IEICE TRANSACTIONS

on Information and Systems

VOL. E101-D NO. 1
JANUARY 2018

The usage of this PDF file must comply with the IEICE Provisions on Copyright.
The author(s) can distribute this PDF file for research and educational (nonprofit) purposes only.
Distribution by anyone other than the author(s) is prohibited.

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY

The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

Changes of Evaluation Values on Component Rank Model by Taking Code Clones into Consideration

Reishi YOKOMORI[†a)], Norihiro YOSHIDA[††], Masami NORO[†], *and* Katsuro INOUE[†††], *Members*

SUMMARY There are many software systems that have been used and maintained for a long time. By undergoing such a maintenance process, similar code fragments were intentionally left in the source code of such software, and knowing how to manage a software system that contains a lot of similar code fragments becomes a major concern. In this study, we proposed a method to pick up components that were commonly used in similar code fragments from a target software system. This method was realized by using the component rank model and by checking the differences of evaluation values for each component before and after merging components that had similar code fragments. In many cases, components whose evaluation value had decreased would be used by both the components that were merged, so we considered that these components were commonly used in similar code fragments. Based on the proposed approach, we implemented a system to calculate differences of evaluation values for each component, and conducted three evaluation experiments to confirm that our method was useful for detecting components that were commonly used in similar code fragments, and to confirm how our method can help developers when developers add similar components. Based on the experimental results, we also discuss some improvement methods and provide the results from applications of these methods.
key words: *component rank, code clone, component graph, use relation*

1. Introduction

The size of many software applications continuously increases with prolonged maintenance due to accommodating many additional features. Therefore, the number of classes that compose such software increases, and relationships between such classes also increase and become exceedingly complicated. When implementing new features, developers add new code fragments to the software. Sometimes these code fragments are very similar to existing code fragments. This is because, implementing a similar feature also requires producing similar code fragments. Such similar code fragments are called code clones [1], [2]. It is generally desirable to remove or not to create a code clone; however, some types of code clones are difficult to resolve. Therefore, similar code fragments are sometimes intentionally left in the source code, and knowing how to manage the software that contains a lot of similar code fragments becomes important when one desires to maintain the source code.

Manuscript received April 10, 2017.
Manuscript revised September 5, 2017.
Manuscript publicized October 5, 2017.
[†]The authors are with Department of Software Engineering, Nanzan University, Nagoya-shi, 466–8673 Japan.
[††]The author is with Graduate School of Informatics, Nagoya University, Nagoya-shi, 464–8601 Japan.
[†††]The author is with Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: yokomori@nanzan-u.ac.jp
DOI: 10.1587/transinf.2017EDP7125

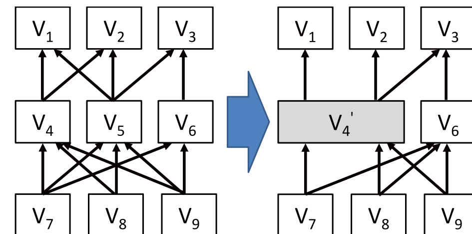

Fig. 1 Idea of our approach: Merging clone-related components.

On the other hand, our research group proposed a ranking model for recommending desirable components from many components by extracting use relations between components and performing repeated computation, and we call it the component rank model [3]. The purpose of the component rank model itself is for recommending desirable components from a viewpoint of how many components use the component.

In this paper, we extend the component rank model by reflecting upon relationships of code clones between components on the component graph. In our approach, components that have similar code fragments are merged in the graph. For example, we consider a part of the component graph in Fig. 1. It has nine nodes that represent nine components, where V_4 and V_5 have similar code fragments to each other. In this case, we merge V_4 and V_5 into V_4', where the edges from the merged components and the edges to the merged components are also merged, respectively. Then, we re-calculate a second evaluation value based on the merged graph to obtain two evaluation values of each component before and after reflection of the code-clone relations.

In this study, we focused on how the evaluation value of each component on the ranking model changes in the above operation, and examined the characteristics of components whose evaluation value have decreased. Such components were mainly components that were used by several components in the merged components. The fact that the several components were using the same component sometimes indicated duplications of similar code fragments, and sometimes indicated that the same component is merely used for another purpose. Depending on the detection technique of code clone, even if the same component was used by outside of similar code fragments, it may be related to the duplication of the similar code fragments if we considered them as gapped clones [4]. Even if it is not clear whether com-

Copyright © 2018 The Institute of Electronics, Information and Communication Engineers

ponents in the merged components were using the component same way in their similar code fragments or not, we made a hypothesis that components whose evaluation value decreases tend to be used by **similar code fragments** in the merged components, and we would suggest to leverage such components.

Before merging, such components could obtain evaluation values from each component in the merged components; however, it became evident to only provide an evaluation value from the merged ones after merging. From Fig. 1, V_1 (V_2) was used by both of V_4 and V_5, so evaluation value of V_1 (V_2) would decrease, and we made a hypothesis that $V_1(V_2)$ would be used in a similar way by V_4 and V_5. Our method is to pick up components, such as V_1 and V_2, that tend to be used by components that have similar code fragments. We proposed that our method can consider a degree of reduction of use relations from the merged components by using a degree of change for the evaluation values, compared with the simplistic approach of making a list of the commonly used components from the merged component.

As a method to leverage such components, we consider a situation where components that realized several similar features had already been implemented in the software. Then, developers are adding new components to realize a similar additional feature. In such situations, it is easy to produce new code fragments that are similar to the existing code fragments, because developers are trying to implement the new feature that is similar to the existing features, and we consider the followings are alternatives of actions by developers.

1. Developers extract similar code fragments by analyzing the existing components based on clone detection tools, and perform a refactoring activity against the existing components and a new component.
2. Developers search components that have realized similar features as examples of the usages of the particular component, and add a new component in the same way as the detected components faithfully.
3. Developers are permitted to add a new component using their own method without any constraints.

