Use-Relationship Based Classification for Software Components

Reishi Yokomori
Nanzan University
Nagoya, Japan
yokomori@se.nanzan-u.ac.jp

Norihiro Yoshida
Nagoya University
Nagoya, Japan
yoshida@ertl.jp

Masami Noro
Nanzan University
Nagoya, Japan
yoshie@nanzan-u.ac.jp

Katsuro Inoue
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

***Abstract*—In recent years, the maintenance period of the software system is increasing. The size of the software system has grown, and the number of classes and the relationship between classes are also increasingly complicated. If we can categorize software components based on information such as functions and roles, we believe that these classified components can be understood together, and are useful for understanding the system. In this paper, we proposed a classification method for software components based on similarity of use relation. For each component, a set of components used by the component was analyzed. And then, for each pair of components, the distance was calculated from the coincidence of the two sets. A distance matrix was created and components were classified by hierarchical cluster analysis. We applied this method to jlGui consisting of 70 components. 8 clusters of 36 components were extracted from the 70 components. Characteristics of the extracted clusters were evaluated, and the content of each cluster was introduced as a case study. In 7 clusters out of the 8 clusters, components of the cluster were strongly similar with each other from the viewpoint of their functions. Through these experiments, we confirmed that our method is effective for classifying components of the target software, and is useful for understanding them.**

***Index Terms*—component, use-relations, hierarchical cluster analysis**

I. Introduction

In recent years, the maintenance period of the software system has increased, by also experiencing the entry of new developers. In order to participate in the maintenance of such a software system, it is the first important task to become accustomed to the environment surrounding the software system [1], and it is necessary to understand the configuration of the target software. In the process of understanding, the developer first understands the diagram and documents representing the composition of the software and then reads the source code of each component of the software in order to grasp the details of each subsystem and the overall image.

In reading the source code, the developer browses the source code of another component one by one on an editor or a source code browser like SPARS-J [2], by referring to the package hierarchy and the name of another class appearing in the source code, and so on. In this way, the source codes of a lot of components are read strategically in order to understand efficiently. In fact, when the developer encounters a component similar to the components the developer confirmed once, it often happens that reworking occurs in order to remember its functions and role. This type of rework needs time, however, it makes their understanding deeper. However, as the number of files and classes increases through a long period of maintenance, the frequency of the rework in understanding increases, and the content to be understood becomes also complicated. The effort required for understanding the internal structure by developers participating later will be much greater than the effort expected by the original developer. By analyzing the source code by the statement unit, a lot of methods for extracting code clone [3], extracting coding-pattern [4], and extracting coding-rule [5] have been proposed. As the scale of the software system increases, the relationship between statements becomes complicated, and the analysis cost gradually increases, so we would like to propose a method with a coarser granularity, based on the anlysis by the class unit.

In the evaluation experiment for the component retrieval system SPARS-J [2], it was effective to guide the component to be browsed next based on its use relationships, and it was popular among developers. If we can categorize software components based on their functions and roles by obtaining information from analysis of use-relationships, we can understand these classified components together and this classification would be useful for understanding the target software. For example, when realizing a specific feature, it is often necessary to use certain components together. We compare use-relations of two components (-A, -B), that means components that component-A uses and components that component-B uses are compared. In the case where there are many common components in the two component set, we consider that component-A and -B are supposed to implement the same specific function or very similar function. We think that understanding of these components can be performed efficiently by also considering components that are commonly used by these components. By being able to understand efficiently, we believe that developers devote much effort to other work and can carry out with high quality. As a result, we think it can improve productivity and quality of maintenance.

In this paper, we proposed a classification method for software components based on similarity of use relation. In this method, for each component, a set of components used by the component was analyzed. And then, for each pair of components, the distance between the two components was calculated from the coincidence of the two sets. In this way, a distance matrix of the target software components was created and software components were classified by hierarchical cluster analysis.

We implemented a prototype for the proposed method, and its analysis target is a software system described in Java. The prototype system treats the Java source files as a component and excludes components defined outside the software, such as components in libraries. Our system generates a distance matrix from the analysis result of Classycle's analyzer [6], and hierarchical cluster analysis is executed by R. We conducted two evaluation experiments. In the first experiment, we applied it to the actual software system of open source project and evaluated the similarity of the components in the obtained cluster from several viewpoints. As a result, many of the components in the cluster showed strong similarity in terms of their functionality, and more than half of similar components were joined as clusters. In the second experiment, we introduced how the components in the clusters were combined, as a case study. Some clusters were composed of a collection of components that have a simple structure and a simple use relationship, and some clusters were composed of a collection of components that realize multiple functions by using several other components. In the case study, knowing the components commonly used by the components in the cluster was effective in understanding the contents of the components in the cluster. Through these experiments, we introduce that the obtained clusters have sufficient accuracy to be used for software understanding, and our method is effective to classify and understand the components of the target software.

