Mining Reusable Software Components from Object-Oriented Source Code of a Set of Similar Software

Anas Shatnawi, Abdelhak-Djamel Seriai
UMR CNRS 5506, LIRMM
Université Montpellier 2 Sciences et Techniques
Place Eugène Bataillon, Montpellier, France
{shatnawi, seriai}@lirmm.fr

Abstract

One of the most important approaches that support software reuse is Component Based Software Engineering (CBSE). Nevertheless the lack of component libraries is one of the major obstacles to widely use CBSE in the industry. To help filling this need, many approaches have been proposed to identify components from existing object-oriented software. These approaches identify components from singular software. Therefore the reusability of these components may be limited. In this paper, we propose an approach to mine reusable components from a set of similar object-oriented software, which were developed in the same domain, ideally by the same developers. Our goal is to enhance the reusability of mined components compared to those mined from single software. In order to validate our approach, we have applied it onto two open source Java applications of different sizes; small and large-scale ones. The results show that the components mined from the analysis of similar software are more reusable than those which are mined from single ones.

Keywords: software component, similar software, mining, reuse, object-oriented, source code, reverse engineering

1. Introduction

It is admitted that reuse improves the software quality and productivity [1]. Component Based Software Engineering (CBSE) is considered as one of the most important approaches supporting software reuse [1, 2, 4]. Nevertheless, one of the major limitations against widely use of CBSE is the lack of component libraries [12]. Therefore, mining reusable components from existing software is an efficient way to supply component libraries. Otherwise, as software components are admitted as more reusable entities than object-oriented ones [12], many approaches have proposed to identify components from existing object-oriented software [3, 5, 6, 7]. These approaches proposed to mine components by analyzing single software. As a result, the mined components may be useless in other software and consequently their reusability is not guaranteed. In fact the probability of reusing a component in new software is proportional to the number of software that has already used it [18]. Moreover software companies often find themselves in the situation where they have developed many software in the same domain, but with functional or technical variations [8]. In most cases, each software variant is developed by adding some variations to an existing software to meet the requirements of a new need. Thus in this paper, we propose an approach to mine reusable components from a set of similar object-oriented software[1] which were developed in the same domain, ideally by the same developers. The goal is to analyze the source code of these software to identify pieces of code that may form reusable components. Our motivation is that components mined from the analysis of several existing software will be more useful (reusable) for the development of new software than those mined from singular ones. To validate our approach, we have applied it onto two open source Java applications of different sizes (i.e. small and large-scale ones). We propose an empirical measurement to evaluate the reusability of the mined components. According to this measurement, the results show that the reusability of the mined components using our approach is better than the reusability of those mined from singular software.

The rest of this paper is organized as follows. In section 2, we present the ROMANTIC approach, which constitute a background for our work. Section 3 presents the proposed approach. The experimental results are presented and discussed in section 4. The related work and conclusion are placed in sections 5 and 6 respectively.

2. Background: the ROMANTIC Approach

In our previous works [3] and [17], we have proposed the ROMANTIC approach which aims to extract a component-based architecture from an object-oriented software. ROMANTIC is mainly based on two models: first an object-to-component mapping model, second a quality measurement model to evaluate the quality of components which are mined from object-oriented source code. In this paper, we rely on these two models to define a process which allows to mine reusable components from similar software.

[1]This work has been funded by grant ANR 2010 BLAN 021902.

2.1. From object to component: the mapping model

A software component is defined based on two parts: internal and external structures [16, 17]. The internal structure implements services provided by the component as well as those used by them. The external structure consists of the accessible services structured as provided and required interfaces. The provided interfaces are the services accessed by other applications/components. The required interfaces represent services that the component needs to perform its provided ones. These are provided by other applications/components. When a component is object oriented (i.e. implemented by an object-oriented language), its internal structure is represented by one or more classes, which can belong to different packages. Fig. 1 shows the object-to-component mapping model.

