Software Clone Management-An Insight

Sandeep Bal[#1], Sumesh Sood[*2]
[#]*Research Scholar, PTU, Kapurthala, Punjab ,India*
[1]sandeep.bal84@gmail.com

[*]*Department of Computer Science,*
Swami Satyanand College of Management & Technology, Amritsar, Punjab, India
[2]Sumesh64@gmail.com

Abstract- **In the current programming scenario, most Software systems contain sections of copied code which ultimately results in software code cloning. Such cloning trends can be seen in different versions of the same software or different software which provide similar kind of services to its users. This situation arose due to the copy-and-paste programming practice. Software maintenance is an ongoing process which works side by side as the system is improved/ updated on timely basis. The concept of software clone management is also a part of such maintenance work. In this paper, an insight to the clone management has been shown where various aspects of clone management are discussed in the form of approaches, metrics, tools and techniques.**

Keywords- **Clone management, Prioritization, Clone change notification system, Maintenance, Software quality.**

I. Introduction

The clones in a system can be managed by assisting the programmer for assigning priorities to sets of clones listed in the code clone reports. The *Software Clone Quality* – quality of cloned code fragments with regard to its adherence to software coding standards can also be determined. The priority of the clones can be decided by the impact of the clones in terms of factors like maintenance, quality aspects, and refactoring. The clones can be fixed on the basis of the metrics and values associated with these factors. A prioritization system [1] along with its prioritization factors and the metrics for prioritization has been mentioned.

The clone management (refactoring, simultaneous editing) has also been supported by a clone change notification system [2] called as Clone Notifier. It helps the developers in notifying newly-appeared and changed clones on a regular basis. 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 whole concept starts up with the categorization of code clones and clone sets based on the evolution patterns between two versions of source code. Prior to that, a clone detection tool helps in detecting the clones among the versions of software.

Another maintenance support environment, Gemini [3] has been developed which provides the user with the useful functions to analyze the code clones and modify them [4]. The tool CCFinder [5] is one of the components of Gemini and used to detect code clones. Gemini primarily provides two diagrams: scatter plot and metrics graph. The scatter plot graphically shows the locations of code clones among source codes. The metrics graph shows metric value of each clone and has a feature to identify the distinctive code clones. Gemini received several practical problems after being evaluated by different software companies through case studies. The identified problems consisted of applying Gemini to refactoring activities [6] and identification of the modified code portions as clone. Refactoring was not easy to be done due to the inappropriateness of the clones to be merged in a single module (procedure, function, macro etc). The reason behind is the detection of the maximal code clones that often include excessive tokens that should be omitted in merging the clones into one routine. The issue of un-identification of modified code portions is found to be the minute changes done in the copy-and-paste programming. Usually, some statements are inserted to the code portion or deleted from it. Gemini cannot find such modified code clones (called gapped clone). The issue got resolved by extending the functionality of Gemini by adding the new function to extract the part of code clone which is easy to merge one module. For the latter issue, a method is proposed to show all the candidates of gapped code clones. The implementation of Gemini to several software helped in evaluating the applicability of the proposed method.

Taking ahead the issue of clone management further lead to study the relation between code clones and reliability and maintainability of a software (software quality) [7]. The modules having code clones (clone-included modules) are more reliable than modules having no code clone (non-clone modules) on average. Nevertheless, the modules having very large code clones (more than

200 SLOC) are less reliable than non-clone modules. Clone-included modules are less maintainable (having greater revision number on average) than non-clone modules; and, modules having larger code clone are less maintainable than modules having smaller code clone. The removal of a clone is suggested if it leads to bad effect on maintenance of the software system.

II. Clone Prioritization System- Factors and Metrics

The clone detection tools are run on a code base which is obtained from the version control system. The results produced are the large reports consisting information about the clones in the selected version of the code base. The prioritization system works on such a selected code base by using results provided by the various analysis tools like – Static analysis tools, Refactoring workbench, Program comprehension tools, Metric tools – results of which are mapped to a selected quality model. The EMISQ quality model [8] (which is based on ISO 9126 standard [9]) has been used for implementing this system. In a way, it helps determine the *Software Clone Quality* – the extent to which maintainability is affected and the impact on the measured code quality in terms of the number of violations and its severity. It also takes into consideration the effects of refactoring the clone. Figure 1 provides a pictorial overview of the discussed method.