The main contributions of our research are as follows:

- We have hypothesized that "When two components use many common components, these two components are implemented for the same or similar purpose", and confirmed it by experiment.
- We showed that the proposed method can combine most of the functionally related components as clusters.
- We showed that the obtained clusters have sufficient accuracy for software understanding, and our method is effective to classify and understand the components of the target software.
- We showed that the components in the cluster could be used as examples when adding new components that use the same components.

Section II introduces the component graph as a background. Section III introduces our method and implementation. Section IV conducts two experiments. Finally, Section V discusses the results and improved ways and introduces related works.

II. Background

A. Component Graph and Use Relation between Components

In general, a component is a modular part of a system that encapsulates its content and whose manifestation is replaceable within its environment [7]. We model software systems by using a weighted directed graph, called a *Component Graph* [2]. In the component graph (V, E) , a node $v \in V$ represents a software component, and a directed edge $e_{xy} \in E$ from node x to y represents a *use relation* meaning that component x uses component y. This *use relation* indicates that a component realizes its own function by utilizing features of other components. If you need to understand how features in a particular component are specifically implemented, you can use use-relations to understand which parts of the software you need to reference together.

B. Classification methods using metrics that represent the characteristics of software components

For a lot of components extracted from multiple software systems, Kobori suggested a method for extracting the same or similar components as the target components [8]. In this method, metrics representing the characteristics of software components are obtained from the source code of each component. These metrics are the number of methods and the number of tokens that represent the scale of the component, and the cyclomatic complexity that represents the complexity of control structure of the component, and so on. In the method, components that have the same or similar values in these metrics are extracted. Through experiments, they showed that extracted components are mainly copied components or copied and modified components from the original ones. This method is a method assumed for using in the software component retrieval system. In the software component retrieval system, it collects a lot of software system. The method could help to extract the almost identical (copied or copied and modified) components across a lot of software systems. Our method is to classify components in one software system, based on the fact that components are using a lot of common components. The purpose of our method is to obtain a set of components using similar functions and to obtain a set of components whose role and features in the software are similar.

III. Proposed Method

When implementing a feature in a new component, we also use the features realized by other existing components as necessary. From this, it can be considered that which components are used by a certain component can also be information for estimating the features realized in the component. In some cases, when using a certain feature, it is necessary to use some other components together. In such a case, there is a commonality in the set of using components among the components that use the feature. Based on the fact that the using components are similar, we thought that these components are similar in function and role. Therefore, we make a hypothesis that "When two components use many common components, these two components are implemented for the same or similar purpose", and confirm it by experiments. We propose a method for classifying components based on the commonality of components that are used by the components. We thought that this method can extract several components groups that are recognizable as collections of components that are similar in function or role. Through evaluation experiments to an open source project, we introduce characteristics of the obtained component groups and components in the obtained component groups have a strong relationship.

A. *Implementation of the analysis tool*

We implemented a classification tool for Java source files of a target software system. The system constructs a component graph for Java source files of the target software system and produces a distance matrix used by the hierarchical cluster analysis on R. For extracting use relations, we used Classycle's analyzer [6], that is an analyzing tool for java class and package dependencies. In the analysis of the Classycle's analyzer, we get the following as use relations: inheritance of class, declaration of variables, a creation of instances, method calls, and reference of fields, and inner classes and use relations about inner classes are merged into the major class in the Java source files. Figure 1 is an overview of the system, and the following is the analysis procedure:

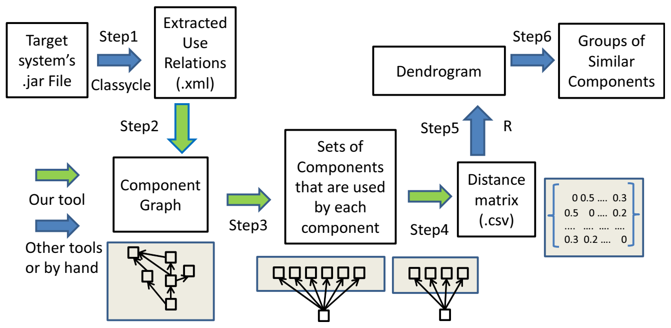

Fig. 1. Overview of the Proposed Method

1. Classycle's analyzer [6] provides the use relations of the target system.
2. Build a component graph based on the use-relations.
3. For each component, a set of components used by the component is obtained from the graph.
4. For each pair of components, a distance between the two components is calculated from a degree of coincidence of the two sets. A distance matrix for the target software components is produced.
5. By using the distance matrix, hierarchical cluster analysis is performed on R.
6. The dendrogram is obtained by R. We decide a threshold value and clusters under the threshold value are recognized as groups of similar components.

