An Empirical Study on Bug Propagation through Code Cloning

Manishankar Mondal   Banani Roy   Chanchal K. Roy   Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{mshankar.mondal, banani.roy, chanchal.roy, kevin.schneider}@usask.ca

Abstract

Code clones are identical or nearly similar code fragments in a codebase. According to the existing studies, code clones are directly related to bugs. Code cloning, creating code clones, is suspected to propagate temporarily hidden bugs from one code fragment to another. However, there is no study on the intensity of bug-propagation through code cloning. In this paper, we define two clone evolutionary patterns that reasonably indicate bug propagation through code cloning. By analyzing software evolution history, we identify those code clones that evolved following the bug propagation patterns. According to our study on thousands of commits of seven subject systems, overall 18.42% of the clone fragments that experience bug-fixes contain propagated bugs. Type-3 clones are primarily involved with bug-propagation. Bug propagation is more likely to occur in the clone fragments that are created in the same commit rather than in different commits. Moreover, code clones residing in the same file have a higher possibility of containing propagated bugs compared to those residing in different files. Severe bugs can sometimes get propagated through code cloning. Automatic support for immediately identifying occurrences of bug-propagation can be beneficial for software maintenance. Our findings are important for prioritizing code clones for management.

Keywords:
Code Clones, Clone-Types, Bug Propagation, Software Maintenance

1. Introduction

Code cloning (i.e., copy/pasting) is a common yet controversial software engineering practice which is often employed by the programmers during development and maintenance for repeating common functionalities. Such a practice results the existence of identical or nearly similar code fragments, also known as code clones, in a code-base. Two code fragments that are similar to each other form a *clone-pair*. A group of similar code fragments forms a clone group or a *clone class*.

Code clones are of great importance from software maintenance perspectives. A great many studies [1, 2, 7, 15, 16, 18, 25, 33, 46, 13] have investigated the impacts of code clones on software evolution. While a number of studies [1, 8, 16, 18, 19] have identified some positive impacts of code clones, other studies [2, 15, 25, 33, 22, 13] have shown strong empirical evidence of negative impacts too. Code clones can be directly related to bugs and inconsistencies in the code-base [22, 23, 44]. It is commonly suspected that bugs can be propagated through code cloning. If a particular code fragment in a code-base contains a temporarily hidden bug, and a programmer copies that fragment to several other places being unaware of the presence of the bug, the bug in the original fragment gets propagated. Fig. 1 shows a possible way of bug-propagation through code cloning.

In Fig. 1, we can see the evolution of two clone fragments $CF1$ and $CF2$. As indicated in the figure, $CF1$ was created in commit operation C_i, and was changed in C_{i+1}. $CF2$ was created in commit C_{i+2} from $CF1$, and the two fragments, $CF1$ and $CF2$, made a clone-pair. In commit C_{i+4}, both of the clone fragments experienced a bug-fix change. Let us assume that the fragments were changed in the same way for bug-fixing. Thus, after experiencing bug-fixing changes, $CF1$ and $CF2$ remained as a clone-pair. From such a phenomenon we realize that $CF1$ contained a bug before $CF2$ was created from it. The bug might be introduced to $CF1$ at the time of its creation (i.e. in commit C_i) or in commit C_{i+1} (i.e., when $CF1$ was changed). However, the bug was not discovered just after being introduced. When $CF2$ was created from $CF1$ (in commit C_{i+2}), the bug in $CF1$ was propagated to $CF2$. Finally, in commit C_{i+4} the bug was fixed by making similar changes to $CF1$ and $CF2$.

Researchers suspect that code cloning can be responsible for propagating bugs. However, there is no study on how frequently bug-propagation occurs during code cloning. Without studying the intensity of bug-propagation we

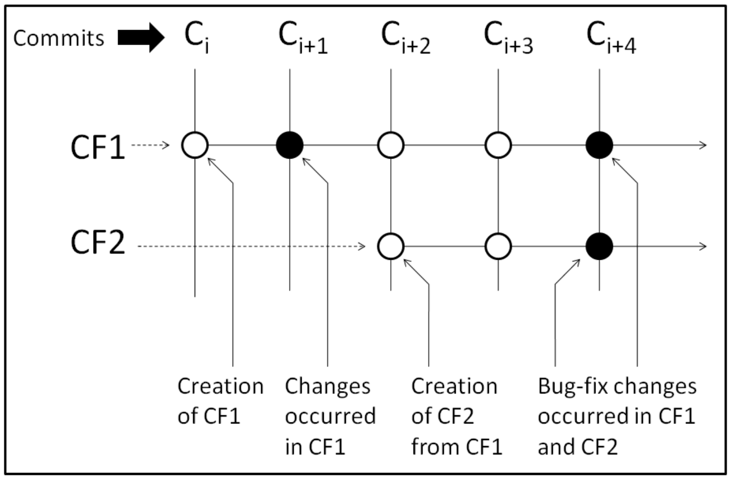


Figure 1: A possible way of bug-propagation through code cloning

cannot properly understand the impacts of code clones on software evolution and maintenance. Focusing on this we perform an empirical study on bug-propagation in code clones. To the best of our knowledge, our study is the first one to investigate bug-propagation through code cloning. We have the following contributions:

- We define two patterns of bug propagation through code cloning.
- We propose an automatic mechanism of detecting these two bug propagation patterns. Our proposed mechanism works in two steps: (i) detecting bug-fix changes in code clones, and (ii) determining whether the bug-fix changes occurred for fixing propagated bugs by analyzing the evolution histories of the code fragments that experienced the bug-fix changes.
- We implement our bug propagation detection mechanism, apply it to seven open-source subject systems written in Java, C and C# and investigate bug propagation through code cloning in these systems.

