Electronic Communications of the EASST
Volume 63 (2014)

Proceedings of the
Eighth International Workshop on
Software Clones
(IWSC 2014)

Late Propagation in Near-Miss Clones: An Empirical Study

Manishankar Mondal, Chanchal K. Roy, Kevin A. Schneider

17 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Late Propagation in Near-Miss Clones: An Empirical Study

Manishankar Mondal[1], **Chanchal K. Roy**[2], **Kevin A. Schneider**[3]

[1] mshankar.mondal@usask.ca, https://homepage.usask.ca/~mam815/
[2] croy@cs.usask.ca, http://www.cs.usask.ca/~croy/
[3] kevin.schneider@usask.ca, http://www.cs.usask.ca/~kas/
University of Saskatchewan, Canada

Abstract:

If two or more code fragments in the code-base of a software system are exactly or nearly similar to one another, we call them code clones. It is often important that updates (i.e., changes) in one clone fragment should be propagated to the other similar clone fragments to ensure consistency. However, if there is a delay in this propagation because of unawareness, the system might behave inconsistently. This delay in propagation, also known as late propagation, has been investigated by a number of existing studies. However, the existing studies did not investigate the intensity as well as the effect of late propagation in different types of clones separately. Also, late propagation in Type 3 clones is yet to investigate. In this research work we investigate late propagation in three types of clones (Type 1, Type 2, and Type 3) separately. According to our experimental results on six subject systems written in three programming languages, late propagation is more intense in Type 3 clones compared to the other two clone-types. Block clones are mostly involved in late propagation instead of method clones. Refactoring of block clones can possibly minimize late propagation. If not refactorable, then the clones that often need to be changed together consistently should be placed in close proximity to one another.

Keywords: Code Clone; Late Propagation; Code Evolution; Software Maintenance; Method Genealogy;

1 Introduction

Software maintenance is one of the most important phases of the software development life cycle. Studies [GH11, GK11, LW08, LW10, Kri07, Kri08, ACD07, TCAP09, BKZ13, KG08, MRR$^+$12, MRS12c, MRS12b, MRS13] show that code clones have both positive [Kri07, GH11, GK11, KG08] and negative [LW08, LW10, MRR$^+$12, MRS12c, MRS12b, MRS13, ACD07, TCAP09] impacts on software maintenance and evolution. Code clones are exactly or nearly similar code fragments scattered in the code-base of a software system. These are mainly created because of the frequent copy-paste activities of the programmers with an aim to repeat the same or similar functionalities during software development and maintenance. If a code fragment is copied from one place of a code-base and pasted to some other places with or without modifications, then the original code fragment and the pasted code fragments become clones of one another.

Evolution of clones [KSNM05, ACD07, TCAP09, BKZ13] has been investigated by different studies in different ways. In this study we investigate a particular clone evolution pattern which is known as late propagation in clones according to the literature [ACD07, TCAP09, BKZ13]. There are strong empirical evidences [ACD07, TCAP09, BKZ13] that late propagation is directly related to bugs [ACD07, TCAP09] and inconsistencies [BKZ13] in the code-base. Researchers have investigated different specific patterns [BKZ13] of late propagation and identified which patterns are more related to bugs, faults, and inconsistencies. However, the existing studies regarding late propagation in clones have the following draw-backs.

(1) None of the studies investigate the intensities of late propagation in different types of clones separately. Such a study is important because, if late propagation is observed to be more intense in a particular clone type compared to the others, we might consider being more conscious while changing clones of that particular type. Also, we might want to refactor clones of that particular type with higher priority.

(2) None of the studies considered Type 3 clones while investigating late propagation. According to a recent study [SRSP13], the number of Type 3 clones can be considerably higher compared to the other two clone types (Type 1, and Type 2). Thus, if we do not consider Type 3 clones in late propagation study, we might miss a significant amount of inconsistencies and faults introduced by late propagation in Type 3 clones.

(3) The existing studies have not emphasized on the identification of the possible causes of late propagation. Also, the possibilities of minimizing late propagation have not been well investigated.

Focusing on the above issues, we investigate late propagation in three types of clones (Type 1, Type 2, and Type 3) separately and answer the following important research questions which were not answered before.

- ***RQ 1.*** *Are the intensities of late propagation different in different types of clones?*
- ***RQ 2.*** *Do the participating clone fragments in a clone pair that experience late propagation generally remain in different files?*
- ***RQ 3.*** *Do block clones or method clones exhibit higher intensity of late propagation?*

According to our experimental results on thousands of revisions of six subject systems written in three different programming languages,

- The intensity of late propagation in Type 3 clones is higher compared to the other two clone types.
- Refactoring of block clones can help us minimize late propagation considerably.
- If not refactorable, the clones that often need to be changed together consistently should remain in close proximity to one another so that while changing a particular clone fragment, a programmer does not forget to look at the other fragments to determine whether changes need to be propagated to these fragments too.

The rest of the paper is organized as follows. Section 2 describes the related terminology, Section 3 elaborates on the detection of late propagation in clones, Section 4 discusses the experimental steps, the experimental results are presented and analyzed in Section 5, Section 6 discusses the related work, Section 7 mentions possible threats to validity and finally, we conclude our paper by mentioning future work in Section 8.

2 Terminology

Types of Clones. We conduct our experiment regarding late propagation considering exact (Type 1) and near-miss clones (Type 2 and Type 3 clones). As is defined in the literature [Roy09], if two code fragments are exactly the same disregarding the comments and indentations, they are Type 1 clones of each other. Type 2 clones are syntactically similar code fragments. In general, Type 2 clones are created from Type 1 clones because of renaming variables or changing data types. Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones.