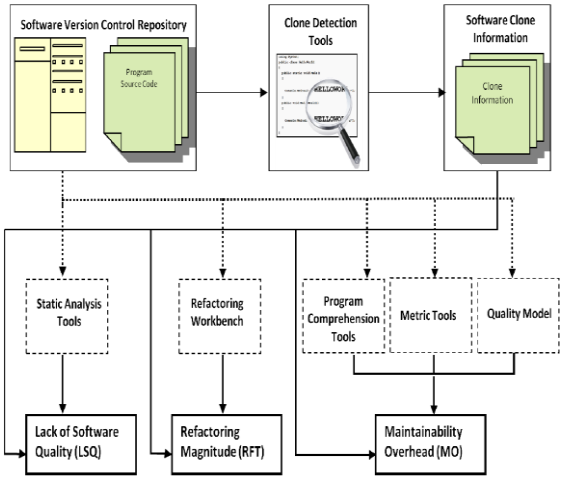

Figure 1: Overview of System for Clone Prioritization

The priority of the clones and clone set is decided by considering different criteria. On the basis of several measures, different factors help in determining the order of the clone results. They help in determining the weight or the priority value for the clone class and for each individual clone in the clone class in a cumulative manner.

A. Factor 1: Lack of Software Quality of the Clone (LSQ)

The quality of the clone code can be determined on the basis of the number of static analysis violations of different severity categories that can be found in the clone class and a particular clone. It is indicative of bug-proneness of the clone. The possible measures and metrics are:

1) Severity of a Rule (Sev_i): This represents severity (also called Message Level by some tools [10]) of the violated static analysis *rules*, identified within the clone fragments.
2) Weight Attached to the Rule (W_i): This refers to the weight associated with the violated static analysis rules computed based on the project specific considerations.
3) Criticality of the Rule ($Crit_i$): This is a function of its *Severity* and its *Weight*. It combines both the factors to decide on the importance of the rule.
4) Count of Violations of a Rule within a Clone Fragment ($CoV^{\,i}_{(n,j)}$): This keeps track of the number of occurrences of the violation of a rule within a clone. Selection of these violations is subject to two metrics as listed below.
   a) Violations with relevance to Quality Attributes.
   b) Likeliness of a violation to be false positive (FP).

The *LSQ* for a single clone fragment $C_{(n,j)}$ as well as for a clone class CC_n can be defined as:

$$LSQ\left(C_{(n,j)}\right) = \frac{\sum_{i=1}^{I} Crit_i \,.\, CoV^{i}_{(n.j)}}{\sum_{i=1}^{I} CoV^{i}_{(n,j)}}$$

B. Factor 2: Refactoring Magnitude (RFT)

It represents the impact of the proposed refactoring on the clone along with the magnitude of the side-effects of the clone. It answers questions on the need for the proposed refactoring. The measures and metrics that contribute towards this factor can be varied and mostly specific to each project. The refactoring requirement should be assigned priority accordingly (some may prefer ease of refactoring over risk of change).

The Extract Method Refactoring (EMR), Pull-Up Method Refactoring (PMR), Parameterization Refactoring (PR), and Template Methods Refactoring (TMR) are the most commonly advocated refactoring techniques to remove cloned code [11, 12]. Object-Oriented Refactoring or Aspect Oriented Refactoring approaches can also be applied [13]. This refactoring activity is further categorized based on its location, since the applicability of any of these does not readily reflect the effort required or the magnitude of impact:

1) Shared library creation (SLC): For clones across classes in different modules.

2) Interface or base class creation (BCC): For clones across classes within the same module.
3) Virtual method creation (VMC): For clones in functions of classes within the same module.
4) Helper method creation (HMC): For clones in functions within same namespace.
5) Common method within a class (CMC): For clones within the same function/class.

