Applying Clone Change Notification System into an Industrial Development Process

Yuki Yamanaka *, Eunjong Choi *, Norihiro Yoshida †, Katsuro Inoue *, Tateki Sano ‡
* Graduate School of Information Science and Technology, Osaka University, Japan
{y-yuuki, ejchoi, inoue}@ist.osaka-u.ac.jp
† Graduate School of Information Science, Nara Institute of Science and Technology, Japan
yoshida@is.naist.jp
‡ Software Process Innovation and Standardization Division, NEC Corporation, Japan
t-sano@cp.jp.nec.com

***Abstract*—Programmers tend to write code clones unintentionally even in the case that they can easily avoid them. Clone change management is one of crucial issues in open source software (OSS) development as well as in industrial software development (e.g., development of social infrastructure, financial system, and medical equipment). When an industrial developer fixes a defect, he/she has to find the code clones corresponding to the code fragment including it. So far, several studies performed on the analysis of clone evolution in OSS. However, to our knowledge, a few researches have been reported on an application of a clone change notification system to industrial development process. In this paper, we introduce a system for notifying creation and change of code clones, and then report on the experience with 40-days application of it into a development process in NEC Corporation. In the industrial application, a developer successfully identified ten unintentionally-developed clones that should be refactored.**

***Index Terms*—Code Clone, Software Maintenance, Refactoring**

I. Introduction

A code clone is a code fragment that has similar or identical code fragments in source code. Many code clone detection tools [1], [2], [3] have been proposed to capture various aspects of source code similarity. A code clone detection tool generally finds all source code clones that match its own definition of code clone; therefore, a tool may report a large number of code clones for large scale software. On the other hand, software developers are interested in only the subset of code clones that are relevant to their activities [4]. For example, although refactoring [5] is one of promising activities to improve the maintainability of code clones, code clone is not always appropriate for refactoring. One of the reasons is that developers sometimes have to repeatedly write code clones that cannot be merged due to the in-expressiveness of a programming language [6], [7], [8]. However, a clone set (i.e., a set of code clones identical or similar to each other) indicates considerable opportunities for developers to merge code clones into one or a few program units (e.g., Java methods) by refactoring [6], [7], [8].

Refactoring aimed to merge code clones is required not only in open source projects but also in industry. A development team at NEC Corporation, a Japanese multinational IT company, has been developed a web application software. Because the team plans long-time maintenance as well as reuse for other system developments, the developers are highly motivated to merge code clones into a single module.

However, the cost of refactoring cannot be ignored especially in industry. Regression test after refactoring takes much cost to preserve behavior after refactoring. The development team at NEC also considers the cost of refactoring. Basically, they do not touch source code after large-scale system test for releasing major version of the software because refactoring after large-scale test leads the re-performance of such costly test. Therefore, they need to know newly-appeared clones regularly, especially before large-scale system test.

In this paper, we present clone change notification system Clone Notifier (see Figure 3) for the promotion of efficient clone management (e.g., refactoring, simultaneous editing). Clone Notifier notifies newly-appeared and changed clones regularly to developers. As an industrial application, we applied Clone Notifier into the process of the web application software development at NEC. The result shows 119 newly-appeared clone sets, and ten out of them are recognized as refactoring candidates by an experienced project manager (i.e., he recognized that each of ten clone sets should be merged into a single module).

As an ex-post analysis, we investigated the characteristics of clone sets recognized as refactoring candidate by the experienced project manager at NEC. The aim of the analysis is data collection for the development of technique to recommend refactoring candidate from all newly-appeared and changed clones. The recommendation is promising to help developers to reduce the cost of finding clone sets should be merged into a single module.

