set of systems that share a common set of functionalities, requirements, or terminology [Neighbors 1992, Pressman 2005]. Therefore, we expect that software systems within the same domain present lexical similarity with respect to

Table 1. Examples of similarity computation

System A	System B	Similarity Rate
Shopp**ing**Cart	ShoppCart	75%
OrderProduc**tId**	OrderProduc	78%
Orderservi**ce**	Orderservi	83%
Reviwe**s**	Reviwe	85%
Client**s**	Client	85%
CartControll**er**	CartControll	85%
Product**s**	Product	87%
Product**s**Controller	ProductController	94%

the names of classes. In this context, similarly named classes may contribute to the comprehension of the characteristics of systems from a given business domain [Cybulski and Reed 2000].

Considering this scenario, our study proposes JReuse, a method for identification of reuse opportunities from software systems. Our method is based on naming lexical similarity of classes. Given a set of software systems from the same domain, JReuse compares names of classes, in pairs, to identify common names among different systems. We believe that recurring names of classes may indicate reuse opportunities in a given domain. Furthermore, frequent names of classes may indicate common behaviors and requirements of these entities [Cybulski and Reed 2000].

In general, similarity rate is not enough for electing a class as a possible reuse opportunity [Ye and Fischer 2005]. We then consider the classes that are more frequent among the systems for recommendation. Note that, for instance, a name of class with matches in 10 different systems is more frequent than a name of class that matches in only 2 systems. Figure 1 illustrates the comparison between classes performed by JReuse. We provide a description of this process as follows.

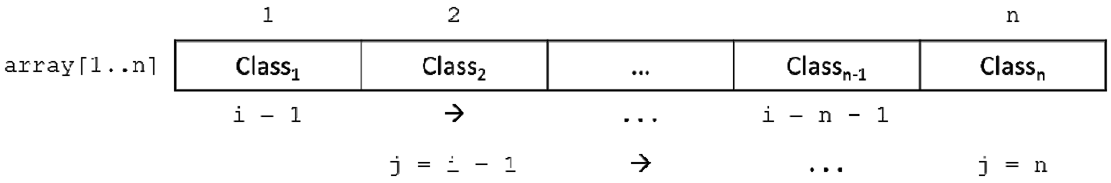


Figure 1. Steps to identify common classes

Consider `array[1..n]` an array of names of classes and two pointers `i = {1, .., n-1}` and `j = {2, .., n}`. For each `i`, we compare `array[i]` with `array[j]` for `j = {i+1, .., n}`. If `array[i]` is similar to `array[j]` with a minimum similarity rate of 75%, then the method registers a reuse opportunity. JReuse compares all classes from the set of systems to identify the similarly named classes. Since our method relies on lexical analysis, we do not perform synonymous analysis. Therefore, we say that classes such as `Client` and `Customer`, that may be similar semantically, are different entities in the source code.

Figure 2 presents the five steps performed by JReuse to identify reuse opportunities in a set of systems. We describe each step as follows.

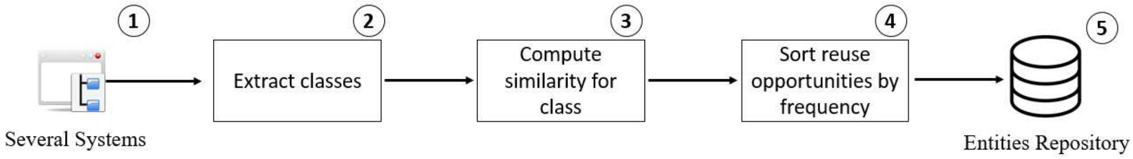

1
2
3
4
5
Several Systems
Extract classes
Compute similarity for class
Sort reuse opportunities by frequency
Entities Repository

Figure 2. Steps of the JReuse method

1. First, the JReuse method receives, as input, software systems from a data set provided by the user. These systems are supposed to belong to the same domain. Then, the method filters non-Java source files, discards every system projects that are for the Android platform, and extracts the names of classes from the Java source files.
2. After, the method extracts the names of classes to compute the similarity rates between pairs of classes from the systems. We highlight that JReuse does not compare names of classes from the same software system.
3. Then, JReuse compares the names of classes in pairs to identify names with at least 75% of similarity. Classes with similar names, called matches, are gathered and each class name receives a score that is the number of systems in which the class occurs. The higher the score, the more relevant may be the class regarding the analyzed domain.
4. After comparing names of classes and computing similarity, JReuse sorts the obtained results, in decreasing order, by the frequency of the identified reuse opportunities.
5. Finally, JReuse composes a repository of candidates to reuse opportunities with the identified classes. This repository may support developers in using such reuse opportunities to implement new systems of the analyzed domain.

3.3. Tool Support

To automate the proposed method, we developed a prototype tool that implements JReuse for Java software systems. We selected Java because (i) it is one of the most popular programming languages[2], (ii) there is an available Java parser to support source code analysis by the generation of an Abstract Syntax Tree (AST), and (iii) many studies have been investigating software reuse in Java systems. Through the Java parser, we may access the source code structure, Javadoc, and comments, for instance. It is also possible to change the AST nodes or create new ones to modify the source code. We also used the Eclipse Java Development Tools (JDT) parser to support the identification of similarly named classes.

The supporting tool performs three steps to identify reuse opportunities. We describe each step as follows.

1. First, the tool retrieves the name of all classes from a software system data set. This step is important to support the similarity computation among classes from different systems.

[2] http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015

2. After, the tool compares the names of classes, in pairs, to identify class names with at least 75% of similarity. Classes with similar name, that is, matches, are gathered and each class name receives a score that is the number of systems in which the class occurs. The higher a score, the more relevant may be the class with respect to the analyzed domain.
3. Finally, the tool persists the classes identified and extracted as reuse opportunities in a database.