If the action #1 is performed, it is possible to gather identical processing for one code fragment, so the maintainability of the software improves and it would provide the best result among the three alternatives from a viewpoint of the maintainability. However, developers cannot always perform action #1 depending on the condition of the source code. In such cases, action #3 is easy to be performed in the actual development when the number of components involved increases or the description of components become complicated, and followed by understanding of the source code becomes more difficult because there are several realization methods to implement similar features. Maintainability of the software decreases substantially because developers have to prepare solutions for each code fragment. If action #2 is performed, developers add new features with an awareness of the presence of the similar code fragments. Thus, a new similar code fragment is added to the software, and the time and effort to maintain consistency of the source code increases. However, the cost for understanding the source code does not change significantly because similar processing was realized by one realization method. Maintainability of the software also would be retained because the developers have to prepare only one type of solution.

We consider that developers should first perform action #1 against code clones that are easy to solve, followed by action #2 against cloned code that are difficult to solve. To achieve action #2, developers must grasp that there are similar code fragments in the existing components, and the new component has some common characteristics with the existing components. Based on the change of evaluation values on the ranking model, we consider that our method can support for keeping maintainability of the software or identifying components that required refactoring, by showing components that were used by similar code fragments. We consider that our methods can help developers in the action #2, because our methods can extract components that would be commonly used in existing similar code fragments, and can support to grasp the common characteristic that a new component is just using the target component.

Based on the proposed approach, we implemented a system that shows the components whose evaluation value has changed after reflection of the code-clone relations. By using the system, we conduct three evaluation experiments. In the first experiment, we applied our method to the several open-source projects and confirmed that the evaluation value of the components used by several components in the merged components would decrease or not. In the second experiment, we searched case samples, and verified how many assumed cases existed in the actual software, and confirmed that a percentage of the reduction of the evaluation value was useful for detecting components that were commonly used in similar code fragments. In the third experiment, we introduce what components were detected by our methods and how these components were used by the merged components, as a case study, and discuss how our method can help developers when developers add similar components.

This paper is a revised version of [5]. In the introduction, we added a description how our approach can help developers, technical explanations about our approach, and case studies to examine what components can be extracted, as the third experiment. From them, we will discuss how the proposed approach can help developers when developers add similar components. We also added considerations of the improvement methods based on the results of three evaluation experiments, and explained further experiments to examine several approach for improvement. We consider that these two things are the major contributions of this version.

In Sect. 2, we introduce the component graph and the component rank model as background. In Sect. 3, we consider how the evaluation value of each component may change when components that have common similar code

fragments were merged on the component graph. In Sect. 4, we introduce our implementation. In Sect. 5, we conduct three evaluation experiments. Finally, in Sect. 6, we discuss the results and further experiments for improvement methods and introduce related works.

2. Component Graph and Component Rank Model

In general, a component is a modular part of a system that encapsulates its content and whose manifestation is replaceable within its environment [6]. We model software systems by using a weighted directed graph, called a *Component Graph* [3]. 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*, in other words, component x uses component y.

Based on the component graph, we proposed the Component Rank Model [3]. In the model, the component graph represents a Markov chain, and movement of the software developer's focus on the target software is represented by a probabilistic state transition. The weight of each node at the steady state is regarded as the evaluation value of the corresponding component, and this model calls the order of the components sorted by the evaluation value component rank of the components. The component rank model was proposed as a part of the component search engine, and we confirmed that a few components obtained the majority of the evaluation values, and these components were very general and core classes. On the other hand, many components had minimum evaluation values or close to the minimum ones, and these were specific and independent classes.

This model introduces several definitions to calculate the evaluation values (weights) for the component graph $G = (V, E)$. Each node v in the component graph G has a nonnegative weight value $w(v)$ where $0 \leq w(v) \leq 1$. The sum of the weights of all nodes in G is 1, and the total weights of the nodes are kept as 1.

The calculation process encompasses the following steps:

1. Set the initial weights for each node.
   Each node has $1/n$ as an initial weight if the target system consists of n components.
2. Calculate the weights of the edges, and re-calculate the weights of the nodes.
   2-1 Calculate the weights of the edges from the weights of the nodes.
   For each edge $e_{ij} \in E$ from v_i to v_j, we define a weight $w'(e_{ij})$ of e_{ij} as follows:

   $$w'(e_{ij}) = d_{ij} \times w(v_i)$$

   Figure 2 (a) depicts this definition. Here d_{ij} is called a *distribution ratio* for edge e_{ij}, where $0 \leq d_{ij} \leq 1$, and if there is no edge from v_i to v_j, $d_{ij} = 0$. The distribution ratio, d_{ij} is used for determining the forward weights of v_i to an adjacent node v_j. For each node which has any outgoing edges, the sum of the distribution ratios for the outgoing edges is always 1. In the current implementation, if a node a has n outgoing edges, then the distribution ratios of the outgoing edges from a are all $1/n$. In the actual calculation, we must treat the target component graph as a strongly-connected graph to guarantee the termination of the repeated computation, so we introduce pseudo use relations between all nodes. This is not included in this paper, so please refer to [3].
   2-2 Re-calculate the weights of the nodes from the weights of the edges.
   The weight of a node v_i is re-defined as the sum of the weights of all incoming edges e_{ki}, such that:

   $$w(v_i) = \sum_{e_{ki} \in \mathrm{IN}(v_i)} w'(e_{ki})$$

   Here, $\mathrm{IN}(v_i)$ is the set of the incoming edges of v_i. Figure 2 (b) shows this definition.
   2-3 Repeat calculation **2-1** and **2-2** until convergence.
3. The weight of each node at the steady state is regarded as the evaluation value of the corresponding component, and the components are sorted by the evaluation value.