The rest of paper is organized as follows: Section II provides a brief explanation of CCFinder, a code clone detection tool. Section III describes categorization of code clones and clone sets based on the evolution patterns between two versions of source code. Section IV explains on our developed clone change notification system, Clone Notifier. Section V describes results of industry application and feedbacks from project manager. Section VI explains ex-post analysis. Section VII discusses threats to validity. Section VIII presents some related

978-1-4673-3091-6/13/$31.00 © 2013 IEEE

ICPC 2013, San Francisco, CA, USA

work. Section IX summarizes our paper with final remarks and indications about our future work.

II. Code Clone Detection Tool : CCFinder

This section briefly explains on *CCFinder* [3] that we use as a clone detection tool. CCFinder is a token-based code clone detection tool. It takes source files as an input and outputs location information of code clones in source code. It detects identical code fragments except for variations in whitespaces and comments. It also detects structurally/syntactically identical fragments except for variations in identifies, literals, types, layout and comments [9].

The clone detection process of CCFinder consists of four steps:

- *Lexical analysis*: Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence, so that finding clones in multiple files is performed in the same way as single file analysis.
- *Transformation*: The token sequence is transformed (i.e., tokens are added, removed, or changed) based on the transformation rules that aims at regularization of identifiers and identification of structures. Then, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code fragments with different variable names to become clone pairs.
- *Match Detection*: From all the substrings on the transformed token sequence, equivalent pairs are detected as clone pairs.
- *Formatting*: Each location of clone pair is converted into line numbers of original source files.

The output file of it is represented in the form of *clone pair* (i.e., a pair of code clones) or *clone set* (i.e., a set of code clones identical or similar to each other).

CCFinder is widely used in the universities as well as industries because of its high speed and accuracy to detect code clones from source code. A division for software engineering at NEC is one of CCFinder users. Several members of the division have worked for the promotion of the use of CCFinder in NEC.

III. Categorization of Clone Clones and Clone Sets

Clone Notifier performs checkup of changed code clones from source code based on categorizations of code clones and clone sets. We defined these categorizations based on the evolution patterns between two versions of source code.

To describe categorization of code clones and clone sets, we defined V_i as source code at the point of time i, and C_i as a set of code clones detected in V_i. Input of this method is V_t (the latest version of source code) and V_{t-1} (the previous version of source code). These categorizations consist of the following steps:

Step1 : Detect C_t and C_{t-1} by analyzing overall V_t and V_{t-1} using CCFinder.

Step2 : Trace code clones between two versions based on a method which is described in section III-A.

Step3 : Categorize code clones in C_t and C_{t-1} based on a definition which is described in section III-B.

Step4 : Categorize clone sets in C_t and C_{t-1} based on a definition which is described in section III-C.

A. Tracing Code Clones

Our previous study [10] traced code clones across multiple versions based on correspondence of the start and end line of them in source code. This study uses the same method with our previous study to trace code clones between two versions.

Also, we defined the *parent-child relationship* to code clones between two versions to trace code clones. When code clone $B \in C_t$ corresponds to code clone $A \in C_{t-1}$, we define B as a *child clone* of A, and A as a *parent clone* of B. The following presents how we defined *parent-child relationship* and trace code clones between two versions in three different cases with figure 1.

Case 1: In Figure 1(a), two lines were inserted to A. This means that code clone $A \in C_{t-1}$ was modified between two versions. In this case, code fragment B in V_t corresponding to A can be traced by counting and inserted lines in A. Thus, the start line number of B is the same as A, and the end line number of B is increased by two lines. When C_t contains B, we define B is a child clone of A.

