Correlation Analysis between Code Clone Metrics and Project Data on the Same Specification Projects

Yoshiki Higo*, Shinsuke Matsumoto*, Shinji Kusumoto*, Takashi Fujinami†, and Takashi Hoshino†
*Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
†Nippon Telegraph and Telephone Corporation, 2-13-34 Konan, Minato-ku, Tokyo 108-0075, Japan

***Abstract*—The presence of code clones is pointed out as a factor that makes software maintenance more difficult. On the other hand, some research studies reported that only a small part of code clones requires simultaneous changes and their negative influences on software maintenance are limited. Besides, some other studies reported that code clones often have positive effects on software development. Currently, the authors are researching exploring the effect of clones on software development and maintenance. In this paper, the authors report their exploratory results on the relationship between clone metrics and project data such as the number of test cases and the number of found bugs. The targets of this exploration are nine web-based software systems. Interestingly, all of them were developed based on the same specification. In other words, they are functionally the same software systems. By targeting such projects, we can explore how implementation differences affect software development. As a result, unit/integration/system testing become more difficult in case that many clones exist in a project.**

I. Introduction

The presence of code clones (simply, clones) has been considered as one of the factors that makes software maintenance more difficult. Simultaneous changes on clones are the main reason why clones are harmful on software maintenance. If multiple clones to be changed simultaneously are not changed simultaneously, unintentional inconsistencies happen on source code. Such unintentional inconsistencies may cause faults later.

To support software development and maintenance against clones, a variety of studies related to clone detection has been conducted [1], [2]. There are also many empirical studies that investigated harmfulness of clones from the viewpoint of simultaneous changes [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. These studies revealed that not all clones are harmful on maintenance but a part of them is surely harmful.

Our research interest is revealing how clones have influences on software development/maintenance and whether clones are technical debt or not. These kinds of research question are very difficult to answer because we need to evolve a software system in two ways: one evolution is a system where code clones are as they are; the other is a system where code clones are removed as much as possible. Then, we can compare the two systems. However, this point is very important because if developers know clone influences, they can decide whether they should conduct refactorings or not on a given system. If clones are not harmful, conducting refactorings is wasting time and human resources.

Currently, the authors are conducting a research project on clone influences on software development and maintenance. In this paper, the authors report experimental results of our correlation analysis between clone metrics and various project data on nine software systems. All the nine systems were developed by different IT vendors. A notable feature of them is that all the systems were developed based on the same specification. By analyzing such systems, we can explore influences of implementation (the amount of duplicated code) on software development and maintenance.

The reminder of this paper is organized as follows: Section II introduces a definition of clones and some related works; Section III explains the design of our experiments and Section IV shows the results; then Section V describes threats to validity on the experiment; finally, Section VI concludes this paper.

II. Code Clone

In this section, the definitions of clones are explained and previous works are introduced.

A. Definition

A pair of similar code fragments (clone pair) is classified into either of the following categories based on the degree of their similarity.

- **TYPE-1**: clone pairs whose two code fragments are exactly identical to each other. They can include only differences of white spaces, tabs, and comments.
- **TYPE-2**: clone pairs whose two code fragments are syntactically identical to each other. They can include token-level differences such as different variable names.
- **TYPE-3**: clone pairs whose two code fragments are syntactically similar to each other to a certain extent. They can include statement-level differences in addition to token-level ones.

Various clone detection techniques have been proposed [1], [2]. Each of the techniques has a unique definition of clones. For example, in token-based detection techniques similar sequences of program tokens are detected as clones [13]. In

AST[1]-based techniques, similar subtrees in ASTs generated from source code are detected as clones [14]. Consequently, different clones are detected by different detection techniques from the same source code. The clone detection technique used in this research is described in Section III-B.

B. Related Work

Inoue et al. proposed a technique to find latent faults by focusing on variable correspondences between two code fragments of clone pairs [7]. Their idea is that, after copying and pasting a code fragment, variables in the pasted code fragment usually correspond to ones in the original code fragment. If variables in the original and pasted code fragments do not correspond to each other, latent faults may exist in the pasted code fragment due to forgetting to replace some variables. They developed a tool to detect clones where variables do not correspond. The tool was applied to two mobile systems and 68 clone pairs were found. The developers of the systems manually investigated each of the clones and found that 26 out of the clone pairs included latent faults.

