Cloning Practices: Why Developers Clone and What can be Changed

Gang Zhang[1], Xin Peng[1], Zhenchang Xing[2], Wenyun Zhao[1]
[1]School of Computer Science, Fudan University, Shanghai, China
[2]School of Computing, National University of Singapore, Singapore
{09110240022, pengxin, wyzhao}@fudan.edu.cn
xingzc@comp.nus.edu.sg

Abstract*— Code clones are similar code segments. Researchers have proposed many techniques to detect, understand and eliminate code clones. However, due to lack of deeper understandings of reasons of cloning practices, especially from personal and organizational perspectives, little effective support can be provided to alleviate maintenance problems caused by code clones. In this paper, we report an industrial study on investigating reasons of cloning practices in large-scale software development from technical, personal, and organizational perspectives. Our study involves code analysis, questionnaire survey, and interviews with developers, and gathers solid empirical data about how developers clone and why during different phases of clones' lifecycle in industrial development. The results of our study suggest that cloning is not simply a technical issue; it must be interpreted and understood in larger context in which code clones occur and evolve. Within these contexts, there are several adjustable factors and two critical points that affect the introduction, existence, and removal of clones. These adjustable factors and critical points reveal opportunities to improve cloning practices in industrial development from technical, personal, and organizational perspectives.

Keywords-Software clones; Clone lifecycle; Clone context; Industrial study

I. INTRODUCTION

Code clones are identical or similar code segments in software programs. Usually, code clones are introduced by cope-paste-modify practices in software development [1]. They can also be introduced due to application of design patterns, implementation of similar requirements, and usage specification of APIs. Although whether code clones are harmful or not is still an open issue [1], it is agreed on that code clones commonly exist in software systems and they must be made explicit so that they can be consistently managed and maintained.

To that end, researchers have proposed many techniques to detect, understand, and eliminate code clones. Clone detection techniques provide automated assistance in identifying code clones in source code by analyzing Abstract Syntax Tree (AST) [8], Program Dependency Graph (PDG) [9], code metrics [10], and program tokens [6][7]. Clone detectors often report large numbers of clones in large-scale industrial systems [7][8][10]. To help understand and manage code clones in large systems, researchers have proposed visualization and query-based filtering techniques [11][12] for inspecting detected code clones. Furthermore, given detected code clones, some researchers [13][14] have proposed to remove code clones by refactoring such as *Extract Method*, *Pull Up Method*, *Replace Conditional with Polymorphism*.

In spite of the success of these clone detection, analysis and refactoring techniques, little work has been done on systematically understanding why developers introduce code clones into a system and why those clones remain in the system over time. Many studies on code clones [6][8] take an artifact-centric view. They often consider copy-paste-modify as a primary reason for introducing code clones into a system. However, copy-paste-modify is only a means to produce code clones. It does not suggest any technical, personal, or organizational reasons why developers clone.

There have been some studies [1][2][16][17] on summarizing intentions and rationales behind cloning practices, such as development strategies, maintenance benefits, language limitations, and developer's capabilities. However, these studies are mainly based on researchers' personal experiences, with little or no empirical support from large-scale industrial studies. Furthermore, these studies focus mainly on the introduction of clones into the system, with little or no study concerning different phases during the lifecycle of clones.

Thus, several important questions remain unanswered:

- From developers' point of view, how frequently do they clone and what are their attitudes towards clones? How do developers' skills and experience affect the ways they deal with clones?
- In addition to technical reasons/constraints, are there any personal and/or organizational reasons/constraints that make developers clone in the development and maintenance of large-scale industrial systems?
- Are there any technical, personal, and/or organizational reasons/constraints that make developers keep (or remove) existing clones? Are there clones less likely to be removed than others? Why?
- Are technical, personal, and organizational reasons/ constraints equally important during different phases of clones' lifecycle? If not, which reasons/constraints are primary driving forces during which phase?
- Are these reasons independent of one another? Or do they correlate to and/or affect one another and how?

In this paper, we attempt to answer the above questions by investigating the entire lifecycle of code clones in a long-lived large-scale industrial system. Our study investigates clones from not only technical perspectives but also personal and organizational perspectives. It consists of three steps: code analysis, questionnaire survey and interviews. Code analysis is to obtain a preliminary understanding of code clones in the subject system and dig out information about relevant developers of these code clones. Questionnaire survey helps us gather information about how developers clone and why in industrial development from technical, personal, and organizational perspectives. Interviews with developers allow us to dig out deeper understandings of root causes for cloning practices.

Our study gathers solid empirical data, especially from personal and organizational perspectives, to answer the above questions about cloning practices in industrial development. Our analysis suggests that cloning is not simply a technical issue; it must be interpreted and understood in larger context in which code clones occur and evolve, including information from personal, organizational, and historical perspectives. Within these contexts, there are several adjustable factors and two critical points that affect the lifecycle of clones. Identifying and understanding these adjustable factors and critical points opens new opportunities to better tool support for clone detection, analysis, and management, which can in turn improve cloning practices in industrial development.

The remainder of the paper is organized as follows. Section II reviews related work. Section III introduces the design and steps of our study. Section IV reports our study results. Section V discusses insights and lessons learned from our results. Section VI discusses the threats to our study. Finally, we concludes and outlines our future plan.

II. Related Work

Much of research on software clone has been focused on (semi-)automatic techniques to detect, understand and eliminate code clones. In particular, researchers have investigated analyzing Abstract Syntax Tree (AST) [8], Program Dependency Graph (PDG) [9], code metrics [10], and program tokens [6][7] to detect clones. The effectiveness of these techniques have been empirically evaluated and compared [19][20]. Some researchers have proposed visualization and query-based filtering techniques [11][12] to help understand detected code clones. Others [13][14] have investigated eliminating clones by refactoring.