B. *Distance between two Components*

Consider two components A and B. L_{A} represents a set of components that are used by A, and L_{B} represents a set of components that are used by B, respectively.

At first, we define the similarity of components A and B $sim(L_{\mathrm{A}}, L_{\mathrm{B}})$ by using the Jaccard coefficient.

$$sim(L_{\mathrm{A}}, L_{\mathrm{B}}) = \frac{|L_{\mathrm{A}} \cap L_{\mathrm{B}}|}{|L_{\mathrm{A}} \cup L_{\mathrm{B}}|}$$

This value has a range of 0 to 1, and the higher this value, the more likely that L_{A} and L_{B} are similar. And then, we define the distance of components A and B $dist(L_{\mathrm{A}}, L_{\mathrm{B}})$ as following; (For representation, the distance was 10 timed, and it does not have other special meaning.)

$$dist(L_{\mathrm{A}}, L_{\mathrm{B}}) = (1 - sim(L_{\mathrm{A}}, L_{\mathrm{B}})) * 10$$

C. *About Linkage Criterion for Hierarchical Clustering*

In the hierarchical clustering, there are candidates to compute the distance for a cluster consists of multiple objects, so the user needs to decide on the linkage criterion to use. For our study, we decided to use the linkage criterions provided by R, since it was intended to create a prototype of an analytical environment. The linkage criterions are roughly divided into two methods: methods to recalculate a distance directly from distances of multiple objects, such as a single linkage, a complete linkage, and UPGMA, and methods to recalculate a distance from the center or the center of gravity of the cluster, such as Ward, and centroid method. From them, we decided to use the former methods. In general clustering, it is considered that the latter methods can be utilized because a set of vectors is used as input and hierarchical clustering is done by calculating the distance between vectors. However, in our analysis, only the distances between components are given as input, so it is meaningless to consider the center of the cluster.

We introduce three types of results, based on the single linkage method, the complete linkage method, and the UPGMA, as a preliminary experiment, and examine which linkage criterion should be used. The target system is GoGui that is a graphical interface program for playing Go and is composed of 181 Java source files. Figure 2 is a dendrogram of the hierarchical clustering using the single linkage method. In the case of the single linkage method, if a component has a similarity to any component in the cluster, the component is also merged into the cluster. So a very large cluster is easily formed, however, similarities between components in a cluster would not be strong. Figure 3 is a dendrogram of the hierarchical clustering using the complete linkage method. In the case of the complete linkage method, a component is merged into the cluster only if the component has similarities with all components in the cluster. A lot of small clusters are formed, however, similarities between components in each cluster would be strong.

Figure 4 is a dendrogram of the hierarchical clustering using the UPGMA. We can confirm a lot of small clusters as in the case of Figure 3, and we can also confirm that these clusters are merged into a large cluster as in the case of Figure 2. In this way, the dendrogram obtained by UPGMA is a balanced result. By adjusting a threshold height for getting clusters, we can obtain very similar clusters obtained by the other two approaches. In this study, we use the UPGMA as a linkage criterion in the hierarchical clustering. The problem of deciding the threshold height for getting cluster is considered to be a problem to be tackled in the future, and we will show how the results change by the upper or lower thresholds in the experiments. As another linkage criterion, we can also

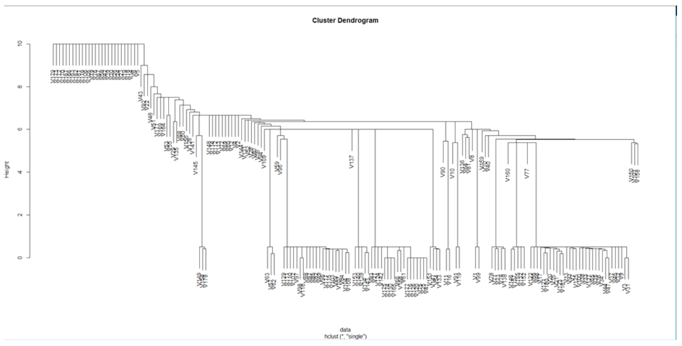

Fig. 2. A Dendrogram of GoGui Components (SLM)