Refactoring Magnitude for a *clone class* can be defined as:

$$RFT(CC_n) = \frac{\sum_{k=1}^{K} RFT(RG_{CC_n}^{k})}{J}, J \text{ is the } Cardinality(CC_n)$$

C. Factor 3: Maintenance Overhead for Cloned Code (MO)

The maintenance overhead factor attempts to quantify the overhead in terms of cost and effort that may be required if the clone is not fixed, and the likeliness of the clone becoming inconsistent. These metrics are related to various maintenance activities centered on the artifacts that need to be maintained. Each of the metric MM^{a} can be described as follows:

1) Size of Clone (MM^{CLNS}): This indicates number of cloned lines in a clone fragment. Larger size indicates higher maintenance.
2) Likeliness of Clone Use (MM^{LCU}): This is a multiplication of the following two metrics. The intuition behind this is, more the clone fragment is used, more maintenance it may require.
   a) Number of times a function containing the clone code is being called based on static call graph of the program: Due to difficulty and cost involved in enumerating all the call strings, this measure can be restricted to a sub set of call strings that are covered in the system test cases.
   b) Frequency of the clone code being used in the function based on the control flow graph of the function: This measure can be obtained by enumerating the static control flow graph of the function.
3) File Change Frequency (MM^{FCF}): The number of times a file containing the clone has been modified in the past.
4) Days since Last File Change (MM^{DLFC}): This is calculated by subtracting the *Number of Days since Last File Modification* from the *Maximum Number of Days since any Modification to the File*. The heuristic followed is that - more recently a file has been changed, more likely it is to be changed again; also a file that has been modified more number of times is likely to undergo more frequent changes. Clones contained in files that change more frequently are more likely to become inconsistent.
5) Age of the Clone in Days (MM^{CA}): It measures how long the clone has lived. It has been suggested that the more long lived the clone is, the more stable it is and hence has lower maintenance overhead.
6) Maintainability Index of the Function Evaluated using Quality Model (MM^{MQM}): This takes into consideration multiple metrics and measures that may affect the maintainability of a function, e.g. eLoC for function, Cyclomatic Complexity, Efferent and Afferent Coupling.

The Maintenance Overhead (MO) of the clone class can be calculated according to the following:

$$MO(CC_n) = \sum_{j=1}^{J} MO(C_{(n,j)})$$

The above discussed factors are defined in a manner suitable for customization based on the project requirements.

III. Clone Change Notification System

Clone Notification System has been applied into the process of the web application software development at NEC Corporation, a Japanese multinational IT company. This Clone Notifier used the clone detection tool *CCFinder* [5]. 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 [14].

Clone Notifier is developed to perform check up of changed code clones in source code because the developers are more interested in code clones that are changed. In order to assist them in the difficult task of checking changed code clones from all of detected code clones, Clone Notifier has been designed.

A. Overview

Figure 2 shows the process of Clone Notifier which takes two versions of source code as an input. It assumes that the developers use version control

system such as Subversion in software development.

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

Step 1: Get the current version of source code from version control system as the latest version V_t.
Step 2: Categorize code clones and clone sets between V_t and V_{t-1} on a pre defined concept of clone categorization.
Step 3: Generate html files for web-based user interface (UI) and a text file for e-mail notification.
Step 4: 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 by an e-mail. Also, Clone Notifier provides web-based code clone viewer for developers who see an e-mail.

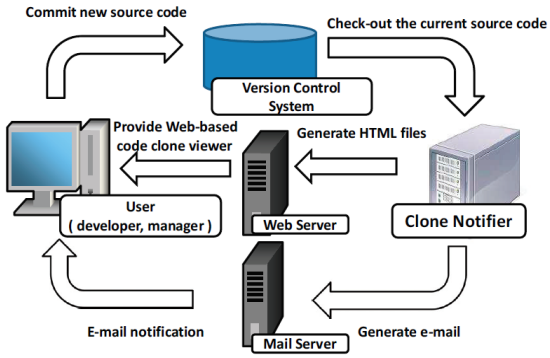

Figure. 2. Process of Clone Notifier

B. E-mail Notification

This e-mail notification is aimed to send an initial report of new code clones and changed code clones. The following information is provided by the e-mail notification:

1. Project information.

- 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 in V_{t-1}.

2. 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.

- 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 V_t .

C. Web-Based UI

This viewer supports developers who see a notification email 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. This user interface consists of the following pages:

- Clone set list page: It displays the list of clone sets. 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. Each code clone is highlighted on this page.