We answer seven important research questions (Table 1) regarding bug-propagation in code clones. According to our investigation on thousands of revisions of seven subject systems:

- A considerable proportion of the code clones in a subject system contain propagated bugs. According to our observation, overall 18.42% of the code clones that experience bug-fixes are involved with bug propagation.

- Near-miss clones (Type 2 and Type 3 clones) exhibit a higher intensity of bug-propagation compared to identical clones (Type 1 clones). Thus, near-miss clones should be given a higher priority for management from the perspective of bug-propagation.

- Overall around 16.74% of the bug-fix changes that are experienced by code clones can occur for fixing propagated bugs.

- According to our manual analysis, the clone fragments that are involved with bug propagation are mostly method clones. Moreover, bug propagation primarily occurs to the clone fragments that are created together in the same revision (i.e., in the same commit operation). Thus, we suggest programmers to prioritize refactoring method clones that were created in the same revision.

- Near-miss clones residing in the same file have a higher possibility of containing propagated bugs compared to the ones residing in different files.

- According to our bug severity analysis, severe bugs can sometimes get propagated through code cloning.

We believe that bug-propagation should be taken into proper consideration when making clone management (such as clone refactoring or tracking) decisions. Tool support for automatic detection of code clones having possibilities of containing propagated bugs can help us get rid of propagated bugs earlier in software evolution. Our prototype tool that we have implemented for our study can be used for identifying code clones that are likely to contain propagated bugs. Thus, it can help programmers in managing code clones from the perspective of bug-propagation.

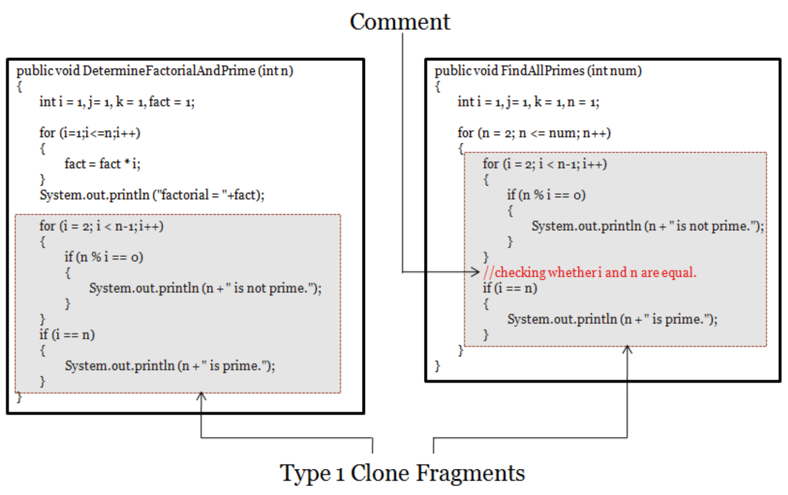
The rest of the paper is organized as follows. Section 2 contains the terminology, Section 3 discusses the experimental steps, Section 4 defines the bug propagation pattern in code clones, Section 5 answers our research questions by presenting and analyzing our experimental results, Section 7

Table 1: Research Questions

SL	Research Question
RQ 1	What percentage of code clones in different clone-types can be involved with bug propagation?
RQ 2	Does clone-size have an effect on the bug-propagation possibility of code clones?
RQ 3	What percentage of the bugs that are experienced by different clone-types can be propagated bugs?
RQ 4	Which pattern of bug-propagation is more intense during evolution?
RQ 5	How often does a propagated bug residing in two clone fragments of a clone-pair get fixed in two different commit operations?
RQ 6	Does bug propagation occur in the same file or across different files?
RQ 7	Do severe bugs get propagated through code cloning?

discusses the related work, Section 8 mentions possible threats to validity, and Section 9 concludes the paper by mentioning future work.

Our study presented in this paper is a significant extension of our earlier work [12] on bug-propagation through code cloning. In our previous study [12], we investigated only four subject systems written in Java and answered three research questions. However, in this extended study, we investigate seven subject systems in total written in Java, C, and C#. We also answer four additional research questions (seven questions in total) in this study. We would like to add that in our previous research [12], we performed a case sensitive search of the bug-fix keywords in the commit messages for identifying bug-fix commits. However, there is a possibility of missing some bug-fix commits with case sensitive search. For this reason, in our extended research we perform a case insensitive search. As a result, the data presented in different tables and graphs are refined in this extended research.