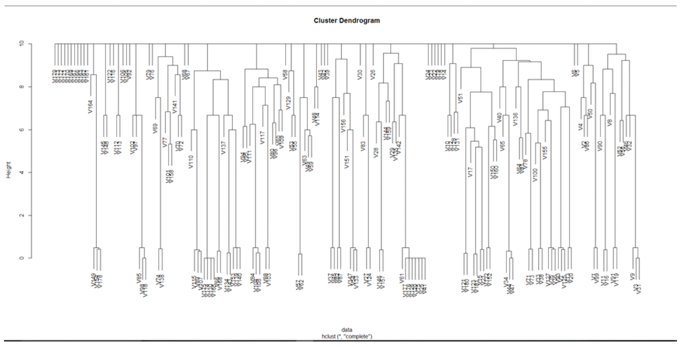

Fig. 3. A Dendrogram of GoGui Components (CLM)

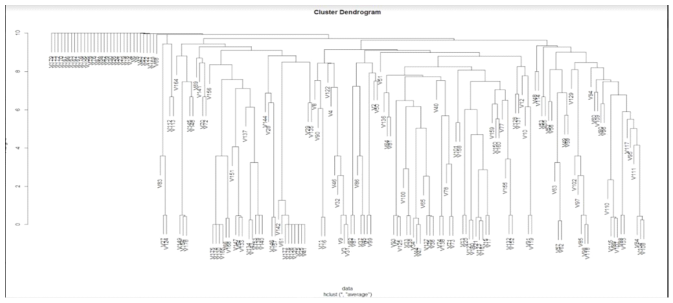

Fig. 4. A Dendrogram of GoGui Components (UPGMA)

consider a method to recalculate a group of using components for each cluster and recalculate the distance between clusters. This also would be a problem to be tackled in the future, however, we consider that it contributes to the improvement of accuracy and the overall trend does not change significantly.

IV. Experiments

For the experiments, we applied our method to jlGui Ver.3.0. It is a graphical music player which supports Java Sound capabilities and consists of 70 Java source files. We obtained a dendrogram of hierarchical clustering using UPGMA. By showing the results of the following two experiments, it is shown that the obtained clusters were composed of components with strong similarity.

1 Can we say that the components in the resulting cluster are strongly related components?
Initially, we prepared several situations that we can imagine when the roles and objectives of the components are similar. Considering some conditions that similar components satisfy, we treated them as criteria to show similarity. For each criterion, we examined how many components in the cluster meets the criteria. It corresponded to the accuracy. Then, we examined how many similar components exist outside the cluster. It corresponded to the recall.

2 What kind of clusters have been obtained?
For some clusters, we introduced the collection of components in the clusters and the role of each component, as a case study. We will show the results classified by our method were realistic ones.

A. *About Criterion for similarity*

At first, we prepared several situations that we can imagine when the roles and objectives of the components are similar.

1) Many of the components that have common processing to achieve similar functionality are derived from the same class or implement the same interface. The processing performed by these components and the APIs provided by them is often quite similar. These components share similar code fragments, so they would be comprehensible at once.
2) Components that realize different objectives for the same target should be considered as a group associated with the target. Common processing would be performed as pre- or post-processing. So these components share similar code fragments and would be comprehensible at once to understand how to use the target.
3) Components that are considered to realize a large feature are also considered as a functional group when observing from a vantage point. The targets of these components are often slightly different, but it is possible to verify the relationship between them. These components will help you to get a rough understanding of what features are provided for a particular element.
4) If the same design pattern is deployed in multiple locations in a software system, you can see that there are multiple components that have a specific role on the design pattern in the software. We believe that using the same design pattern has a common intention, so these components are considered to be comprehensible to understand a part of software design. In addition, these components sometimes share similar code fragments to perform the common processing.

In such cases, we thought that the components used by these components had a commonality. Besides, the similarity of components may be confirmed by the similarity of package classification and their names. Based on those, seven criteria were set to confirm that the two components are similar, and they were used for calculating the precision and the recall.

C1 The two components were derived from the same class or implemented the same interface.

C2 The two components belonged to the same package.
C3 The two components used the same word as a part of the name.
C4 The two components shared similar code fragments.
C5 When comparing the components used by the two components, the proportion of matching components was over 50%.
C6 The two components were thought to constitute a part of a specific functional group.
C7 There were several methods with the same signature in the two components.

B. About precision and recall

In order to calculate the precision, we randomly extracted a component from each cluster and treated it as component A. Next, at the given criterion, components related to component A were searched from the components in the cluster. We calculated the number of related components in the cluster divided by the number of components in the cluster and we regarded it as the precision for the criteria. If there is no related component without component A, it is represented by ×.