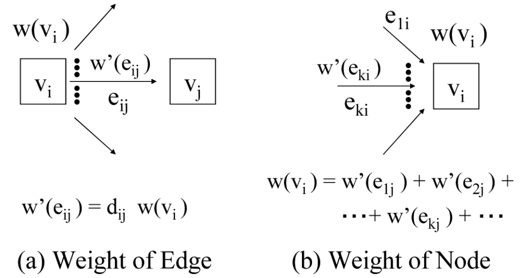

Fig. 2 Definition of weights.

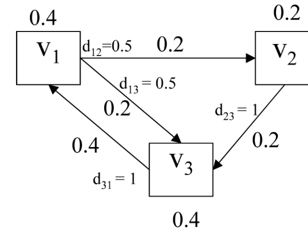

Fig. 3 An example of the stable weights assigned to the nodes and edges.

Figure 3 shows a component graph with the evaluation value set to the steady state. Also, v_1 has two outgoing edges, and weight = 0.4 is evenly divided between the two outgoing edges with 0.2 each (i.e., $d_{12} = d_{13} = 0.5$). For example, v_3 has two incoming edges, each with weight = 0.2, so that the total weight of v_3 is 0.4.

3. Representing Code Clone Relations on the Component Graph

A code clone is a code fragment that has identical or similar code fragments to that in the source code [1]. Code

clones appear not only in a single component, but also appear between different components. It is generally desirable not to create a code clone; however, code clones are sometimes woven with a certain intention. In such cases, developers must share information about the code that has similar code fragments, to satisfy software maintainability. This is because developers should review other code fragments to maintain consistency when one such code fragments should be modified.

In this paper, we extend the component rank model by revealing the relationships of the code clones. As a method for revealing code clone relationships, we merge the nodes which have the same or similar code fragments. For example, we consider a part of the component graph in Fig. 1. It has nine components, and V_4 and V_5 have similar code fragments to each other. In this case, we merge V_4 and V_5 into V'_4, and the edges are also merged. In the merged component graph, the edges are drawn from V_x to V'_4 if V_x uses V_4 or V_5, and the edges are also drawn from V'_4 to V_x if V_4 or V_5 uses V_x, respectively.

In this way, we obtain two component graphs, before and after merging, based on the relationships of the code clone and calculate two component ranks based on the two component graphs. We focus our attention on the components that are not merged, and calculate the differences of the evaluation value of each component. In this study, we have an interest regarding how the evaluation value of each component changes by revealing the relationships of the code clones. We examine how the evaluation value of the components would change as follows:

- Incoming edges to V_4 and V_5 are aggregated to one incoming edge to V'_4. So, the number of outgoing edges from V_7 and V_8 decreases and the evaluation value of these edges would increase. So, the evaluation value of each incoming edge increases.
- The evaluation value of V'_4 is a sum of all incoming edges, and hence would be larger than that of either V_4 or V_5. However, the evaluation value of V'_4 would not exceed that of the sum of V_4 and V_5. This is because the incoming edges to V_4 and V_5 are aggregated to one incoming edge. For example, we consider a situation where x nodes are merged, and the number of outgoing edges from a certain node becomes n to $n - x + 1$. The evaluation value of each edge becomes $\frac{n}{n-x+1}$ times; however, $\frac{n}{n-x+1}$ is equal to or less than x, so the evaluation value of the merged incoming edge would not exceed the sum of the incoming edges before merging.
- We consider a case of a component that is used by both merged components. V_1 is used by both V_4 and V_5. Before merging, V_1 obtains evaluation values via the edges from V_4 and V_5. After merging, the outgoing edges from V_4 or V_5 are also aggregated to one outgoing edge, and V_1 obtains an evaluation value via the edges from V'_4. The evaluation value of V'_4 does not exceed the sum of that of V_4 and V_5, so the evaluation value of V_1 decreases.
- We consider a case of a component that is used by only one of the merged components. V_5 uses V_3, but V_4 does not use V_3. Before merging, V_3 obtains an evaluation value via the edge from V_5. After merging, V_3 obtains an evaluation value via the edge from V'_4. The evaluation value of V'_4 is larger than that of V_5, so the evaluation value of V_3 would increase.
- All nodes are affected by the following factors:
  - In the repeated calculation, a decrease or increase of the evaluation values are spread to other nodes through their outgoing edges.
  - When a closed path is built by the merged components, components on the closed path give a part of their evaluation value to other components on the closed path, so the evaluation values increase.

Thus, the evaluation value of the component, which is used by several components among the merged ones, would decrease. We tried this operation in some examples, and confirmed that the changes of most of the evaluation values were about several tens of percent, and the variability of the evaluation values in the upper ranked and lower ranked components did not change so much. On the other hand, the changes on the ranking were quite different between the upper ranked components and the lower ranked ones. Even if the evaluation value had decreased, the ranking of higher-ranked components did not change so much, this is because it had still sufficient evaluation value after the operation. At the same time, the ranking of lower-ranked components were very sensitive about the change of the evaluation value, this is because there were many components with low evaluation values.

It was difficult to evaluate all components properly from a viewpoint of the changes on the ranking, so our method only considered changes on the evaluation values. We also believe that there is a difference of the decreasing degree by how the component is used by other components. Components that receive evaluation values from merged components and other components would have a low decreasing degree. On the other hand, components that are only used locally by the merged components would exhibit a high decreasing degree. Hence, we set a hypothesis that "Components whose evaluation value decreases tend to be used by similar code fragments in the merged components."

4. Implementation of the Analysis Tool

We implemented a tool that calculates two component ranks and compares the difference of each component' s evaluation value. In this tool, our analysis target is a Java software system, and we choose a .java file as a component. This tool is implemented by PHP, and Fig. 4 is an overview of the system. The following is the analysis procedure:

1. CCFinder [1] get information about the code clone. We treat two .java files to have a clone relation if they have

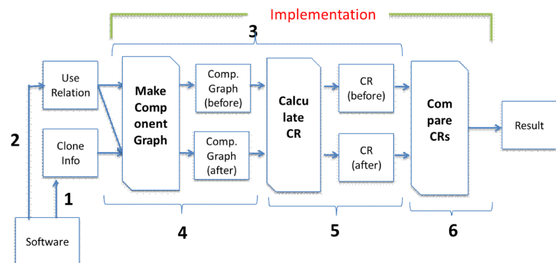

Fig. 4 Overview of the analysis tool.

similar code fragments that are longer than 30 tokens.