In spite of the promising results of these clone detection and analysis techniques, code clone remains a big challenge to software maintenance of large-scale systems. Researchers have conducted many empirical studies to better understand code cloning in real-world systems. Wang et al. [21] conducted an empirical study of cloning among SCSI drivers for Linux. Roy and Cordy [25] presented an empirical study of near-miss function clones in open-source software and provided a complete catalogue of different clones.

Recently, more and more empirical studies focus on evolution and management of code clones. Mondal et al. [22] reported an empirical study on stability of cloned and non-cloned code in different languages. Bettenburg et al. [23] investigated the impacts of inconsistent changes to code clones on software quality. Their results suggested that developers are able to effectively manage and control the evolution of cloned code at release level. Thummalapenta et al. [26] investigated to what extent clones are consistently propagated and the relationship between clone evolution patterns and clone characteristics. Kim and Murphy [18] investigated clone genealogy information extracted from open-source systems and found that refactoring may not always improve software with respect to clones. Cai and Kim [24] conducted an empirical study of long-lived code clones, and found that evolutionary characteristics of clones may be a better indicator for refactoring needs than static or spatial characteristics of clones.

All these studies examined open-source systems, and focused on code analysis and did not consider reasons and intentions for the introduction, existence and removal of clones. Furthermore, they took an artifact-centric view of code clones and did not analyze information from developer and organizational perspectives.

There has been some work on investigating intentions and reasons why developers clone. Kim et al. [16] constructed a categorization of programmers' intentions for copy-paste practice by inferring their intentions from the editing procedures of clone instances and asking questions in follow-up interviews. Their interviews aimed at confirming intentions inferred from code analysis. In contrast, the goal of our questionnaire survey and interviews is to dig out deeper understanding of reasons for cloning, especially from personal and organizational perspective.

Kapser and Godfrey [2] provided several patterns of cloning that are used in real software systems, analyzed their motivations. Al-Ekram et al. [5] reported their findings of potential reasons for large numbers of accidental clones in several open-source systems. These studies examined only why clones are introduced into the system. Furthermore, the motivations and reasons in these studies were mainly derived from researchers' personal interpretation of code clones. In contrast, our study investigates reasons for cloning over the entire lifecycle of clones, and our results are derived from 21 developers of a large-scale industrial system.

The survey by Roy and Cordy [1] proposed a classification of intentions and reasons for cloning. Although the proposed classification is very comprehensive, it has little or no empirical support from large-scale industrial studies. As pointed out by Roy and Cordy [1], a large-scale industrial study like ours is essential to build effective support for detecting clones as well as alleviating maintenance problems caused by code clones.

III. Study Design and Steps

We now discuss the rationales behind the design of our study. We also describe the key steps of our study.

A. Study Design

The design of our study is driven by our understandings of the lifecycle of code clones and our goals to discover fundamental reasons why developers clone and what can be changed in their cloning practices.

We propose to model the lifecycle of code clones in state diagram shown in Figure 1. A developer may intentionally duplicate a piece of code (e.g. by copy-past-modify) in his local working copy of a software system, and thus create some *tentative clones*. Such clones are tentative because developers have not fully worked out entire solutions yet. Furthermore, the impact of such tentative clones is local; they do not affect others who also work on the system. Tentative clones may be removed, or they may be integrated into the baseline of the system and thus become *baseline clones*. The impact of baseline clones is global; they affect everybody who maintains the system. Alternatively, a developer may introduce some code clones for example due to application of design patterns or implementation of similar requirements. Such clones are accidental because developers are not aware that they create code clones. Due to this unawareness, accidental clones will usually be integrated into the baseline of the system and thus become baseline clones. Once entered the baseline of the system, code clones will remain in the system over time unless they are explicitly removed at some time point.

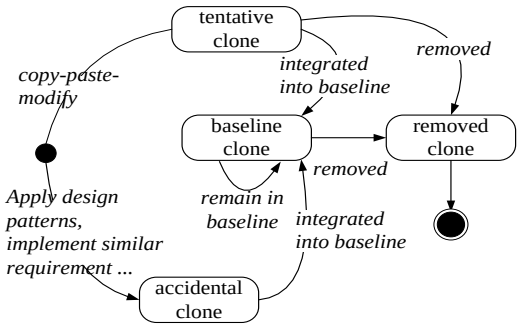

Figure 1. The lifecycle of code clones

The goal of our work requires us capturing and analyzing reasons for cloning from the following three perspectives:

- **Technical perspective** captures reasons originated from the nature of technical problems to be solved, and they usually affect different developers and organizations in the same way.
- **Personal perspective** captures reasons that are related to developers' habits, skills, experience, etc.
- **Organizational perspective** captures reasons that are related to the structure and management strategy of software projects and teams in an organization.

B. Study Steps

Let us now review the key steps of our study, from selection of subject system, code analysis, questionnaire survey, and interviews with developers.

0) Selection of subject system

The subject system we used in this study is a large-scale telecommunication system. The system is written in C/C++ and it has been actively developed and maintained for more than 10 years. The latest version of the system that we examined contains 1,164 modules, 36,634 files and 14,805,698 lines of code. There have been in total 1,773 developers who have worked on the system; about 400 developers are currently working on the latest version.

The selection of this subject system is driven by the following four reasons. First, the system has a large number of code clones at different levels of granularity; these clones are introduced for various reasons. Second, the system has a long history of evolution; this facilitates our observations and analysis of different phases of code clones during their lifecycle. Third, the first author of this paper used to be a key developer of the subject system and is very familiar with the design and implementation of the system. This allows us to select representative meaningful clones for detailed analysis. Finally, the system involves a large number of developers with diverse levels of skills and experiences, so that we can select representative developers of different capabilities for questionnaire survey and interview. Furthermore, the first author still maintains a close relation with the current team of the system. This enables effective questionnaire survey and interviews that require active involvement of developers.