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 1, n = 1;

    for (n = 2; n <= num; n++)
    {
        for (i = 2; i < n-1; i++)
        {
            if (n % i == 0)
            {
                System.out.println (n + " is not prime.");
            }
        }
        //checking whether i and n are equal.
        if (i == n)
        {
            System.out.println (n + " is prime.");
        }
    }
}
```

Figure 2: A pair of Type 1 clone fragments

2. Terminology

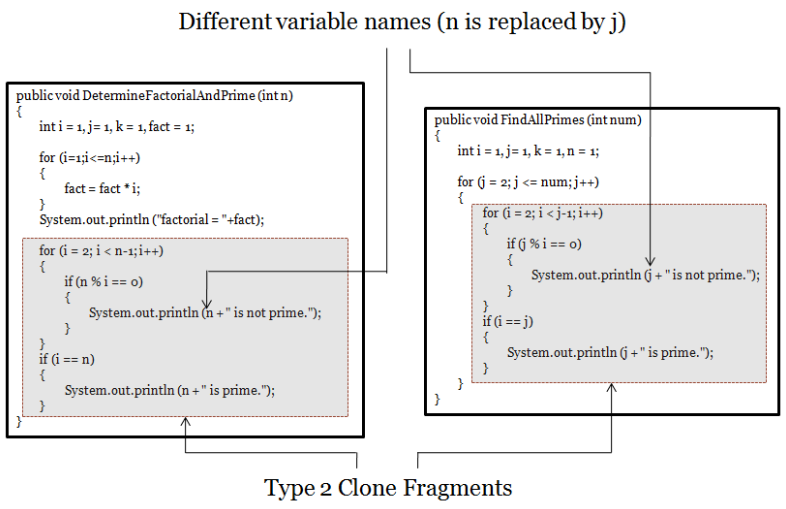
2.1. Types of Clones

We investigate bug propagation considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones) [39, 38].

According to the literature, if two or more code fragments in a particular code-base are exactly the same disregarding the comments and indentations, these code fragments are called **exact clones** or **Type 1 clones** of one another. Fig. 2 shows two methods named 'DetermineFactorialAndPrime(int n)' and 'FindallPrimes(int num)'. The two highlighted portions in these two methods disregarding the comment at the right-hand-side method are Type 1 clones of each other.

Type 2 clones are syntactically similar code fragments in a code-base. In general, **Type 2 clones** are created from Type 1 clones because of renaming identifiers and/or changing data types. The two highlighted portions in the two methods in Fig. 3 are Type 2 clones of each other. We see that the variable, n, at the left side clone fragment has been renamed as j at the right side fragment.

Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones. **Type 3 clones** are also known as *gapped clones*. Fig. 4 shows two methods. The two highlighted



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 1, n = 1;

    for (j = 2; j <= num; j++)
    {
        for (i = 2; i < j-1; i++)
        {
            if (j % i == 0)
            {
                System.out.println (j + " is not prime.");
            }
        }
        if (i == j)
        {
            System.out.println (j + " is prime.");
        }
    }
}
```

Figure 3: A pair of Type 2 clone fragments

portions in these methods are Type 3 clones of each other. We see that the clone fragment residing in the right side method contains a line, 'k = k + 1;', which is not present in the clone fragment at the left side method.

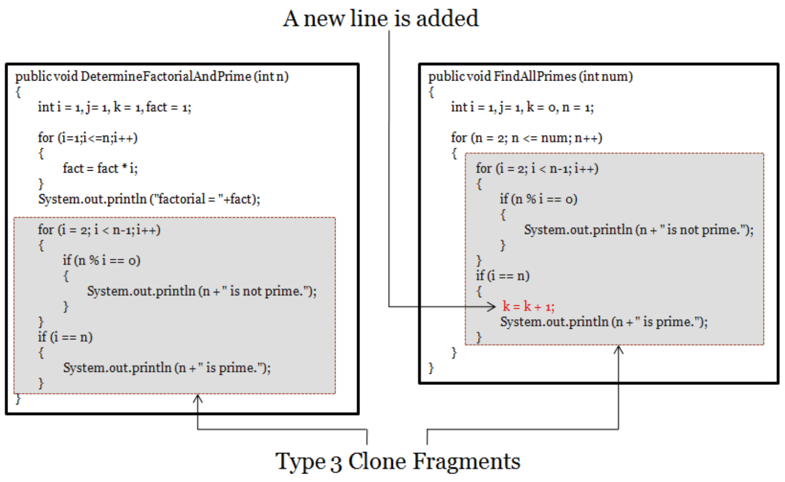
Clone-pair: If two code fragments in a particular code-base are similar to each other according to the above definitions of code similarity, we call them a clone-pair. A clone-pair can be of Type 1, Type 2, or Type 3.