2. Classycle's analyzer[†] provides the use relations. In the analysis of the Classycle's analyzer, we get the following as use relations: inheritance of class, declaration of variables, creation of instances, method calls, and reference of fields. Classycle's analyzer's results are based on relations between the classes, so the results are mapped into relations between the files.

3. Calculate a component rank before merging.

4. Merge components that have a clone relation, and the use relations are also re-considered on the merged component graph.

5. Calculate a component rank after merging.

6. Compare the two component ranks and the result of the comparison is outputted as a table.

5. Experiments

To confirm that we can detect components that would be used by similar code fragments in a similar way, we performed the following two experiments.

1. In the previous section, we discussed how the evaluation value of each component changes by merging, and we conjectured that the evaluation value of the component that is used by several components among the merged ones, would decrease. We focus attention on the non-merged components, and these are then categorized, and we apply our method to the source code of the open-source projects. In this experiment, we confirm that the evaluation values of the Group A components decrease more than the other components (Group B or C). We would like to check the adequacy of our consideration, and consider actual use.

   Group A These components are used by several components in the merged components. We can imagine a situation that there are several sets of the merged components in a component graph. In such a case, if the component meets the requirement against at least one of the set, then the component belongs to this group.

   Group B These components are used by only one of the merged components. This means that the component of this category is used only once at most from all the sets of the merged components, and some of the sets may not use the component.

   Group C Other components are components that are not used by any sets of the merged components.

2. We obtain Group A components using experiment #1. For each Group A component, we investigate whether the merged components use the Group A component in a similar way or not, by visually reviewing the code. We check the adequacy of our hypothesis that, "Components whose evaluation value decreases tend to be used by similar code fragments in the merged components."
3. In experiment #2, we found several Group A components whose evaluation value had decreased. As a case study, we investigated how such Group A components were used by the merged components that have similar code fragments. Through the explanation of the case study, we evaluate how our method can support existing codes implemented by Action #1∼#3, described in Introduction.

5.1 Rate of Variability

The number of nodes decreased by merging the components; however, the evaluation value is distributed relatively evenly between each component, and the sum of the evaluation values are always 1. So, we must compare two evaluation values in view of the decrescence of the nodes.

To compare two evaluation values, we suppose the following situation; first, the number of nodes in the component graph before merging are represented as V_{before}, and an evaluation value of node v_i is $w(v_i)_{before}$. After merging, the number of nodes decreased into V_{after}, and the evaluation value of the node became $w(v_i)_{after}$. We define a rate of variability of node v_i as following;

$$\frac{w(v_i)_{after} \times V_{after}}{w(v_i)_{before} \times V_{before}} \times 100 \tag{1}$$

5.2 Experiment #1: Changes in Evaluation Values

We selected 45 open-source projects in SourceForge. This selection was performed via a search function of SourceForge, and we selected projects that have both source code files (.java) and bytecode (.jar) and we did not have any other intentions. For each project, we got one version of the software and applied our method to it, and calculated a rate of variability for each component. We categorized the non-merged components into Groups A, B, and C, and calculated an average variability rate for Groups A, B, and C, respectively.

Table 1 shows an average of the rate for components in Groups A, B, and C per project, respectively. Some projects did not have any components belonging to Group A (B, or C), so the number of projects for each category was less

[†]http://classycle.sourceforge.net/

Table 1 Average of rate of variability for each project.

	Projects	Average	Standard Deviation	Increased	Decreased
Group A	43	88.5%	19.6	3	40
Group B	41	107.7%	16.2	19	22
Group C	43	104.3%	6.4	20	23

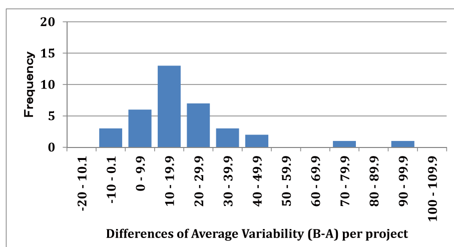

Fig. 5 Histogram of the difference of average rate (B -A) per project.

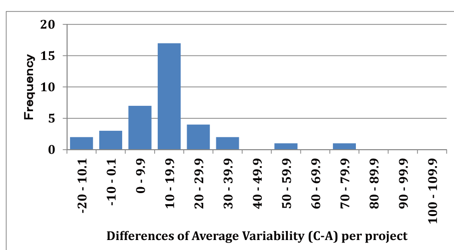

Fig. 6 Histogram of the difference of average rate (C -A) per project.

than 45. We confirmed that an average rate of variability for Group A components decreased 10%; whereas, that for the components in Groups B and C increased slightly. In this way, the average rate of variability for the components in Group A decreased more than that observed for Groups B and C. Group C components were not directly affected by the merging nodes, so the standard deviation of Group C was smaller than that of Groups A and B.

Some projects did not have any components belonging to Group A (B, or C); however, components were present in both Group A and B in 36 projects, whereas components were present in both Group A and C in 37 projects, respectively. For each respective project, we calculated the difference between the average rate of variability for the components in Groups A and B, and also the difference between that for Group A versus that for Group C. Figure 5 shows a distribution of the differences between Group A and Group B, and Fig. 6 shows a distribution of the differences between Group A and C, respectively. In a few project, there were some Group A components whose evaluation values increased after merging components. This is because propagations of indirect values caused by the change of the graph structure were larger than the direct changes in values, and details will be discussed in the discussion. Many projects were plotted in a range of 10% from 20%, and in 80-90% of the projects, we confirmed that Group A components lost evaluation values more than the other components.