1) Preliminary understanding by code analysis

The objective of code analysis is to establish a preliminary understanding about code clones in the subject system so that we can sample representative clones and come up with potential reasons why these clones are there. These representative clones and their potential reasons provide inputs for our questionnaire design and survey.

We analyzed the subject system with CCFinderX [6]. It detects 47,527 clone classes with 286,983 clone instances from the system. Those clone instances are distributed in 18,298 files of 1,164 modules.

We used the technique proposed by Basit and Jarzabek [3] to aggregate simple clones to file- and module-level clones. We obtained 80 module-level clone classes involving 417 modules for further investigation. For the rest of clone classes that are not included in any module-level clone classes, we calculated the distribution of the size of these clone classes in terms of the numbers of clone instances they have. We sampled 141 clone classes covering different sizes of clone classes for subsequent investigation.

Given sampled clone classes, we manually investigated relevant code. This helps us establish a preliminary understanding about potential reasons for why those clones were introduced and evolved to their current state.

2) Soliciting reasons for cloning by questionnaire

The objective of questionnaire survey is to solicit reasons for cloning at different phases of the lifecycle of code clones (see Figure 1) from technical, personal, and organizational perspectives.

We designed a questionnaire based on three types of inputs. First, we considered potential reasons of sampled representative code clones that are inferred from code analysis. Second, we complemented potential reasons obtained through code analysis with intentions and rationales for code clones that are compiled in the survey paper of clone research [1]. Third, we interviewed three people (one

project manager, one people manager, and one senior developer) before our study. This pre-study interview provides additional inputs to our questionnaire design from personal and organizational perspectives.

TABLE 1. QUESTIONNAIRE DESIGN

start→ tentative clone	q1. Based on my experience, I clone most often during: ___
	q2. I seldom|sometimes|often copy-paste-modify code
	q3. I copy-paste-modify code because: ___
start→ accidental clone	q4. I seldom|sometimes|often notice code clones accidently
	q5. Based on my experience, accidental code clones are often introduced because___
tentative clone→ removed clone	q6. I seldom|sometimes|often remove tentative clones that I created before they enter the baseline of the system
	q7. If you have ever removed some tentative clones, what are your considerations for the removal? ___
tentative clone→ baseline clone	q8. I seldom|sometimes|often leave tentative clones unattended, and thus they become baseline clones in the baseline of the system
	q9. Why do not you remove tentative clones? ___
baseline clone	q10. I seldom|sometimes|often encounter code clones in baseline versions of the system
	q11. I seldom|sometimes|often feel uncomfortable with code clones in baseline versions of the system
baseline clone → removed clone	q12. I am seldom|sometimes|often willing to remove code clones in baseline versions of the system
	q13. I am aware of the following techniques for removing code clones___
	q14. I seldom|sometimes|often remove baseline code clones during development
	q15. If you have ever removed some code clones from baseline versions of the system, what are your considerations for the removal? ___
baseline clone→ baseline clone	q16. I seldom|sometimes|often remove code clones in baseline versions that I think are harmful
	q17. What make you not remove harmful baseline clones? ___

Table 1 summarizes the resulting questionnaire. It contains 17 questions. These questions cover all state transitions during the lifecycle of code clones (see Figure 1). Nine questions are multiple-choice questions with three choices: seldom, sometimes, and often. These questions are concerned with frequency of developers' cloning practices. Six questions are semi-structured with both pre-defined choices and open answers. The rest two questions are free-form questions: Question 7 and Question 15. These questions aim at gathering reasons why developers remove tentative and baseline clones. These two questions are free-form because we cannot come up with specific reasons except general ones such as better understandability and maintainability.

We selected 21 developers from the team of the subject system for questionnaire survey. These developers have 3-12 years (on average 8 years) of industrial experiences. They all have worked on some of sampled code clones. In this study we adopted self-administered questionnaire [15], which means that we talked with each developer face to face during questionnaire survey. We explained the purpose and meaning of questions/answers in the questionnaire. During the survey, we also provided each participant with a set of code clones that he has introduced and/or maintained before. We found that participants were highly inspired by code clones that we provided. Those representative code clones allow participants to better recall their decisions and practices regarding code clones.

After we completed survey with 21 participants, we performed statistics analysis of questionnaire responses. This helps us determine: 1) different attitudes of developers towards code clones, 2) rate of clones produced by different reasons; and 3) clones that are less likely to be removed.

Note that due to space limitation we cannot list the entire questionnaire in this paper. We cannot provide all responses from 21 participants either. Readers can find our questionnaire and the responses from 21 participants at http://www.se.fudan.edu.cn/research/clonestudy2012/.

3) Root-cause analysis through interviews

The objective of interviewing with developers is to dig out deeper understandings of reasons for cloning practices gathered in questionnaire survey. Based on their responses, we selected nine questionnaire participants and conducted follow-up interviews with them. We employed five-why technique [4] for this follow-up interview. Five-why technique is a root-cause exploration technique commonly used in management field to identify root causes of a problem. It starts with an observation of the problem and explores potential causes iteratively by asking questions until root causes are believed to be identified. Note that five-why just refers to such a root-cause exploration technique. The questions being asked may not be exactly five.

Figure 2 shows an example of the application of five-why technique in our interview. In this example, we identified five identical copies for a sort algorithm. These duplicated copies were put under different namespaces (for example, one is in *Protocol* domain, another is in *Kernel Service* domain). During the interview with the developer of this sort algorithm, we provided him with cloned sort algorithm and had the conversation shown in Figure 2. This interview suggests that improper process of module creation is the root cause of the cloned sorting algorithm.

Q: Why did you copy this algorithm?
A: Because the original copy is in a different namespace. I just followed the existing solution.
Q: Why was it put in the different namespace?
A: Because there is no common module to put common algorithms such as this sorting algorithm. So the developer of the original implementation of the algorithm put it into the client module.
Q: Why has the common module not been created?
A: Because there are a lot of managerial process overheads to create common module.