Monden et al. investigated relationships between clones and revision numbers of source files on a legacy COBOL system [12]. As a result, they found that source files tend to be modified more frequently in either of the following situations:

- source files where 80% or more code are duplicated, or
- source files where 200 LOC or larger clones exist.

There are some cases that coding with copying and pasting operations is a reasonable way of implementation [15]. One of such cases is adding a new function to an in-service software system. Firstly, a developer copies and pastes a code fragment of an existing function and pastes it to another place of the software. Then he/she modified the pasted code fragment to make out a similar new function. For a while, the system is operated with the situation. When, the behavior of the new function gets stable, the pasted code fragment is merged with the original code fragment. Adding new functions with this procedure can prevent faults from occurring in the running system. Kapser et al. investigated two open source systems and found that 71% of clones have good influences on software maintenance.

Yamanaka et al. developed a clone management system, which notifies developers whether clones have been changed in the latest commit [16]. The system checks differences between the latest version and the second latest version in the source code repository. If clones in the second latest version have been changed, the information related to the changes was sent to developers via email. They applied the system to a software project in a company. In the application, there were some commits where developer changed clones of which the developers did not know the presence. The developers were able to recognize the changes on clones by the notifications and the developers were able to take some actions such as refactoring for the clones.

[1]Abstract Syntax Tree

Chatterji et al. conducted an experiment to investigate whether developers were able to locate code fragments causing given bugs with clone information [17]. The research participants were 43 graduate students. In cases where the students used clone information after they had found a code fragment causing a given bug, they were able to locate other code fragments causing the same bug by using clone information. On the other hand, before locating any code fragments causing a given bug, using clone information did not help the students to locate the buggy code fragments.

Zhan et al. conducted an investigation for 21 developers of an industrial system that had been maintained for more than ten years [18]. As a result, they found a variety of reasons why the developers had generated clones. Some of the reasons were technical ones such that (1) developers wanted to avoid inducing new bugs by changing existing code and (2) merging clones as a single module was not easy. There were also organizational reasons such that the developers did not afford to merge clones due to the development schedule. They found there were also personal reasons such that some of the developers had wanted to improve their coding skills by reusing other skillful developers' code.

Göde and Koschke investigated changes in clones on three open source systems written in C or Java [4]. They found that 88% clones had not been changed at all after they had been generated. In their investigation, 15% changes in clones caused unintentional inconsistencies in source code.

Hotta et al. compared change frequencies of duplicated code with non-duplicated code [6]. The targets were 15 open source systems written in C/C++ or Java. As a result, they found that non-duplicated code had been changed more frequently than duplicated code.

Tsunoda et al. showed that using multiple kinds of clone detection results for models of fault-prone module prediction can achieve higher accuracy than a single kind of clone detection results [19]. They used four detection tools: CCFinderX, PMD's Copy/Paste Detector, Simian, and Nicad. Each tool was executed with two different settings, so that they obtained eight detection results in total. They calculated duplicated ratio of source code based on each of the detection results and added the ratio values to the prediction models. Then, they compared accuracy of 10 models: one model including all the eight values, one model including none of the values, and eight models including a single value. As a result, the model including all the eight values achieved the highest accuracy. Some of the models including a single value were worse than the model including none of the values.

Saini et al. investigated whether cloned methods tend to have different metrics values from non-cloned methods [20]. The target metrics are 27 in total. Some of them are well-known and traditional metrics such as McCabe's cyclomatic complexity [21] and Halstead's software science [22], others are simple ones such as the number of parameters, and the number of loops. The results were that the size of cloned methods was smaller than non-cloned methods by 29%. However, there was no difference in the other metrics.

III. Experimental Design

In this section, the designs for the experiment are explained.

A. Target Systems

Figure 1 shows an overview of the target system. It is a web-based inventory control system. The J2EE technology is used in the system to realize online transaction processing. The system has 11 frames, two kinds of ledger sheets, 32 kinds of processing, and two interfaces. Its function points are 113 FP[2]. JSP is used in the client-side programs and Java is used in the server-side programs, respectively.

A company made its specification and nine different vendors implemented the systems independently. Hereafter, we call the vendors V_A, V_B, $\cdots$, V_I, respectively. Every vendor made a service contract with the company separately. These nine systems are based on the same specification but all of them have different implementations. The nine systems were not for real usage but for the experiment in the company to collect various data on software developments.