We also analyzed these differences statistically by using the Welch's t-test, and confirmed that there was a statistically significant difference between Groups A and B ($\alpha = 0.01, d = 1.04$, and $1 - \beta = 0.99988$), and there was also a statistically significant difference between Groups A and C ($\alpha = 0.01, d = 0.94$, and $1 - \beta = 0.99932$), respectively. From these results, we concluded that the evaluation value of the component that was used by several components of the merged components tended to decrease more than the cases for the other components.

5.3 Experiment 2: Components Used by Merged Components

We selected 34 open-source projects from 45 projects in Experiment #1 due to time constraints, and the non-merged components were categorized in the same manner as Experiment #1. In Experiment #2, we focused on the Group A components. For each Group A component, we analyzed how the component was used by the merged components by visually reviewing the code, and whether the component was used in a similar way or not. This means that the component was used by similar way inside or outside of the detected code clone in the merged components, and their behaviors using the component were almost identical. Mainly, we judged it based on whether there were common descriptions that used the component in the processing, or before or after the processing. This is because CCfinder [1] cannot detect gapped clones [4], so even if the same component is used by outside of similar code fragments, it may be related to the duplication of the similar code fragments if we considered them as gapped clones. We categorized how the component was used in the merged components, and discussed the points to be considered when we analyze code clones with considering use relations.

From the 34 open-source projects, we extracted 423 components that belonged to Group A, and checked whether the component was used in a similar way or not. We confirmed that 269 components (64%) were used by the merged components in a similar way. On the other hand, 154 components (36%) were used by merged components in a different way. Hence, almost two-thirds of Group A components were used by the merged components in a similar way. There were several merged sets, so we confirmed 339 examples from the 269 components. We classified these 339 examples based on how the component was used. In 195 cases, we confirmed that the components were used by a method-call or creation of an instance, and so on, in the similar code fragments. In many of the remaining cases, the components were inherited by several subclasses, and these subclasses had common descriptions that were recognized as code clones. In other cases, the Group A components were exception classes, and these components were used in similar exception handlings.

Among the 34 projects, there were 24 projects that have more than three components that belong to Group A. For each project, we calculated a value that are obtained by dividing the number of Group A components that were used

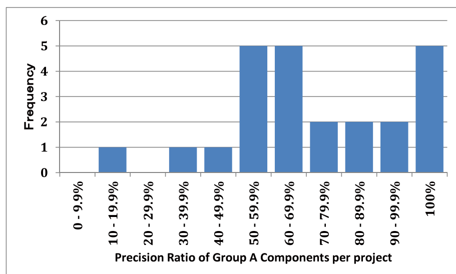

Fig. 7 Histogram of the precision for each project.

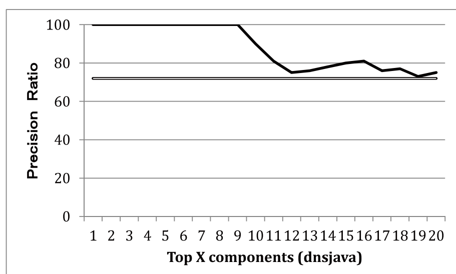

Fig. 8 Precision ratio for the top X cases (dnsjava).

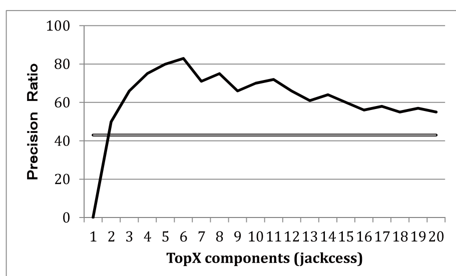

Fig. 9 Precision ratio for the top X cases (Jackcess).

by the merged components in a similar way by the number of Group A components, and considered it as a precision ratio of the project. Figure 7 shows a distribution of the precision ratio for each project. We confirmed that there was a distribution peak around the average (64%); on the other hand, there were several projects whose precision ratio was 100%.

In the next analysis, we evaluated based on cases that were represented by pairs of Groups A component and merged components. We selected 2 projects, dnsjava[†] Ver.3.4.1. and Jackcess[††] Ver.2.0.4. that had more than 20 cases. For each project, these cases were arranged in order of decreasing rate of the evaluation value of the Group A component, and we calculated a precision ratio for Top X ($X = 1, \cdots, 20$) cases as the same manner of the previous one. Figure 8 is a transition of the precision ratio for the Top X dnsjava cases, and Fig. 9 is that for the Top X Jackcess cases, respectively. In these figures, a solid line shows the precision ratio for the Top X cases, and a horizontal line shows the precision ratio for all cases in the projects.

For dnsjava, Group A component was used by the merged components in a similar way in the cases whose decreasing rate were from the first to the ninth, so the solid line in Fig. 8 keeps 100% till the ninth. For Jackess, Group A component was used by the merged components in a different way in the case whose decreasing rate was the largest, however, Group A components were used by the merged components in a similar way in the cases whose decreasing rate were from the second to the sixth, so the solid line in Fig. 9 was increasing tendency from the second to the sixth. With a few exceptions, the line of the precision ratios always remained above the precision ratio for all cases in the projects, so we considered that the components whose evaluation value had decreased significantly fitted our hypothesis well.

5.4 Experiment 3: Case Studies

For each top five Jackess and dnsjava cases in the last analysis of Experiment #2, we investigated how Group A component was used by the merged components that have similar code fragments as a case study. Table 2 and 3 is a summary of this investigation, and it represents top five cases and contains the name of Group A component, the names of the merged components that have similar code fragments, the Group A component was used in similar way or not, common points, and the results of estimate of what the developer had taken in Action #1-#3. About what actions are expected when developers attempt to add similar functions, we presented it at the Introduction as Action #1-#3. We estimated it from the similarity of implementation methods of class members related to the Group A component. As a result, we couldn't find cases implemented by Action #1 and detected only cases implemented by Action #2 and #3. This is because our method is based on code clones, and code clones would not be detected when developer had taken Action #1.