2.2. Clone Fragment

We frequently use the term 'clone fragment' in our paper. A clone fragment is a particular code fragment which is exactly or nearly similar to one or more other code fragments in a code-base. Each member in a clone group or a clone-pair is a clone fragment.

2.3. Clone Genealogy

We detect clone genealogies for the purpose of our investigations. We define a clone genealogy in the following way. Let us assume that a clone fragment was created in a particular revision and was alive in a number of consecutive revisions. Thus, each of these revisions contains a snapshot of the clone fragment. The genealogy of this clone fragment consists of the set of its consecutive snapshots from the consecutive revisions where it was alive. Each clone fragment in a particular revision belongs to a particular



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 0, n = 1;

    for (n = 2; n <= num; n++)
    {
        for (i = 2; i < n-1; i++)
        {
            if (n % i == 0)
            {
                System.out.println (n + " is not prime.");
            }
        }
        if (i == n)
        {
            k = k + 1;
            System.out.println (n + " is prime.");
        }
    }
}
```

Figure 4: A pair of Type 3 clone fragments

clone genealogy. In other words, a particular clone fragment in a particular revision is actually a snapshot in a particular clone genealogy. By examining the genealogy of a clone fragment we can determine how it changed during evolution.

We automatically detect clone genealogies using the SPCP-Miner [27] tool. In our research, by examining the genealogy of a clone fragment we determine which commit operation(s) made changes to it.

2.4. Bug Propagation through Code Cloning

Existing studies [22, 23, 44] reveal that code clones can be related to bugs in software systems. It is also suspected that bugs in a code-base can get propagated through code cloning (copy/pasting). If a particular code fragment contains a bug which has not been discovered yet, then creating copies of that code fragment (i.e., cloning that code fragment) actually propagates the bug in all the created copies. If this bug gets discovered at a particular point of evolution, it should be fixed in each of the clone fragments that contains it. Thus, code cloning can increase bug-fixing effort during software evolution. Bug propagation tendencies of code clones should be considered when prioritizing them for management. Code clones with higher tendencies should be given higher priorities. Section 4 discusses the details on bug propagation patterns. In our definition of bug propagation patterns, we use

Table 2: Subject Systems

Systems	**Lang.**	**Domains**	**LLR**	**Revs**
Carol	Java	Game	25,091	1700
Freecol	Java	Game	91,626	1950
jEdit	Java	Text Editor	191,804	4000
Jabref	Java	Reference Management	45,515	1545
GLGraphics	Java	Library for processing programming languages	15,509	464
Ctags	C	Code definition generator	33,270	774
MonoOSC	C#	Formats & Protocols	14,883	355

LLR = LOC in the Last Revision Revs = No. of Revisions

the term *Similarity Preserving Co-change*. We discuss similarity preserving co-change in the following subsection.

2.5. Similarity Preserving Co-change of Clone Fragments

Mondal et al. [31] introduced the term *similarity preserving co-change*. We describe this in the following way. Let us consider two code fragments, CF1 and CF2, which are clones of each other in revision r of a subject system. A commit operation was applied on revision r and both of these two fragments were changed (i.e., the clone fragments co-changed) in such a way that they were again considered as clones of each other in the next revision $r+1$ (i.e., created because of the commit). In other words, the clone fragments preserved their similarity even after experiencing changes in the commit operation. Thus, this co-change of clone fragments (i.e., change of more than one clone fragment together) is a Similarity Preserving Co-change (SPCO).

In a similarity preserving co-change (SPCO), two or more clone fragments from the same clone class are changed together consistently (i.e., the clone fragments are changed in the same way) [31].

3. Experimental Steps

We perform our investigation on seven subject systems written in Java, C, and C# as listed in Table 2. We download these systems from an online SVN repository [34]. We select our subject systems emphasizing their diversity in sizes, implementation languages, and revision history lengths. We also see that the systems are of different application domains.

3.1. Experimental Steps