Every vendor conducted testing for the system. Then, the final version of the code, which passed the testing, was sent to the company. The targets of this experiment are Java source files, which are the client-side of the system. The size of the client-side ranged between 20,000 and 44,000 lines of code. The company also conducted testing to check the quality of each client-side system. Hereafter, we call the testing conducted by vendors *vendor testing* and also we call the testing conducted by the company *acceptance testing*. Both vendor testing and acceptance testing consist of the following three kinds of testings.

UT (Unit Testing): verifying each modules.

IT (Integration Testing): verifying interfaces between modules, data interactions with databases.

ST (System Testing): verifying whether the target system operates appropriately when it takes dummy inputs like real ones.

B. Clone Detection

Currently, there are various clone detection techniques. In this research, the authors adopt a token-based technique to detect clones from the target projects. Token-based clone detection techniques have the following features.

- Detections are quick. Token-based techniques take only 10 minutes or so to detect clones from million lines of code [13], [23].
- Detection results tend to become huge. In other words, token-based techniques are good at detecting clones without overlooking while they tend to find a large amount of trivial clones [24], [25].
- Basic token-based detection algorithms are designed to detect TYPE-1 and TYPE-2 clones.

The authors are also conducting research on improvements of token-based clone detection. Our latest clone detection tool is **CloneGear**[3]. CloneGear is a multi-linguistic token-based detection tool. Currently, it can handle C/C++, Java, JavaScript, JSP, Python, and PHP code. CloneGear has the following features.

- CloneGear does not detect clones from code of repeated instructions. For example, in C/C++ or Java, trivial duplicated code such as consecutive if-else statements or consecutive switch-case entries are not detected as clones.
- Even if there are different instructions in two similar code fragments, the two similar code fragments are detected as Type-3 clones.

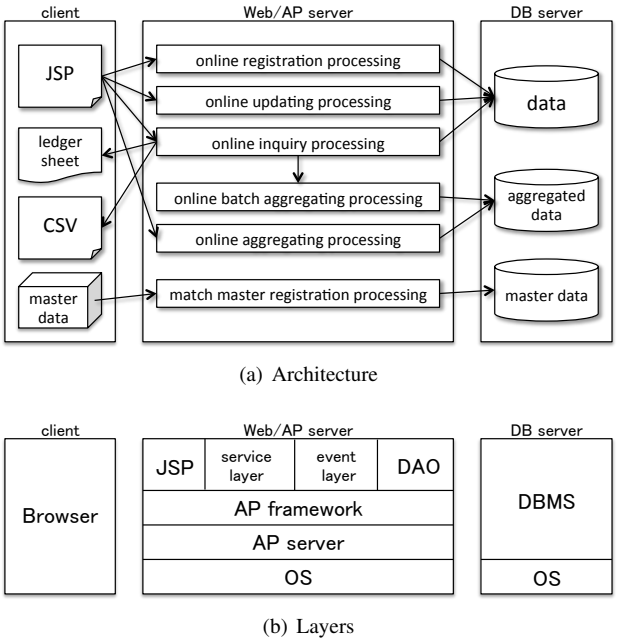

Fig. 1. An overview of the target systems

The first feature has been realized by utilizing our technique folding repeated instructions in advance of clone detection [26]. The second feature has been realized by utilizing Smith-Waterman algorithm [27], which is a well-used algorithm to detect similar base arrangements in DNA or RNA. We have applied the algorithm to clone detection [28]. In our previous research, we have already confirmed that both features are beneficial for clone detection [26], [28]. In this research, we use CloneGear to detect clones from the target projects.

In this research, the authors measure the following clone metrics, all of which are normalized ones because the target projects are different source code sizes. All the metrics are measured for every project.

DOC (Density Of Clones): the number of clones per 1,000 lines. This metric is calculated by dividing the number of clones detected from a given project by its kilo lines of code.