IV. Gemini: A tool for Maintenance support

This maintenance support environment based on code clone analysis, also called as Gemini has been developed in [15]. The system architecture can be seen in Figure 3(a). The gray parts (a gray quadrilateral and ellipse) have been proposed in [22] and the black parts (enlarged in Figure 3(b)) have been proposed in [4]. Basically, Gemini delivers the source files to the code clone detector, CCFinder[5], and then shows the information of the detected code clones to the user through various GUIs.

An interactive code clone analysis can be seen with the help of following view windows which are possible due to Gemini since it is a GUI-based code clone analysis environment:

- Scatter plot view,
- Metric graph view, and
- Source code view.

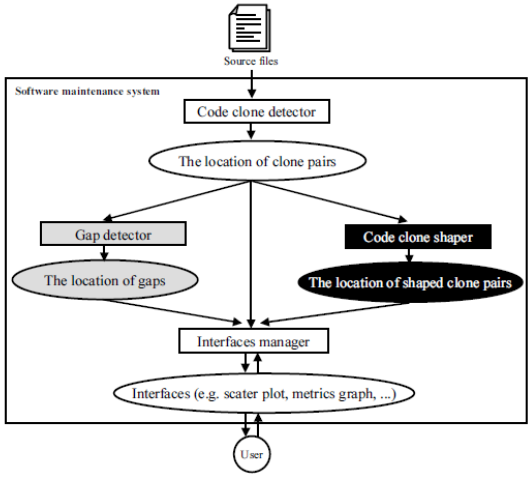

Figure 3(a) Architecture

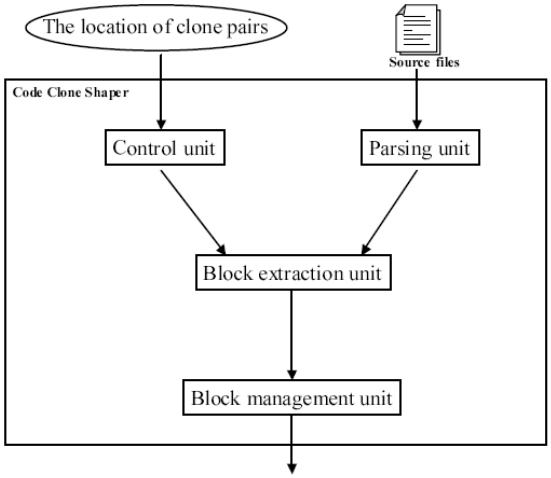

Figure 3(b) Code Clone Shaper

The existence of existing clone pairs in source files can be seen visually by using Scatter plot. It provides the state of distribution of code clone at a glance in early phase of code clone analysis therefore it is very effective mechanism where the user can select clone pairs by mouse dragging.

The selection of clones by their quantitative characteristics can be performed by the Metric graph view. User can select clone pairs or classes in metric graph view, by the values of metric for each clone class to easily select the distinctive ones.

The source code view works co operating the scatter plot view on the metric graph view. The user can obtain the actual source code corresponding to clones selected in the other views.

Problems found in case studies

There occurred many problems during the case studies of Gemini (and CCFinder) implementation on several commercial software products. Following are the two serious and repeated problems:

The developers usually do not reuse the code portion as it was- but they partially modifies the code and then reuse it in the case of 'copy-and-paste' reuse. For example, additional statements would be inserted into it. Thus, some differences exist between the original code portion and the copied-and-pasted one. Here, we call the each difference "gap" and such code clone as "gapped clone". As the minimum length of a code clone must be set in CCFinder beforehand, so when the code portion is found too short, CCFinder does not identify it as a code clone. Conversely, if we set a small value to the minimum length, then a lot of code clones are detected and the information is practically useless.

In [4], a solution was proposed to this problem. In the paper, we could refer to a certain set of gapped clones by representing visually exact/renamed clones and gaps themselves on scatter plot. In fact, the complexity of detecting all gapped clones one by one is massive (square of number of exact/renamed clones). So, we took the alternative solution.