JReuse provides an abstraction for the design organization of a system given a domain. In other words, the developers may use the reuse opportunities identified by JReuse to compose a partial design for any system that belongs to the analyzed domain in terms of frequent classes. For this purpose, the tool provides output as a CSV file. Each line of the file contains (i) the name of a class identified as reuse opportunity and (ii) the absolute path of the class. JReuse sorts the output file, in decreasing order, by the frequency of the identified reuse opportunities.

4. Evaluation Settings

This section describes an empirical evaluation of the method proposed in Section 3. For this purpose, we designed an exploratory study conducted in environment controlled based on guidelines of Wohlin et al. (2012). Since JReuse aims to identify the main reuse opportunities from software systems, our evaluation consists of analyzing the reuse opportunities identified by the proposed method.

Section 4.1 presents the study goal and research questions designed to guide our study. Section 4.2 describes the steps to evaluate our method through a prototype tool. Section 4.3 discusses the steps for collecting the target systems from GitHub. Section 4.4 describes the exclusion criteria to compose the final set of systems for analysis. Section 4.5 presents the strategy adopted by JReuse to compute the similarity between classes. Finally, Section 4.6 presents the data set used to evaluate the JReuse method.

4.1. Goal and Research Questions

In this study, our goal is to assess whether JReuse is able to identify frequent classes in a specific software domain. We are also interested in assessing the relevance of the results provided by our method. For this purpose, we chose four domains to evaluate, namely accounting, restaurant, hospital, and e-commerce. We also designed the following research questions (RQs) to guide our study.

RQ1. *What are the most frequent classes in software systems for each selected domain?*

Through RQ1, we are interested in investigating whether the most frequent identified classes are useful as recommendations for software systems for the respective domain. We expect that JReuse is able to provide a list of classes whose recommendations for reuse are relevant for the respective domains.

RQ2. *How distributed are the most frequent classes through systems per domain?*

With RQ2, we aim to understand to what extent the same class, identified as one of the most frequent classes, occur in different software systems from a given domain.

For instance, we aim to understand if the same name of class can occur in all analyzed systems, or in most of them.

4.2. Evaluation Steps

To evaluate JReuse in identifying reuse opportunities, we chose systems from domains of accounting, restaurant, hospital, and e-commerce. We chose such domains for the following reasons. First, software systems from these domains encompass several basic business features, such as user and product management. Second, there is a significant number of systems, per domain, available for download in GitHub. Third, from the perspective of the authors of this study, the four chosen domains are well-defined in terms of requirements and, therefore, we believe that it might be possible to find several reuse opportunities among systems of these domains.

We extracted the systems that compose our data set from GitHub repositories. We performed the selection of systems for the e-commerce domain in January 2015 and in May 2016 for the other domains. We selected the software systems based on the ranking of starred systems and system length in terms of storage space. In GitHub, stars are a meaningful measure for repository popularity among the platform users, and may support the selection of relevant systems for study.

Figure 3 presents the three study steps we followed to investigate the two research questions described in Section 4.1. We list the steps as follows.

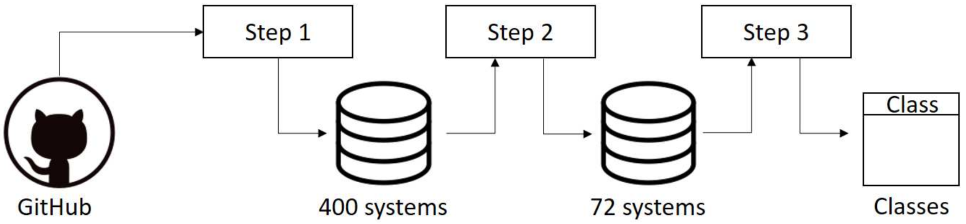


Figure 3. Steps of the exploratory study

Step 1: Automated Search. This step consists of colleting a set of software systems from GitHub for analysis (see Section 4.3).

Step 2: Exclusion Criteria. This step is a filtering of the collected systems, aiming to discard the inappropriate systems for analysis (see Section 4.4.).

Step 3: Class Name Similarity. This step consists of running JReuse to identify the reuse opportunities in class-level (see Section 4.5).

4.3 Automated Search

In order to clone automatically several systems from GitHub, we needed to define appropriate search strings per domain since there is a diverse terminology to represent the same software domain. For instance, we may refer to the *e-commerce* domain as *ecommerce*, without hyphenation. Thus, to collect the software systems that compose our data set, we developed an algorithm to clone GitHub repositories individually, with the respective systems, based on a specific search string per analyzed domain. Since the goal of our study is to identify reuse opportunities from different software systems, given large system sets per domain, we defined the search strings presented in Table 2.

Table 2. Search string per domain

Domain	Search String
Accountancy	*Accountancy* OR *Accounting*
Restaurant	*Restaurant* OR *Eatery* OR *Restaurants*
Hospital	*Hospital* OR *Infirmary* OR *Lazaretto*
E-Commerce	*E-Commerce* OR *Ecommerce* OR *Electronic Commerce*

4.4 Exclusion Criteria

We did a rigorous and transparent selection of the target systems, and attempted to minimize the risk of bias due to process of mined of projects from GitHub to a minimum by applying strict exclusion criteria. Table 3 presents the exclusion criteria applied in the selected systems. We collected 400 Java systems from GitHub, 100 for each domain in order descending sorted by stars. We then discarded systems according to the four following exclusion criteria.

Table 3. Exclusion criteria applied to the data set

Domain	Discarded Systems per Exclusion Criteria				Selected System
	Non-Java	Android	< 1 KLOC	Not English	
Accounting	12	17	51	9	11
Restaurant	4	27	53	3	13
Hospital	7	24	40	16	13
E-Commerce	21	3	20	21	35
All	**44**	**71**	**164**	**49**	**72**

First, non-Java software systems, since GitHub do not verify automatically the main programming languages of the systems. Second, Java projects developed for Android platform, because Android systems tend to have a different architectural design and code implementation when compared with traditional Java systems. Third, systems with less than 1,000 lines of code (LOC). Fourth, systems written in other languages rather than English, since our method relies on a lexical similarity technique and, then, natural language may affect significantly the results provided by our method.