The recall was calculated only for items where related components existed. At the given criteria, components related to component A were searched from all components in the software. We calculated the number of related components in the cluster divided by the number of related components in the software and we regarded it as the recall for the criteria. If there were no related components in the cluster without component A, we express it by ×. For both cases, the component A was also counted for the number of components.

C. Experiment 1 : Can we say that the components in the resulting cluster are strongly related components?

1) Overview: For 70 jlGui components, we performed the hierarchical clustering based on UPGMA. Figure 5 is a dendrogram of the analysis, and we cut the dendrogram by a red line on Figure 5 and got the eight sets of components joining under the red line as clusters. These clusters were numbered as shown in Figure 5, and we investigated the precision and recall for each criterion for each cluster.

TABLE I
THE PRECISION FOR JLGUI CLUSTERS

	NC	C1	C2	C3	C4	C5	C6	C7
#1	5	4/5	4/5	4/5	×	5/5	5/5	4/5
#2	4	4/4	4/4	4/4	3/4	4/4	4/4	4/4
#3	9	4/9	7/9	7/9	×	7/9	6/9	8/9
#4	4	4/4	4/4	4/4	4/4	4/4	4/4	4/4
#5	5	4/5	5/5	5/5	3/5	5/5	5/5	5/5
#6	3	3/3	×	3/3	3/3	3/3	3/3	3/3
#7	4	×	2/4	2/4	×	4/4	3/4	×
#8	2	×	×	×	×	×	×	×

The Table I is a summary of the precision. For each cluster, the table shows the number of components in the cluster (NC) and the precision for each criterion, represented by the number of similar components in the cluster and the number of components in the cluster. In cluster #1-#7, the high percentage was shown on most criteria, and you can confirm strong relationships among components in the clusters. For these components, the structure, location, and origin of the name were very similar and provides similar methods, so these components can be recognized as functional groups. Some of these components shared similar code fragments, others do not. In experiment 2, we detailed the contents of these clusters. On the other hand, we could not confirm any strong relationship between the components in cluster #8. The cluster #8 consisted of two components, one was the delegation component for the music playlist and the other was the component to search music files. Since the two components used the **Playlist** and **Playlistitem** in common, there was a common point that the two components manipulate the playlist. However, we could not confirm any relevance other than that.

TABLE II
RECALL FOR JLGUI CLUSTERS

	NC	C1	C2	C3	C4	C5	C6	C7
#1	5	4/4	4/6	4/5	×	5/5	5/5	4/5
#2	4	4/5	4/7	4/7	3/8	4/8	4/7	4/7
#3	9	4/8	7/21	7/10	×	7/12	6/10	8/10
#4	4	4/8	4/10	4/10	4/4	4/10	4/9	4/13
#5	5	4/8	5/10	5/10	3/3	5/10	5/9	5/13
#6	3	3/8	×	3/5	3/7	3/6	3/5	3/3
#7	4	×	2/4	2/6	×	4/17	3/7	×
#8	2	×	×	×	×	×	×	×

Next, the Table I is a summary of the recall for each criterion for each cluster. For each cluster, the table shows the number of components in the cluster (NC) and the recall for each criterion, represented by the number of similar components in the cluster and the number of similar components in the software. From the result, we confirmed that most of the related components were collected into clusters #1. Clusters #4 and #5 seemed to be integrated because the correct set of components were almost identical for them. Approximately half of the related components were collected into clusters #2, #3 and #6. There was no big difference between the criteria, and the overall trend was high. Overall, we found that many of the components considered to be similar were included in the clusters. However, as in the case of #7 and #8, there were cases where no strong association was found.

D. Experiment 2 : What kind of clusters have been obtained?

As a case study, we will explain components in the cluster #1-#7 of the Figure 5. First, we introduced the components obtained by the red lines in the Figure 5, and explained about components that were commonly used. We also introduced which components were newly added when moving the line upward. Through these introductions, we will show that the results classified by this method are realistic.