Next, as for the second problem, the clone detection process of CCFinder is very fast but only lexical analysis is performed where the detected clones are just maximal and not always semantically cohesive. Hence it is necessary for the user of CCFinder to extract semantically cohesive portions manually from the maximal. Sometimes the semantically cohesive clones has more important meaning than maximal (just longest in local) ones during the refactoring process. Coincidental Cloning might be another issue around the scenario. There might be a common logic between similar processes which has lead to code cloning.

In [16] and [17], the semantically cohesive code clones are detected using program dependence graph (PDG) for the purpose of procedure extraction and so on. However, currently, there are no examples of the application of their approaches to large scale software since the cost to create PDG is very high.

To solve this problem, a two-step approach is taken in which maximal clones are detected first and then semantically cohesive clones are extracted from the results.

The easy to be reused code clones can be detected in less time by using this approach. The details are explained in next section.

Approach

The Shaped Clone are defined as the merge-oriented code clone extracted from the clones detected by CCFinder. The extracting process consists of the following three steps:

STEP 1: CCFinder is performed and clone pairs are detected.

STEP 2: By parsing the inputted source files and investigating the positions of blocks, semantic

information (body of method, loop and so on) is given to each block.
STEP 3: Using the information of location of clone pairs and semantics of blocks, meaningful blocks in the code clone are extracted. Here, intuitively, meaningful block indicates the part of code clone that is easy to merge.

Implementation
The implementation of the shaped clone detection function (Code Clone Shaper in Figure 3(b)) has been done in Gemini. The implementation of the proposed shaped clone detection method is explained below which includes the following units:
– Control unit
– Parsing unit
– Block extraction unit
– Block management unit

Control unit invokes the Parsing unit, Block extraction unit, and Block management unit through reading the code clone information (output from CCFinder).

Parsing unit conducts lexical and syntax analysis for the inputted source files. Here, Block is defined as code portion enclosed by a pair of brackets.

Block extraction unit extracts the block from the code clones detected by CCFinder using the stored data and analysis results from CCFinder.

Block management unit puts the blocks extracted by Block extraction unit in an appropriate order. It is necessary to obtain the consistency of the data used in Gemini.

V. Software Quality Analysis

It was noticed while reviewing previous work that considerable parts (5-50%) of large software are code clones[18][19][20]. The reasons found were reusing code by copying a pre-existing program fragment [21][19][22]; and addition of new functionalities to an existing system while performing maintenance/ up gradation. It ultimately leads to poor software quality such as low readability and changeability [19]. If one revises a copy of duplicated code sections, he/she must update all the other copies, and this may raise the maintenance cost. Moreover, if he/she overlooks one of the copies, a fault will remain in the copy, and this may lower the reliability of the system. However, the influence of code clones on software quality has not been quantitatively clarified yet.

The main goals of this study are:

1. Clarify the relation between code clones and the reliability.
2. Clarify the relation between code clones and the maintainability.

It is possible to estimate the reliability of a system which can be calculated by measuring the number of faults found over a specific time period. A system with fewer faults is considered more reliable than a system with a greater number of faults. So, measuring the number of faults of an existing system will help in analyzing the relation between code clones and the reliability of that system.
On the other hand, it is not easy to measure the maintainability of a system. Since, maintainability is related to the maintenance cost (person-hours) which means that a system of poor maintainability requires more cost in doing maintenance works than that of higher maintainability. Another way to estimate the maintainability is using software (product) metrics. Many software metrics have been proposed to measure the complexity of software such as McCabe's Cyclomatic number, Halsted's metrics, and Chidamber & Kemerers' metrics, etc [23][24][25][26] but these are not useful in calculating reliability and maintainability because code clones are essentially independent from these metrics. Moreover, a module with low maintainability and a large cyclomatic number (per SLOC) does not specify if the software quality is affected by cloning.
Another solution for measuring maintainability is to use the revision number of software modules. The repeated revision (adding and changing functionalities), makes it more complicated and more difficult to be maintained.

Module based analysis
In order to clarify the relation between software quality and code clones, a module-based analysis is conducted. The clone pairs are classified into following two types (Figure 4.)

(1) In-module clone pair: A code fragment pair is known as "in-module clone pair" if both fragments in the pair exist in the same module.
(2) Inter-module clone pair: A code fragment pair is known as "inter-module clone pair" if each fragment in the pair exists in the different module.