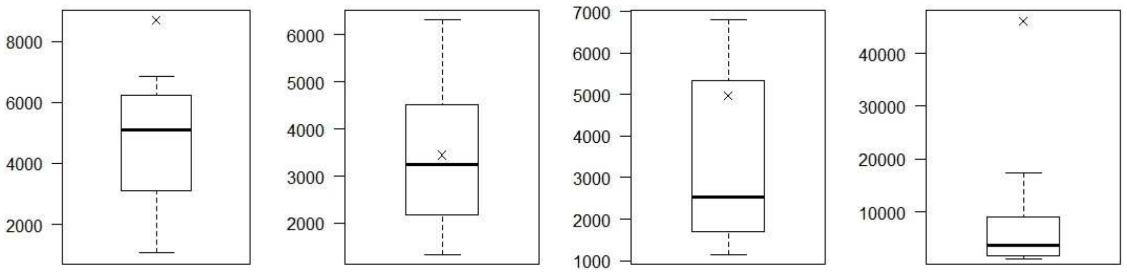
4.5. Class Name Similarity

From each selected domain as described in the previous steps, we performed analysis through JReuse. To identify and extract reuse opportunities, we executed the tool that provides support the developed method for 72 collected software systems from GitHub. These systems were submitted to JReuse for extraction of reuse opportunities. JReuse compares the names of classes in pairs to identify names with at least 75% of similarity. Classes with similar names, called matches, are gathered and each class name receives a score that is the number of systems in which the class occurs. The higher the score, the more relevant may be the class regarding the analyzed domain. After the automated analysis for each domain, JReuse provided a list with the most frequent classes that occur in the domain.

4.6 Data Set

The systems that compose our data set were retrieved from GitHub. For each selected system, we considered only the last release. This process was necessary to discard different versions of the same system, which probably contain several classes with similar names. Finally, we obtained 72 Java systems for evaluation of the JReuse method, as indicated in Table 4.To better characterize systems in the four domains, Figures 4 and 5 presents software metrics for systems per domain: LOC and number of classes (NOC), respectively. We plotted twelve boxplots, one for each metric. However, because of the heterogeneity of the sample of our data set, we decided to eliminate "outliers" for each metric. Therefore, all boxplots presented a brief overview of each analyzed domain.

Let us consider Figure 4 in the following analysis of LOC. In this figure, we represent the mean of each distribution with "X". Table 4 provides addition descriptive data, i.e. the data that compose the boxplots from Figure 4. With respect to the accounting domain, we observe that the mean of LOC for the systems is 8,690. Moreover, the median is 5,112, i.e., half of the accounting systems has at least 4 KLOC. That is, a significant number for analysis and identification of reuse opportunities.



(a) Accounting **(b) Restaurant** **(c) Hospital** **(d) E-commerce**

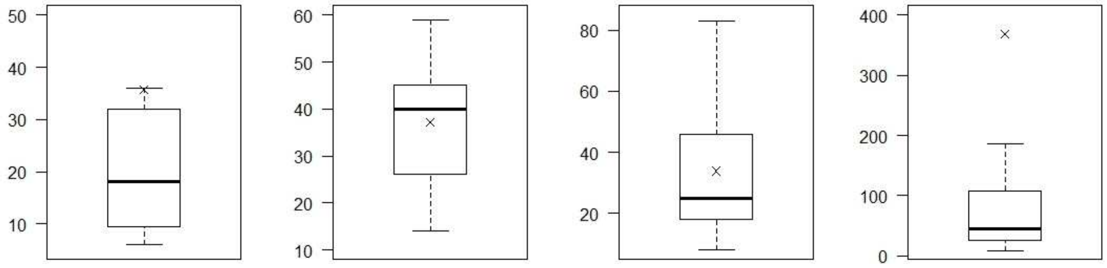
Figure 4. LOC of the systems per domain

Table 4. Descriptive analysis of LOC per domain

Domain	1st Quantile	Median	3st Quantile	Mean	Std. Dev.
Accounting	3,112	5,112	6,229	8,690	11,952.08
Restaurant	2,187	3,256	4,519	3,447	1,527.15
Hospital	1,700	2,534	5,346	4,964	6,223.94
E-Commerce	1,805	3,730	8,691	46,100	107,045.3

Regarding the restaurant domain, the mean of LOC is 3,447. In addition, the median is 3,256. Again, we conclude that these systems have a significant LOC for analysis. For the hospital domain, the mean is 4,964 and the median is 2,534 of LOC. Although these values are smaller than the obtained values for the other domains, it remains significant for the study. Finally, with respect to the e-commerce domain, we observe a mean LOC of 46,100 and a median of 3,730. In general, systems from this domain have the highest numbers of LOC and, therefore, they may have several reuse opportunities.

With respect to the following analysis of NOC, consider Figure 5. In this figure, we represent the mean of each distribution with "X". Table 5 provides addition descriptive data, i.e. the data that compose the boxplots from Figure 5. Regarding the accounting domain, note that the mean of NOC for the systems is 35.73. Furthermore, the median is 18, i.e., half of the accounting systems has at least 18 classes. This number is significant for analysis because we are interested in finding similarly named classes within a pairwise comparison. Therefore, we expect a comparison of 18 * 18 = 324 pairs that may be reuse opportunities.



(a) Accounting (b) Restaurant (c) Hospital (d) E-commerce

Figure 5. NOC of the systems per domain

Table 5. Descriptive analysis of NOC per domain

Domain	1st Quantile	Median	3st Quantile	Mean	Std. Dev.
Accounting	9.5	18	32	35.73	48.01
Restaurant	26	40	45	37.23	14.35
Hospital	18	25	46	33.85	24.19
E-Commerce	26	45.5	100.2	368	819.72

Regarding the restaurant domain, the mean of NOC is 37.23. In addition, the median is 40. Again, we conclude that these systems have a significant NOC for analysis. For the hospital domain, the mean is 33.85 and the median is 25 of NOC. Finally, with respect to the e-commerce domain, we observe a mean NOC of 368.9 and a median of 45.5. In general, systems from this domain has the highest numbers of NOC and, therefore, there is a significant possibility of identifying reuse opportunities.