We perform the following experimental steps before analyzing bug propagation in code clones: **(1)** Extraction of all revisions (as mentioned in Table 2) of each of the subject systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [6]; **(3)** Detection and extraction of code clones from each revision by applying the NiCad [5] clone detector; **(4)** Detection of changes between every two consecutive revisions using *diff*; **(5)** Locating these changes to the already detected methods as well as clones of the corresponding revisions; **(6)** Locating the code clones detected from each revision to the methods of that revision; **(7)** Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [25]; **(8)** Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy; **(9)** Detecting clone fragments that experienced bug-fix changes; and **(10)** Analyzing the evolution of bug-fix clones to identify bug propagation. For completing the first eight steps we use the tool SPCP-Miner [27]. In Section 3.2 we will discuss the technique that we apply for detecting bug-fix changes in code clones. We will define possible bug propagation patterns, and discuss how we detect such patterns in Section 4.

Clone Detection. We use the well-known NiCad [5] clone detector for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [41, 42]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold and blind renaming of identifiers. For different settings of a clone detector the clone detection results can be different and thus, the experimental findings can also be different. For this reason selection of reasonable settings (i.e., detection parameters) is important. We used the mentioned settings in our research, because in a recent study [45] Svajlenko and Roy show that these settings provide us with better clone detection results in terms of both precision and recall. We should note that before using the NiCad outputs for Type 2 and Type 3 cases, we pre-processed them in the following way.

(1) Every Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.

(2) Every Type 3 clone class that exactly matched any Type 1 or Type 2 class was excluded from Type 3 outputs.

We performed these preprocessing steps because we wanted to investigate bug propagation in each of the three types of clones separately. When we detect Type 2 clones using NiCad, the clone results include Type 1 clone classes. As we wanted to consider Type 1 clone results separately, this is important to exclude Type 1 clone classes from Type 2 clone results. If we do not perform this exclusion, some Type 1 clone classes will be investigated twice and this is not expected. Without this exclusion, we cannot make a fair comparison between clone types. Finally, after filtering out Type 1 clone classes from Type 2 results, we consider purely Type 2 clones in our research. In the same way, we identify purely Type 3 clones by excluding Type 1 and Type 2 clone classes from Type 3 clone results.

Clone Genealogies of Different Clone-Types. As we wanted to investigate bug-propagation in three clone-types (Type 1, Type 2, and Type 3) individually, we detect clone genealogies for each of these three clone-types separately. We perform the following steps for clone genealogy detection.

- **Step 1:** In this step, we detect code clones of three clone-types separately using the NiCad clone detector as we have just described. Thus, for each clone-type, we have a separate set of clone detection result.

- **Step 2:** In this second step, we apply the SPCP-Miner tool [27] on the clone detection result of each clone-type separately. SPCP-Miner detects clone genealogies from the clone detection result obtained from all the revisions of a subject system. After completing this step, we get three separate sets of clone genealogies for three clone-types.

As we obtain a separate set of clone genealogies for each clone-type, we can easily investigate bug-propagation in each clone-type separately and can make a comparison of the bug-propagation tendencies of the clone-types.

Tackling Clone-Mutations. Xie et al.[48] found that mutations of the clone fragments (i.e., a particular clone fragment may change its type) might occur during evolution. If a particular clone fragment is considered of a different clone-type during different periods of evolution, SPCP-Miner extracts a separate clone-genealogy for this fragment for each of these periods. Thus, even with the occurrences of clone-mutations, we can clearly distinguish which bugs were experienced by which clone-types.