Figure 2. An example of five-why tecnique in our interview

IV. RESULTS

Let us first review empirical data gathered in our study for answering questions about cloning practices listed in introduction.

A. How frequent developers clones and what are their attitudes towards clones?

Responses of 21 participants to Question 2 and Question 10 suggest that cloning is a frequent practice from

developers' point of view. 33% of developers claim that they often copy-paste-modify code; 67% claim that they sometimes do so; none of the participants claim that they never clone. Furthermore, 90.4% of developers claim that they often come across code clones in their daily work.

To further understand developers' attitude towards code clones, we cluster 21 participants based on the similarity of their responses to our questionnaire. Our analysis identifies seven clusters among 21 participants of our questionnaire survey. Four of these clusters contain only one developer, and we consider them as outliers. The rest three clusters contain 3, 3 and 11 developers respectively.

We analyze responses of these 17 developers in terms of their level of uncomfortableness towards existing code clones (i.e. attitude), ways known to them to remove clones (i.e. skill), and how frequent they remove clones (i.e. behavior). Table 2 summarizes our findings. Clearly, developers of the three clusters have distinct characteristics. It is interesting to note that our clustering results are consistent with the reputation and assessment of these developers in the organization.

TABLE 2. DEVELOPERS' ATTITUDE/BEHAVIOR ON CLONE

cluster id	count(ratio)	Characteristics		
		attitude	skill	Behavior
A	3(17.6%)	strong	very good	often
B	3(17.6%)	strong	good	sometimes
C	11(64.8%)	middle	middle	sometimes

Three developers of cluster-A represent most experienced and skillful developers. These developers feel strongly uncomfortable with existing code clones. Thus, they often remove existing clones from the baseline of the system. Furthermore, they rarely let tentative clones enter the baseline of the system and become baseline clones. They can do so because they are able to apply many ways to handle clones in different situations. The responses of these three developers to questions 2 show that they clone less frequently than developers of the other two clusters. Figure 3 shows an excerpt of our interview with one of the developers of cluster-A. To such developers, cloning can be a powerful ways for the tasks at hand.

Sometimes I have to clone some code due to constraints such as time limitation, risk avoidance … Sometimes I also intentionally clone code to explore program that I am not familiar with for example the X library to read a datafile. … So I copy source code from existing programs, and modify it to make it fulfill my work. … Once I finish learning, I will refactor my code to remove duplications.

Figure 3. Interview with one developer from cluster-A

Compared with developers of cluster-A, three developers of cluster-B also feel strongly uncomfortable with code clones. Furthermore, they are also aware of many ways to remove clones. However, their responses to Question 6 and Question 14 suggest that they less likely remove tentative clone and baseline clones. Furthermore, they more likely let tentative clones become baseline clones based on their response to Question 8. We interviewed one of the developers of cluster-B. Our interview suggests that although the developer knows many techniques to remove clones, he does not do so frequently because he does not feel confident to apply these techniques. Consulting with the manager of this developer confirms that this developer is a good developer and has a lot of potential. He just lack of experiences which take time to accumulate.

In contrast to developers of cluster-A and cluster-B, the 11 developers of cluster-C are not so sensitive to code clones and know less about techniques to remove clones. These participants represent a large portion of the developers in the subject organization. They are relatively weak in terms of their skills and experience of handling code clones.

B. Why do developers introduce tentative clones?

Question 3 in our questionnaire is concerned with why developers introduce tentative clones. Table 3 summarizes nine reasons that we collected from the responses of 21 participants. Six of these reasons can be classified as technical (the first to the sixth), two as organizational (the seventh to the eighth), and one as personal (the ninth). Clearly, technical reasons are driving force to introduce tentative clones.

TABLE 3. REASON FOR INTRODUCING TENTATIVE CLONES

Reason	Count (Percentage)
Following existing solutions: There already exist an solution for the problems to be solved.	10 (47.6%)
Avoiding breaking existing features: Reusing needs to change existing code which may break existing features.	9 (42.9%)
Difficult to be reused: The original code contains content irrelevant to the task at hand	9 (42.9%)
Duplicate code is short: Clone is not harmful if duplicate code is short	7 (33.3%)
Lack of proper framework: Has to copy-paste-modify due to lack of proper framework	7 (33.3%)
Easy for separate evolution: Duplicate code so that they can be evolved independently.	1 (4.8%)
Time limitation: Pressured by project progress or so	10 (47.6%)
Issues of code ownership: Code to be reused is owned by other developers, teams, or even organizations	5 (23.8%)
Learning and discovery: Copy-paste-modify for experiment or learning purpose.	9 (42.9%)

It is important to note that most of reasons for introducing tentative clones are not independent of one another. Figure 4 shows part of the cause-effect relationships among these reasons. In the figure, those underlined are reasons identified from questionnaire responses, while others are identified during root cause analysis.

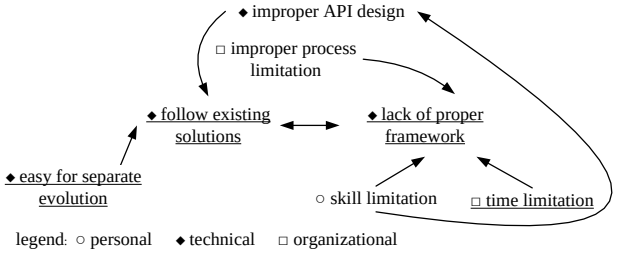

Figure 4. Relationship among reasons

For example, *following existing solutions* may correlate with *lack of proper framework*. We identified many data-configuration related clones in the subject system. All these

clones deal with verifying parameters, save configuration to database, and notify other registered clients about data changes. However, the types of data being processed can be different. Because there is no proper framework for systematic reuse, developers have to copy-paste the implementation of an existing data-configuration, and modify it for handling different types of data. Recently, the organization reconstructed the subject system and provided a new data-configuration framework for systematic reuse. We interviewed a developer who has experiences with both old and new framework. Figure 5 shows an excerpt of our interview. Clearly, this developer appreciates more the new framework so that he does not have to clone any more.