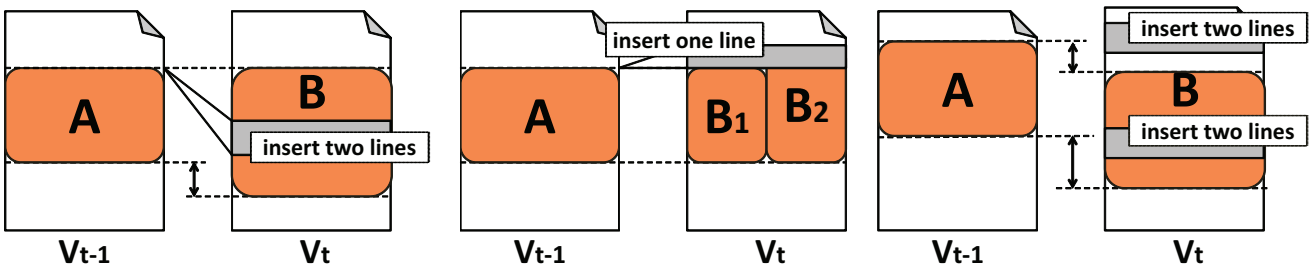
Case 2: In Figure 1(b), one line was inserted to the edge of A. This means that edge of code clone $A \in C_{t-1}$ was modified between two versions. If we regard one line was inserted above A, code fragment B_1 corresponds to A. In contrast, if we regard one line was inserted to A, code fragment B_2 corresponds to A. If C_t contains both of B_1 and B_2, we define the child clone as which has the nearest number of lines with A. In this case, because A and B_1 are identical, we define B_1 as a child clone of A.

Case 3: In Figure 1(c), two lines were inserted above A and two lines were also inserted to A. This means that code fragment which is located above code clone $A \in C_{t-1}$ and code clone $A \in C_{t-1}$ were modified between two versions. In this case, code fragment B in V_t corresponding to A can be traced by counting inserted lines. Thus, the start line numbers of B is increased by two lines and end line numbers of B is increased by four lines. When C_t contains B, we define B as a child clone of A.

B. Categorization of Code Clones

All code clones in V_t and V_{t-1} are categorized based on evolution patterns of them. To explain categorization of code clones, we defined propositional function about code clone $X_c \in C_t$ and $X_p \in C_{t-1}$.

- $P(X_c)$: Parent clone of X_c exists in V_{t-1}.
- $C(X_p)$: Child clone of X_p exists in V_t.
- $M(X_c)$: X_c was modified between two versions.
- $R(X_c)$: A pair of X_c and its parent clone is a clone pair.



(a) Case 1 (b) Case 2 (c) Case 3

Fig. 1. Tracing code clones: parent-child relationship of code clones

The followings are categorization of code clones that we defined. We explain this categorization with aforementioned propositional function.

- *Stable clone*: When code clone X_c satisfies $P(X_c) \wedge \neg M(X_c)$, we defined each of X_c and the parent clone of X_c as a stable clone respectively. A pair of stable clones between the versions V_{t-1} and V_t means that the code clone in V_{t-1} is unmodified until V_t.
- *Modified clone*: When code clone X_c satisfies $P(X_c) \wedge M(X_c) \wedge CP(X_c)$, we defined each of X_c and the parent clone of X_c as a modified clone respectively. A pair of modified clones between the versions V_{t-1} and V_t means that the code clone in V_{t-1} is modified but contained in the same clone set until V_t.
- *Moved clone*: When code clone X_c satisfies $P(X_c) \wedge M(X_c) \wedge \neg CP(X_c)$, we defined each of X_c and the parent clone of X_c as a moved clone. A pair of moved clones between the versions V_{t-1} and V_t means that the code clone in V_{t-1} is modified and then moved different clone set until V_t.
- *Added clone*: When code clone X_c satisfies $\neg P(X_c)$, we defined X_c as an added clone. Added clone in V_t means that the code clone is newly added in V_t .
- *Deleted clone*: When code clone X_p satisfies $\neg C(X_p)$, we defined X_p as a deleted clone. Deleted clone in V_{t-1} means that the code clone is deleted in V_{t-1}.

C. Categorization of Clone Sets

All clone sets in V_t and V_{t-1} are categorized based on evolution patterns of them. We defined the following categorization of clone sets according to the categorization of code clones that they contain.