Exclusion of Auto-Generated Code and Test Code. Before conducting the experiments, we excluded the auto-generated (i.e., system generated) code from our consideration, because the target of our research is the code which is written by programmer(s). In our research, we have investigated one C system, one C# system, and five Java systems. The subject system, Ctags, which is written in C does not contain any auto-generated code. We manually checked the files to ensure this. For the Java systems, we automatically searched the entire source code to identify if there is any auto-generated code. In Java source files, auto-generated code generally stay within the tags <editor-fold defaultstate="collapsed" desc="Generated Code"> and </editor-fold>. Moreover, the auto-generated Java code often contains a method named 'initComponents'. Thus, we searched the entire source code to find the key-words, 'Generated Code' and 'initComponents'. However, we did not get these key-words in any source code files of our Java systems. Thus, our investigation does not involve auto-generated code in Java. The auto-generated C# files contain the tags: <auto-generated> and </auto-generated>. We automatically searched the code-base of our C# system to find if there is any auto-generated source file. We obtained four files with the mentioned tags. We excluded these files from our investigation.

We found that our subject systems Carol and GLGraphics contain some test code in the folders named as 'test'. We excluded code from these folders during our investigation. We did not find any test code in our other subject systems. The source code of the last revision of each of our subject systems is available on-line [50].

3.2. Bug Detection in Code Clones

For a particular candidate system, we first retrieve the commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically examine the commit messages using the heuristic proposed by Mockus and Votta [26] to identify those commits that occurred for the purpose of fixing bugs. Then we identify which of these bug-fix commits make changes to clone fragments. If one or more clone fragments are modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) was necessary for fixing the corresponding bug. In other words, the clone fragment(s) are related to the bug. In this way we examine the commit operations of a candidate system, analyze the commit messages to retrieve the bug-fix commits, and identify those bug-fix commits that affected code clones.

The way we detect the bug-fix commits was also previously followed by Barbour et al. [2]. They detected bug-fix commits in order to investigate whether late propagation in clones is related to bugs. They at first identified the occurrences of late propagations and then analyzed whether the clone fragments that experienced late propagations are related to bug-fix. In our study we detect bug-fix commits in the same way as they detected, however, our study is not limited to late propagation clones only. We perform our study considering all clone fragments of a software system. Barbour et al. did not investigate bug-propagation. We investigate bug-propagation through code cloning in our study. Also, they did not consider Type 3 clones in their study. We consider Type 3 clones in our bug-propagation study. After detecting bug-fix changes in code clones, we automatically detect whether the bug-fix clones evolved following a bug-propagation pattern.

Investigating the availability of commit messages in the commit-log. As we detect bug-fix commits by analyzing the commit messages, it is important to investigate whether commit operations are often associated with commit messages or not. We automatically check each of the commit operations recorded in the commit log and determine whether these commits are associated with commit messages. Fig. 5 shows the percentages of commit operations with or without commit messages. We see that the percentage of commit operations with commit messages is almost 100% for most of our subject systems except MonoOSC. For MonoOSC, this percentage is 57.18% which is greater than the percentage of commits without any message. Considering all the commit operations of all the subject systems we realize that only 2.66% of the commit operations were performed without any commit messages. Thus, commit operations are mostly performed with commit messages. Such a finding makes us realize that detecting bug-fix commits by analyzing the commit messages is reasonable.

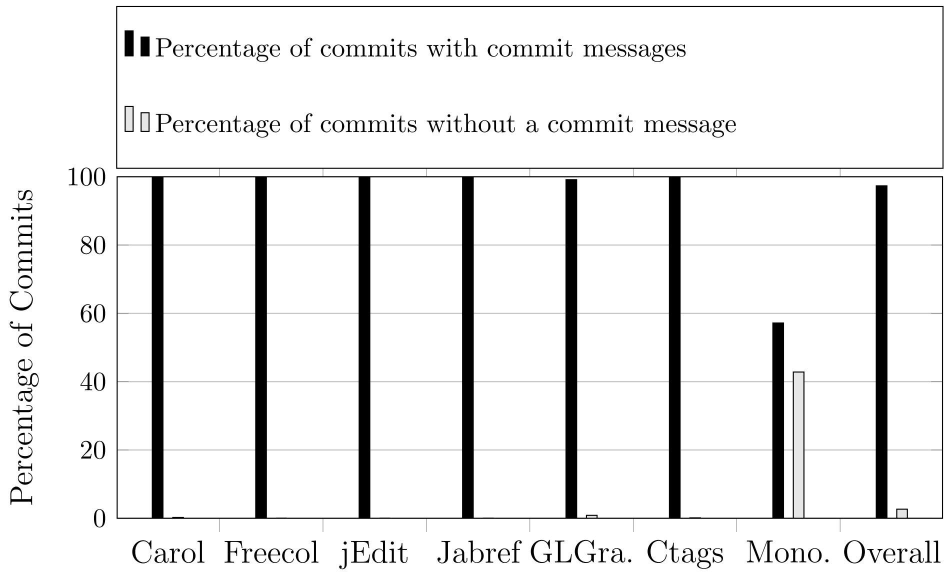


Figure 5: Percentage of commits with or without commit messages

4. Bug Propagation Patterns