Q: What is your feeling on the new framework?
A: The new framework is far better. We do not need to copy-paste anymore. The development is much more efficient and the code size is only 1/3 compare with the old framework.
Q: Before the new framework was provided, why you and other developers preferred to copy-paste instead of creating a better framework for systematic reuse?
A: Because all people who ever work on it write code in the similar way. We just follow existing solutions.
Q: Why was the new framework built in past months?
A: Because we employed an external technical coach who has very solid design background, and he opened our eyes to a better solution. Meanwhile, the organization allocated us more resources for developing this new framework.

Figure 5. Relation between following existing solutions and lack of proper framework

Following existing solutions can be resulted from *improper API design*. Note that *improper API design* is not in our pre-defined choices for Question 3. It is an example of root cause that we dig out through interview with developers. For example, we identified many clones of invoking a sequence of APIs in such places as distributed service creation, pool-based memory allocation in the subject system. We interviewed the designer of the distributed-service-creation APIs and he commented that:

Originally I was thinking that such APIs are flexible to use. It seems that in most of cases the API clients always use them in a few common scenarios, so a better solution would be to design some facade APIs to hide the complexity … instead of ask developers to call many APIs which could be error prone

Figure 6. Following existing solutions caused by improper API deisgn

Time limitation is one of the two organizational reasons for introducing tentative clones, provided by the participants of our questionnaire survey. *Time limitation* may result in lack of proper framework because developers are pressured to deliver a working product and they often do not have time to restructure the system for better design.

Learning and discovery is the only personal reason for introducing tentative clones that our questionnaire-survey participants mentioned. Figure 3 presents an example for this reason.

C. Why do tentative clones become (or not become) baseline clones?

Question 7 and Question 9 in our questionnaire are concerned with why developers remove tentative clones and why they let tentative clone enter the baseline of the system.

The responses of Question 7 suggest that developers remove tentative clones mainly for quality-driven purposes. Table 4 list four most frequent words mentioned in the responses of 21 participants in our questionnaire survey.

TABLE 4. REASON FOR REMOVING TENTATIVE CLONE

	count	percentage
understandability	11	61.1%
maintainability	9	50.0%
overall code quality	9	50.0%
extensibility	5	27.8%

In contrast to similar quality-driven reasons for removing tentative clones, reasons for tentative clones becoming baseline clones are more diverse. Table 5 summarizes nine reasons that we collected from the responses of 21 participants to Question 9.

TABLE 5. REASON FOR TENTATIVE CLONE BECOMING BASELINE CLONE

Reason	Count (Percentage)
Difficult to change existing code: tentative clones becomes difficult to remove when developer want to do so	8 (38.1%)
Not considered harmful: For some clones, developers do not think they are harmful.	6 (28.6%)
Be consistent with existing clones: Because there are already clones for the similar problem, developer has to keep consistent with them.	5 (23.8%)
Time limitation: tentative clones enter baseline because there is no time to remove them.	14 (66.7%)
Risk avoidance: in the later phase of the project, developers are afraid that changing source code may break the system	14 (66.7%)
Not my responsibility: Developer does not think they have responsibility to prevent baseline clones	4 (19.0%)
Performance evaluation by LOC: code lines produced is used as an productivity measurement	1 (4.8%)
No awareness of harmfulness of clone: developers lack of awareness of how and why clones can be harmful	7 (33.3%)
Knowledge/skill limitation: Developers do not know how to remove clones	6 (28.6%)

Organizational reasons are primary driving force for tentative clones becoming baseline clones. Among four organizational reasons, *time limitation* and *risk avoidance* are the two most significant reasons. To better understand the impact of these organizational reasons on tentative clones, we further interviewed six developers who considered *learning and discovery* as a beneficial reason for introducing tentative clones. This further interview shows that all these developers plan to remove tentative clones at the time they introduce them. However, this plan will not always be carried out. They claim that they remove only 20% - 60% (on average 43%) of tentative clones that they introduce. Two major reasons they gave for not removing tentative clones are in fact *time limitation* and *risk avoidance*. It is consistent with the results shown in Table 5. Even though these developers have to leave some tentative clones in the baseline, all of them are aware of the potential harmfulness of doing so for the maintenance of the system in the future.

Compared with organizational reasons, they impact of technical and personal reasons for tentative clones becoming baseline clones is relatively minor. They show that *difficulty*

to change existing code, be consistent with existing clones, and *no awareness of harmfulness of clone, knowledge and skill limitation* may also make developers not remove tentative clones. As a result, tentative clones become baseline clones.

D. Why are baseline clones (not) removed over time?

Questions 10 to 17 in our questionnaire are concerned with whether developers feel troubled by these clones and whether developers will remove them and why.

For 19 developers who ever remove baseline clones, their responses to Question 15 suggest that they remove baseline clones mainly for quality-driven purposes (see Table 6). This is similar to our finding in reasons for removing tentative clones (see Table 4 in Section IV.C).

TABLE 6. REASON FOR REMOVING BASELINE CLONES

	count	percentage
understandability	6	31.6%
maintainability	4	21.1%
extensibility	11	57.9%
code quality	1	5.3%

Similar to the diversity of reasons for not removing tentative clones, reasons for not removing baseline clones are very diverse. Table 7 summarizes 18 reasons that we collected from the responses of 21 participants to Question 17. Eight of these reasons can be classified as organizational, seven as technical, and two as personal.

Again, similar to reasons for not removing tentative clones, organizational reasons are driving forces to not remove baseline clones as well. In fact, some reasons that participants gave for not removing baseline clones overlap those for not removing tentative clones, for example *time limitation*, *risk avoidance*, *performance evaluation*, *not my responsibility*. In addition, participants of our questionnaire survey gave a few more organizational reasons for not removing baseline clones. These additional reasons suggest that whether an organization appreciates the efforts to removing baseline clones and setting clear plans for do so greatly affect whether developers will remove baseline clones.