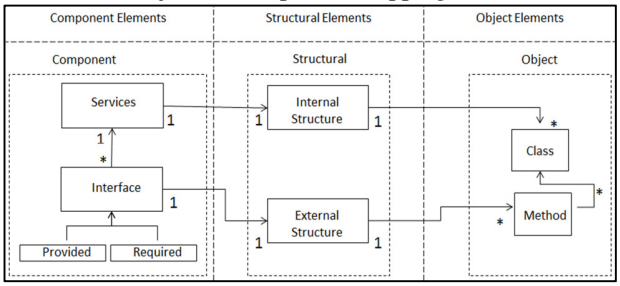


Figure 1. shows the object-to-component mapping model.

2.2. From object to component: the quality measurement model

According to [4, 15, 16], a component is defined as "a software element that (a) can be composed without modification, (b) can be distributed in an autonomous way, (c) encapsulates the implementation of one or many functionalities, and (d) adheres to a component model" [3]. Based on this definition, we identified three quality characteristics of a component: composability, autonomy and specificity [3]. Composability is the ability of a component to be composed without any modification. Autonomy means that it can be reused in an autonomous way. Specificity characteristic is related to the fact that a component must implement a limited number of closed functionalities. Based on these characteristics we proposed a quality measurement model for object-oriented components. The basis of this model is that characteristics are mapped to object-oriented metrics following ISO model 9126 [10]. First of all, the above characteristics are refined into sub-characteristics. Then, these sub-characteristics are refined into properties related to the external structure of a component. Next, these properties are mapped to the properties of the internal structure of a component. Finally, these properties are refined into object-oriented metrics. Fig. 2 shows how the component characteristics are refined following the proposed measurement model.

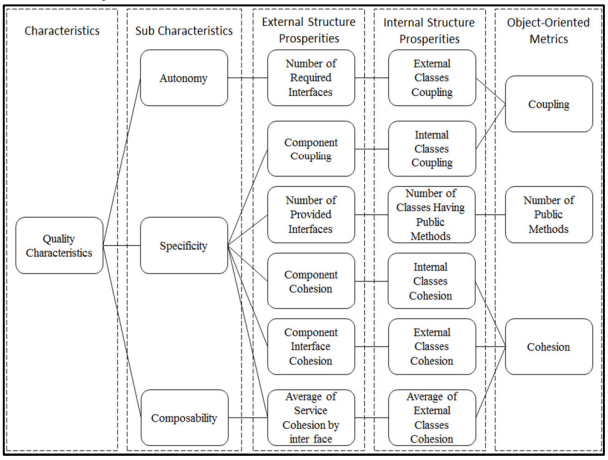


Figure 2. Component quality measurement model.

Based on this measurement model, we defined a fitness function to measure the quality of an object-oriented component based on its characteristics [3]. This function is given bellow:

$$Q(E) = \frac{1}{\sum_i \lambda i} \left(\lambda_1 * \mathrm{S(E)} + \lambda_2 * \mathrm{A(E)} + \lambda_3 * \mathrm{C(E)}\right) \quad (1)$$

Where:

- E is an object-oriented component composed of a group of classes.
- $S(E)$, $A(E)$ and $C(E)$ refer to the specificity, autonomy, and composability of E respectively.
- λ_1, λ_2, λ_3 are weight values, situated in [0-1]. These are used by the architect to weight each characteristic as needed.

We have proposed a specific fitness function to measure each of these characteristics. For example, the specificity characteristic of a component is calculated as follows:

$$S(E) = \frac{1}{5} * \left(\frac{1}{|I|} * \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Coupl(E) + noPub(I)\right) \quad (2)$$

This means that the specificity of a component E depends on the following object-oriented metrics: the cohesion of classes composing the internal structure of E ($LCC(E)$), the cohesion of all classes composing the external structure of E ($LCC(I)$), the average cohesion of all classes composing the external structure of E ($\frac{1}{|I|} * \sum_{i \in I} LCC(i)$), the coupling of internal classes of E ($Coupl(E)$ which is measured based on the number of dependencies between the classes of E), and the number of public methods belong to the external structure of E ($noPub(I)$). LCC (Loose Class Cohesion) is an object-oriented metric that measures the cohesion of a set of classes [11]. For more details about the quality measurement model please refer [3, 17].

3. The Proposed Approach

The aim of our approach is to mine reusable components based on the static analysis of the source code of a set of similar object-oriented software.