5. Results and Discussion

In this section, we present and discuss the main results of our empirical evaluation with JReuse. Section 5.1 presents the most frequent classes identified by JReuse per domain. Section 5.2 focus on the distribution of the most frequent classes through the systems of each domain. Finally, Section 5.3 provides an overview and discusses lessons learned.

5.1 Frequent Classes per Domain

In a first moment, we present the results regarding the most frequent classes per analyzed domain. Therefore, we answer RQ1 as follows.

***RQ1**. What are the most frequent classes in software systems for each selected domain?*

In this study, we analyzed the frequency of similarly named classes for the systems of each domain. Table 6 presents software metrics for systems per domain: lines of code (LOC) and number of classes (NOC). This table categorizes NOC in two types: (i) analyzed, i.e., the number of entities analyzed by the tool and (ii) recommended, that is, entities identified by the tool as reuse opportunities. In general, from Table 6 we observe that JReuse identified good results as candidates for reuse opportunities. For instance, for domain e-commerce, JReuse identified 75 classes as reuse opportunities.

In order to present and discuss the most frequent classes extracted as reuse opportunities, we considered the following exclusion criteria of classes. For each domain, we discarded classes that occur in a maximum of two different systems.

Table 6. Software metrics computed for the systems per domain

Domain	Number of Systems	LOC	Number of Classes	
			Analyzed	Recommended
Accounting	11	95,588	493	25
Restaurant	13	44,813	484	17
Hospital	13	65,297	446	21
E-Commerce	35	1,567,337	12,598	75

We made this decision because our method compares classes in pairs and, then, three occurrences may not be significant to a reuse recommendation. To validate the lists of most frequent classes per domain, we submitted such lists to a group of experts in the e-commerce domain (also called domain experts). Four software engineers of a Software Engineering laboratory compose this group of experts. The group was responsible for analyzing the relevance of the results provided by JReuse in the context of each domain.

Tables 7, 8, 9, and 10 present classes identified as reuse opportunities for e-commerce, accounting, restaurant, and hospital, respectively. We selected only the classes with at least 15%[3] of occurrence in the systems of the respective domain. Each table has a "Domain-Specific" field. This field indicates the viewpoint of the domain experts regarding a given class to be specific for the analyzed domain. We use three symbols to represent the domain experts viewpoint in the tables. The (✔) symbol indicates that the domain experts agreed that the class is specific for the domain under analysis. The (X) symbol indicates that the domain experts disagreed that the class is indicated for the domain. Finally, a blank field (i.e. Unconfirmed) indicates that the domain experts did not converge to a specific opinion on the class. Moreover, each table has a "Labels" filed to inform the level of relevance of the class identified by JReuse as reuse opportunity.

Scale to Indicate the Level of Relevance of the Entities Identified. To support the identification of the most recommended classes for each domain, we defined scales to represent levels of recommendation for the classes. These scales rely on the frequency of the classes identified as reuse opportunity. The weak label (from 0% to <50%) indicates that the class is weakly or moderately recommended as a reuse opportunity given a domain. In turn, the strong label (from 50% to 100%) indicates that the class is highly recommended as a reuse opportunity.

Table 7 presents results with respect to the accounting domain. In our analysis, we discarded 161 classes because they presented less than 15% of frequency among systems. For this domain, the classes from `Users` to `TransactionManager` belong to the strong label and, therefore, they are the highly recommended classes for accounting systems. On the other hand, the domain experts did not consider the classes `Users`, `DatabaseConnection`, and `Util` as specific classes for the accounting domain. In addition, the classes from `AddFinancialsAction` to `RawMaterial` belong to the weak label. The remainder classes have exactly two or three occurrences in different systems from the accounting domain. Therefore, they are weakly recommended and were omitted from this table.

[3] The percentage depends on the number of systems under analysis given a domain

Table 7. Classes with at least 15% of occurrences in the accounting domain

Label	Class	Frequency	% of Systems	Domain Specific
Strong	Users	13	100%	X
	DatabaseConnection	13	100%	X
	CashFlow	11	85%	✓
	Util	10	77%	X
	BalancesAssets	9	69%	✓
	CashBanks	9	69%	✓
	ShareholderEquity	9	69%	✓
	BalancesLiabilities	8	62%	✓
	ChartAccounts	8	62%	✓
	AccountingMovement	8	62%	✓
	AccountsReceivable	8	62%	✓
	AccountsPayable	6	46%	✓
	Transactions	7	54%	✓
	Log	7	54%	X
	FinancialReportsPoeHelper	7	54%	X
	InventoryManager	7	54%	✓
	TransactionManager	7	54%	✓
Weak	AddFinancialsAction	6	46%	✓
	Accounts	6	46%	✓
	FeaturesAnalysis	6	46%	✓
	RawMaterial	6	46%	✓

Key: Agree (X), Disagree (✓), and Unconfirmed (blank field)

Table 8 presents results for the restaurant domain. We discarded 13 classes since they presented less than 15% of frequency among systems. The classes `Login` and `User` belong to the strong label. From the domain experts viewpoint, these classes are not specific classes in the domain. However, they are relevant in restaurant systems. The classes from `Client` to `Order` belong to the strong label and are relevant for the restaurant domain from the domain experts viewpoint. Note that, for many of the classes identified by JReuse, the experts considered such classes as relevant reuse opportunities for restaurant systems, even in the weak label, such as `RestaurantMenu`, `Delivery`, and `Customer`.