In the cases of dnsjava, cases that were used by two component-groups were ranked high. One is a group consisting of 30 components that implements various types of DNSRecord and protocol, and the other is a group consisting of two components. In the former case, about a half of 30 components used the corresponding Group A component in similar way, however, the remaining components were unrelated ones that didn't use the Group A component. Before merging, the Group A component obtained evaluation value from each half of these 30 components. After merging, the Group A component obtained evaluation value only from the merged ones, so evaluation values for such Group A components have decreased a lot. In the 1st case of dnsjava, contents related to Mnemonic were very simple, so class members related to Mnemonic were implemented with consistency. And some cases used by two classes were not complicated, so we consider that such case would be before the derivation. On the other hand, in the case of 2nd and 4th, we confirmed that there were several kinds of imple-

[†]http://sourceforge.net/projects/dnsjava/
[††]http://jackcess.sourceforge.net/

Table 2 Summary of top 5 dnsjava cases.

	GroupA	Merged Components that have similar code fragments	Similar Way?	Common Point	Estimated Approach
1	Mnemonic	30 Files(CERTRecord,DSRecord, EDNSOption, KEYBase, KEYRecord, SIGBase, SOARecord, TKEYRecord, TSIGRecord, and so on) . . . (A)	Yes	For keeping configuration in 11 Classes	Action #2
2	Tokenizer	(A)	Yes	For realization of rdataFromString() in 18 Classes	Action #3
3	Tokenizer	APLRecord and TXTBase	Yes	For realization of rdataFromString()	Action #2
4	Compression	(A)	Yes	Described as a parameter of rrToWire() in 18 Classes	Action #3
5	Compression	APLRecord and TXTBase	Yes	Described as a parameter of rrToWire()	Action #3

Table 3 Summary of top 5 Jackcess cases.

	GroupA	Merged Components that have similar code fragments	Similar Way?	Common Point	Estimated Approach
1	ComplexValue	ColumnImpl, CursorImpl IndexCursorImpl, IndexImpl and ComplexColumnInfoImpl	No		
2	ComplexValue	AttachmentColumnInfoImpl and VersionHistoryColumnInfoImpl	Yes	For Management of Id information in toValue()	Action #3
3	ComplexDataType	AttachmentColumnInfoImpl and VersionHistoryColumnInfoImpl	Yes	To use information of ComplexDataType in getType()	Action #2
4	QueryFormat	AppendQueryImpl and UpdateQueryImpl	Yes	To use static variable in QueryFormat	Action #2
5	QueryImpl	AppendQueryImpl and UpdateQueryImpl	Yes	They are derived from QueryImpl. For preprocessing in toSQLString()	Action #3

mentation among 30 components, and it is difficult to grasp the content effectively even if we review only the methods of the same name. The 6-9th cases were not introduced in the Table 2, however, these cases were also similar to the 2nd-fifth cases, and the implementation approaches of related methods derive into several ones.

For Jackess, Group A component was used by the merged components in a different way in the 1st case, however, Group A components were used by the merged components in a similar way in the cases from the second to the fifth, and these similar descriptions were minimum required and small in size. In case of such the small one, two similar descriptions were sometimes implemented by different approaches.

We consider that there are many software systems that contain a lot of similar components in order to manage various formats and protocols. The detail of these components would be different, so descriptions of these components would be easily derived from the difference. For efficient understanding, refactoring is needed to divide such similar descriptions into common and original sections. We consider that our method is not only useful to support action #2, but also useful to support action #3, this is because detected components could be a trigger of such refactoring.

6. Discussion

6.1 Experimental Results

From the results of Experiment #1, we confirmed that the evaluation values of the Group A components, those that were used by several components among the merged components, decreased more on average than those of Groups B and C. On the other hand, we also confirmed that the evaluation values of some components in Group A increased and those of the components of Group B or C decreased severely. Especially, there were three projects where the average rate of variability of the Group A components increased, and we investigated how the nodes in the component graph were merged for such cases.

- Components with low evaluation values were affected by indirect propagations caused by the change of evaluation value of components with high evaluation values. When a change of the graph structure was too large, components with low evaluation values were also affected.
- We confirmed that there were several groups of merged components. Some of the Group A components were used by several components in a certain group; however, some components were also used by only one component in another group. In such cases, the evaluation value of such components tended to increase.
- When a change of the graph structure was too large, or when the components whose role was quite different were merged, several closed paths were produced in the merged graph. In such cases, the distribution of the evaluation value was affected by the created closed paths.

From these results, we consider that it is not desirable to merge components gratuitously, and our method would be improved by setting a restriction for merging of the components based on relativeness between the components. These adjustments would be thought of as a configuration choice problem [7], and we will show a brief result of these adjustments in the next subsection.

From the results of Experiment #2, we confirmed that almost two-third of the Group A components were used by the merged components in a similar way. For actual use, we consider the ratio was insufficient to realize the method by simply extracting the components that were commonly used by the code clone-related components. We consider that sorting the components based on the variability rate seems to be efficient, as indicated in Figs. 8 and 9, and decreasing of evaluation value a lot due to the decreasing of its incoming edges caused by merging a lot of components that have similar code fragments. Our hypothesis that, “Components whose evaluation value decreases tend to be used by similar code fragments in the merged components.” worked to some extent, for components whose evaluation value decreased a lot.