The mining process is based on the following steps: first, each software is independently analysed to identify all potential components. These are identified based on the evaluation of their quality characteristics. Next, we identify similar components among all potential ones. Similar components are those providing, mostly the same services and differing compared to few others. After that, we rely on the similarity of each group of components to build a single component, which will be representative of this group; this will be considered as a reusable component. Only classes constituting the internal structures (i.e. the implementation) of the reusable components are identified in this step. Next, we identify their external structure: their provided and required interfaces. Finally, the last step of the mining process aims at documenting the mined components. This documentation includes suggestions to describe the services that components provide. Fig. 3 summarizes the mining process.

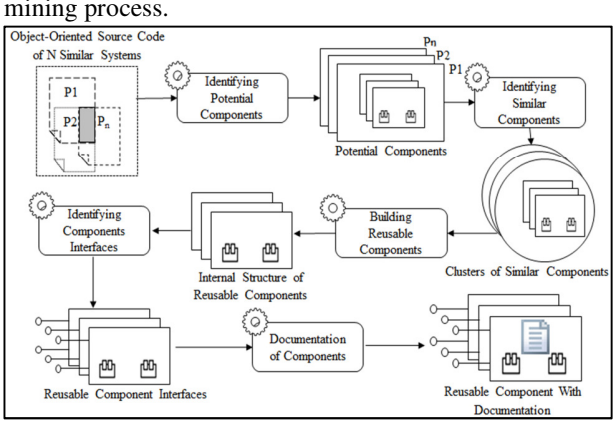


Figure 3. The process of reusable components mining.

3.1. Identifying potential components

Potential components are mined based on the analysis of each object-oriented software. Each potential component is composed of a set of classes where the corresponding value of the quality fitness function is satisfactory (i.e. its quality value is higher than a predefined threshold). The classes composing a potential component are gradually identified starting from a core class. Each class of the analyzed software can be selected to be a core one depending on if an accepted component can be formed starting from this one. This is decided as the result of the next steps.

The selection of the classes to be added at each step is decided based on the value of the quality function of the formed component. In other words, classes are ranked based on the obtained value of the quality function when it is gathered to the current group. The class obtaining the highest quality value is selected to extend the current group. We do this until all classes are grouped into a single group. The quality of the formed groups is evaluated at each step (i.e. each time when a new class is added). Some classes of this group will be excluded. These are those added after the quality function reaches the peak value.

For example, in Fig. 4, classes 7 and 8 are put aside from the group of classes related to component 2 because when they are been added the quality of the component is decreased compared to the peak value. Thus classes retained in the group are those maximizing the quality of the formed component. After identifying all potential components of such software, the only ones retained are those where the quality values are higher than a predefined quality threshold. For example, in Fig. 4, component 1 does not reach the predefined threshold and, thus, not retained as a potential component. This means that the starting core class is not suitable. Algorithm 1 below illustrates the process of potential components mining. In this algorithm, Q refers to the quality fitness function and Q-threshold is a predefined quality threshold.

Algorithm 1: PotentialComponents(OO source code): potential components

```
classes ← extractInformation(OO source code);
for each C in classes
  component ← C;
  bestComponent ← component;
  while (|classes – component.classes| > 1) do
    c1 ←getNearestClass(component,classes–component);
    component ← component + c1;
    if (Q(component)) > Q(bestComponent) )then
       bestComponent ← component;
    end if
  end while
  if (Q(bestComponent) > Q-threshold)
    add(Results, bestComponent);
  end if
end for
return Results;
```

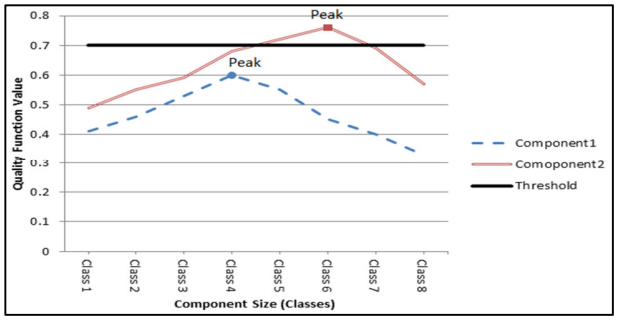


Figure 4. Forming potential components by incremental selection of classes.