- *Stable clone set:* Stable clone set contains code clones that did not changed between two versions such as clone set A in Figure 2. These clone sets contain stable clones that involved in V_t and V_{t-1} respectively.
- *Changed clone set:* Changed clone set contain one of the modified clone, moved clone, added clone, and deleted clone such as clone set B in Figure 2. These clone sets contain modified, moved, deleted and added clones that involved in V_t and V_{t-1} respectively.
- *New clone set:* New clone sets contain added clones such as clone set C in Figure 2. These clone sets involved in only V_t.
- *Deleted clone set:* Deleted clone sets contain deleted clones such as clone set D in Figure 2. These clone sets involved in only V_{t-1}.

IV. Clone Change Notification System

According the feedbacks from the CCFinder users in NEC, we found that developers are interested in code clones that are changed, but it is difficult for them to check changed code clones from all of detected code clones. Therefore, we developed a clone change notification system, Clone Notifier that performs checkup of changed code clones in source code.

A. Overview

Figure 3 shows the process of Clone Notifier. Clone Notifier takes two versions of source code as an input. It assumes that the developers use version control system such as Subversion in software development.

To detects all of code clones from each of the two versions, CCFinder is used as an clone detection tool in Clone Notifier. A main reason why we use CCFinder is high applicability and understandability of detection. Although syntax-based and semantic-based detection tools are accurate, those tools are not applicable in the case of unexpected source code causes errors of syntax or semantic analyses. In order to catch up minor language upgrade, and deal with incomplete source code, token-based tools such as CCFinder is more promising than syntax-based and semantic-based ones. The understandability of detection result is important for industrial application because developers would like to know the reason why each pair of code is recognized as clones to consider the reason of duplication and ways to merge those clones.

The process of this system is comprises of following four steps:

Step1 : Get the current version of source code from version control system as the latest version V_t. [1]

Step2 : Categorize code clones and clone sets between V_t and V_{t-1} which is described in section III.

[1]We use source code that were analyzed in the last time as the previous version V_{t-1}.