We also classified 339 examples based on how the component was used by the merged components. From a viewpoint of clone analysis based on the use relation, this type of classification would be useful for identifying a particular kind of use relations that are closely related to producing code clones. For example, we consider that the following support would be realizable: when developers introduce some classes that inherit a particular class, it can be supported by providing coding comments as there are already some good examples that inherit the class, and these classes serve as a useful reference. Based on such an approach, we can control the existing code clone that derives and increases in various forms in the software. Since such differentiated code clones make it more difficult to detect code clones, our method would be useful for improvement of the quality of the software.

From the results of Experiment #3, we evaluate how our method can support existing codes implemented by Action #1~#3, mentioned in the Introduction.

- The code implemented by Action #1 is considered that some measures had already been undertaken, so no further support would be needed.
- If developers can take Action #2, similar descriptions would be simple ones, or would be spread only to a few components. Our method can support developers by presenting existing examples, as already described in Introduction. However, detailed support is required at the time of recommendation. For example, non-relevant components should be removed from the candidates of examples, and supporting system have to present which methods are associated with the use of the component.
- If developers take Action #3, similar descriptions have been already derived and are difficult to grasp efficiently. Our method can support developers by showing these derived descriptions that using the detected component, and by encouraging to perform refactoring against these derived descriptions. This support can improve a quality of the existing codes into the direction of Action #1 or #2.

Table 4 Results: Restriction method based on the package distance(CardMe).

Package Distance	9	8~3	2	1	0
Group A Components	33	31	32	12	9
Ave. Rate (Group A) (%)	99	98	96	60	39
Ave. Rate (Group B) (%)	113	113	114	115	112
Ave. Rate (Group C) (%)	121	120	121	113	112
Precision (%)	70	71	69	75	78
Recall (%)	96	92	92	38	29
F-measure	0.81	0.80	0.79	0.5	0.42

6.2 Application of Improvement Methods

As examples of restriction for merging of components, we attempted the following approaches for a few projects:

- We considered the distance between components on a package hierarchy when merging components.
- We changed the definition of the clone relation so that we only consider larger code fragments, namely those that are longer than 40 (50, 70) tokens.

At first, we applied a restriction method based on the distance between components on the package hierarchy when merging components. We defined the distance between two components as the number of movements when moving from one package to another package along a tree structure. We did not merge the components whose distance was longer than a threshold value. For example, we only merged components that belonged to the same package under the circumstances where the threshold value was 0, and we only merged components that belonged to the same package, or that were a parent and child relationship in a package hierarchy under the circumstances where the threshold value was 1, respectively.

Table 4 shows results when we changed the threshold value from 0 to 9. The target system was CardMe[†], which was an implementation of RFC 2426 - VCard. For each threshold value, we calculated a value that are obtained by dividing the number of components in Group A that were used by the merged components in a similar way by the number of components in Group A, and considered it as a precision ratio for the threshold value. To define a recall, we searched components that have similar code fragments. And then, we searched components that are used by these components in a similar way, and treat them as a population at recall. Hence, the recall shows how many components in the population were extracted as Group A components for a specific threshold. Components merged by clone relation don’t belong to the Group A-C, so some of the components in the population were undetectable by any threshold. The results indicate that an average rate of Group A components tend to decrease by making the distance restriction stringent. Precisions were also kept or improved by making the distance restriction stringent; however, the recalls decreased significantly when the restriction was made too

[†]https://sourceforge.net/projects/cardme/

Table 5 Results: Restriction method based on the size of similar code fragments (JMDNS).

Tokens	30(orig.)	40	50	70
Group A Components	15	13	11	3
Ave. Rate (Group A) (%)	103	90	92	76
Ave. Rate (Group B) (%)	110	103	105	102
Ave. Rate (Group C) (%)	87	102	99	102
Precision (%)	67	69	72	66
Recall (%)	90	81	72	18
F-measure	0.76	0.74	0.72	0.28

stringent. By applying this approach to several projects, the result was slightly improved when we set the distance to 2 or 4. This means that the average rate of Group A and its precisions were maintained or improved, and its recalls did not decrease.

Next, we changed the threshold value of the code clone detection in CCfinder. In this approach, we could consider only larger similar code fragments by ignoring small fragments, and we treated two .java files to have a clone relation if they have similar code fragments that are longer than 40 (50, and 70) tokens.

Table 5 shows the results when we changed the threshold value from 30 to 70 and precision and recall were calculated by the same manner of the previous one. The target system was JMDNS†, a Java implementation of the multicast DNS. In the case of 30 tokens, the evaluation values of Group A components increased on average. There were several similar code fragments that greatly change the structure of the graph by merging, and the changes of the evaluation values by indirect propagations were large. When the threshold value was small, several similar code fragments were coincidentally matched in the token structure, and the tool recognized them as code clones. By increasing the threshold value for detecting similar code fragments, the evaluation values of Group A components decreased on average. We considered that the changes of the evaluation values close to the assumption were caused by the exclusion of similar code fragments which matched coincidentally. We determined that the average rates of Group A components tend to decrease by making the length restriction stringent. Precisions were also improved by making the length restriction stringent; however, recalls decreased significantly. The condition of recall with 30 tokens was not 100% because some of the merged components were not combined under the other condition, and some of those components belonged to Group A under the other condition. By applying this approach to several projects, the result was improved when we set the length to 40 or 50. This means that the average rate of Group A and its precisions were improved, without making its recalls decrease significantly.

We found that both ways could reduce the merging components whose relationships were tenuous, and the average rates of Group A components and its precisions were improved. So these cases would be successful cases. On the other hand, the recalls dropped for both ways. We also found that making stringent restrictions had an opposite effect, and applying several restriction methods little-by-little appeared to be a good approach.

We can use also other restriction method by changing extraction method of code clone to make the relationship between merged components more predictable. By taking into account gapped clone [4], similar code fragments can be captured on a larger scale. For example, we consider that only components that are highly relevant would be merged by merging when more than 50% of the descriptions are included in their similar code fragments. Hence, we would like to try other restriction methods for improving the performance of our approach.