1) Cluster #1: 5 components: Four of the five components were components such as **OggVorbisInfo**, which handled information for several compression types, such as Ogg Voris,

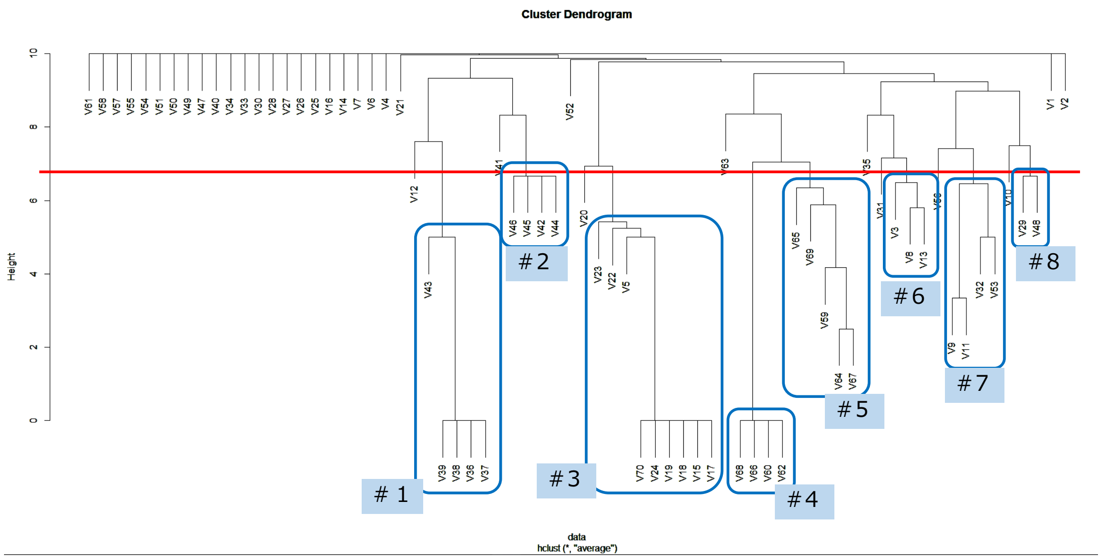

Fig. 5. A Dendrogram for jlGui (UPGMA)

Mpeg, Flac, APE etc. and these were derived from **TagInfo**. The fifth component was **EmptyDialog**. This also used **TagInfo**, and was a part of the dialog related components. These five components were clustered because they used **TagInfo** in common. And any other components were not used in common. If you move the red line upward, **Playlistitem** representing each playlist item will be joined to the cluster.

2) Cluster #2: 4 components: Four components were components such as **OggVorbisDialog**, which handled dialogs for several compression types, such as Ogg Voris, Mpeg, Flac, APE etc. and these were derived from **TagInfoDialog**, and used a corresponding component in Cluster #1. These four components were clustered because they used **TagInfoDialog** in common. And any other components were not used in common. If you move the red line upward, **TagInfoFactory**, that was a factory class for **TagInfo** and **TagInfoDialog** joined to the cluster #2, and finally, cluster #1 and #2 were integrated.

3) Cluster #3: 9 components: Nine components were several types of GUI components, such as icons, bars, sliders, popups, buttons, and so on. These components were derived from the swing components in the standard library for GUI. These components were clustered because they used **AbsoluteConstraints** in common. And any other components were not used in common. **AbsoluteConstraints** was a component for managing information about size and position on the screen. There were 15 components using **AbsoluteConstraints**, and they were scattered across Cluster #3, #6 and #8. If you move the red line upward, **ActiveJLabel** joined the cluster #3 because it also used **AbsoluteConstraints**.

4) Cluster #4: 4 components: Four components were components like **VisualizationPreference**, which handled the preferences of some items. These components were derived from **PreferenceItem**, and these components were clustered because these components used **PreferenceItem**. And any other components were not used in common. If you move the red line upward, cluster #4 and #5 were integrated. Components in Cluster #4 only used **PreferenceItem**, not other components.

5) Cluster #5: 5 components: As in the case of cluster #4, five components were components like **Preferences**, which handled the preferences. These components were also derived from **PreferenceItem**, and these components were clustered because these components used **PreferenceItem**. And any other components were not used in common. If you move the red line upward, cluster #4 and #5 were integrated. Components in Cluster #5 used **PreferenceItem** and several other components and this was a difference.

6) Cluster #6: 3 components: Three components were components that realized user interfaces, such as **PlayerUI**, **PlaylistUI** and **EqualizerUI**. These components were clustered because these components used 8-10 GUI components in common, including **Loader**, **Skin** and components in Cluster #3. If you move the red line upward, **Skin** and **Standalone-Player** joined to the cluster #6, because these components managed a core feature using several GUI components.

7) Cluster #7: 4 components: First, **BasePlaylist** and **PlaylistFactory** joined the cluster #7. These two components used **PlayList** and **Config**, and realized playlist-related features. After that, **SkinLoader** and **FileUtil** joined the cluster #7. The latter two components didn't use **PlayList**, however, they used **Config** and other classes in common. If you move the red line upward, components using **Config** joined to the cluster. There were 15 components using **Config**, which were scattered across Cluster #5, #6 and #7.