Table 8. Classes with at least 15% of occurrences in the restaurant domain

Label	Class	Frequency	% of Systems	Domain Specific
Strong	Login	10	77%	X
	User	10	77%	X
	ConnectionManager	9	70%	X
	Client	9	70%	✓
	Table	8	62%	✓
	PaymentType	8	62%	✓
	Dish	8	62%	✓
	Employee	7	54%	✓
	Order	7	54%	✓
Weak	RestaurantMenu	6	47%	✓
	Delivery	6	47%	✓
	ItemOrdered	6	47%	✓
	Customer	4	31%	✓

Key: Agree (X), Disagree (✓), and Unconfirmed (blank field)

Consider Table 9 for analysis of the hospital domain. We discarded a set of 25 classes because they have occurred in less than 15% of the systems. Observe that the classes from `Patient` to `Microbiology` belong to the strong label and, therefore, they are highly recommended classes as reuse opportunities. Note that, from the viewpoint of the domain experts, the three most frequent classes are specific from hospital systems. In fact, classes such as `Patient` and `Doctor` are meaningful in the domain. In addition, classes from `PatientCondition` to `OperationsWithCards` are from the weak label. Finally, the remainder classes have less than 10% of the occurrences.

Table 9. Classes with at least 15% of occurrences in the hospital domain

Label	Class	Frequency	% of Systems	Domain Specific
Strong	Patient	13	100%	✓
	Doctor	13	100%	✓
	Disease	11	85%	✓
	User	10	77%	X
	Login	9	69%	X
	Diagnose	9	69%	✓
	Symptoms	9	69%	✓
	PatientDisease	8	62%	✓
	HealthPlan	8	62%	✓
	Immunology	8	62%	✓
	Haematology	8	62%	✓
	Medication	7	54%	✓
	Surgery	7	54%	✓
	MedicalRecords	7	54%	✓
	TypePayment	7	54%	
	Microbiology	7	54%	✓
Weak	PatientCondition	6	46%	✓
	LaboratoryExams	6	46%	✓
	Log	6	46%	X
	HistoPathology	6	46%	✓
	Connection	6	46%	X
	Paycash	5	38%	
	Util	5	38%	X
	OperationsWithCards	3	23%	

Key: Agree (X), Disagree (✓), and Unconfirmed (blank field)

Finally, consider Table 10 for the analysis and discussion regarding the e-commerce domain. For this domain, we discarded 3,573 classes because they were present in less than 15% of the analyzed systems. Note that the classes `Product` to `ClientDao` belong to the strong label. That is, they are highly recommended classes for e-commerce systems, because they were present in more than 50% of the analyzed systems. In addition, the classes `Item` to `ShoppingCartService` are the weakly recommended classes. As aforementioned, we omitted the classes with less than 15% of the occurrences.

In general, we observed that the classes identified by JReuse are relevant to their respective system domains, from the viewpoint of the domain experts. Although some classes in the weak label are considered relevant, most of the group agreement was related to classes in the strong label. Therefore, our data suggests that our method is able to identify interesting candidates to reuse. We then are able to assess how distributed are such classes among different systems from the same domain.

Table 10. Classes with at least 15% of occurrences in the e-commerce domain

Label	Class	Frequency	% of Systems	Domain Specific
Strong	Product	28	80%	✓
	PaymentType	24	69%	✓
	Client	20	58%	✓
	ProductDao	18	52%	✓
	ClientDao	18	52%	✓
Weak	Item	17	49%	✓
	ShoppingCart	17	49%	✓
	User	17	49%	X
	Customer	14	40%	✓
	Category	12	35%	✓
	ProductService	10	29%	✓
	Order	9	26%	✓
	LoginController	7	20%	X
	UserDao	6	18%	✓
	ProductServiceImpl	6	18%	✓
	ShoppingCartController	6	18%	✓
	OrderedProduct	5	15%	✓
	ShoppingCartService	5	15%	✓

Key: Agree (X), Disagree (✓), and Unconfirmed (blank field)

5.2 Distribution of Frequent Classes

After presenting the most frequent classes from systems of each system domain under analysis, we present the results with respect to the distribution of classes through systems from the same domain. Therefore, we answer *RQ2* as follows.

RQ2. *How distributed are the most frequent classes through systems per domain?*

Figure 6 presents the top-ten most frequent classes for the accounting domain, based on the number of occurrences for each class. Such classes are, in decreasing order of frequency, `Users`, `DatabaseConnection`, `CashFlow`, `Util`, `BalancesAssets`, `CashBanks`, `ShareholderEquity`, `BalancesLiabilities`, `ChartAccounts`, and `AccountingMovement`. We observe that, although only `CashFlow` is specific to the given domain from the viewpoint of the domain experts, all classes from this label are meaningful in accounting systems.

Regarding the restaurant domain analysis, Figure 7 presents the top-ten classes with the highest occurrences, namely `Login`, `User`, `ConnectionManager`, `Client`, `Table`, `PaymentType`, `Dish`, `Employee`, `Order`, and `RestaurantMenu`. These classes have a high to medium level for recommendation according to our scale defined in Section 5.1. The classes with the highest occurrences in this domain are `Login` and `User`, respectively. Both classes were present in 77% of the analyzed information systems. Nevertheless, they are not specific classes of restaurant systems. In turn, JReuse was able to identify several frequent classes such as `Client`, `Table`, `PaymentType`, and `Dish`.