ROC (Ratio Of Clones): the ratio of clones against the whole source code. This metric is calculated by dividing the number of tokens included in any clones by the number of whole tokens in a given project. If

[2]Function Point

[3]https://github.com/YoshikiHigo/CloneGear

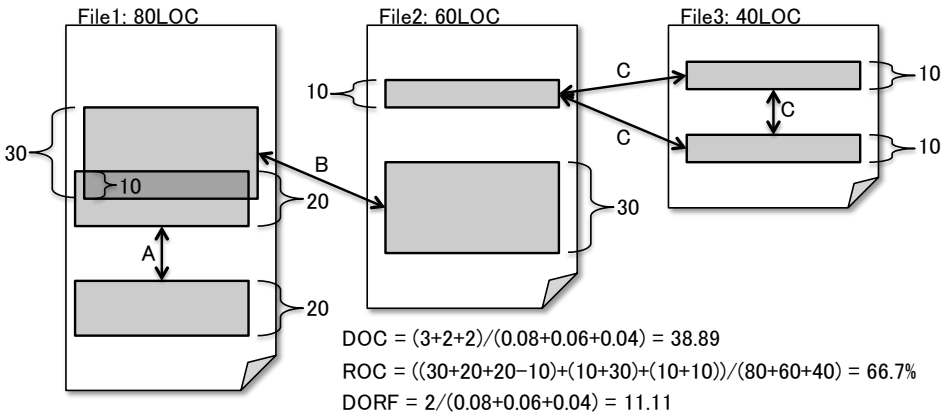

Fig. 2. Example of metrics measurement

there is no clone in a given project, ROC becomes the minimum value 0%. If the whole source code is duplicated, ROC becomes the maximum value 100%.

DORF (Density Of Related Files): the number of file pairs sharing clones per 1,000 lines. This metric is calculated by dividing the number of file pairs sharing clones in a given project by its kilo lines of code.

DOC is a normalized version of the number of clones. The number of clones is a well-used metric in clone analysis studies. ROC is also a well-used metric. This metric is sometimes called *clone coverage*. DORF is a new metric that the authors contrived in this research. This metric is the authors' intuition that clones shared with different files are more harmful than ones within a single file.

Figure 2 shows a measurement example of the three metrics. In this figure, there are three source files. Three clone groups A, B, and C have been detected. Each code fragment in A, B, or C includes 20, 30, or 10 lines of code. Herein, for simplicity, we assume every code line consists of only a program token. In this situation, the three metrics are calculated as shown in the bottom of the figure.

For each project, the authors detected clones three times. In each time, the authors used different thresholds 50, 100, or 150. The thresholds mean the minimum token number of code fragments to be detected as clones. Clone detection results are better if they include the smaller number of false positives and they miss the smaller number of undetected clones to be detected. However, the number of false positives and the number of missing clones have a trade-off relationship. If we configure clone detection tools not to detect false positives, we will miss many clones to be detected. On the other hand, if we configure tools not to miss clones to be detected, clone detection results will include many false positives. Consequently, in this research, the authors used three different values as thresholds of minimum clones to be detected.

50 tokens: this value is often used in token-based clone detections. With this value, detection tools tend not to miss clones to be detected but it includes many false positives [13], [29].

100 tokens: this value is intended to reduce false positives. However, more true clones tend to be missed.

150 tokens: this value is intended not to detect false positive as much as possible.

Table I shows values of the clone metrics for the three thresholds on each project just for reference. All the clone metrics become smaller values as a larger threshold is used.

TABLE I
MEASUREMENT RESULTS OF CLONE METRICS

Vendor	Threshold: 50			Threshold: 100			Threshold: 150		
	DOC	ROC	DORF	DOC	ROC	DORF	DOC	ROC	DORF
V_A	11.66	12.3%	3.69	2.57	7.5%	0.59	1.07	4.4%	0.27
V_B	61.17	43.1%	5.24	35.93	38.7%	2.64	22.06	33.8%	1.75
V_C	24.85	22.4%	6.64	7.85	18.3%	1.77	3.76	13.7%	0.89
V_D	20.92	26.2%	10.56	6.30	19.1%	3.46	2.10	10.1%	0.95
V_E	30.18	22.1%	10.01	10.99	18.1%	3.03	4.99	13.1%	1.78
V_F	14.60	20.6%	2.81	5.93	17.3%	1.00	3.45	14.6%	0.60
V_G	17.38	19.6%	9.58	5.76	15.0%	3.50	1.46	6.7%	1.49
V_H	33.52	34.1%	7.95	13.62	29.4%	3.79	5.45	19.5%	2.23
V_I	32.86	32.1%	3.86	15.71	29.5%	2.78	5.20	17.9%	2.42

C. Project Data

In this section, the authors introduce project data used in this research. When we started this research, the nine vendors had finished developing the target systems. We were not able to ask the vendors to collect/measure project data that we wanted. We used available project data, which the company had asked the vendors to collect/measure to check quality and progress of the developments. Due to contractual reasons with the company, the authors cannot provide actual values of the project data.