3.2. Identifying similar components

Potential components are mined based on the analysis of a set of similar software. As a consequence, some of them may be similar. Similar components are those providing mostly the same functionalities and differing in few ones. These can be considered as variants of a common component. The similarity as well as the difference between components appears compared to their internal structures composed of object-oriented classes. Thus similar components are those sharing the majority of their classes and differing considering the other ones. We gather similar component into groups from which we mine common ones.

Groups of similar components are built based on a lexical similarity metric. Thus components are identified as similar compared to the strength of similarity links between classes composing them. We use cosine similarity metric [9]. Following this metric each component is considered as a text document which consists of a list of component classes' names.

We use a hierarchal clustering algorithm to gather similar components into groups. It starts by considering individual components as initial leaf nodes in a binary tree. Next, the two most similar nodes are grouped into a new one (i.e. as a parent of them). This is continued until all nodes are grouped as a binary tree. This tree is composed of all candidate clusters. To identify the best ones (clusters), we use a depth first search algorithm. Starting from the tree root to find the cut-off points, we compare the similarity of the current node with its children. If the current node has a similarity value exceeding the average similarity value of its children, then the cut-off point is in the current node, otherwise, the algorithm continues through its children (c.f. Algorithm 2). The results of this algorithm are clusters where each one groups a set of similar components.

Algorithm 2: ComponentsClustering(Potential Components): clusters of potential components

---

```
binaryTree ← PotentialComponents
while (|binaryTree| > 1) do
  c1, c2 ← nearestNodes(binaryTree);// cosine similarity
  c ← newNode(c1, c2);
  remove(c1, binaryTree);
  remove(c2, binaryTree);
  add(c, binaryTree);
end while
clusters ← depthFirstSearch.getBestClsuters(binaryTree);
return clusters;
```

3.3. Reusable component mining from similar potential ones

As previously mentioned, similar components are considered as variants of a common one. Thus, from each cluster of similar components, we extract a common component which is considered as the most reusable compared to the members of the analyzed group. It is composed based on all shared classes and some selected non-shared ones. Shared classes form the core of the reusable component. These classes may not form a correct component following our quality measurement model. Thus some non-shared classes are added based on the following criteria:

- The quality of the component obtained by adding a non-shared class to the core ones.

- The density of a non-shared class in a cluster of similar components which refers to the occurrence ratio of the class compared to the components of this group. We consider that a class, which has high density, contributes to build a reusable component.

Consequently the following algorithm generates classes forming the reusable components. First, for each cluster of similar component, we extract all candidate subsets of classes among the set of non-shared ones. Then, the subsets that reach a predefined density threshold are only selected. The density of a subset is the average densities of all classes in this subset. Next, we evaluate the quality of the component formed by grouping core classes with classes of each subset resulting from the previous step. Thus the subset maximizing the quality value is grouped with the core classes to form the reusable component. Only components with a quality value higher than a predefined threshold are retained.

Nevertheless the above algorithm is NP-complete (i.e. the complexity of identifying all subsets of a collection of classes is 2^n-1). This means that the computing time will be accepted only for components with a small number of non-shared classes. This algorithm is not scalable for a large number of non-shared classes (e.g. 10 non-shared classes need 1024 operations, while 20 classes need 1048576 operations).

Consequently, we propose the following heuristic algorithm as an alternative. First of all, non-shared classes are evaluated based on their density. The Classes that do not reach a predefined density threshold are rejected. Then, we identify the greater subset that reaches a predefined quality threshold when it is added to the core classes. To identify the greater subset, we consider the set composed of all non-shared classes as the initial one. This subset is grouped with the core classes to form a component. If this component reaches the predefined quality threshold, then it represents the reusable component. Otherwise, we remove the non-shared class having the lesser quality value compared to the quality of the component formed when this class is added to the core ones. We do this until a component reaching the quality threshold or the subset of non-shared classes becomes

empty. Algorithm 3 shows the process of reusable components mining, where Q refers to the quality function (1), Q-threshold refers to the predefined quality threshold.