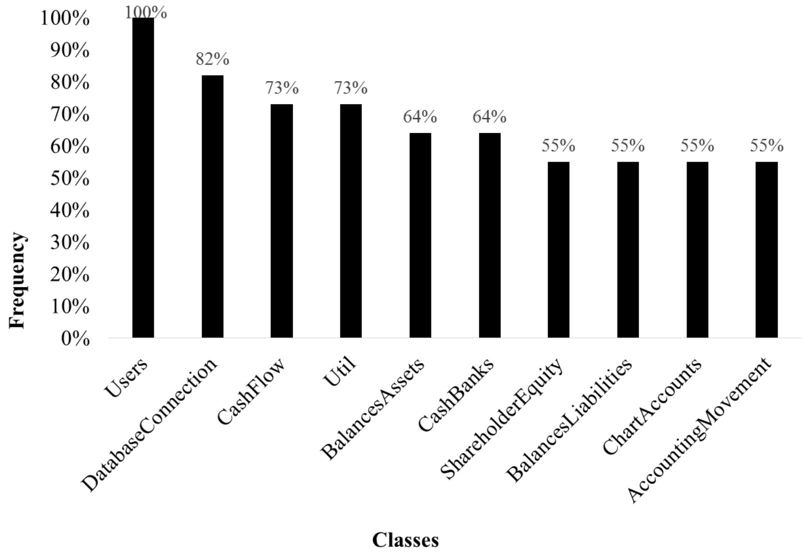


Figure 6. Distribution of frequent classes through accounting systems

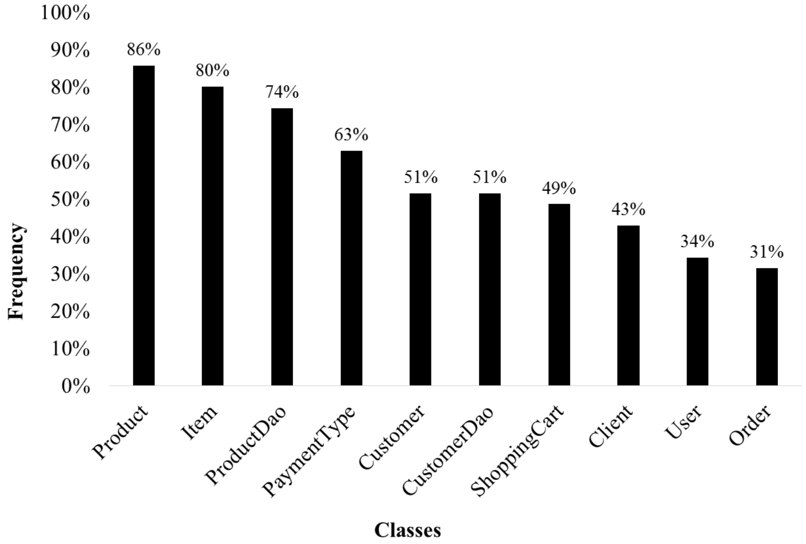


Figure 7. Distribution of frequent classes through restaurant systems

Figure 8 presents the most frequent classes identified for the hospital domain, in decreasing order of frequency. For the 13 systems we collected from this domain, JReuse extracted some relevant entities, such as `Patient`, `Doctor`, and `Disease`, from the domain experts agreement. The classes presented in this figure belong to the strong label. Note that the classes `Patient` and `Doctor` were present in 100% of the evaluated systems. Similarly, to the other domains, JReuse identified some classes that are generic, i.e. classes expected in systems from other domains, such as `User` (77%).

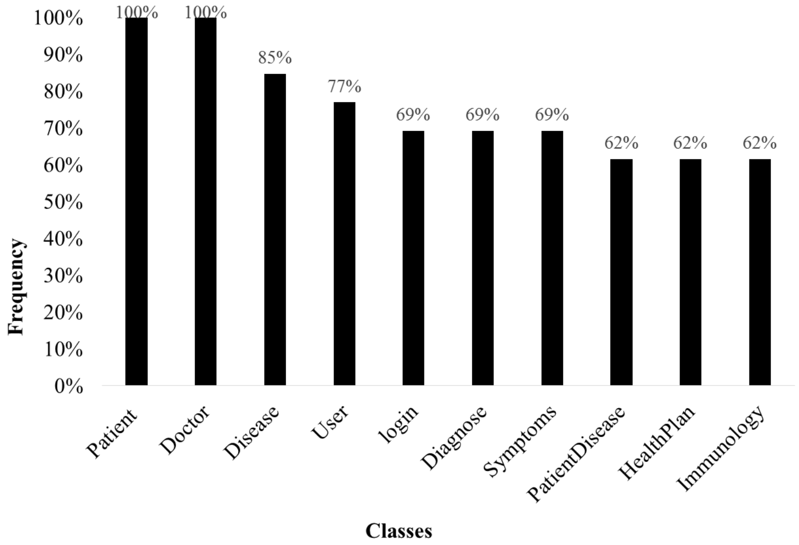


Figure 8. Distribution of frequent classes through hospital systems

Finally, Figure 9 presents the top-ten most frequent classes for e-commerce systems. We sorted the classes in decreasing order of frequency. The most frequent entities are, respectively, `Product`, `PaymentType`, `Client`, `ProductDao`, `ClientDao`, `Item`, `ShoppingCart`, `User`, `Customer`, and `Category`. Note that, according to the domain experts the classes `Product`, `Payment`, `ShoppingCart`, `Customer`, and `Client` are elementary entities, i.e. we expect them in an e-commerce system. In turn, although `User` is one of the most frequent classes identified by JReuse (49% of the systems contain this class), `User` is not specific of the e-commerce domain. However, this class is meaningful for information systems in general.