Clone Class. A group (i.e., two or more) of clone fragments that are the same (Type 1) or similar (Type 2 or Type 3) to one another form a clone class. We detect clones using NiCad [CR11, RC08] clone detector that reports clones by grouping them into classes.

Clone Pair. Two clone fragments belonging to the same clone class form a clone pair. Thus, every possible pair (i.e., combination) of two clone fragments in a particular clone class is a clone pair. From each of the clone classes reported by NiCad [CR11] we determine all possible clone pairs for conducting our experiment.

Cloned Method. If a method contains cloned (Type 1, Type 2, or Type 3) lines, we call this method a cloned method. If all lines of a method are cloned lines, then this method is a fully cloned method. If a method contains both cloned and non-cloned lines, we call this method a partially cloned method.

Method Clones. If two or more methods are clones (Type 1, Type 2, or Type 3) of one another, we refer to these as method clones. Method clones are fully cloned methods.

Here, we should note that we conduct our experiment considering - (1) method clones and (2) block clones that reside in methods as was done in a previous study [BKZ13].

Late Propagation in a Clone Pair. Let us consider a pair of clone fragments. We say that this clone pair has experienced late propagation if it receives a diverging change followed by a converging change.

- ***Diverging Change.*** Let us assume a particular commit C_i where one or both of these two clone fragments were changed. Because of this change, the fragments were not considered as clones of each other. In other words, the clone fragments diverged. Such a change is called a *diverging change* for the pair.

- ***Converging Change.*** Let us assume a later commit C_{i+n} ($n >= 1$) where one or both of these fragments were changed, and because of this change, the fragments were again considered as clones of each other. In other words, the fragments converged. The change for which the fragments converged is termed as a *converging change*.

A particular clone pair may experience late propagation more than once during evolution. Fig.

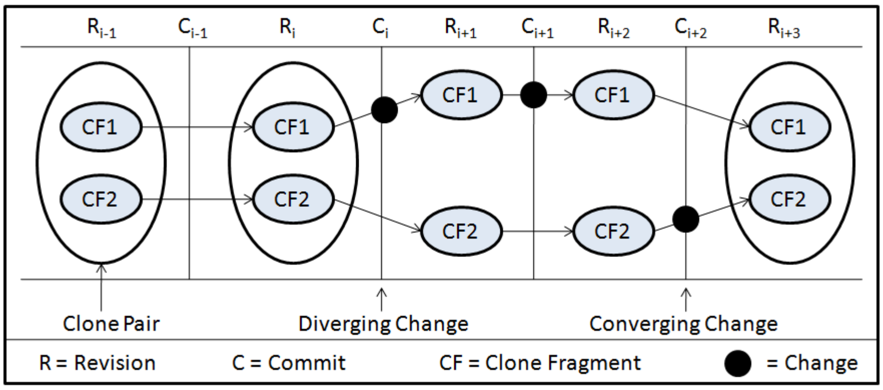


Figure 1: A possible example of late propagation

1 shows a possible example of late propagation experienced by a clone pair (*CF1*, *CF2*). The commit C_i applied on revision R_i modified *CF1* and as a result, *CF1* and *CF2* diverged. However, in commit C_{i+2}, the fragment *CF2* changed and *CF1* and *CF2* again became clones of each other in R_{i+3}.

We conduct our late propagation study considering the granularity of clone pairs as was done by each of the previous studies. While it would be good to conduct such a study considering clone classes, consideration of clone classes might cause the loss of important information regarding late propagation. Let us assume a clone class consisting of six clone fragments in revision R_i. The subsequent commits might affect only two of these six clone fragments leaving the other four fragments as they are. There is a possibility that these two clone fragments (that are getting changed) will experience a late propagation (i.e., the changes occurred in one clone fragment will propagate to the other one with some delay) in future evolution. However, the other four clone fragments might not require to be changed during the whole period of evolution. In other words, the changes occurred to the two clone fragments might not ever need to be propagated to the other four clone fragments. In such a situation, consideration of all these six clone fragments for late propagation is not reasonable. While pairs of clone fragments in a particular class might experience late propagation, the whole class might not.

3 Detection of Late Propagation

At the very beginning, we assume a global list of clone pairs each of which has the potential of experiencing late propagation. We call such a clone pair a CPLP (Clone Pair having the potential of experiencing Late Propagation). We call this list the GLOBAL LIST.

Clone Pair having the potential of experiencing Late Propagation (CPLP). We consider a pair of code fragments, (*CF1* and *CF2*), which are clones of each other in revision R_i. A commit operation C_i was applied on R_i and one or both of these fragments changed. However, because of this change, *CF1* and *CF2* were not considered as clones of each other in revision R_{i+1}. In other words, the change in C_i is a diverging change for the pair (*CF1*, *CF2*). This pair is considered as a CPLP because, there is a possibility that in a future commit operation, the fragments *CF1* and *CF2* will converge (i.e., *CF1* and *CF2* will again be considered as clones of each other).

The GLOBAL LIST remains empty initially. We examine the commit operations sequentially from the very beginning one. We only consider those commits where there were changes to one or more clone fragments of a particular clone type. As we examine the commit operations, we update the GLOBAL LIST and mark some clones pairs (i.e., some CPLPs) in this list as the late propagation clone pairs. Suppose, C_i is such a commit which was applied on revision R_i and the immediate next revision R_{i+1} was created as a result. We perform the following steps sequentially considering C_i.

Step 1. Determining the list of affected clone fragments. We identify the list of clone fragments (in revision R_i) that received some changes during C_i. We call this list the LIST OF AFFECTED CLONE FRAGMENTS.