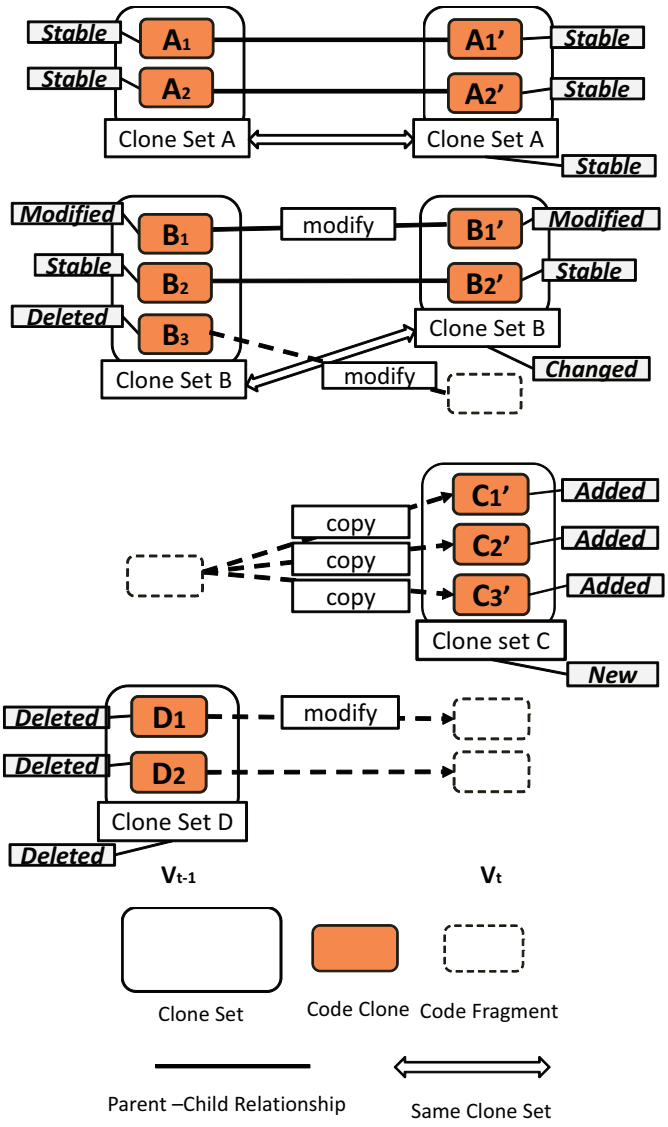


Fig. 2. Example of categorization of code clones and clone sets

Step3 : Generate html files for web-based user interface (UI) and a text file for e-mail notification [2].

Step4 : Send an e-mail with generated text file to developers on changed clones.

As described above, Clone Notifier provides information of changed code clones and clone sets between the two versions s by an e-mail.Also, Clone Notifier provides web-based code clone viewer for developers who see an e-mail.

B. E-mail Notification

This e-mail notification is aimed to send an initial report of new code clones and changed code clones. Figure 4 shows the parts of a text file for e-mail notification of two versions

[2]Note that the analysis of the system is stored for two months as html files.

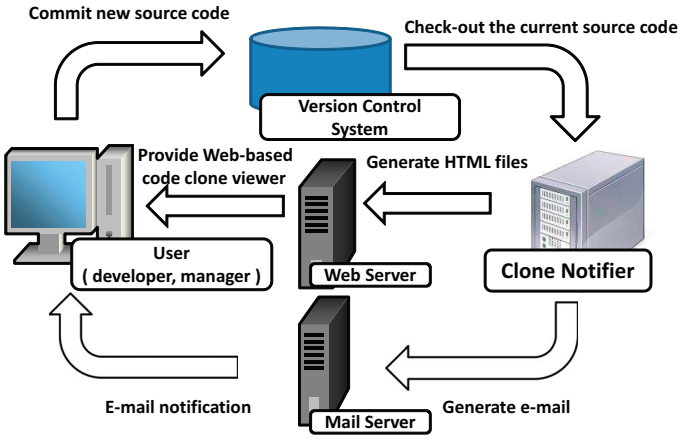


Fig. 3. Process of Clone Notifier

of Apach Ant[3]. The following information is provided by the e-mail notification.

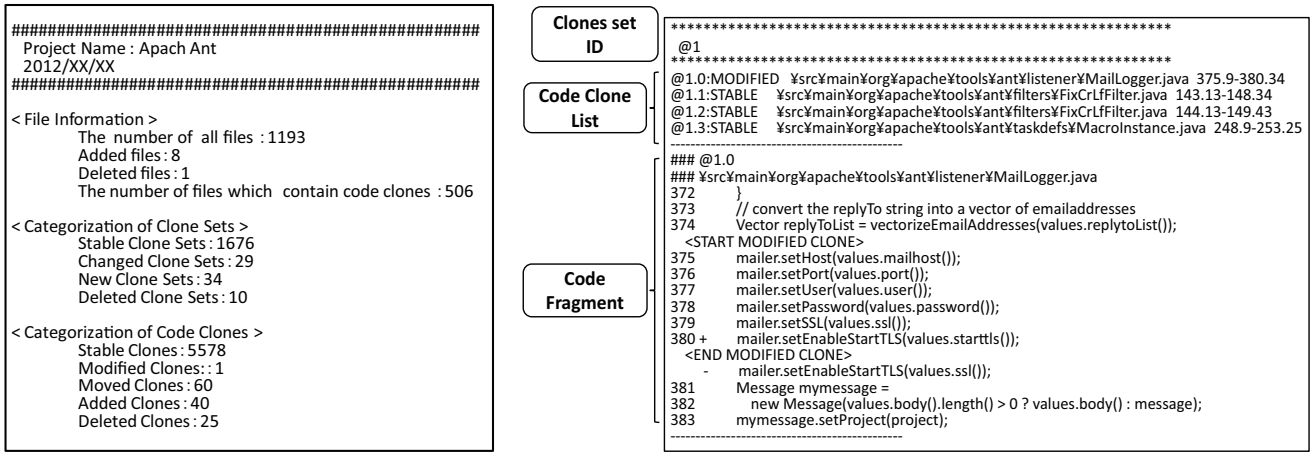
- Project information (the Figure 4(a)).
  - File information: the number of all files, added files, deleted files, and files that contain code clones
  - Categorization of clone sets: the number of stable, changed, new and deleted clone sets in V_t and V_{t-1}
  - Categorization of code clones: the number of stable, modified and added clones in V_t and deleted clones inV_{t-1}
- Clone set list: the list of changed clone sets which categorized into changed, new and deleted clone sets. The following information on each clone set is provided as shown in Figure 4(b).
  - Clone set id: the index assigned for each clone set in V_t and V_{t-1}
  - Code clone list: the list of code clones involved in each clone set between two versions
  - Code fragment: each code clone with the line number on the source file in Vt (g+h represents added line and g-h represents deleted line.)