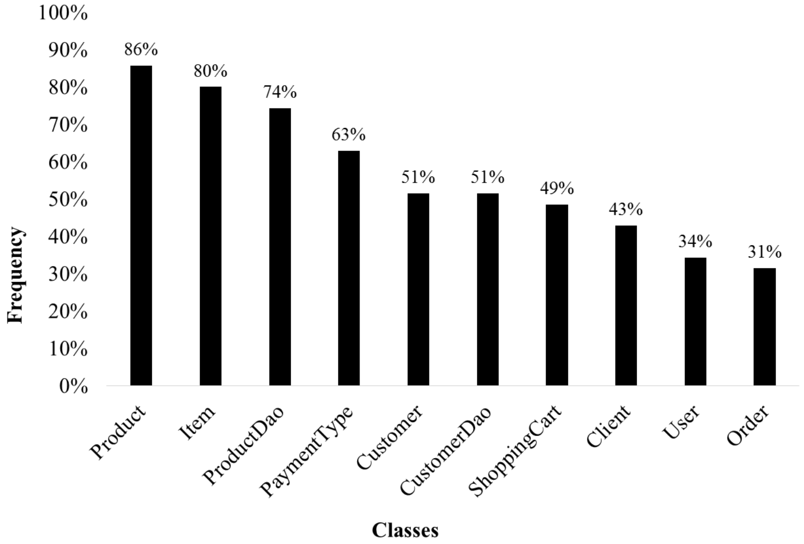


Figure 9. Distribution of frequent classes through e-commerce systems

5.3 Lessons Learned

In this study, we learned a lot regarding interesting research topics such as software reuse, reuse opportunities identification, and recommendation systems. For this propose, we take as an example the e-commerce domain, especially by the popularity and size of these systems on GitHub. We discuss some of the main lessons learned with support of the following questions.

How much a lexical analysis may support the identification of reuse opportunities assets? As discussed in Section 2, there are many approaches to support software reuse in literature. Lexical analysis is a simple one. However, as pointed by the results of Section 5, it may be effective to identify reuse opportunities in systems from a single domain. Moreover, we initially conceived our method to gather classes with names that are semantically similar. However, through our study we identified some occurrences of similar entities in an intuitive fashion that do not represent the same real-world concept.

In our exploratory study, which was conducted in a controlled environment (see Section 4), we found for instance, frequent classes such as `Client` and `Costumer` have distinct behaviors although intuitively they represent the same real-world abstraction. Some classes named as `Client` implement a simplistic system client, which register data. In turn, `Costumer` classes generally implement system clients with more robust features, such as data management. Therefore, we conclude that lexical analysis performs satisfactorily to identify reuse opportunities at least in this domain.

Are names of classes suitable to the entities they represent in a business domain? We discuss in Section 3.2 that names of classes may be useful for reuse opportunities identification. In fact, we observed that naming similarity identification might support reuse opportunities identification. However, to retrieve similarly named classes may be uninteresting if they are not representative in a specific domain. Section 5 highlights the identified classes that fit to e-commerce domain. These entities are the most frequent that our tool detected.

Therefore, we believe that names of entities are, in general, sufficiently representative. Moreover, we observed in this study that our method is able to identify reuse opportunities in randomly mined systems from GitHub, provided by different development teams. Therefore, we expect to obtain results that are even more relevant in the context of a specific organization.

How to apply our reuse opportunities identification tool in a reuse recommendation system? Classes are elementary entities of object-oriented software systems. Knowing this type of source code entities, we are able to describe the architecture of a system. Therefore, with results provided by our tool, we see an opportunity for reuse recommendation through software modeling using class diagrams, for instance.

To the best of our knowledge, we have not found many recent studies with respect to reuse opportunities identification, supported by tools for this activity, and methods to support the building of reuse repositories with similar approach. Therefore, as an interesting research topic, we lack more quantitative data to measure and compare different techniques that support software reuse.

6. Threats to Validity

We based our study on related work to support the method definition and the development of the supporting tool, both called JReuse. Regarding the evaluation of our method and tool, we conducted a careful empirical study to assess effectiveness of the method in identifying reuse opportunities. However, some threats to validity may affect our research findings. We discuss the main threats and respective treatments as follows, based on the categories presented by Wohlin et al. (2012).

Construct Validity. Before running our reuse opportunities identification method, we conducted a careful filtering of information systems from GitHub repositories. However, some threats may affect the correct filtering of systems, such as human factors that wrongly lead to discard a valid system for evaluation. Considering the exclusion criteria for selection of systems (see Section 4.2), we implemented an algorithm to automate this process and, then, discard inappropriate systems for analysis. However, we may have discarded relevant software systems by using our algorithm, such as systems misidentified as non-Java systems.

Internal Validity. We conducted a lexical classification of entities that are prone to some threats. To treat this possible problem, we selected a sample of 10 e-commerce systems from our data set, with diversified number of entities. Then, we manually identified the names of entities from source code to find synonyms. We compared our manual results with the results provided by the tool and observed a loss of 10% in synonym terms identified through the automated process.

Conclusion Validity. After running our identify tool, we gathered manually classes that seemed to represent the same real-world object. For instance, we considered classes named as *Client* and `Costumer` as the same type of class. However, this process is subjective and human factors may have affected it. In this first exploratory study, we decided for not unifying terms (for instance, `Customer` and `Client`) in the quantitative analysis.

External Validity. We evaluated our method with a set of 72 systems, extracted from GitHub. Considering that they may not represent the four analyzed domains, our findings may be not be generalized. Furthermore, we evaluated only four system domains, accounting, restaurant, hospital, and e-commerce. However, the collected systems are the most popular on GitHub that is a largely used platform. Finally, we evaluated systems implemented only in Java programming language. Although it is one of the most popular languages worldwide, our results may not generalize to other programming languages.

7. Conclusion and Future Work

In a previous work, we proposed JReuse, a method to identify reuse opportunities from software system of a specific domain. Our method relies on lexical analysis to compare names of classes and identify the most frequent ones. In addition, we present a prototype tool that implements the method for analysis of Java software systems. Finally, we conduct a preliminary evaluation of our method with 38 software systems of the e-commerce domain, collected from GitHub. Given the limitations of our previous effort to evaluate the proposed method, this paper extends our earlier contributions with a more extensive evaluation of JReuse in the context of four different software domains: accounting, restaurant, hospital, and e-commerce. For this purpose, we analyzed a large set of 72 Java systems also collected from GitHub.