Step 2. Determining the list of affected clone pairs. We make a list of clone pairs that involve one or more clone fragments in the LIST OF AFFECTED CLONE FRAGMENTS. We denote this list of clone pairs as the LIST OF AFFECTED CLONE PAIRS.

Step 3. Updating the GLOBAL LIST using the LIST OF AFFECTED CLONE FRAGMENTS. We identify those clone pairs in the GLOBAL LIST each of which involves any of the clone fragments in the LIST OF AFFECTED CLONE FRAGMENTS. There is a possibility that such a clone pair in the GLOBAL LIST has converged. In order to check this we determine whether the fragments in such a pair are considered as clones of each other in revision R_{i+1} which was created because of commit C_i. If this is true, then we understand that this clone pair in the GLOBAL LIST has experienced late propagation. We mark this clone pair as a late propagation pair.

Step 4. Updating the GLOBAL LIST using the LIST OF AFFECTED CLONE PAIRS. If any pair in the LIST OF AFFECTED CLONE PAIRS already appears in the GLOBAL LIST, we do not need to consider this pair because, this has already been handled in the previous step. Considering the remaining pairs (in the LIST OF AFFECTED CLONE PAIRS), we determine the CPLPs (i.e., the clone pairs that have the potential of experiencing late propagation). If the two fragments in a remaining pair are not considered as clones of each other in revision R_{i+1}, then this pair is a CPLP. We include the CPLPs in the GLOBAL LIST.

For each of the commit operations we follow the above four steps, update the GLOBAL LIST and mark some CPLPs in this list as the late propagation pairs if they converge. After examining all the commit operations we get all the late propagation clone pairs of a particular clone type.

Now, let us assume that a particular pair in the GLOBAL LIST has been marked as a late propagation pair during the examination of the commit operation C_i. This pair has the following three possibilities during future evolution.

- The pair may again experience late propagation.
- The fragments in the pair may evolve independently.
- One or both fragments may form new pair(s) with other fragments of the same or other clone types.

In case of the third possibility, our implementation considers the new pairs in calculation because they can experience late propagation. For this first two cases we do not need to do any further processing because, the clone pair has already been detected as a late propagation pair and our aim is to identify whether any clone pair has ever experienced late propagation during evolution. After experiencing late propagation the fragments in the pair might evolve independently, however, this is not our concern in this research work.

Detection of late propagation considering an individual clone-type. Suppose we are detecting late propagation considering the clones of Type j where j = 1, 2, or 3. The clone fragments *CF1* and *CF2* are clones of this type in revision R_i. Because of the commit C_i on revision R_i, the fragments *CF1* and *CF2* diverged. Let us assume that in commit C_{i+n}, the fragments converged and they were again considered as clones of Type j. Then, we consider this late propagation as a late propagation of Type j. It might be the case that after converging, *CF1* and *CF2* were not considered as clones of Type j. They were considered as clones of Type k where k= 1, 2, or 3 and $j \neq k$. In this case we do not consider a late propagation, because the fragments changed their types. While detecting late propagation in the clones of Type k, the fragments *CF1* and *CF2* are considered to determine whether they experienced a late propagation of Type k. However, we plan to investigate the intensity of such mixed type late propagations (i.e., where the participating fragments were considered of one clone type before divergence but of another clone type after convergence) as a future work.

Late propagation in Type 3 clones. We know that Type 3 clone fragments contain some non-cloned lines. Our implementation can detect late propagation caused by the diverging changes in the cloned lines of the Type 3 clones. However, there might be situations when significant changes (i.e., increase) in the non-cloned portions can cause two Type 3 clones to diverge because the proportion of non-cloned portions might exceed the dissimilarity threshold limit. These fragments might again converge and be considered as Type 3 clones of each other. Our implementation detects late propagation in these situations too. We do not disregard these situations, because convergence might happen because of significant changes (i.e., increase) in cloned portions so that the proportions of non-cloned portions again become within threshold limit. In this research work, we do not investigate the intensity of such late propagations (i.e., caused by the changes in non-cloned portions of Type 3 clones). We plan to investigate this as a future research work.

4 Experimental Steps

We conducted our experiment on six subject systems listed in Table 1. We downloaded the revisions of each of these systems from a open-source SVN repository SOURCEFORGE[1]. For each of the subject systems we considered each of the revisions beginning from the first one. Detection of late propagation by mining the revisions of a particular subject system requires the following experimental steps to be done sequentially - (i) Extraction of methods from each of the revisions, (ii) Detection of method genealogies, (iii) Extraction of clones from each of the revisions, (iv) Locating these clones to the already detected methods, (v) Extraction of changes between every two consecutive revisions, (vi) Reflecting these changes to the already detected methods and clones residing in these methods, and (vii) Detection of clone pairs that experienced late propagation.

[1] Sourceforge: http://www.sourceforge.net

Table 1: Subject Systems

Systems	Lang.	Domains	LOC	Revisions
Ctags	C	Code Def. Generator	33,270	774
QMailAdmin	C	Mail Management	4,054	317
jEdit	Java	Text Editor	1,91,804	4000
Freecol	Java	Game	91,626	1950
OpenYMSG	Java	Yahoo Messenger	15,553	297
GreenShot	C#	Multimedia	37,628	999