STEPS was used to consider size of the target projects.

STEPS: a value to represent size of source code. It is similar to LOC but blank lines and comment lines are not considered to measure it.

DP, DM, and PPW were used to consider the aspect of development cost for the target projects.

DP (Development Period): a period that a vendor took to develop the target project. The period that the company took to conduct acceptance testing is not included in DP.

DM (Development Man-hour): man-hour that a vendor took to develop the target project.

PPW (Progress Per Week): average code size that a vendor implemented per week.

The authors used the number of vendor test cases. However, the authors did not use the number of acceptance test cases because the same acceptance testing was conducted for all the projects. In other words, the number of test cases in the acceptance testing for all the projects is exactly the same.

#VUT, #VIT, #VST: the number of vendor unit, integration, or system test cases.

The authors also used the density of vendor testing. This data was calculated by dividing the number of test cases by the code size of that time.

DOVUT, DOVIT, DOVST: density of vendor unit, integration, or system testing.

Data related to bugs were also targets of our experiment. The authors used the number of bugs found in vendor and acceptance testing.

#BIVUT, #BIVIT, #BIVST: the number of bugs found in vendor unit, integration, or system testing.

#BIAUT, #BIAIT, #BIAST: the number of bugs found in acceptance unit, integration, or system testing.

The authors also used the bug density in vendor testing. The bug density was calculated by dividing the number of found bugs by the code size of that time. The authors did not use the bug density in acceptance testing because such data was not recorded in the projects.

BDOVUT, BDOVIT, BDOVST: bug density of vendor unit, integration, or system testing.

D. Calculating Correlation Coefficient

The authors calculated correlation coefficient between clone metrics shown in Table I and project data shown in Section III-C. In other words, the authors investigated a similarity between the two orders of the projects: one is a descending order of a given clone metric; the other is a descending order of given project data. The similarity was calculated by using Spearman's ρ. The authors obtained clone detection results for three threshold values. Consequently, the authors derived three values of the correlation coefficient for each pair of clone metrics and project data.

IV. Results

Figure 3 shows the correlation coefficient between the clone metrics and the project data. There is a line segment for every clone metric on all the project data. The line segments represent the range of correlation coefficient between given project data and a given clone metric. The authors detected clones with three thresholds, so that there are three values of the correlation coefficient for each clone metric. "$\times$" on the line segments mean they are median values. The number following the metric names means the number of correlations where p-value is 0.017 or less. For example, "DOC_3" in the figure means that each of all three variations of DOC metric was significant with p-value 0.017. The reason why we used 0.017 as p-value is that $(1-0.017)^3 = 0.95$. The formula means at least 98.3% probability is required for the significance of each correlation coefficient to ensure that all of the three correlation coefficient are significant at 95% probability.

The following pairs of the clone metrics and the project data have at least 1 significant correlation coefficient.

1) Positive correlation on PPW$\times$ROC
2) Negative correlation on #VUT$\times$DORF
3) Positive correlation on #BIVIT$\times$DORF
4) Positive correlation on #BIAUT$\times$DORF
5) Positive correlation #BIAIT$\times$DOC
6) Positive correlation #BIAIT$\times$ROC
7) Positive correlation #BIAIT$\times$DORF
8) Negative correlation on BDOVUT$\times$DORF
9) Positive correlation BDOVIT$\times$DORF

1) seemingly shows that high ROC projects developed more lines of code than low ROW ones per week. In other words, projects including more clones can be developed more efficiently. If we assume that clones were generated by copy-and-paste coding, this result is quite natural because copy-and-paste coding is much faster than writing line-by-line code. However, there is no significant correlation coefficient between DM and any of the clone metrics. Consequently, we cannot conclude that copy-and-paste coding contributes to reducing development cost.

3) and 9) show that more bugs tend to be detected in integration testing if more files share clones. However, 2) and 8) shows less bugs tend to be detected in unit testing on projects where many files share clones These results imply that bugs tend to be detected late in such projects.