C. Web-Based UI

This viewer supports developers who see a notification e-mail and would like to understand the detail of new and changed clone sets. Once a developer select one of clone sets, this Web-based UI shows source code and also highlights code clones in the source code. Figure 5 shows clone set list page and source file page in the web browser. This user interface consists of the following pages

- Clone set list page : It displays the list of clone sets (Figure 5(a)).Users can move to the corresponding source file page by clicking the links of each code clone.
- Source file page : It displays code clones that are involved in the selected clone set in clone set list page (Figure 5(b)). Each code clone is highlighted on this page.

[3]http://ant.apache.org/



(a) Project information

(b) Clone set list

Fig. 4. Example of e-mail notification

V. Industrial Application

To confirm that Clone Notifier is able to inform changed code clone information, Clone Notifier was applied to process of an web-application software development in NEC. Developers in NEC think that checking code clones that are comprised of small fragments is useless because they are rarely meaningful code clones according their experiences. Therefore, code clones consists of less than 30 tokens were excluded in this application.

The target system consists of approximately 350 files and 12 KLOC written in Java. The duration of the application was about 40 days, from December 2011 to January 2012.

A. Application Result

During 40 days application, Clone Notifier notified overall 152 changed clone sets, 20 deleted clone sets and 119 new clone sets. Also, every time, it notifies approximately 870 stable clone sets.

In particular, between 119 clone set of newly-appeared clone set, the project manager recognized 10 clone set as clone sets should be refactored. Each of two of the ten clone sets was merged into a single function during the 40 days. The other eight clone sets were designated as refactoring candidates that will be merged during next maintenance project.

B. Feedback from the Project Manager

We regularly interview with the project manager and got several feedbacks on application of Clone Notifier. According to feedback from the project manager, without Clone Notifier, he could not notice all of the changed code clones. Especially, he satisfied with the result that ten clone sets were recognized as candidates should be merged, and two of ten clone sets were merged during 40 days.

Also, he request us to implement the feature to present numerical criteria of selecting code clones, such as length of a code clones, the number of code clones in a clone set for next time. He suggested that numerical criteria help a user to understand the benefit of merging each of clone set.

VI. Manual Observation of Newly-Appeared Clone Sets

As an ex-post analysis, we investigated the characteristics of clone sets recognized as refactoring candidate by the experienced project manager at NEC. The aim of the analysis is data collection for the development of technique to recommend refactoring candidate from all newly-appeared and changed clones. The recommendation is promising to help developers to reduce the cost of finding clone sets should be merged into a single module.

We manually checked the differences between ten clone sets should be merged and the other 109 clone sets. As a result, we learned interesting insights about ten clone sets should be merged.