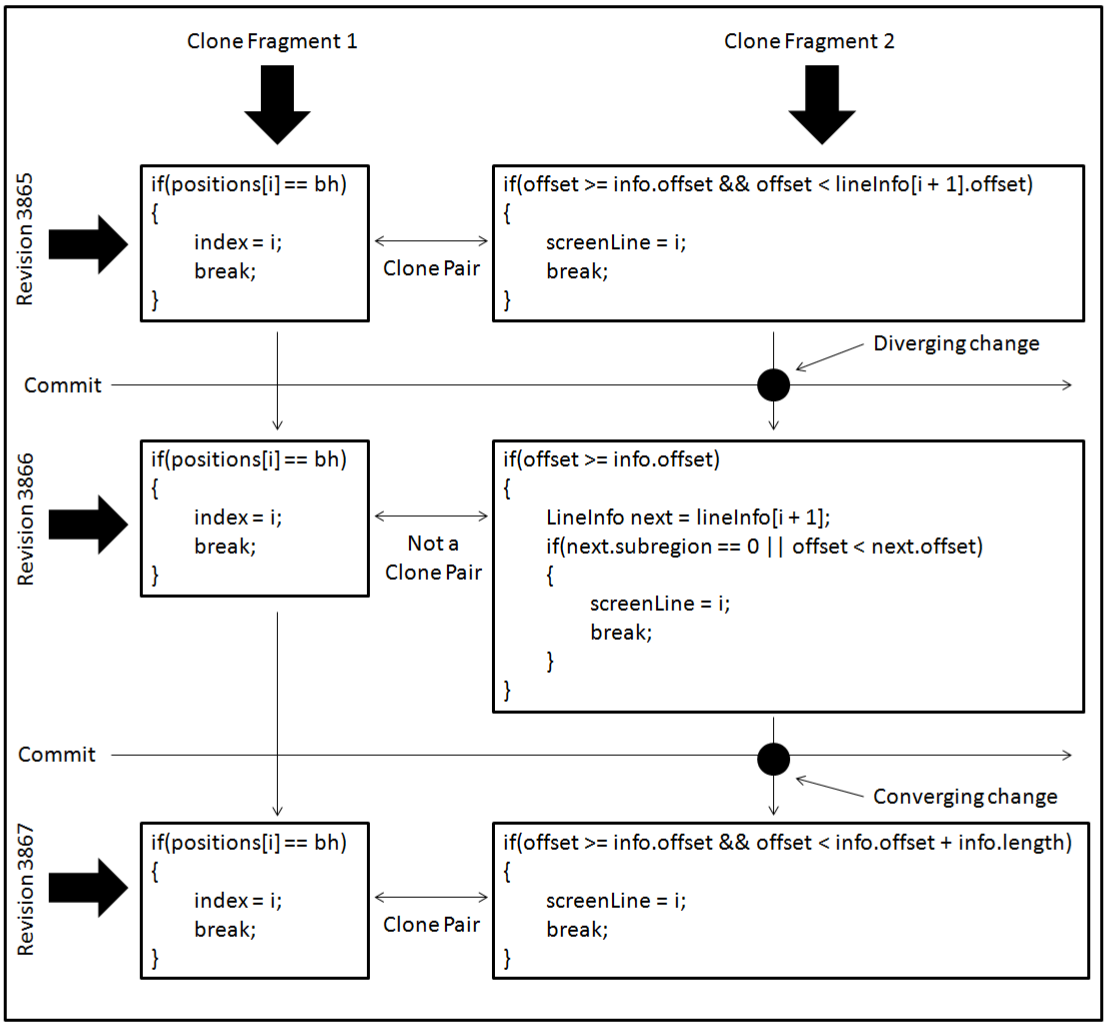
Table 2: Intensity of Late Propagation in Different Types of Clones

	Type 1		Type 2		Type 3		
System	TG	PL	TG	PL	TG	PL	CTHP
Ctags	89	0 %	97	2.06 %	258	5.42 %	T 3
QMailAdmin	33	12.12 %	27	22.22 %	41	17.07 %	T 2
jEdit	9007	0.24 %	1108	2.52 %	3991	1.70 %	T 2
Freecol	253	7.5 %	387	4.39 %	935	7.5 %	T 1, T 3
OpenYMSG	584	0 %	85	0 %	270	2.59 %	T 3
GreenShot	1896	0.53 %	478	1.67 %	964	4.46 %	T 3

T i = Type i **TG** = Number of total clone genealogies in the system
PL = Percentage of clone genealogies that exhibited late propagation
CTHP = Clone type exhibiting the highest intensity of late propagation

We extract methods using CTAGS[2]. For detecting method genealogies we follow the procedure proposed by Lozano and Wermelinger [LW08]. The genealogy of a particular method helps us to understand how a particular method evolved during software evolution. As we detect method genealogies, by locating clones to methods we can determine how a particular clone fragment changed over the evolution. We use NiCad clone detector for detecting and extracting clones from each revision of a subject system. The main purpose of choosing NiCad is that it can detect clones of different clone-types separately including Type 3 with high precision and recall [RC09, RCK09]. For detecting Type 3 clones, we considered a dissimilarity threshold of 20% with blind renaming of identifiers. For the details of the first six steps mentioned above and for NiCad setup, we refer the interested readers to our earlier work [MRS12a]. The last step (i.e., step vii) has been described in Section 3.

[2] Ctags: http://sourceforge.net/projects/ctags/?source=directory



```
if(positions[i] == bh)
{
    index = i;
    break;
}
```

```
if(offset >= info.offset && offset < lineInfo[i + 1].offset)
{
    screenLine = i;
    break;
}
```

```
if(positions[i] == bh)
{
    index = i;
    break;
}
```

```
if(offset >= info.offset)
{
    LineInfo next = lineInfo[i + 1];
    if(next.subregion == 0 || offset < next.offset)
    {
        screenLine = i;
        break;
    }
}
```

```
if(positions[i] == bh)
{
    index = i;
    break;
}
```

```
if(offset >= info.offset && offset < info.offset + info.length)
{
    screenLine = i;
    break;
}
```

Figure 2: An example of late propagation in a Type 3 clone pair from subject system jEdit

5 Experimental Results and Discussion

We applied our implementation on each of the six subject systems in Table 1 and identified the clone pairs that experienced late propagation considering each of the three clone types (Type 1, Type 2, Type 3). As late propagation in Type 3 clones was not investigated before, we present an example of late propagation occurred to a Type 3 clone pair of our subject system jEdit. We automatically detect this late propagation by applying our late propagation detection tool. We present Fig. 2 for describing the late propagation example.