Algorithm 3: MiningReusableComponents(Clusters of Components) : reusable components

```
---------------------------------------------------------------------
for each cluster in Clusters of Components do
    shared ← getSharedClasses(cluster);
    nonShared ← getNonSharedClasses(cluster);
    component ← shared;
    removeClassesLessThanDensityThreshold(nonShare);
   while (|nonShare|>0) do
if(Q(component + nonShare)>=Q-threshold)
            add(Results,component);
   break while;
end if
removeLessQualityClass(NonShare, shared);
    end while
end for
return Results;
```

3.4. Identifying structure of the reusable components

As it is illustrated in section II, a component is used based on its provided and required interfaces. For an object-oriented component, provided interfaces are composed of the public methods of classes that compose its external structure. The required interfaces are composed of the methods that are used from the other components (i.e. the provided interfaces of the other components). We rely on the following heuristics to identify these interfaces. First, we consider that when a group of methods belongs to the same object-oriented interface, then they may belong to the same component's interface. Second, cohesive and lexically similar methods have high probability to belong to the same interface. Third, when a component provides services for another component, it provides them through the same interface. Finally, when methods are called many times together, this is an indicator of a high correlation of use. We consider these methods as belonging to the same provided interface.

According to the above heuristics, we defined the following function. It is used to measure the quality of a component's interface.

$$Interface(M) = \frac{1}{\sum_i \lambda_i}\left(\lambda_1 * SI(M) + \lambda_2 * SM(M) + \lambda_3 * CU(M) + \lambda_4 * CI(M)\right) \quad (3)$$

Where:

- *M*: a set of methods.
- *SI*: measures how much a set of methods M belongs to the same object-oriented interface.
- *SM*: measures how much a set of methods M is similar using cosine and cohesion (LCC) metrics.
- *CU*: measures how many times a set of methods M has been called together by the same component.
- *CI*: measures how many times a set of methods M is invocated together.

Based on the above function we use a hierarchical clustering algorithm to partition a set of public methods into a set of clusters, where each cluster is a component's interface. First, this clustering algorithm produces a binary tree that contains all candidate clusters. Then we use a depth first search algorithm to travel through the binary tree, in order to identify the best partition of the methods.

3.5. Documentation of Components

The documentation of a component helps the developers to find a component that meets their needs. The description of the component functionalities forms an important part of its documentation. Thus we propose to identify for each mined component its main functionalities. We do this based on two steps: the identification of the component functionalities and the generation of a description for each of them. These steps are detailed below.

3.5.1. Identifying the component functionalities. As we mentioned it in section II, the quality function is based on three sub-characteristics. One of them is used to measure the specificity of a component. It is related to the functionalities provided by this component. The specificity depends on three properties. The first is that the number of public methods is proportional to the number of functionalities. The second is that classes providing the same functionalities must be cohesive. The last property is that elements of source code participating in the same functionality must have a high cohesion with themselves and low coupling with other parts in the component. Thus, we use equation 2 (Cf. section 2.2) as a fitness function in a hierarchical clustering algorithm in order to decompose component classes into partitions, where each one represents one of the functionality of the analyzed component.

3.5.2. Generation of the functionality description. In the previous step, the component classes are partitioned according to their functionalities. In this step, we present how the description of each partition (i.e. functionality) is generated. This description consists of the most frequent words in the partition classes' names. We consider that in an object-oriented language, a class name is often a set of nouns concatenated by the camel-case notation. These

nouns are representing a meaningful name for the main purpose of the class. Usually, the first noun in a class name holds the main goal of the class, and so on. Accordingly we propose the following three steps. First, tokens are extracted by separating the words which form the class name according to the camel-case syntax (e.g. MediaControllerAlbum is divided into Media, Controller, and Album). Second, a weight is affected to each extracted token. The tokens which are the first word of a class name are given a large weight. Other tokens are given a small weight. The weight is calculated as follows:

$$Weight(w) = \frac{1}{\sum_i N_i} * (1 * N_1 + 0.75 * N_2 + 0.50 * N_3 + 0.25 * N_4) \quad (4)$$

Where:

- *W*: refers to a word.
- N_i refers to the number of occurrence of the word *w* in the position *i*.