4) 5) 6) and 7) mean that acceptance unit/integration testing detected more clones on the projects having higher clone metrics values. The bugs detected in acceptance testing means that they had not been detected in vendor testing. These results

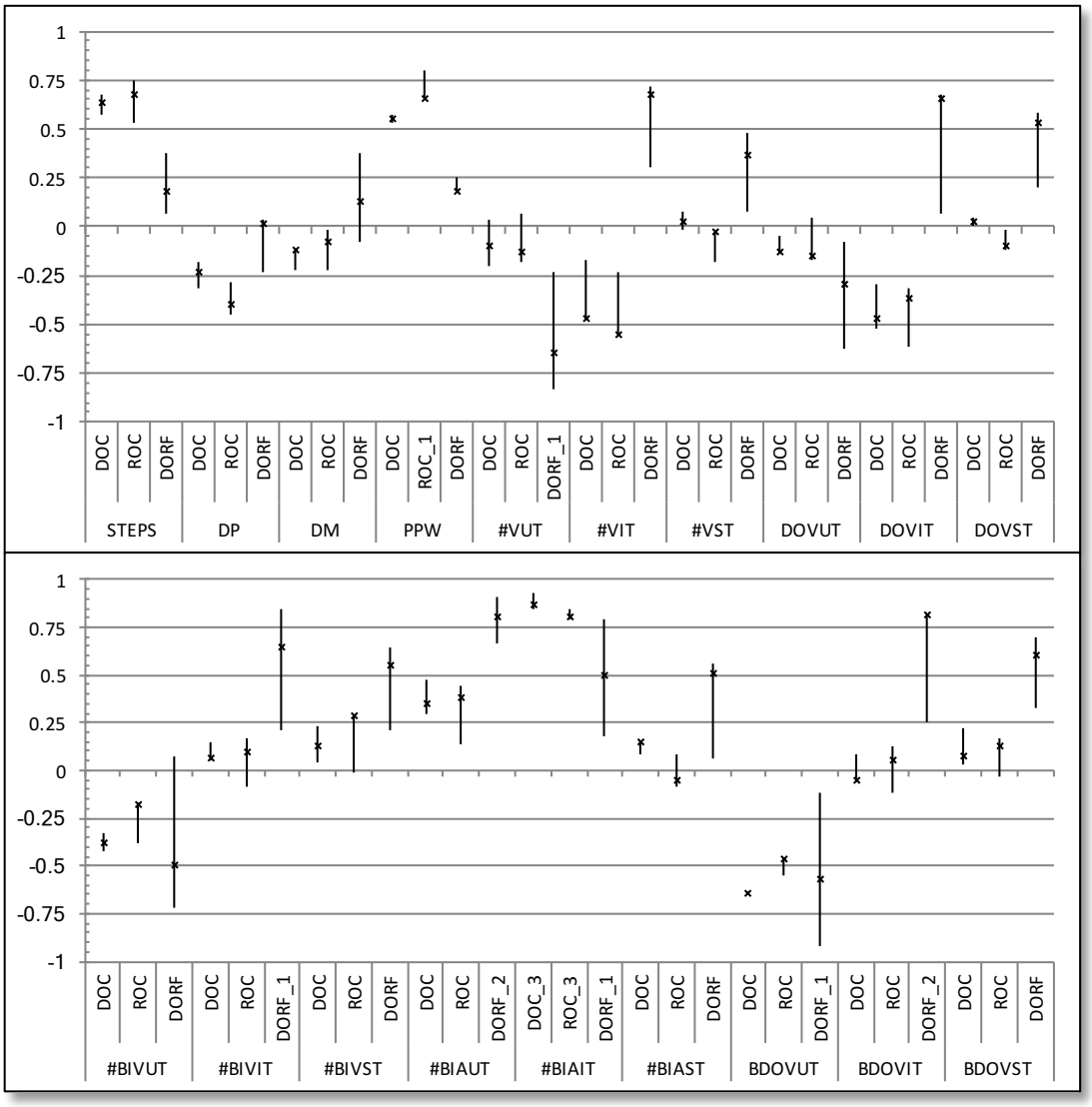

Fig. 3. Correlation coefficient between clone metrics and project data. The number following the metrics names means the number of significant correlations (p-value is 0.017 or less).

imply that the presence of clones makes it more difficult to conduct testing for vendors.

As a result of above discussion, we conclude that the presence of clones makes testing more difficult and there are few benefit of clones for software development.

V. Threats to Validity

In this experiment, the authors used 0.017 as p-value. This is because there are three correlation coefficient for each pair of the project data and the clone metrics and the authors wanted to ensure that all of three correlation coefficient are significant at 95% probability. The authors utilized 19 project data and three clone metrics, so that there were 57 correlations in this experiment. Thus, conducting multiple comparisons such as FDR[4] is another way for investigation.