In this way, most of the components in the cluster were functionally related components, and these components were divided into two large categories. The components of the first category had a simple structure and a simple use-relation and realized a single function by derivation. The purpose of these components was consistent with the purpose of commonly used components. The components of the second category used several components and realized multiple functions. The purposes of the commonly used components were integrated and it becomes the purposes of the components. When we have to realize another screen for the software, these components might be useful as examples. In this way, understanding the commonly-using components was essential to understand the roles of the components in the cluster.

Table III lists the jlGui components in order of the number of components that use the component. Most of the components in the list were mentioned in our case study, so we thought the classification result was appropriate as a whole.

TABLE III
THE NUMBER OF COMPONENTS USING EACH JLGUI COMPONENT

Components	Used by
javazoom.jlgui.player.amp.skin.AbsoluteConstraints	15
javazoom.jlgui.player.amp.util.Config	15
javazoom.jlgui.player.amp.PlayerUI	10
javazoom.jlgui.player.amp.tag.TagInfo	10
javazoom.jlgui.player.amp.util.ui.PreferenceItem	9
javazoom.jlgui.player.amp.tag.ui.TagInfoDialog	7
javazoom.jlgui.player.amp.playlist.Playlist	6
javazoom.jlgui.player.amp.playlist.PlaylistItem	6
javazoom.jlgui.player.amp.skin.Skin	6

V. CONSIDERATION

A. Evaluation of Experiments

We classified jlGui components based on the similarity of the components used by each component and confirmed that many of the components in the obtained clusters showed strong relationships with each other. It is effective to understand the role of the components in the cluster based on the role of commonly used components and this approach is effective as a systematic method to efficiently understand the inside of the software. As in the situation that components using **AbsoluteConstraints** distributed in several clusters, components that use the same components may spread across multiple clusters. In that case, we think that the use-relations may be different depending on the role difference, so these components are classified into different clusters.

However, there are cases where the roles of commonly used components do not directly indicate the role of the components, and the roles of commonly used components are ambiguous. There seem to be some useless cases in this way, but we consider that our method is effective as a whole.

B. Refining Approaches for the Proposed Method

In the case of jlGui, there were about 20 components which did not use any classes in jlGui, and these were not classified. In the maximum case, about 30% to 50% of the components may not be classified. Some components were realized by not using components in jlGui but using only library components. So the number of classified components would be increased if we also consider use relations to the external components. Classification would be affected by how use-relations to the external components are treated in the calculation of similarity. So we would like to confirm whether consideration of external components is effective or not, and propose a refined method.

Moreover, for Cluster #1 and #2, and for #4 and #5, an integrated cluster may be appropriate categorization when we consider functional groups in a broad sense. In some cases, the result would be improved by moving the threshold height up and increasing the size of the cluster. On the other hand, in other cases, the result would be improved by moving the threshold height down and selecting components carefully. We would like to confirm the appropriate threshold trend.

C. Usages of the Proposed Method

First of all, as a method of using the proposed method, we would like to use it as a method for the efficient grasp of components and inside of the software. Next, we also believe that some contributions can be made to maintenance work. For example, we consider that it can also be used for checking for omissions of correction in maintenance work. As a method of assisting such checking work, you may use the commit history or similar code fragments, and we expect the same effect as those. Moreover, we think that it is effective to suggest components with similar use-relationships, as reference examples, when adding new components to implement additional features in maintenance. Finally, we consider that our method supplies a good viewpoint to express the software's evolutionary process. By comparing the dendrograms of each version in the middle of development, you can grasp which part of the software is growing and can grasp the direction of evolution and take countermeasures for it.

D. Comparison with Other Existing Methods

Currently, a comparison is not done as an experiment, and comparison with existing classification method is a problem to be tackled in future. In the feeling of calculating the recalls of Experiment 1, our method seems to perform better classification within the scope of analysis between classes, compared to the result of classification based on package hierarchy and based on similar code fragments. In the case of package-based classification, classification is done with a larger granularity, so a certain package contains several types of components. Our method can classify them components based on their purposes. In the case of code clone-based analysis, as the modification after copying increases, a similar code fragment becomes a gaped clone and it becomes difficult to grasp. Descriptions using other classes remain at the root, so the method is often highly resistant to modification.