From technical perspective, *avoiding re-learning* is an important reason for not removing baseline clones. Developers often feel that removing code clones would affect the understandability of the programs that they are already very familiar with. As a result, developers tend to not remove these clones in order to avoid relearning.

Keeping solution consistent is another important technical reason. For example, on average 80 module-level clone classes identified in our study contain 5.2 modules. Because there are many instances in a clone class, it becomes impossible to remove all of them due to limited project budget and resources, while removing only some of them will introduce inconsistencies among relevant modules.

Lack of initial and historical context is another important technical reason why developers do not remove baseline clones. One of our participants commented that "*it is very difficult to understand the context when the clone is introduced and how it evolves over time*". Interviewing with this participant shows that lack of initial and historical context can be resulted from: 1) the original developer of the code clone is no longer available for consultation; 2) the code quality is poor so that the clone is very difficult to understand; 3) the original project is already closed; there is no enough information and related support (e.g. no testing team, no testing equipment) for removing clones.

We identified two personal reasons for not removing baseline clones. These two reasons are related to developers' knowledge, skills, and their personal interests in software development.

TABLE 7. REASON FOR NOT REMOVING BASELINE CLONES

Reason	Count (Percentage)
Avoiding re-learning: Developer are already familiar with cloned codes	16 (76.2%)
No automation tests: no automation tests to protect the code that would be affected by clone removal	14 (66.7%)
Keeping solution consistent: Due to the existence of many copies, it is impossible to remove all of them, while removing some of them will introduce inconsistencies	10 (47.6%)
Avoiding breaking existing features: Clone removal needs tochange existing code that may break existing features.	10 (47.6%)
Difficult to change existing code: the cloned codes become difficult to remove when developer want to do so.	8 (38.1%)
Lack of tool support: no proper tool support for clone analysis or removal	8 (38.1%)
Lack of initial and historical context: there is little or no information about why clones are introduced and how they evolve to current state	4 (19.0%)
Risk avoidance: remove duplication will increase the project risk	17 (81.0%)
Cost limitation: no budget in the project for clone removal	11 (52.4%)
Time limitation: no time for clone removal	11 (52.4%)
Avoiding cascading effects in the project: clone removal may create additional tasks (e.g. communicating with other developers)	10 (47.6%)
Performance evaluation: not related to my performance evaluation	7 (33.3%)
Not appreciated: clone removal is not appreciated by organization	7 (33.3%)
No plan: clone removal is not listed as a target of the project.	7 (33.3%)
Not my responsibility: the developers do not feel that it's their responsibility to remove others'clone	5 (23.8%)
Knowledge/skill limitation: Keeping the clone is the only choice which developer can use because developers do not know how to remove clones	4 (19.0%)
Not interested: developers are more interested in writing new code than removing clones	2 (9.5%)

E. Why do developers introduce accidental clones?

Question 5 in our questionnaire is concerned with why developers introduce accidental clones. Responses from participants of our survey identify three technical reasons, *lack of knowledge of existing solutions*, *automatically generated code*, and *incomplete code merging*.

Lack of knowledge of existing solutions means that sometimes developers simply do not know a function that he needs has already been implemented somewhere else in the system. For example, a function *isDigit()* was implemented twice in *Network Driver* module and *Voice Service* module respectively. These two *isDigit()* were not the results of

copy-paste after we consulted the developer of *Voice Service* module, which appeared later in code repository. The discussion shows that this function has been placed in *Network Driver* module instead of a common utility module. As a result, the developer of *Voice Service* module did not know existing *isDigit()* in *Network Driver* module.

Automatically generated code is another reason for accidental clones. In our study, we did not expect such responses because the subject company does not allow developers to put automatically generated code into code repository. However, one of the 21 participants reports automatically generated code as the reason for introducing accidental clones. In the follow-up interview with this developer, she suggested that in some cases developers have to do so if they need to modify the automatically generated code. As a result, developers have to commit such generated code into the code repository.

Incomplete code merging is the third reason for accidental clones. For example, we identified two similar modules named *SnmpAgent* and *SnmpAgent-M*; both of them provide *SNMP Agent* service. In fact, these two modules were from two product variants originated from the same ancestor a few years ago, and the two product variants were merged together recently. We interviewed the developer who knows the history of the merging of the two product variants. The developer commented that:

> *We felt very painful when we merged the two products. In fact there are many modules-level clones between the two products because they were originated from the same ancestor. We had spent a lot of effort on merging these cloned modules , and we succeeded in most cases. But for the two modules you showed me, because they are already quite different from each other due to different evaluation direction, we cannot merge them. As a result, they were both kept in the current system.*

Figure 7. Interview with the developer on code merging

We did not find any personal and organizational reasons for introducing accidental clones in the participants' responses to Question 5. However, as discussed above, interviewing with developers shows that there are more fundamental organizational reasons behind technical reasons for introducing accidental clones. For example, *lack of knowledge of existing solutions* may be caused by lack of proper mechanism in the organization to create common modules or lack of communication mechanism between developers. *Incomplete code merging* may be due to lack of proper product splitting/merging strategy in the organization.

V. DISCUSSION

Conducting this study gives us some interesting insights into how we may improve the current cloning practices in industrial development. We discuss them in this section.

A. Context-sensitive view of cloning practices

Our study suggests that artifact- and technical-centric analysis of code clones is not enough. To determine the impacts of code clones on software quality (i.e. harmfulness of clones) and take proper actions, we must examine clones in larger contexts in which code clones occur and evolve.