In the following two subsections we first provide formal definitions of two bug-propagation patterns, and then describe an automatic procedure that we have used for detecting these patterns. We should note that the two patterns we are going to describe include all possible ways of bug propagation through code cloning where the propagated bug was fixed in at least two clone fragments from the same clone class. Intuitively, a propagated bug should be fixed in the clone fragment that primarily contained the bug, and also, in the other clone fragments where the bug was propagated.

4.1. The First Bug Propagation Pattern

Propagation Pattern. Let us consider that two code fragments were created in a particular revision. These code fragments are similar to each other, and thus form a clone-pair. We also assume that a similar code fragment was not preexisting. As these two fragments were created in the same revision (i.e., in the same commit operation) it is likely that one fragment was first created by the programmer, and then she created the second one from the first one (possibly by copy/pasting). In this case, any bug that was introduced in the first fragment during its creation can be propagated to the second one. Considering such a way of bug-propagation we define the following bug-propagation pattern.

Pattern Definition. Let us consider that a clone-pair consisting of two clone fragments, CF1 and CF2, resides in revision r of a subject system. For fixing a bug, these clone fragments were changed together (i.e., were co-changed) in the commit operation c which was applied on revision r. We consider that this bug-fix change experienced by the two clone fragments in commit c is the fixing of a propagated bug if the following conditions hold:

- **Condition 1:** The two clone fragments, CF1 and CF2, were created in the same revision $r_{created}$. Here, $r_{created} < r$ (i.e., $r_{created}$ is older than r). No other similar code fragment was preexisting. In other words, a code fragment which is similar to CF1 and CF2 was not existing in revision $r_{created} - 1$. Here, $r_{created} - 1$ is the revision which was created just before the revision $r_{created}$.

- **Condition 2:** None or only one of the two clone fragments was changed during the evolution from $r_{created}$ to r. In other words, none or only one of the two fragments was changed in any of the commits that were applied on the revisions $r_{created}$ to $r-1$. Here, $r-1$ is the revision that was created just before revision r.

- **Condition 3:** The two clone fragments, CF1 and CF2, experienced a similarity preserving co-change in the bug-fix commit operation c which was applied on revision r. We have defined *similarity preserving co-change* in Section 2. In a similarity preserving co-change, the clone fragments are updated consistently (i.e., the fragments are changed in the same way).

The first condition (*Condition 1*) implies the likeliness that one of the two fragments was created from the other one (possibly by copy/pasting). *Condition 2* confirms that after being created, at least one of the two fragments remained unchanged before they were both changed in the bug-fix commit c. *Condition 3* implies that each of the clone fragments was changed in the same way for fixing the bug, and thus, each fragment contained the same bug.

By analyzing the second condition (*Condition 2*) we realize that if none of the clone fragments got changed during the intermediate evolution (i.e., in the commits that were applied on the revisions $r_{created}$ to $r - 1$), then

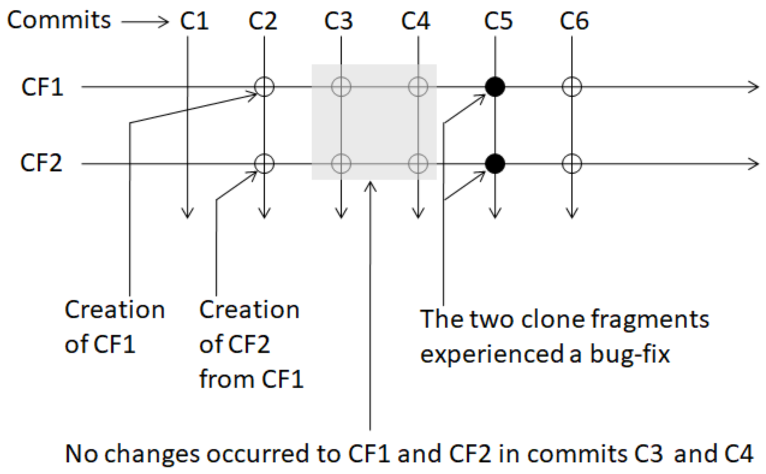

Commits
C1
C2
C3
C4
C5
C6
CF1
CF2
Creation of CF1
Creation of CF2 from CF1
The two clone fragments experienced a bug-fix
No changes occurred to CF1 and CF2 in commits C3 and C4

Figure 6: An example of the first bug-propagation pattern

the bug that was fixed in the two fragments in the commit operation c (the commit c was applied on revision r) was certainly introduced to them at the time of their creation (i.e., in the revision $r_{created}$). However, we see that *Condition 2* allows only one of the fragments to be changed in the intermediate evolution. Even with such a flexibility in the condition it is still confirmed that the bug that was fixed by consistently changing CF1 and CF2 in commit c was not introduced by any change during the intermediate evolution. Let us assume that the fragment CF2 was changed in a commit in the intermediate evolution. However, *Condition 3* confirms that both of the fragments were changed in the same way for bug-fixing in commit c. In other words, each of the changed fragment (i.e., CF2) and unchanged fragment (i.e., CF1) contained the same bug. Reasonably, any change in the intermediate evolution did not introduce the bug that was fixed in commit c, because CF1 also experienced the same fix even after remaining unchanged in the intermediate evolution. The bug was introduced (in one fragment) and propagated (to the other fragment) at the time of creation of the two fragments.