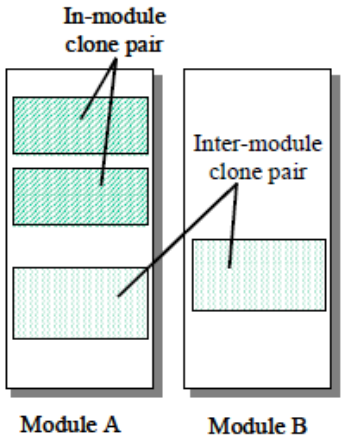

Figure 4. Types of Code Clone Pairs

These two types of clone pairs may have different influence on software quality. Inter-module clones may implicitly increase the functional coupling between modules, while in-module clones do not affect the strength of coupling between modules.

Based on above classification, the modules are classified into following four types (Figure 5.)

1. Non-clone module: A module containing no clones.
2. Clone-included module: A module containing at least one code clone pair. This type of module is classified into following three modules.
   - 2.1 Closed module: A module containing in-module clone pairs only.
   - 2.2 Related module: A module containing inter-module clone pairs only.
   - 2.3 Composite module: A module containing both in-module and inter module clone pairs.

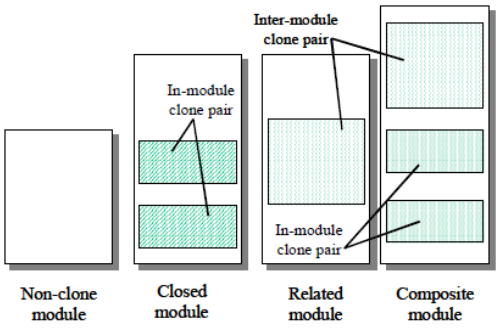

Figure 5. Module Classification

The analysis is performed on a legacy software developed about 20 years ago. It has been continuously maintained till today.

The analysis consisted of code fragments having at least 30 same lines. The measurements of the experiment are as follows:

1. LOC (Lines of code): Lines of code of each module.
2. AGE (Module age): The number of days from the date each module is initially developed to the present.
3. REV (Revision number): The number of revisions made upon each module till present. The revision includes any kind of modifications done to each module such as fixing faults and adding and changing functionalities.
4. Faults (The number of faults): The number faults found from each module in recent years (past six years in this experiment).
5. MAXLEN (Length of maximum clone): The length (LOC) of the largest code clone included in each module.
6. COVERAGE (Coverage of clone): The percentage of lines that include any portion of clone in each module.

Figure 6 shows an example of code clone metrics. MAXLEN of module B is 20 because module B contains two clones and both of them are of 20 LOC. Similarly, MAXLEN of module D is 40 because the largest clone included in module D is of 40 LOC.

COVERAGE of module B is 80% (= 40 / 80 * 100) because total clone size is 40 LOC (= 20 + 20) and the module size is 80LOC. Similarly, COVERAGE of module D is 80% (= 80 / 100 * 100) because total clone size is 80 LOC (= 40 + 20 + 20) and the module size is 100 LOC.

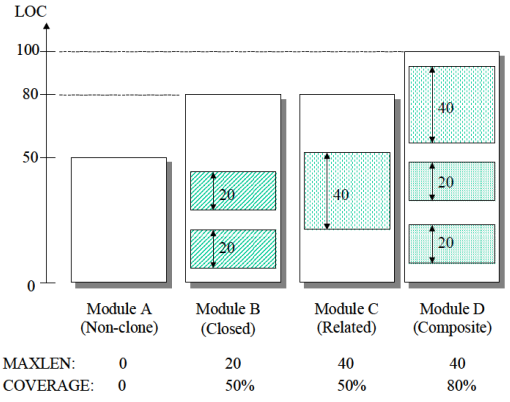

Figure 6. Code Clone Metrics

In order to evaluate the reliability of modules, number of faults per line is taken as a reliability measure. A comparison of reliability between non-clone modules and clone-included modules is shown in figure 7, which shows that clone-included modules are more reliable than non-clone modules. Clone-included modules are 1.7 times as reliable as non-clone modules on average. One possible interpretation for this result is that copying code from trusted part can lessen the fault injection compared with writing the code from scratch. Another possible interpretation is that code fragments created by copy-and past programming do not have new types of functionality, so that there may be little chance of introducing unknown types of faults in the fragments.
The reliability of each type of modules is shown in figure 8. Closed modules, related modules, and composite modules are all more reliable than non-clone modules on average.