One type of important contextual information is developers who introduce and maintain code clones. Take the module-level clone shown in Figure 7 as example. During preliminary understanding step of our study, we considered this clone as a good example of cloning for clean and understandable architecture [2] based on code analysis of the clone. However, interviewing with the developer who maintained this clone told an opposite story. The developer claimed that "*We felt hard to merge these two similar modules during product merging, so we left this clone pair in the code*". In this case, considering developers's knowledge turns this clone from useful to harmful.

Another important contextual information is historical context in which clones occur and evolve. A useful clone may become harmful due to changes in historical context over time. For example, tentative clones may be introduced for several beneficial reasons, such as learning and discovery. Our follow-up survey (see Section IV.C) suggests that all participants plan to remove or optimize such tentative clones after achieving learning objectives in order to avoid negative impacts of these clones on future development. However, only 43.3% of the participants actually stick to their plan, which indicates that many tentative clones for learning and discovery will not be removed and thus become baseline clones in the system.

Goals of developers' tasks in a software project and evaluation policies of an organization is the other important type of contextual information that affects developers' attitudes and actions on code clones. As discussed in Section IV.D, if removing a clone aligns well with the goal of the undertaking task such as developing a new feature or fixing a clone-related bug, the clone will more likely be removed and developers consider such actions as beneficial. Otherwise, developers tend to leave clones as is because they do not consider removing clones is their responsibility and do not want to incur cost of relearning and retesting.

B. Adjustable factors for improvingcloning practices

Based on root causes discussed in Section IV, we further identified a set of adjustable factors that can be changed for improving current cloning practice in industrial development. We categorize these adjustable factors again from technical, personal, and organizational perspectives. Note that we consider a root cause for cloning as an adjustable factor if there are feasible and cost-effective ways to change this cause in reality, and such change will affect the lifecycle of code clones.

Adjustable factors from technical perspective are concerned with the needs for more systematic methodologies and better automatic tool supports for cloning. In addition to the application of clone detectors before introducing new functions to avoid cloning [1], a few of further improvements should also be considered to help avoiding clones or eliminating clones. For example, distributed version control system with light-weight branching strategy such as git supports experimental branching without unnecessary integration of clones into code repository. As another example, to avoid accidental clones introduced by code generation, an effective improvement could be applying techniques such as bidirectional transformations [27] to keep high-level models in sync with generated code, and thus

eliminate generated code from code repository. For accidental clones introduced by incomplete code merging, a systematic software product line approach can help to reduce project branching/merging practices and enable better management of product variants.

Adjustable factors from personal perspective are concerned with skills and experience of developers. As discussed in Section IV.A, in most cases skilled and experienced developers are willing to and can eliminate harmful clones before it turns hard to eliminate them. Furthermore, they can utilize tentative clones as a means for learning and discovery and without leaving them as baseline clones. This suggests that we may have more diligent cloning practices by raising awareness of harmfulness of clones, training developers with various techniques of removing clones, and encouraging them applying these techniques in practice.

Adjustable factors from organizational perspective are concerned with improvements on organization and project management. One of such improvements is to encourage high-quality code and abandon size-based effort evaluation, e.g. by lines of code. Furthermore, our analysis in Section IV.D suggests that lack of organization's appreciation to clone removal is an important reason that hinders developers' willingness to remove clones. It may also be helpful if an organization can schedule and allocate more resource for periodic code refactoring.

C. Two critial points for clone removal

Our study identifies two critical points during the lifecycle of code clones for clone removal. Once these two critical points are passed, clones will seldom be removed from the system.

The first critical point is the time when tentative clones become baseline clones. Developers act differently on tentative clones and baseline clones. As discussed in Section IV.C, developers tend to remove tentative clones in order to achieve various software qualities. This shows that developers seem to be more accountable for the clones they create before they become baseline clones in the system. In contrast, as discussed in Section IV.D, developers tend to not remove baseline clones unless doing so aligns well with the goals of their maintenance tasks. One primary reason for the developers' reluctance to remove baseline clones is changes of historical context. Due to changes of code ownership and evolution of clones, developers may feel that they do not have enough knowledge about the initial context and evolution history of clones. As a result, it might be risky to remove baseline clones.

The second critical point is the time when the third copy of cloned code appears. As discussed in Section IV.B and Section IV.D, *following existing solution* is an important reason for cloning. This shows that developers prefer to follow existing solutions when there are already many similar copies. Furthermore, once a piece of code is cloned many times, developers tend to not remove such clones, because they want to *keeping solutions consistent* with and *avoiding re-learning* of codes that people are familiar with. As a result, the more times a piece of code is cloned, the more likely it will be cloned in the future, and the less likely it will be removed. This finding is consistent with "broken windows" theory in the area of criminology and social management, which suggests that, problems should be fixed when they are small, otherwise problems may become more and more serious as people tend to follow earlier examples.

D. Automatic tool supports

Several tools have been proposed to support management of code clones over the lifecycle of clones, such as [6], [11], [12]. These tools support exploring mainly information extracted from code base. However, our study suggests that to effectively manage code clones one must explore and integrate different sources of information beyond code base, for example, developers' reasons and intentions for cloning.

We envision a tool that can record and keep track of reasons and developers' intentions of cloning through different phases of clones' lifecycle. These reasons and intentions may be then used to understand code clones over their lifespan. Furthermore, our study suggests that developers clone for various reasons; rate of clones produced by each reason varies greatly. Cloning statistics collected based on these reasons may enable more context-sensitive detection of code clones, and provide more context-sensitive support for what developers are currently working on, for example suggesting clone refactoring at critical points

In addition to refactoring suggestions for individual clones, the tool may also suggest potential improvements at system level. For example, an accidental clone caused by *lack of knowledge of existing solutions* often implies that code is not placed in the right module so that it is difficult to find it. Therefore, if the tool detects a number of instances of accidental clones caused by the same reasons, the system architect may be alerted to consider moving the code being cloned to the right module. This can help to avoid introducing more clones in the future. For clones caused by personal and organizational reasons, the tool may also provide some general suggestions for improvement. For example, if the tool records a lot of clones resulted from time limitation, it would be helpful for the project manager to consider scheduling more time and allocating more resources for dealing with clones.