Last, we use tokens which have the highest weight to construct the functionality description in an orderly manner. Meaning, the token that has the highest weight will become the first word of the functionality description and so on. The architect defines the number of words as needed.

4. Experimental Results and Evaluation

To validate the proposed approach, we applied it onto product variants of two open source Java applications. These are Mobile Media[1] [13] as a small-scale software, and ArgoUML-SPL[2] [14] as a large-scale one.

Mobile Media is a software product line. It is used to manipulate music, video and photo on mobile devices. Using the latest version, the user can generate 200 variants. In our experimentation, we use 8 variants, where each variant contains 43.25 classes on average.

ArgoUML-SPL [14] is a UML modeling tool. It is developed based on software product line. We applied our approach on 9 variants, where each variant is generated by changing a set of the needed features. Each variant contains 2198.11 classes on average.

4.1. Identifying potential components

To consider that a group of classes forms a component, its quality function value should exceed a predefined quality threshold. We tested the quality threshold value from 0 up to 1 by incrementing it 0.05 in each run. The results obtained from Mobile Media and ArgoUML are respectively shown in Fig. 5.a and Fig. 5.b. where the value of the threshold is in the X-axis, and the average number of the mined components in a variant is in the Y-axis.

Table 1 shows the total number of potential components (TNOCV) mined based on the analysis of all variants, the average number of classes (size) of these components (ASOC), the average value of the specificity characteristic (AS), the average value of the autonomy characteristic (AA) and the average value of the composability characteristic (AC). We assign 0.70 and 0.83 as threshold value respectively for Mobile Media and ArgoUML case studies.

Table 1. The results of potential components extraction.

Product Name	TNOCV	ASOC	AS	AA	AC
Mobile Media	24.5	6.45	0.56	0.71	0.83
ArgoUML-SPL	811	11.38	0.64	0.83	0.89

As an example of a potential component extracted from ArgoUML-SPL, consider the one identified by considering GoClassToNavigableClass as the core class Fig. 6 shows how this component is formed and when the quality fitness function reaches the peak after adding the 18th classes. Thus the 18 first classes form this potential component. The remaining classes are rejected.

4.2. Identifying similar components

The results of the clustering algorithm are presented in Table 2. For each case study, Table II shows the number of clusters (NOC), the average numbers of components in the identified clusters (ANOC), the average number of shared classes in these clusters (ANSC), the average value of the specificity characteristic (ASS), the average value of the autonomy characteristic (AAS), and the average value of composability characteristic of the shared classes (ACS) in these clusters.

Table 2. The results of component's clustering.

Product	NOC	ANOC	ANSC	ASS	AAS	ACS
Mobile Media	42	5.38	5.04	0.59	0.71	0.89
ArgoUML-SPL	325	5.26	8.67	0.57	0.87	0.93

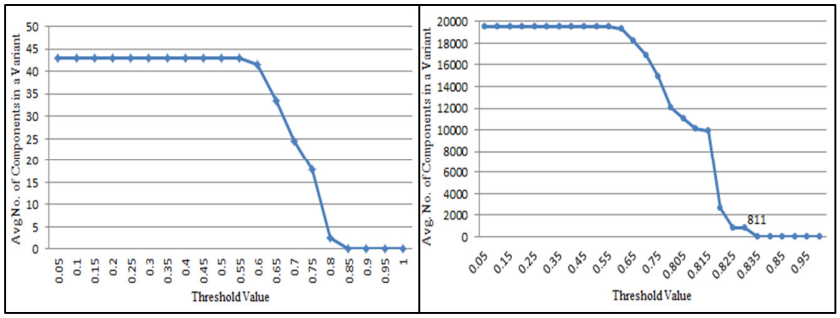
4.3. Reusable component mining by analyzing similar potential ones

Table 3 summarizes the final set of reusable components mined using our approach. We assign 0.50 to the density threshold value. For each product, we present the number of the mined components (NOMC), the average component size (ACS), and the average value of the specificity (AS), the autonomy (AA), and the composability (AC) of the mined components.

Table 3. The final set of mined components.