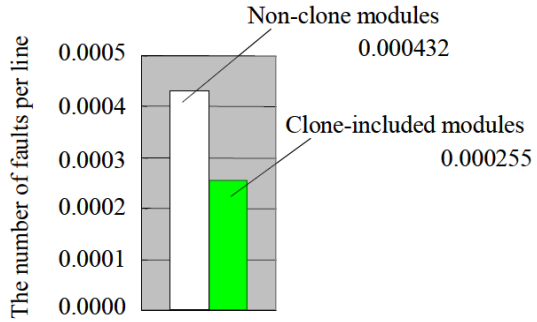

Figure 7. Relation between reliability and clones

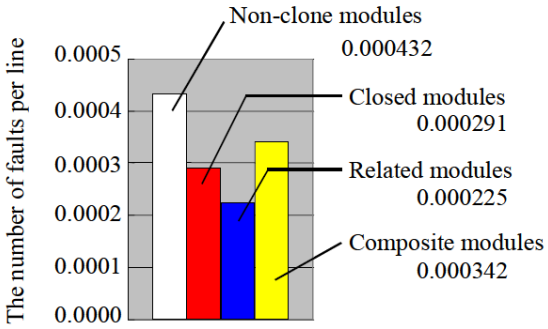

Figure 8. Reliability of modules

Revision number is chosen as a maintainability measure for evaluating the maintainability of modules. A comparison of the maintainability between non-clone modules and clone-included modules is shown in figure 9 which shows that clone included modules are less maintainable than non-clone modules. Figure 10 shows the maintainability of each type of modules. Closed modules, related modules, and composite modules are all less maintainable than non-clone modules on average.

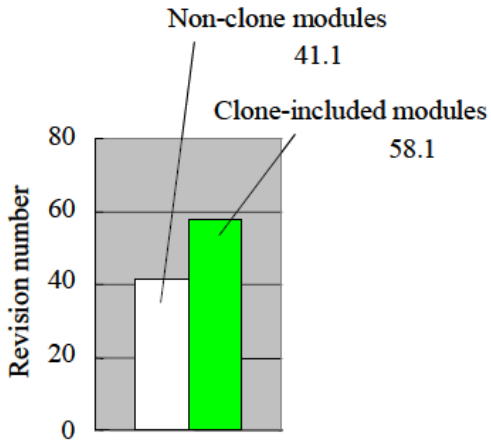

Figure 9. Relation between maintainibility and clone

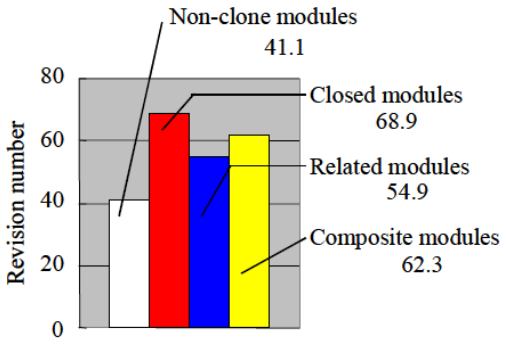

Figure 10. Maintainibility of different modules

VI. Summary

In this paper, various clone management issues were defined. Priorities were assigned to the various detected clones in software in order to perform effective clone management. The factors take into consideration the *Software Clone Quality*, the cost incurred due to the clone maintenance and the effects of refactoring the clones.

A clone change notification system, Clone Notifier was elaborated that notifies newly-appeared and changed clones regularly to developers.

The functionality of a maintenance support environment Gemini was extended to easily merge code clones into one code portion. It supported the maintenance activity more efficiently.