A. Code Clones Introduced without Code Addition

As the first insight, in the case of clone sets that were newly-appeared by adding new code, the project manager frequently recognized them as ones should be merged. On the other hand, clone sets were sometimes accidentally created by only the replacement or the deletion of statements. In other words, even if no line is added to a code fragment, it sometimes became a code clone in a clone set together with other code fragments when one (or greater than one) character is changed in it. In such case, the project manager mostly decided to leave those duplicates as it is.

From our observation of the 119 clone sets, we found that only coding idioms (e.g., programming or API/library specific idioms) are involved in clone sets that were newly-appeared by only the replacement or the deletion of statements. Basically, such idioms are difficult to merge or have an overall positive effect on maintenance and development [7] therefore they should be eliminated from refactoring candidates.

According to this observation, we eliminated clone sets that were newly-appeared by only the replacement or the deletion

Clone Set ID:66			
ID	Category	File Name	Location
79	DELETED	¥src¥main¥org¥apache¥tools¥ant¥filters¥ExpandProperties.java	73-87
117	STABLE	¥src¥main¥org¥apache¥tools¥ant¥filters¥PrefixLines.java	86-100
138	STABLE	¥src¥main¥org¥apache¥tools¥ant¥filters¥SuffixLines.java	87-101

Clone Set ID:44			
ID	Category	File Name	Location
232	MODIFIED	¥src¥main¥org¥apache¥tools¥ant¥listener¥MailLogger.java	375-380
82	STABLE	¥src¥main¥org¥apache¥tools¥ant¥filters¥FixCrLfFilter.java	143-148
87	STABLE	¥src¥main¥org¥apache¥tools¥ant¥filters¥FixCrLfFilter.java	144-149
823	STABLE	¥src¥main¥org¥apache¥tools¥ant¥taskdefs¥MacroInstance.java	248-253

(a) Clone set list page

```
92        */
    [START ID:78(Deleted Clone) CLONESET:39(Deleted Clone Set)]
93      public int read() throws IOException {
94  +       if (index > EOF) {
95  +           if (buffer == null) {
96  +               String data = readFully();
    -
    [START ID:79(Deleted Clone) CLONESET:66(Deleted Clone Set)]
    -       int ch = -1;
    -
    -       if (queuedData != null && queuedData.length() == 0) {
    -           queuedData = null;
    -       }
    -
    -       if (queuedData != null) {
    -           ch = queuedData.charAt(0);
    -           queuedData = queuedData.substring(1);
    -           if (queuedData.length() == 0) {
    -               queuedData = null;
    -           }
    -       } else {
    [END ID:78]
    -           queuedData = readFully();
    -           if (queuedData == null || queuedData.length() == 0) {
    [END ID:79]
    -               ch = -1;
    -           } else {
```

(b) Source file page

Fig. 5. Example of web-based UI

of statements from the 119 clone sets. The result shows that the elimination not only left the all of clone sets should be refactored but also reduced 119 newly-appeared clone sets by approximately 87%.

B. Syntactically Incomplete Clone Sets

As the second insight, code sets include whole parts of loop or branch statements were considered as ones should be merged. Meanwhile, the project manager rarely recognized clone sets include only parts of loop or branch statements as ones should be merged because it is difficult to merge syntactically incomplete clone sets.

According to this observation, we eliminated syntactically incomplete clone sets from the 119 clone sets. The result shows that the elimination not only left the all of clone sets should be refactored but also reduced 119 newly-appeared clone sets by approximately 90%.

VII. Threats to Validity

Even though Clone Notifier successively applied to process of development in NEC and we found some characterization of code clones that should be merged in this study, our study have several limitations. At first, detected code clones in this study rely on detection results from CCFinder. Secondly, we have applied Clone Notifier into only the development process of only one project

To overcome these limitations, we are planning to apply Clone Notifier to other software systems during long period. Moreover, we consider using outputs of other code clone detection tools.