In Fig. 2 we see a Type 3 clone pair in revision 3865 of our candidate system jEdit. As we can see, the participating clone fragments (denoted as *Clone Fragment 1* and *Clone Fragment 2*) are two if-blocks. NiCad detects these Type 3 clones by considering a dissimilarity threshold of 20% and applying blind renaming of identifiers. These two clone fragments belong to two different source code files[3] [4]. The names of the container methods of these two clone fragments are *removePosition* and *getScreenLineForOffset* in revision 3865 . The commit operation applied on revision 3865 changed the clone fragment at the right hand side (i.e., Clone Fragment 2). Because of this change they were not considered as a clone pair in revision 3866. Thus, this change is a diverging change for the clone pair. However, the commit operation applied on revision 3866 again changed the fragment at the right hand side and the fragments converged (i.e., became a clone pair) in revision 3867. Thus, this clone pair experienced a late propagation.

In the following subsections, we answer the research questions mentioned in the introduction by presenting and analyzing our experimental results.

5.1 RQ 1: Are the intensities of late propagation different in different types of clones?

Rationale. Answering this research question is important. If it is observed that late propagation in a particular clone-type is more intense compared to the other clone-types, then it is an implication that clones of that particular clone type have a higher probability of introducing bugs and inconsistencies to the code-base compared to the other types. Thus, it would be beneficial if we could refactor clones of that particular type with higher priority. By minimizing these clones we can minimize the possibility of faults and inconsistencies to the code-base.

Methodology. For answering this research question we applied our prototype tool on each of the candidate systems and determine the following measures considering each of the three types of clones of each of the subject systems.

- The number of total clone genealogies
- The percentage of clone genealogies that experienced late propagation

We show these in Table 2. We also calculate the overall proportion of clone genealogies involved with late propagation considering each clone type. These proportions are shown in Fig.

[3] Source code file for Clone Fragment 1: trunk/org/gjt/sp/jedit/buffer/OffsetManager.java

[4] Source code file for Clone Fragment 2: trunk/org/gjt/sp/jedit/textarea/ChunkCache.java

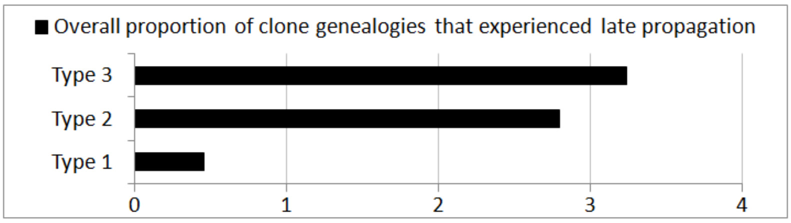


Figure 3: Overall proportion of clone genealogies that experienced late propagation (considering each type of clones)

Table 3: Number of Clone Pairs that Experienced Late Propagation

Lang.	System	Type 1	Type 2	Type 3
C	Ctags	0	1	9
C	QMailAdmin	2	3	7
Java	jEdit	13	21	62
Java	Freecol	12	10	49
Java	OpenYMSG	0	0	4
C#	GreenShot	5	4	37

3. Overall proportion was calculated in the following way.

$$OP_{T_i} = \frac{\sum_{s \varepsilon S} LP_{T_i}(s)}{\sum_{s \varepsilon S} TG_{T_i}(s)} \times 100 \quad (1)$$

Here, OP_{T_i} is the overall proportion of late propagation genealogies of clone-type *Type i* where i = 1, 2, or 3. $LP_{T_i}(s)$ is the number of late propagation genealogies of clone-type *Type i* in subject system s. Finally, $TG_{T_i}(s)$ represents the total number of clone genealogies of clone-type *Type i* in s. S is the set of all subject systems.

Analysis. From Table 2 we see that for most of the subject systems (four out of six), Type 3 clones exhibit the highest intensity of late propagation in comparison with the other two clone types (Type 1, and Type 2). Moreover, Fig. 3 clearly demonstrates that the overall proportion of Type 3 clones that exhibit late propagation is higher compared to the other two types.

Answer to RQ 1. *According to our experimental results, the intensity of late propagation in Type 3 clones is generally higher compared to the intensity of late propagation in other two clone-types. Thus, possibly Type 3 clones have higher probability of introducing faults and inconsistencies to a code-base than the clones of other two types.*

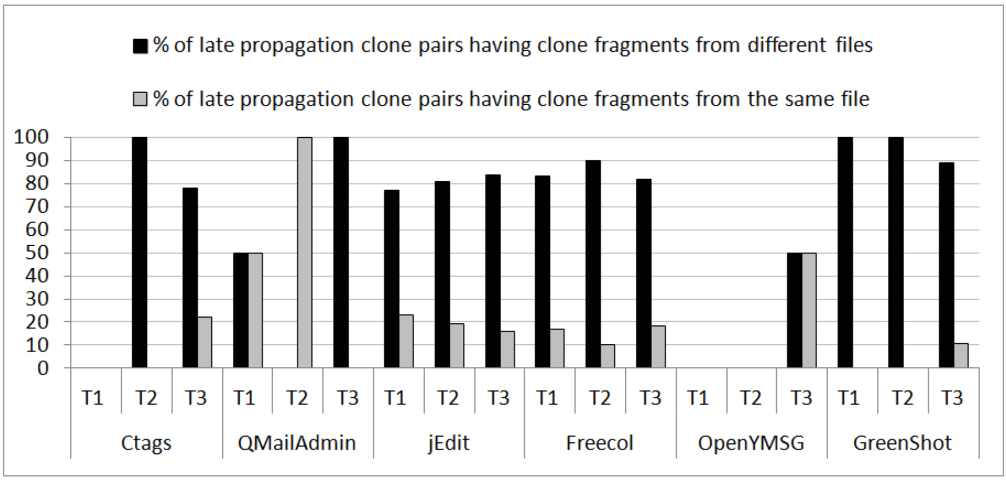


Figure 4: Proportions of late propagation clone pairs having clone fragments from different files or the same file (considering each clone type)