VI. THREATS TO VALIDITY

Our questionnaire may be incomplete, because choices for questions, for example potential reasons for removing tentative clones, were derived from our preliminary understanding of code clones in the subject system, existing literatures, and a pre-study interview. Although we encouraged open-ended answers during questionnaire survey, most participants still preferred choosing from given choices, instead of providing open-ended answers. Furthermore, although we explain questions/choices to participants during survey, they may still misunderstand some questions and/or choices. As a result, rate of different reasons may be biased.

Many findings in this study were subject to the quality of participants' questionnaire responses and interview records. Some participants cannot provide valuable inputs for our

questionnaire survey and root cause analysis. Furthermore, although we emphasized that the objective of this study is purely research-oriented and the study results will not be used for evaluation of participants' performance, some participants may still conceal their real thoughts about personal and organizational issues in our questionnaire survey and interview.

Our findings are also subject to our interpretation of questionnaire responses and interview records. Because we do not have complete knowledge about background of participants and project and organizational context of code clones in the subject system, our analysis may be biased.

We investigated only one subject system. Although the subject system is a long-lived large-scale industrial system that contains many representative code clones, it may not cover all kinds of situations of clone practices. The system is developed in C/C++, some of our findings, especially those from technical perspective, may not be valid for systems developed in other languages. Furthermore, we only selected some representative code clones and their relevant developers for questionnaire survey and interview. Finally, organizational reasons we collected in this study may only reflect the characteristics of the subject company. All these limitations may limit the generalizability of our study.

VII. CONCLUSION

In this paper, we reported an industrial study of cloning practices in large-scale industrial development from technical, personal, and organizational perspectives. Our study investigated 80 module-level clone classes and 141 function-level clone classes. We surveyed and interviews 21 developers of these clones for soliciting and understanding reasons and root causes of their cloning practices during different phases of the lifecycle of code clones.

Our results gathered solid empirical data to answer a set of important questions about cloning practices, such as developers' attitudes and behaviors on clones, which reasons are driving forces during different phases of clones' lifecycle, and how different reasons correlate and affect one another. Our study suggested that cloning is not simple a technical issue. Instead, it must be interpreted in personal, organizational, and historical context. Furthermore, our study identified a few adjustable factor and critical points that can be changed for improving current cloning practices in industrial development.

In the future, we plan to conduct more empirical study with different kinds of systems from different companies to generalize our findings. Furthermore, we are interested in developing an automatic tool that can better support clone detection, analysis, and management in terms of not only code similarities but also developers' reasons and intentions for cloning (or not) as well as adjustable factors from different contexts.

REFERENCES

[1] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report No. 2007-541, School of Computing, Queen's University, Canada, 2007.

[2] C. Kapser and M. W. Godfrey. "Clones considered harmful" considered harmful. WCRE 2006, pp. 19-28.

[3] H. A. Basit, S. Jarzabek: A data mining approach for detecting higher-level clones in software. IEEE Trans. Soft. Eng., Vol. 35(4): 497-514, 2009.

[4] B. Andersen and T. Fagerhaug. Root cause analysis: simplified tools and techniques. ASQ Quality Press, 2006.

[5] R. Al-Ekram, C. Kapser, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. ISESE 2005, pp. 376-385.

[6] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. on Soft. Eng., Vol. 28(7): 654- 670, 2002.

[7] B. Baker. On finding duplication and near-duplication in large software systems. WCRE 1995, pp. 86-95.

[8] I. Baxter, A. Yahin, L. Moura, and M. S. Anna. Clone detection using abstract syntax trees. ICSM 1998, pp. 368-377.

[9] J. Krinke. Identifying similar code with program dependence graphs. WCRE 2001, pp. 301-309.

[10] J. Mayrand, C. Leblanc, E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. ICSM 1996, pp. 244-253.

[11] M. Rieger, S. Ducasse, and M. Lanza. Insights into system wide code duplication. WCRE 2004, pp. 100-109.

[12] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. ICSM 2008, pp. 376-385.

[13] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring support environment based on code clone analysis. SEA 2004, pp.222-229.

[14] F. V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. ASE 2004, pp. 336-339.

[15] A. N. Oppenheim. Questionnaire design, interviewing, and attitude measurement. Print pub, 2006.

[16] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. ISESE 2004, pp. 83- 92.

[17] M. Rieger. Effective clone detection without language barriers. Ph.D. Thesis, University of Bern, 2005.

[18] M. Kim and G. Murphy. An empirical study of code clone genealogies. ESEC/SIGSOFT FSE 2005, pp. 187-196.

[19] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. IEEE Trans. on Soft. Eng., Vol. 33(9): 577-591, 2007.

[20] E. Burd, J. Bailey. Evaluating clone detection tools for use during preventative maintenance. SCAM 2002, pp. 36-43.

[21] W. Wang and M. Godfrey. A study of cloning in the linux scsi drivers. SCAM 2011, pp. 95-104.

[22] M. Mondal, C.K. Roy, M.S. Rahman, R.K. Saha, J. Krinke, and K.A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. SAC 2012.

[23] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An empirical study on inconsistent changes to code clones at release level. WCRE 2009, pp. 85-94.

[24] D. Cai and M. Kim. An empirical study of long-lived code clones. FASE 2011, pp. 432-446.

[25] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. J. Softw. Maint. Evol.: Res. Pract., Vol. 22(3): 165-189, 2010.

[26] S. Thummalapenta, L. Cerulo, L. Aversano, M. D. Penta. An empirical study on the maintenance of source code clones. Empirical Soft. Eng., Vol. 15(1): 1-34, 2010.

[27] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, K. Hiroyuki, and L. Montrieux. Maintaining invariant traceability through bidirectional transformations. ICSE 2012.