VIII. Related Work

A lot of studies have been done for investigating and supporting clone evolution [11].

Kim et al. studied genealogies of code clone [8]. They defined a model of clone genealogy in order to study evolution of code clones across multiple versions of source code. We presented a clone change notification system with the feature of the categorization of clone evolution, which is based on the opinions of industrial developers in NEC, and then reported an industrial application of the clone change notification system. Also, other models of clone evolution have been proposed, and discussed [12], [13], [14], [15].

Duala-Ekoko et al. presented *Clone Region Descriptors* to track code clones moved to other locations in source code [16], [17]. The tracking code clones in proposed system is based on text difference and similarity. For more accurate tracking code clones, we should integrate *CloneTracker* into proposed system.

Nguyen et al. have developed a clone management tool *JSync* to notify developers change and its inconsistency of code clones in source files [18]. Also, Jiang et al. [19] and Li et al. [20] have been proposed on the inconsistency detection of code clones from a single version of source code. Proposed system is promising to support inconsistency detection of code clones but unable to show inconsistency between code clones explicitly. We should add the feature of inconsistency detection by them to proposed system.

Our earlier workshop paper [21] introduced Clone Notifier and early-stage of the industrial application. In this paper, we present not only completed industrial application but also manual observation of newly-appeared clone sets as an ex-post analysis.

IX. Summary and Future work

In this paper, we present clone change notification system, Clone Notifier that notifies newly-appeared and changed clones regularly to developers. We applied Clone Notifier into the process of the web application software development at NEC. The application result shows 119 newly-appeared clone sets, and ten out of them are recognized as refactoring candidates.

In ex-post analysis, we also investigated the characteristics of clone sets recognized as refactoring candidate by the experienced project manager at NEC for data collection for the development of technique to recommend refactoring candidate from all newly-appeared and changed clones.

As future work, we plan to integrate the filtering techniques with the clone change notification system, and then conduct longer-term case study at NEC. The further case studies of other industrial projects are needed for the generalization of the findings.

Acknowledgment

We express our great thanks to Ms. Fusako Mitsuhashi and Mr. Shin'ichi Iwasaki at NEC Corporation for data collection. This work is partially supported by JSPS, Grant-in-Aid for Scientific Research (A) (21240002).

References

[1] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, 2007, pp. 96–105.

[2] Z. Jiang and A. Hassan, "A framework for studying clones in large software systems," in *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, 2007, pp. 203–212.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 654–670, 2002.

[4] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proceedings of the 5th International Workshop on Software Clones (IWSC 2011)*, 2011, pp. 7–13.

[5] M. Fowler, *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[6] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 435–461, 2008.

[7] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[8] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 2005)*, 2005, pp. 187–196.

[9] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transations on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2007.

[10] S. Kawaguchi, M. Matsushita, and K. Inoue, "Clone history analysis using configuration management system," *IEICE Transactions (in Japanese)*, vol. J89-D, no. 10, pp. 2279–2287, 2006.

[11] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone Evolution: A Systematic Review," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 25, no. 3, pp. 261–283, 2013.

[12] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, 2007, pp. 81 –90.

[13] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, 2007, pp. 24–33.

[14] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007, pp. 170 –178.

[15] J. Harder and N. Gode, "Modeling clone evolution," in *Proceedings of the 3rd International Workshop on Software Clones (IWSC 2009)*, 2009, pp. 17–21.

[16] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, 2007, pp. 158–167.

[17] ——, "Clone Region Descriptors: Representing and Tracking Duplication in Source Code," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 1, pp. 3:1–3:31, 2010.

[18] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone management for evolving software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, 2012.

[19] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, 2007, pp. 55–64.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[21] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Industrial application of clone change management system," in *Proceedings of the*

6th International Workshop on Software Clones (IWSC 2012), 2012, pp. 67–71.