Product	NOC	ACS	AS	AA	AC
Mobile Media	39	5.61	0.58	0.74	0.90
ArgoUML-SPL	324	9.77	0.61	0.84	0.84

[1]Available at http://homepages.dcc.ufmg.br/~figueiredo/spl/icse08
[2]Available at http://argouml-spl.tigris.org/



(a) (b)

Figure 5. Changing threshold value to extract all potential components.

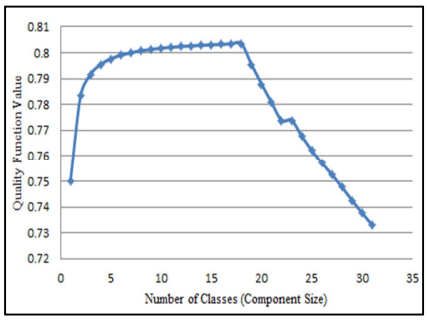


Figure 6. An instance of a potential component extraction.

Table 4 shows some of the reusable components that are mined based on the analysis of Mobile Media. DOF is the description of the functionalities provided by the considered component. NOV is the number of variants that contains this component. NOC represents the number of classes that forms the component. S, A and C represents respectively the specificity, the autonomy, and the composability of each component.

As shown in Table 4, the second component provides two functionalities, which are *Add Constants Photo Album*, and *Count Software Splash Down Screen*. The former one deals with adding a photo to an album. The letter dedicated to splash screen service.

Table 4. Some components.

DOF	NOV	NOC	S	A	C
New Constants Screen Album Image	6	6	0.59	0.75	0.94
Add Constants Photo Album Count Software Splash Down Screen	8	10	0.57	0.75	0.89
Base Image Constants Album Screen Accessor List Controller Image Interface Thread	6	9	0.67	0.50	0.85

4.5. Reusability validation

In order to validate the reusability of components that are mined based on our approach, we compare their reusability with ones that are mined from singular software. We consider that the reusability of a component in a collection of software is evaluated by calculating the ratio between the number of software that can reuse this component to the number of all software. The component can be reused in a software if it provides functionalities required by this software. In other words, we analyze the software functionalities, and then we check if a component provides some of these functionalities. The functionalities are identified based on potential components in this software.

We measure the reusability of the mined components based on K-fold cross validation method [9]. K-fold is an evaluation model that is used to validate the results of the mining model by dividing the data set into two parts: train data, and test data. Train data are used to learn the mining model, while test data are then used to validate the mining model. The idea of K-fold method is to divide the data set into K parts. The validation is applied K times by considering K-1 parts as train data and the other one as test data. After that, the validation result is the average of all K trails. In the same manner, we validate our approach by dividing the variants of the product into K parts. Then, we mine components from train variants only (i.e. K-1 parts). Next, we validate the reusability of these components in the test variants. We evaluate the result by assigning 2, 4, and 8 to the K in each validation time. Due to limited space, we give only the results obtained from the Mobile Media case study (c.f. Table 5). These results show that the reusability of the components which is mined from a collection of similar software is better than the reusability of components which is mined from singular software. Also, the reusability is decreased when the number of K is increased because of the number of test variants is decreased (i.e. when K=8, there is only one test variant). The slight difference between the reusability results comes from the nature of our case studies, where these case studies are very similar. Consequently, the resulting components are closely similar (i.e. there are many groups of similar components containing exactly the same classes which resulted the same reusable component). Therefore, there is very small difference in the results, as shown in Table 5.

Table 5. The reusability validation

K	Similar Software	Singular Software
2	0.32	0.28
4	0.18	0.15
8	0.09	0.07

5. Related Work

Numerous approaches have been presented to address the problem of component identification from object-oriented software such as [3], [5], [6], and [7]. All existing approaches mined components from single software.

In [3], the authors presented an approach to extract components from object-oriented software. Classes composing the extracted components form a partition. Mined component are considered as a part of the component-based architecture of the corresponding software.

The authors in [5] presented an approach to migrate an object-oriented software into a component-based software. The authors extract services from the software, and then these services are converted into components. They depended on use case, sequence diagrams, and class diagrams to identify the structural relationship between the objects, and object usage. The limitation of this approach is that sequence diagrams, use case, and class diagrams are not always available.