In this paper, we choose a .java file as a component. The granularity of a component also would affect the effectiveness. For the usage that introduced in the Introduction, we considered that the accuracy of information obtained becomes ambiguous one when the granularity of component is coarse, and the accuracy of information obtained becomes clarified one when the granularity of component is fine. We also would like to try other granularity, and then we also have to consider meaning under other conditions.

6.3 Related Works

From the use relation (dependency) analysis viewpoint, architecture recovery is an active research area. Zhang represents the object-oriented system by using the Weighted Directed Class Graph, and proposed a clustering algorithm for recovering high-level software architecture [8]. Constantinou represents hierarchical relationships between components as a D-layer, by contracting closed paths in a component graph, and investigated the relationships between the architectural layers and design metrics [9]. In the past research, we proposed a metric representing a change of component rank as an impact on source code updates [10]. We also focused on the change of the evaluation value of each component in this work.

From the clone analysis viewpoint, 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 [4]. Antoniol analyzed the evolution of code duplications in 19 versions of the Linux kernel [11]. Yoshida et al. proposed an approach to support clone refactoring based on code dependency (e.g., caller-callee, shared variable) among code clones [12]. Their approach presented a coherent set of code clones as a refactoring opportunity. Our research used the component rank model as a basis for consolidating clone information and displaying the results. The characteristics of our research paid attention to the components commonly used by cloned code. In [13], Yamanaka suggested a daily reporting system about modifications related to cloned codes. We consider that changes of component rank may also benefit a reporting system.

†http://jmdns.sourceforge.net/

7. Conclusion

In this paper, we extended a component rank model by considering code clones between components, and those components that had a similar code fragment were merged onto the component graph. We implemented a system based on the method, and we conducted three evaluation experiments. In the first experiment, we confirmed that the evaluation value of the component, which was used by several components among the merged ones, decreased. In the second experiment, we confirmed that almost two-thirds of the components that were used by the several merged components, were used by merged components in a similar way, and sorted the components based on a rate of variability which seemed to be efficient. In the third experiment, we confirmed that the proposed approach would be useful for preventing components from deriving more than necessary, and also be useful for suggesting the need for factoring.

From these results, we could confirm a certain degree of its utility, and a rate of variability seemed to be efficient for filtering non-related components. We also suggested improvement methods, and characteristics of these improvement methods were confirmed by the application results. After further improvement methods were estimated, we would like to implement a tool which helps to reduce code clones in various forms. The tool would contribute to support the software-maintenance work and the improvement of the quality of the source code.

Acknowledgments

This research is supported by Nanzan University Pache Research Subsidy I-A-2 for the 2017 academic year. I would like to gratefully and sincerely thank to E. Senga. Original idea of this research is based on his master thesis written in Japanese [14]. We are supervisors of the thesis, and we added all of the experiments and considerations for this paper and [5].

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," IEEE Transactions Software Engineering, vol.28, no.7, pp.654–670, 2002.
[2] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," Proceedings of International Conference on Software Maintenance, pp.368–377, 1998.
[3] 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.
[4] M. Mondal, C.K. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider, "Comparative stability of cloned and non-cloned code: An empirical study," Proceedings of the 27th ACM Symposium on Applied Computing, pp.1227–1234, 2012.
[5] R. Yokomori, N. Yoshida, M. Noro, and K. Inoue, "Extensions of component rank model by taking into account for clone relation," Proceedings of the 10th International Workshop on Software Clones, pp.30–36, 2016.
[6] C.W. Krueger, "Software reuse," ACM Computing Surveys, vol.24, no.2, pp.131–183, 1992.
[7] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp.455–465, 2013.
[8] Q. Zhangs, D. Qiu, Q. Tian, and L. Sun, "Object-oriented software architecture recovery using a new hybrid clustering algorithm," Proceedings of the Seventh International Conference on Fuzzy Systems and Knowledge Discovery, pp.2546–2550, 2010.
[9] E. Constantinou, G. Kakarontzas, and I. Stamelos, "Towards open source software system architecture recovery using design metrics," Proceedings of the 15th Panhellenic Conference on Informatics, pp.166–170, 2011.
[10] R. Yokomori, M. Noro, and K. Inoue, "Evaluation of source code updates in software development based on component rank," Proceedings of 13th Asia Pacific Software Engineering Conference, Bangalore, India, pp.327–334, 2006.
[11] 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.
[12] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," Proceedings of the 11th IEEE International Software Metrics Symposium, pp.16:1–16:10, 2005.
[13] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Industrial application of clone change management system," Proceedings of the 6th International Workshop of Software Clones, pp.67–71, 2012.
[14] E. Senga, "Evaluation of software components considered code clone," Master's thesis, Dept. of Mathematical and Information Science, Nanzan University, 2013. (in Japanese).

Reishi Yokomori received his Master and Ph.D. from Osaka University in 2001 and 2003, respectively. He has been with Nanzan University since 2005 and now he is an associate professor in the department of software engineering of Nanzan University. His research interests are program analysis and software development environments. He is a member of the IEEE, and the IEEE Computer Society.

Norihiro Yoshida received his B.E. from the Kyushu Institute of Technology in 2004 and his Master and Ph.D. from Osaka University in 2006 and 2009, respectively. He is an associate professor at Nagoya University. Before joining Nagoya University in 2014, he was an assistant professor at the Nara Institute of Science and Technology from 2010. His research interests include program analysis and software development environments. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Masami Noro received his Ph. Dr. degree from Keio University in 1988. He has been with Nanzan University since 1996 and now he is a professor in the department of software engineering of Nanzan University. His research interests are programming language semantics and software architecture.

Katsuro Inoue received the B.E., M.E., and D.E degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984 to 1986. He was a research associate at Osaka University from 1984 to 1989, an assistant professor from 1989 to 1995, and is a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environments. He is a member of the IEEE, the IEEE Computer Society, and the ACM.