E. Threats to Validity

In this study, we applied our method to only one software system, so it is necessary to discuss generalization by applying to various software systems. The main concern is whether the same tendencies can be confirmed in other domain software and whether similar tendencies can be confirmed in different software systems of different sizes. The purposes of components in one software system are considered consistent, so it may have derived good results. In order to work well, a certain degree of commonality is required for target components, so it might not work well if the target of the analysis was multiple software systems. In this regard, the classifications may be effective if components generated by a certain product line were analyzed. There is a possibility that the approach of the class definition in Java and our classification method have successfully engaged. Whether our method works well for software written in other programming languages is unknown.

F. Related Works

From the viewpoint of use-relation analysis, architecture recovery is an active research field. Zhang represented the object-oriented system using a class graph and proposed a clustering algorithm to restore the high-level software architecture [9]. Constantinou represented the hierarchical relationships between components as a D-layer and examined the relationship between the architecture layer and the design metrics [10]. Kula et al. proposed the Software Universe Graph (SUG) Model as a structured abstraction of the evolution of software systems and their library dependencies over time and demonstrated its usefulness by showing several views of visualizations [11].

There are many studies to extract patterns from the call history of API on the source code. Zhong at el. proposed a system to extract call sequences of API from open source software's repositories [4], and showed that the system is useful for learning how to use the library's API. Li at el. proposed a system to extract function call sequences from C codes as programming rules, and the system provided a mechanism for detecting a portion that violates the rules [5]. Our method differs from these methods in that the granularity of our analysis is between components. We believe that it can extract informative information even in component-based approach.

As a method to extract similar code fragments, developers can use code clone analysis. Mondal analyzed the stability of several kinds of cloned codes, and they reported that Type3 clones, known as gapped clones, have higher stability than other clones [12]. Antoniol analyzed the evolution of code duplications in 19 versions of the Linux kernel [13]. Yoshida et al. proposed an approach to support clone refactoring based on code dependency among code clones [14].

VI. Conclusion

In this paper, we proposed a classification method for software components based on similarity of use relation. For each pair of components, the distance was calculated from the coincidence of their using components, and the hierarchical clustering is performed based on the distances. From the result of experiments, we confirmed that many components of the cluster were strongly similar with each other from the viewpoint of their functions, and the purpose of commonly used components greatly helps to understand the role of components in the cluster. We confirmed that our method is effective for classifying components, and is useful for understanding them. For future works, we would like to compare properties with other classification methods including stochastic block model, confirm its generality, and improve accuracy. Moreover, we would like to realize supporting tools for software understanding and for maintenance activities.

Acknowledgment

This research is supported by Nanzan University Pache Research Subsidy I-A-2 for the 2018 academic year. We would like to thank H. Kagami, N. Mase, and M. Sawai. This research is based on their bachelor thesis [15]. We were supervisors of it and added experiments and evaluations.

References

[1] B. Dagenais, H. Ossher, R. K. E. Bellamy, M. P. Robillard, and J. P. de Vries, "Moving into a new software project landscape," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering (ICSE2010)*, 2010, pp. 275–284.

[2] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code," *IEEE Transactions. Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[4] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, 2009, pp. 318–343.

[5] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *proceedings of the 10th European software engineering conference*, 2005, pp. 306–315.

[6] Classycle, http://classycle.sourceforge.net/.

[7] C. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[8] K. Kobori, T. Yamamoto, M. Matsushita, and K. Inouee, "Classification of java programs in spars-j," in *proceedings of the International Workshop on Community-Driven Evolution of Knowledge Artifacts*, 2003.

[9] Q. Zhangs, D. Qiu, Q. Tian, and L. Sun, "Object-oriented software architecture recovery using a new hybrid clustering algorithm," in *Proceedings of the Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, 2010, pp. 2546–2550.

[10] E. Constantinou, G. Kakarontzas, and I. Stamelos, "Towards open source software system architecture recovery using design metrics," in *Proceedings of the 15th Panhellenic Conference on Informatics*, 2011, pp. 166–170.

[11] R. G. Kula, C. D. Roover, D. M. German, T. Ishio, and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem," in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER2018)*, 2018, pp. 288–299.

[12] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," in *Proceedings of the 27th ACM Symposium on Applied Computing*, 2012, pp. 1227–1234.

[13] G. Antoniol, U. Villano, E. Merio, and M. D. Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[14] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *Proceedings of the 11th IEEE International Software Metrics Symposium*, 2005, pp. 16:1–16:10.

[15] H. Kagami, N. Mase, and M. Sawai, "Evaluation of software parts classi cation method based on commonality of parts to be used and use source," in *Bachelor Thesis of Nanzan University*, 2018, (In Japanese).