At last, the relation between code clones and the software reliability and maintainability of a software was explained which led to the following conclusions:

- Clone-included modules are 1.7 times as reliable as non-clone modules on average.
- Closed modules, related modules, and composite modules are all more reliable than non-clone modules on average.
- The modules having very large code clones (more than 200 lines) are less reliable than non-clone modules.
- Clone-included modules are less maintainable (having greater revision number) than non-clone modules on average.
- Closed modules, related modules, and composite modules are all less maintainable than non-clone modules on average.
- The modules having larger code clone are less maintainable than modules having smaller code clone.

References

[1] Venkatasubramanyam, R. D., Gupta, S., & Singh, H. K. (2013, May). Prioritizing code clone detection results for clone management. In *Proceedings of the 7th International Workshop on Software Clones* (pp. 30-36). IEEE Press.

[2] Yamanaka, Y., Choi, E., Yoshida, N., Inoue, K., & Sano, T. (2013, May). Applying clone change notification system into an industrial development

process. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on* (pp. 199-206). IEEE.

[3] Higo, Y., Ueda, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2002). On software maintenance process improvement based on code clone analysis. In *Product Focused Software Process Improvement* (pp. 185-197). Springer Berlin Heidelberg.

[4] Ueda, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2002). On detection of gapped code clones using gap locations. In *Software Engineering Conference, 2002. Ninth Asia-Pacific* (pp. 327-336). IEEE.

[5] Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, *28*(7), 654-670.

[6] Fowler, M. (2009). *Refactoring: improving the design of existing code*. Pearson Education India.

[7] Monden, A., Nakae, D., Kamiya, T., Sato, S. I., & Matsumoto, K. I. (2002). Software quality analysis by code clones in industrial legacy software. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on* (pp. 87-94). IEEE.

[8] Plosch, R., Gruber, H., Hentschel, A., Korner, C., Pomberger, G., Schiffer, S., ... & Storck, S. (2007, March). The EMISQ method-Expert based evaluation of internal software quality. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE* (pp. 99-108). IEEE.

[9] ISO/IEC 9126-1:2001, Software Engineering – Product Quality– Part 1: Quality Model. 2001.

[10] http://msdn.microsoft.com/enS/library/bb429379(v=vs.80).aspx

[11] Roy, C. K., & Cordy, J. R. (2007). *A survey on software clone detection research*. Technical Report 541, Queen's University at Kingston.

[12] Koni-N'Sapu, G. G. (2001). A scenario based approach for refactoring duplicated code in object oriented systems. *Master's thesis, University of Bern*.

[13] Schulze, S., Kuhlemann, M., & Rosenmüller, M. (2008, October). Towards a refactoring guideline using code clone classification. In *Proceedings of the 2nd Workshop on Refactoring Tools* (p. 6). ACM.

[14] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., & Merlo, E. (2007). Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, *33*(9), 577-591.

[15] Ueda, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2002). Gemini: Maintenance support environment based on code clone analysis. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on* (pp. 67-76). IEEE.

[16] Komondoor, R., & Horwitz, S. (2001). Using slicing to identify duplication in source code. In *Static Analysis* (pp. 40 56). Springer Berlin Heidelberg.

[17] Krinke, J. (2001). Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on* (pp. 301-309). IEEE.

[18] Baker, B. S. (1995, July). On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on* (pp. 86-95). IEEE.

[19] Ducasse, S., Rieger, M., & Demeyer, S. (1999). A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on* (pp. 109-118). IEEE.

[20] Lague, B., Proulx, D., Mayrand, J., Merlo, E. M., & Hudepohl, J. (1997, October). Assessing the benefits of incorporating function clone detection in a development process. In *Software Maintenance, 1997. Proceedings., International Conference on* (pp. 314-321). IEEE.

[21] Baxter, I. D., Yahin, A., Moura, L., Anna, M. S., & Bier, L. (1998, November). Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on* (pp. 368-377). IEEE.

[22] Monden, A., Nakae, D., Kamiya, T., Sato, S. I., & Matsumoto, K. I. (2002). Software quality analysis by code clones in industrial legacy software. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on* (pp. 87-94). IEEE.

[23] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, *20*(6), 476-493.

[24] Fenton, N. (1991). Softwre Metrics: A Rigorous Approach.

[25] Halstead, M. H. (1977). *Elements of software science* (Vol. 7, p. 127). New York: Elsevier.

[26] McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*, (4), 308-320.