5.2 RQ 2: Do the participating clone fragments in a clone pair that experience late propagation generally remain in different files?

Rationale. Answering this question is important. According to a number of studies [DLL09, VPV10], the program entities that often need to be changed together (i.e., that often require corresponding changes) should remain in close proximity to each other so that while changing a particular entity the developer does not miss to look at other entities that may require corresponding changes. Considering this fact we suspect that possibly the clone fragments in a clone pair that exhibit late propagation generally remain in two different files and as a result, the developers often forget to make corresponding changes to these clone fragments. We investigate in the following way to look into this matter.

Methodology. We have already said that considering each of the clone types of each of the subject systems we identify the clone pairs that experienced late propagation. The counts of these clone pairs are shown in Table 3. For each of these pairs we determined whether the participating clone fragments remain in different files or in the same file. We determined two percentages - (i) the percentage of the clone pairs having clone fragments from different source code files and (ii) the percentage of clone pairs consisting of clone fragments from the same file. These percentages are shown in Fig. 4. We also determined the overall percentages for each clone type considering all subject systems using an equation similar to Eq. 1. Fig. 5 contains these percentages.

Analysis. Considering the percentages regarding each clone-type of each of the subject systems in Fig. 4 we can say that in general, the clone fragments in a clone pair that experience late propagation belong to different source code files. From the figure we see that for most of the cases, the percentage of late propagation clone pairs consisting of clone fragments from different files is much higher than the percentage of late propagation clone pairs having clone fragments from the same file. The overall percentages regarding each clone-type in Fig. 5 also demonstrate

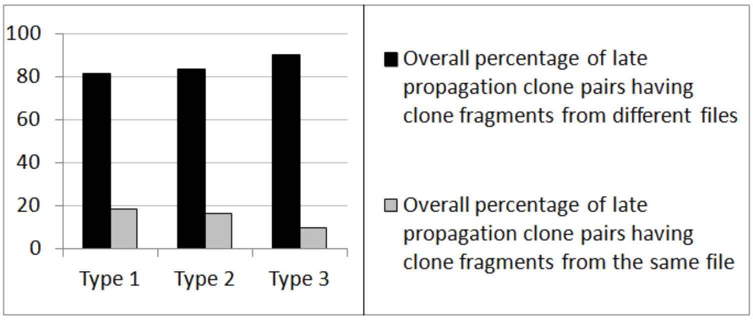


Figure 5: Overall proportions of late propagation clone pairs having clone fragments from different files or the same file (considering each type)

that most of the clone pairs that exhibit late propagation are formed with clone fragments belonging to different source code files. Moreover, intensity of such a scenario in Type 3 case is higher compared to the other two types (in Fig. 5).

Answer to RQ 2. *In general, the participating clone fragments in a clone pair that experience late propagation remain in different source code files*. This can be a possible reason of late propagation, because without proper tool support a programmer might often forget or even unable to propagate the changes occurred to a clone fragment in one file to another clone fragment remaining in another file. Thus, according to our findings, if it is necessary that the clone fragments in a particular clone pair often be changed consistently, then if possible, it is better to place the clone fragments in close proximity to each other (if they cannot be merged) so that while changing one fragment a programmer does not miss to look at the other fragment to decide whether changes also need to be propagated to this other fragment.

5.3 RQ 3: Do block clones or method clones exhibit higher intensity of late propagation?

Rationale. Intuitively, copying a block of statements from one method and pasting that block to several other methods is more difficult compared to copy-pasting a whole method. While pasting a block of statements into a method, the variable names and data types in the block might need to changed in accordance with the variables and data types in that method. If there is a problem in making such correspondence and as a result, the variables are not changed correctly, then this will create inconsistency in future evolution. If a number of block clones (forming a clone class) are created with such inconsistencies, these inconsistencies in different clone fragments will be discovered at different times during evolution and as a result, late propagation will happen.

Also, blocks might not have well defined boundaries as of methods. For this reason, keeping track of block clones might seem to be more difficult compared to method clones to a programmer.