In [6], the authors depended on dynamic dependencies between software classes, in order to reengineer an object-oriented software into a component-based software. They relied on the use-case diagram to identify the execution trace scenarios. Classes that frequently occur in the execution traces are grouped into a component.

In [7], the authors proposed an approach to extract stable components. The authors identify a set of candidate components from requirements and use case using formal concept analysis. The extracted components represent the functional units that can be reused in the future [7]. The authors focused on the component stability rather than the component reusability.

7. Conclusion

Mining components from similar software provides more guarantees for the reusability of the mined components rather than depending on single software. In this paper, we proposed an approach to mine reusable components from a set of similar object-oriented software. We validate our approach by applying it on two sets of variants of two open source Java applications.

There are two aspects to be considered regarding the hypothesis of our approach. First, we consider that the variability between software is in the class level (i.e. classes that have the same name should have the same implementation). Second, forming a component by adding a non-shared class to the core ones may cause a dead code (i.e. a piece of code which is executed but there is no need for its result).

Our future directions will focus on migrating similar software into component based software product line.

8. References

[1] W.B. Frakes, K. Kang, "Software reuse research: status and future," IEEE Transactions on Software Engineering, vol.31, no.7, pp.529-536, 2005.

[2] N.M.J. Basha, S.A. Moiz, "Component based software development: A state of art," International Conference on Advances in Engineering, Science and Management (ICAESM), pp.599-604, 2012.

[3] S. Kebir, A.-D. Seriai, S. Chardigny, A. Chaoui, "Quality-Centric Approach for Software Component Identification from Object-Oriented Code," Joint Working Conference on Software Architecture IEEE/IFIP WICSA and ECSA, pp.181-190, 2012.

[4] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] S. K. Mishra, D. S. Kushwaha, A. K. Misra, "Creating reusable software component from object-oriented legacy software through reverse engineering," Journal of Object Technology, vol. 8, no. 5, pp. 133-152, 2009.

[6] S. Allier, H. A. Sahraoui, S. Sadou, Vaucher S., "Restructuring object-oriented applications into componentoriented applications by using consistency with execution traces," in Proceedings of CBSE'10. Berlin, Heidelberg: Springer-Verlag, pp. 216–231, 2010.

[7] H.S. Hamza, "A Framework for Identifying Reusable Software Components Using Formal Concept Analysis," Conference on Information Technology: New Generations. ITNG '09. Sixth International, pp.813-818, 2009.

[8] J. Rubin and M. Chechik. Locating distinguishing features using diff sets. InProceedings of ASE 2012. ACM, New York, NY, USA, 242-245..

[9] J. Han, M. Kamber, Data Mining Concepts and Techniques, 2nd Edition. Elsevier Inc, 2006.

[10] ISO, "Software engineering – Product quality – Part 1: Quality model," International Organization for Standardization, Tech. Rep. ISO/IEC 9126-1, 2001.

[11] J. M. Bieman, B.-K. Kang, "Cohesion and reuse in an object-oriented software," in Proceedings of SSR '95. New York, NY, USA: ACM, pp. 259-262, 1995.

[12] O. Nierstrasz, L. Dami, "Component-Oriented Software Technology," Object-Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (Eds.), Prentice Hall, pp.3-28, 1995.

[13] E. Figueiredo, N. Cacho, C. Sant'Anna, et al., "Evolving Software Product Lines with Aspects: an empirical study on design stability," ICSE, pp. 261-270. 2008.

[14] M.V. Couto, M.T. Valente, E. Figueiredo, "Extracting Software Product Lines: A Case Study Using Conditional Compilation," CSMR 2011 , pp.191-200, 2011.

[15] G. T. Heineman, W. Councill T., Eds., "Component-based software engineering: putting the pieces together," Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[16] C. Luer, A. V. D. Hoek,"Composition environments for deployable software components," Tech. Rep., 2002.

[17] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit,"Extraction of component-based architecture from objectoriented software," in Proceedings of WICSA 2008. Washington, DC, USA: IEEE Computer Society, pp. 285–288, 2008.

[18] J Sametinger,"Software Engineering with Reusable Components," Springer Verlag Berlin Heidelberg New York, 1997.