[4]False Discovery Rate

In this experiment, the targets are only a single set of projects. If the authors conduct the same investigation to other sets of projects, the authors may obtain different kinds of correlations between project data and clone metrics. However, our target is very special data, which include nine isofunctional software systems. The authors think using this data match with our research context, which reveals influences of clones on software development.

We were not able to obtain detailed data for the bugs in the nine projects. We were not able to conduct detailed analyses for each of the bugs. Thus, it is unclear how many of the bugs are related to code clones.

The project data used in the experiment were captured by vendors themselves. There may be some vendors that did not precisely capture the project data. There is also a possibility that some project data were captured precisely while the others were not in the same vendors.

VI. Conclusion

In this paper, the authors reported experimental results on the relationship between code clones and project data. The target systems are nine web-based systems developed based on the same specification. As a result, the authors found the followings.

- If many files share clones a project, it becomes more difficult to detect bugs in unit testing. In such projects, more clones are detected in integration testing.
- If many clones exist in a project, vendor testing become more difficult. In such project, more bugs were detected in acceptance testing.

References

[1] D. Rattan, R. Bhatia, and M. Singh, "Software Clone Detection: A Systematic Review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[2] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[3] L. Barbour, F. Khomh, and Y. Zou, "An Empirical Study of Faults in Late Propagation Clone Genealogies," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1139–1165, 2007.

[4] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 311–320.

[5] Y. Higo and S. Kusumoto, "How Often Do Unintended Inconsistencies Happen? —Deriving Modification Patterns and Detecting Overlooked Code Fragments—," in *Proceedings of the 28th International Conference on Software Maintenance*, 2012, pp. 222–231.

[6] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is Duplicate Code More Frequently Modified Than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software," in *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010, pp. 73–82.

[7] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, "Experience of Finding Inconsistently-changed Bugs in Code Clones of Mobile Software," in *Proceedings of the 6th International Workshop on Software Clones*, 2012, pp. 94–95.

[8] J. Krinke, "Is Cloned Code Older Than Non-cloned Code?" in *Proceedings of the 5th International Workshop on Software Clones*, 2011, pp. 28–33.

[9] 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.

[10] A. Lozano and M. Wermelinger, "Assessing the Effect of Clones on Changeability," in *Proceedings of the 24th International Conference on Software Engineering*, 2008, pp. 227–236.

[11] M. Mondal, C. K. Roy, and K. A. Schneider, "Does Cloned Code Increase Maintenance Effort?" in *Proceedings of the 11th International Workshop on Software Clones*, 2017, pp. 38–44.

[12] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software," in *Proceedings of the 8th International Symposium on Software Metrics*, 2002, pp. 87–94.

[13] 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. 7, pp. 654–670, 2002.

[14] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 368–377.

[15] 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.

[16] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Applying Clone Change Notification System into an Industrial Development Process," in *Proceedings of the 21th International Conference on Program Comprehension*, 2013, pp. 199–206.

[17] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft, "Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 20–29.

[18] W. Zhao, Z. Xing, X. Peng, and G. Zhang, "Cloning Practices: Why Developers Clone and What Can Be Changed," in *Proceedings of the 28th International Conference on Software Maintenance*, 2012, pp. 285–294.

[19] M. Tsunoda, Y. Kamei, and A. Sawada, "Assessing the Differences of Clone Detection Methods Used in the Fault-prone Module Prediction," in *Proceedings of the 10th Internal Workshop on Software Clones*, 2016, pp. 15–16.

[20] V. Saini, H. Sajnani, and C. Lopes, "Comparing quality metrics for cloned and non cloned java methods: A large scale empirical study," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, 2016, pp. 256–266.

[21] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[22] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977.

[23] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling Code Clone Detection to Big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[24] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[25] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and Implementation for Investigating Code Clones in a Software System," *Information and Software Technology*, vol. 49, no. 9-10, pp. 985–998, 2007.

[26] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Folding Repeated Instructions for Improving Token-Based Code Clone Detection," in *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, 2012, pp. 64–73.

[27] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[28] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Gapped Code Clone Detection with Lightweight Source Code Analysis," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 93–102.

[29] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th International Conference on Program Comprehension*, 2008, pp. 172–181.