Methodology. Considering each of the clone types of each of the subject systems, we at first determine those clone pairs that exhibited late propagation and then we determine whether the clone fragments in such a pair are block clones or method clones. We calculate - (i) the

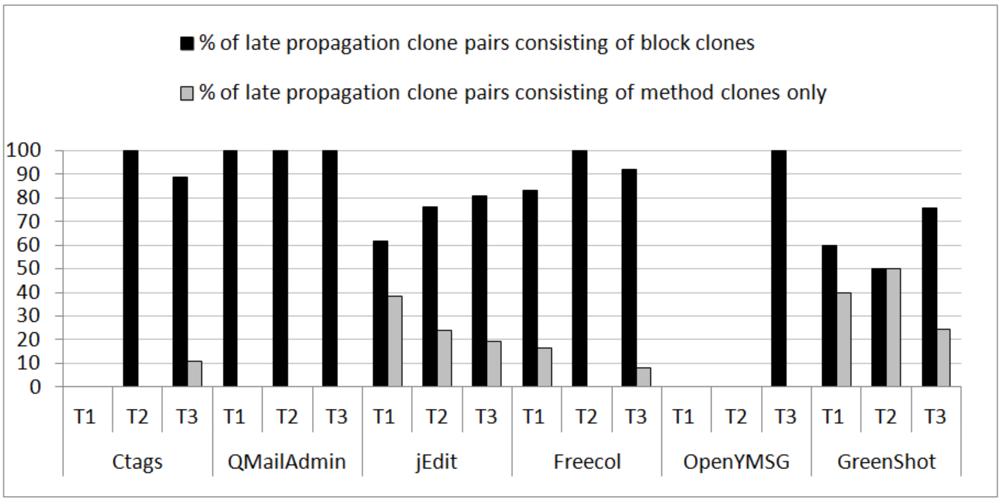


Figure 6: Proportions of late propagation clone pairs involving block clones or method clones (considering each type of clones)

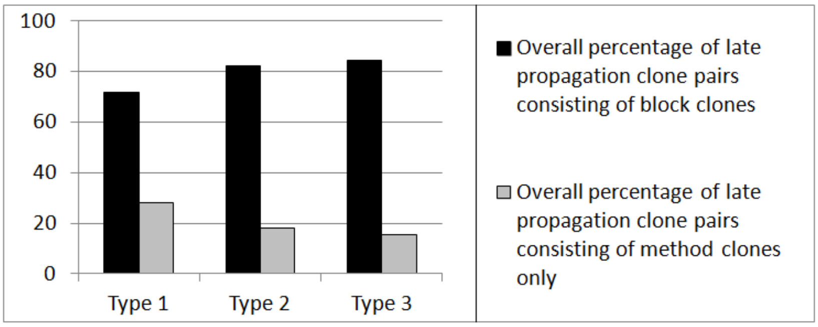


Figure 7: Overall proportions of late propagation clone pairs involving block clones or method clones (considering each type of clones)

percentage of clone pairs each consisting of at least one block clone and (ii) the percentage of clone pairs consisting of method clones only. These percentages regarding three clone types of each of the subject systems are shown in Fig. 6. We determined the overall percentages for each of the clone types considering all candidate systems using an equation similar to Eq. 1. These overall percentages are shown in Fig. 7.

Analysis. From Fig. 6 we see that for almost all of the cases, the percentage of late propagation clone pairs involving block clones is much higher (100 % in many cases such as Type 2 case of Freecol) than the percentage of late propagation pairs consisting of only method clones. Although we determine the percentage of late propagation clone pairs having at least one block clone, for most of the cases we observed that both of the clones in such a pair are block clones. However, for a very few cases (Type 1 case of Ctags and Type 1 and Type 2 cases of OpenYMSG) we did not get any clone pair experiencing late propagation.

The overall proportions regarding three types of clones in Fig. 7 demonstrates that in case of each clone type, the overall proportion of late propagation pairs involving block clones is much higher compared to that of the late propagation pairs consisting of method clones.

Answer to RQ 3: *The clone pairs that experience late propagation generally consist of block clones instead of method clones. Block clones exhibit much higher intensity of late propagation than method clones.* Such an observation implies that block clones possibly have higher probability of introducing inconsistencies to a code-base compared to the method clones. Thus, we should possibly consider refactoring (if possible) block clones with higher priority.

5.4 Discussion

From our answers and observations regarding the previous three research questions we understand that the best way of minimizing late propagation is to minimize block clones by refactoring. A recent study [ZR13] shows that extract method refactoring can possibly be used for removing block clones. However, there could be situations where removal of block clones might not be possible. In these situations, if it is necessary that the clones often be changed together consistently, then it is better to place the clone fragments (i.e., the methods containing the clone fragments) in close proximity to each other so that a programmer can look at all these fragments while making changes to any one. Alternatively, we can build a database of such clones with automatic tool support so that if a programmer attempts to change a particular clone fragment, he/she will be able to look at the other clone fragments (stored in the database) that also have the possibility of getting changed together. Such an automatic system with clone database can help us considerably in minimizing late propagation.

6 Related Work

A number of studies have already been done on clone evolution and late propagation in clones during evolution. Kim et al. [KSNM05] studied clone evolution by defining and extracting clone genealogies from two Java systems using CCFinder[5] as the clone detector. Krinke [Kri07] studied the consistent and inconsistent changes to the Type 1 clones considering the evolutions

[5] CCFinder. http://www.ccfinder.net/ccfinderxos.html

of five open source subject systems using Simian[6] clone detector. He also studied the stability of clones [Kri08] in comparison with non-cloned code. Göde et al [GH11, GK11] analyzed clone evolution and its effect on software maintenance by enhancing Krinke's study [Kri08].

In a recent study, Barbour et al. [BKZ13] investigated eight different patterns of late propagation by studying three open-source subject systems written in Java and identified two patterns that have higher likelihood of introducing inconsistencies to a code-base. They used three clone detection tools NiCad, CCFinder, and Simian in their study. However, Type 3 clones were not considered in this study. Aversano et al. [ACD07] investigated clone evolution on two subject systems to determine how clones are maintained. According to their observation 18% of the clones experienced late propagation. They show that late propagation in clones can directly be related to bugs and thus late propagation is risky. In another study Thummalapenta et al. [TCAP09] investigate late propagation in clones considering four subject systems and reported that late propagation is often related to faults and inconsistencies.