We evaluate JReuse through an exploratory study conducted in a controlled environment, considering two aspects. First, we assess whether the most frequent classes provided by our method are relevant for the respective domains. Second, we assess the distribution of frequent names of classes among different systems of the same domain. Our findings suggest that our method was able to suggest relevant classes for systems from the four analyzed domains. The opinion of a group of domain experts reinforces our findings. This group validated our results and indicated high rates of agreement with respect to the relevance of the reuse opportunities provided by JReuse. In addition, our data suggest that the most frequent classes provided as candidates for reuse were present in a significant number of different systems of the respective analyzed software domains.

As future work, we intend to enhance JReuse to suggest source code to developers based on the most frequent classes identified by the proposed method. In addition, we aim to combine lexical with semantic analysis to improve the identification of reuse opportunities. For instance, the semantic analysis may support analysis of synonyms and improve our results of identification. In addition, we may explore alternative techniques for similarity computation. We also intend to implement our method targeting other object-oriented programming languages.

References

Caldiera, G. and Basili, V. R. (1991). Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2):61–70.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R. (2009). A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering (TSE)*, 35(5):684–702.

Cybulski, J. and Reed, K. (2000). Requirements Classification and Reuse: Crossing Domain Boundaries. In *Proceedings of the 6th International Conference on Software Reuse (ICSR),* pages 190–210.

Guo, J. and Luqi (2000). A Survey of Software Reuse Repositories. In *Proceedings of the 7th International Conference and Workshops on the Engineering of Computer Based Systems (ECBS),* pages 92–100.

Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M., and Kusumoto, S. (2005). Ranking Significance of Software Components based on Use Relations. *IEEE Transactions on Software Engineering (TSE),* 31(3):213–225.

Kawaguchi, S., Garg, P., Matsushita, M., and Inoue, K. (2006). MUDABlue: An Automatic Categorization System for Open Source Repositories. *Journal of Systems and Software (JSS),* 79(7):939-953.

Koziolek, H., Goldschmidt, T., Gooijer, T., Domis, D., and Sehestedt, S. (2013). Experiences from Identifying Software Reuse Opportunities by Domain Analysis. In *Proceedings of the 17th International Software Product Line Conference (SPLC),* pages 208–217.

Krueger, C. (1992). Software Reuse. *ACM Computing Surveys (CSUR),* 24(2):131–183.

Kuhn, A., Ducasse, S., and Gírba, T. (2007). Semantic Clustering: Identifying Topics in Source Code. *Information and Software Technology (IST),* 49(3):230–243.

Lee, J., Kang, K. C., and Kim, S. (2004). A Feature-Based Approach to Product Line Production Planning. In *Proceedings of the 3rd International Conference on Software Product Lines (SPLC)*, pages 183–196.

Li, J., Zhang, Z., and Yang, H. (2005). A Grid Oriented Approach to Reusing Legacy Code in ICENI Framework. In *Proceedings of the 3rd International Conference on Information Reuse and Integration (IRI),* pages 464–469.

Liu, H. and Lu, R. (2008). Word Similarity based on an Ensemble Model using Ranking SVMs. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT),* pages 283–286.

Maarek, Y., Berry, D., and Kaiser, G. (1991). An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering (TSE),* 17(8):800–813.

Mende, T., Koschke, R., and Beckwermert, F. (2009). An Evaluation of Code Similarity Identification for the Grow-and-Prune Model. *Journal of Software Maintenance and Evolution: Research and Practice,* 21(2):143–169.

Michail, A. and Notkin, D. (1999). Assessing Software Libraries by Browsing Similar Classes, Functions and Relationships. In *Proceedings of the 21st International Conference on Software Engineering (ICSE),* pages 463–472.

Mohagheghi, P. and Conradi, R. (2007). Quality, Productivity and Economic Benefits of Software Reuse: A Review of Industrial Studies. *Empirical Software Engineering (ESE),* 12(5):471–516.

Monroe, R. and Garlan, D. (1996). Style-Based Reuse for Software Architectures. In *Proceedings of the 4th International Conference on Software Reuse (ICSR),* pages 84–93.

Morisio, M., Ezran, M., and Tully, C. (2002). Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering (TSE),* 28(4):340–357

Neighbors, J. (1992). The Evolution from Software Components to Domain Analysis. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE),* 2(3):325–354.

Oliveira, J. Fernandes, E., Souza, M., and Figueiredo, E. (2016). A Method Based on Naming Similarity to Identify Reuse Opportunities. *In Proceedings of the XII Brazilian Symposium on Information Systems (SBSI).*

Oliveira, M., Goncalves, E., and Bacili, K. (2007). Automatic Identification of Reusable Software Development Assets: Methodology and Tool. In *Proceedings of 5th the International Conference on Information Reuse and Integration (IRI),* pages 461–466.

Pressman, R. (2005). Software Engineering: A Practitioner’s Approach. McGraw-Hill Education.

Ramler, R., Moser, M., and Pichler, J. (2016). Automated Static Analysis of Unit Test Code. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 25–28.

Ravichandran, T. and Rothenberger, M. (2003). Software Reuse Strategies and Component Markets. *Communications of the ACM*, 46(8):109–114.

Sojer, M. and Henkel, J. (2011). License Risks from Ad Hoc Reuse of Code from the Internet. *Communications of the ACM*, 54(12):74–81.

Tian, Y., Lo, D., and Lawall, J. (2014). SEWordSim: Software-specific word similarity database. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 568–571.

Wang, Z., Xu, X., and Zhan, D. (2005). A Survey of Business Component Identification Methods and Related Techniques. *International Journal of Information Technology*, 2(4):229–238.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). Experimentation in Software Engineering. Springer Science & Business Media.

Ye, Y. and Fischer, G. (2005). Reuse-Conducive Development Environments. *Automated Software Engineering (ASE)*, 12(2):199–235.

Yujian, L. and Bo, L. (2007). A Normalized Levenshtein Distance Metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 29(6):1091–1095.

Zhen, Z., Shen, J., and Lu, S. (2008). WCONS: An Ontology Mapping Approach based on Word and Context Similarity. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pages 334–338.