We see that while there are number of great studies, none of these focus on the intensity of late propagation separately in different types of clones. Also, no study investigated late propagation in Type 3 clones. Moreover, the existing studies did not emphasize on the causes of late propagation and on the possibilities of minimizing late propagation considering the causes. In this study, we investigate these issues by answering three research questions. We believe that our findings are important and have the potential to help us in better clone maintenance.

7 Threats to Validity

The number as well as the percentage of clone genealogies that experienced late propagation may vary because of the variation of the detection parameters of the clone detection tool (NiCad in our study). However, the settings that we have used for NiCad are considered standard and with these settings NiCad can detect clones with higher precision and recall [RC09, RCK09]. Moreover, NiCad can report three types of clones (Type 1, Type 2, Type 3) separately and thus, it helped us to investigate the intensity of late propagation on these clone-types separately.

The number of subject systems that we have used in our experiment is not sufficient to take a concrete decision regarding the possible causes of late propagation. However, we selected our subject systems focusing on the diversity of sizes (from very small to large) and application domains (six different application domains) to generalize our findings. Thus, we believe that our findings are important and have the potential to minimize late propagation in clones.

8 Conclusion

In this paper, we investigate late propagation in three types of clones (Type 1, Type 2, and Type 3) separately. Through our experiment we tried to answer three important research questions (mentioned in the Introduction) regarding the intensity, possible causes, and minimization of late propagation. According to our study on six subject systems written in three different programming languages, Type 3 clones experienced the highest intensity of late propagation among the

[6] Simian. http://www.harukizaemon.com/simian/index.html.

three types of clones. The clone fragments that experienced late propagation are block clones (not method clones) for most of the cases. Thus, refactoring of block clones can considerably minimize late propagation. Our findings also suggest that - if not refactorable, then we can build a database with proper tool support for those clone fragments that often require to be changed together consistently so that while changing a particular clone fragment a programmer can look at the other clone fragments that might need to be changed correspondingly. As a future work, we plan to investigate whether programming languages as well as application domains of the subject systems can bias the intensity of late propagation. Also, considering Type 3 clones, we plan to investigate different late propagation patterns, their frequencies and effects on software maintenance.

Bibliography

[ACD07] L. Aversano, L. Cerulo, M. Di Penta. How Clones Are Maintained: An Empirical Study. In *CSMR*. Pp. 81–90. IEEE Computer Society, 2007.

[BKZ13] L. Barbour, F. Khomh, Y. Zou. An empirical study of faults in late propagation clone genealogies. *Software: Evolution and Process* 25:1139 – 1165, 2013.

[CR11] J. R. Cordy, C. K. Roy. The NiCad Clone Detector. In *Tool Demo Track, ICPC*. Pp. 219 – 220. 2011.

[DLL09] M. D'Ambros, M. Lanza, M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering* 35:720 – 735, 2009.

[GH11] N. Göde, J. Harder. Clone Stability. In *CSMR*. Pp. 65–74. 2011.

[GK11] N. Göde, R. Koschke. Frequency and risks of changes to clones. In *ICSE*. Pp. 311 – 320. 2011.

[KG08] C. Kapser, M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering* 13:645 – 692, 2008.

[Kri07] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE*. Pp. 170 – 178. 2007.

[Kri08] J. Krinke. Is cloned code more stable than non-cloned code? In *SCAM*. Pp. 57 – 66. 2008.

[KSNM05] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy. An empirical study of code clone genealogies. In *ESEC-FSE*. Pp. 187 – 196. 2005.

[LW08] A. Lozano, M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM*. Pp. 227 – 236. 2008.

[LW10] A. Lozano, M. Wermelinger. Tracking clones' imprint. In *IWSC*. Pp. 65 – 72. 2010.

[MRR+12] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, K. A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *ACM SAC*. Pp. 1227–1234. ACM, 2012.

[MRS12a] M. Mondal, C. K. Roy, K. A. Schneider. Connectivity of co-changed method groups: a case study on open source systems. In *CASCON*. Pp. 205 – 219. 2012.

[MRS12b] M. Mondal, C. K. Roy, K. A. Schneider. Dispersion of changes in cloned and non-cloned code. In *IWSC*. Pp. 29 – 35. 2012.

[MRS12c] M. Mondal, C. K. Roy, K. A. Schneider. An Empirical Study on Clone Stability. *ACM SIGAPP Applied Computing Review* 12(3):20–36, 2012.

[MRS13] M. Mondal, C. K. Roy, K. A. Schneider. An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study. *Science of Computer Programming*, 2013.
doi:10.1016/j.scico.2013.11.027

[RC08] C. K. Roy, J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *ICPC*. Pp. 172–181. IEEE Computer Society, 2008.

[RC09] C. K. Roy, J. R. Cordy. A mutation / injection-based automatic framework for evaluating code clone detection tools. In *Mutation*. Pp. 157–166. 2009.

[RCK09] C. K. Roy, J. R. Cordy, R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74:470 – 495, 2009.

[Roy09] C. K. Roy. Detection and analysis of near-miss software clones. In *ICSM*. Pp. 447–450. 2009.

[SRSP13] R. K. Saha, C. K. Roy, K. A. Schneider, D. E. Perry. Understanding the Evolution of Type-3 Clones: An Exploratory Study. In *MSR*. Pp. 139 – 148. 2013.

[TCAP09] S. Thummalapenta, L. Cerulo, L. Aversano, M. D. Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 15:1 – 34, 2009.

[VPV10] A. Vanya, R. Premraj, H. V. Vliet. Interactive Exploration of Co-evolving Software Entities. In *CSMR*. Pp. 260 – 263. 2010.

[ZR13] M. F. Zibran, C. K. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring. *IET Software* 7:167 – 186, 2013.