Fig. 6 graphically shows an example of the first bug-propagation pattern. We see the evolution of two clone fragments CF1 and CF2 through a number of commit operations. The clone fragment CF1 was created in the commit operation C2. The fragment CF2 was also created in the same commit operation from CF1. They experienced the same bug-fix in commit C5. Before experiencing bug-fix they did not experience any other changes. Thus, the example in Fig. 6 complies with our definition of the first bug-propagation pattern.

4.2. The Second Bug Propagation Pattern

Propagation Pattern. Let us assume that a code fragment CF2 was created in a particular revision from a similar preexisting code fragment CF1 (possibly by copy/pasting). Consequently, these two clone fragments made a clone-pair. In such a phenomenon it is possible that an unreported bug (i.e., a bug which has not yet been discovered) which was preexisting in the fragment CF1 will be transferred (i.e., propagated) to CF2. Considering this way of bug propagation we define the following bug propagation pattern.

Pattern Definition. Let us consider that two clone fragments, CF1 and CF2, make a clone-pair in revision r of a subject system. These two fragments were created at two different revisions: $r_{created1}$ and $r_{created2}$ respectively. Here, $r_{created1} < r$ and $r_{created2} < r$. In other words, both $r_{created1}$ and $r_{created2}$ are older than r. We also assume that $r_{created1} < r_{created2}$. Thus, CF1 was preexisting (i.e., CF1 is older than CF2). For fixing a bug, these two clone fragments were changed together (i.e., were co-changed) in the commit operation c which was applied on revision r. We consider that this bug-fix change experienced by the two fragments in commit c is the fixing of a propagated bug if the following three conditions hold:

- **Condition 1:** Just after the creation of the younger fragment (i.e., CF2), it was considered similar to the older one (i.e., CF1). Thus, CF1 and CF2 made a clone-pair from revision $r_{created2}$. We should note that $r_{created2}$ is older than r.

- **Condition 2:** None or only one of the two clone fragments was changed during the evolution period between the revisions $r_{created2}$ and r. In other words, none or only one of the fragments was changed in the commit operations applied on the revisions $r_{created2}$ to $r - 1$. We have already mentioned that the revision r experienced the commit operation c which made changes to CF1 and CF2 for fixing a bug.

- **Condition 3:** The co-change of the two clone fragments (i.e., CF1 and CF2) in the commit operation c is a *similarity preserving co-change.*

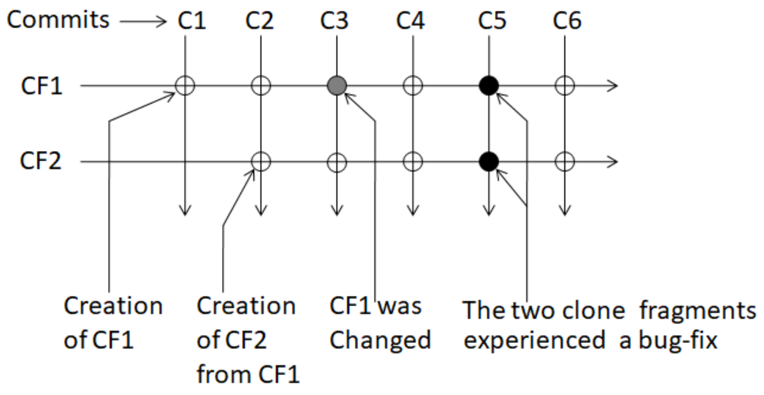


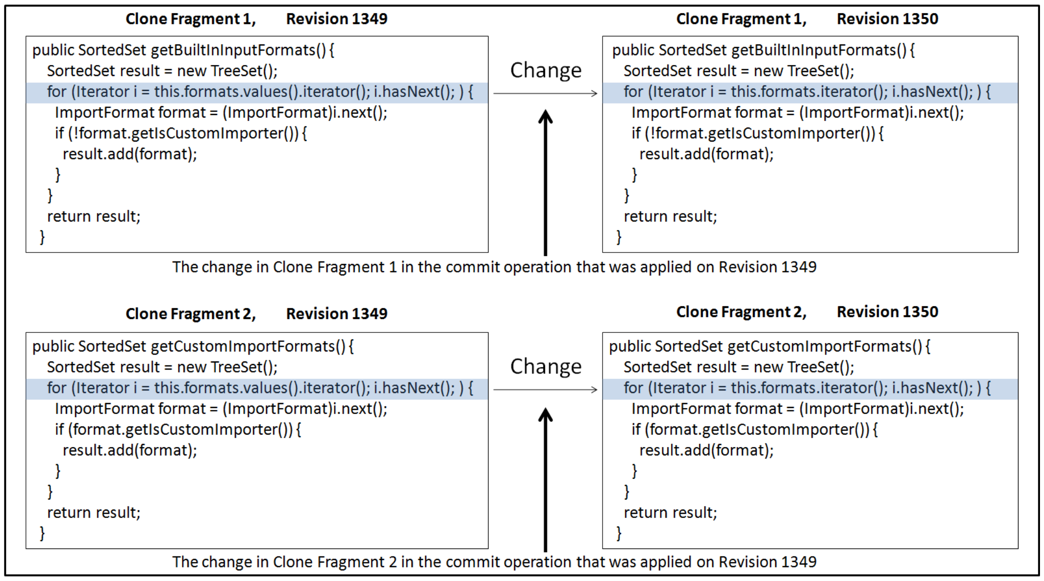
Figure 7: An example of the second bug-propagation pattern

As the clone fragments (CF1 and CF2) were created at two different commits, it implies the possibility that the younger code fragment CF2 might be created from the similar preexisting code fragment CF1. *Condition 1* implies that the bug that was fixed in commit *c* was introduced to CF2 at the time of its creation. *Condition 2* and *Condition 3* are similar to the corresponding conditions in the first pattern described in Section 4.1. This section (i.e., Section 4.1) also contains the discussions of these two conditions. Finally, the above three conditions reasonably imply the fixing of a bug that was preexisting in the older fragment CF1 and was propagated to the fragment CF2 through code cloning.

Fig. 7 shows an example of the second bug-propagation pattern. The clone fragment CF1 was created in the commit operation C1. The clone fragment CF2 was created at a later commit operation, C2, from CF1. Both fragments experienced the same bug-fix at Commit C5. We also notice that only the clone fragment CF1 experienced a change in the commit operation C3 before experiencing the bug-fix. Thus, Fig. 7 complies with our definition of the second bug-propagation pattern.

4.3. Automatic Detection of Bug Propagation Patterns

By following the procedure described in Section 3.2 we detect the bug-fix commits that affected code clones. Let us consider such a bug-fix commit which we call BFC. Let us further assume that a clone-pair was changed


Clone Fragment 1, Revision 1349
Clone Fragment 1, Revision 1350
Change
The change in Clone Fragment 1 in the commit operation that was applied on Revision 1349
Clone Fragment 2, Revision 1349
Clone Fragment 2, Revision 1350
Change
The change in Clone Fragment 2 in the commit operation that was applied on Revision 1349

```
public SortedSet getBuiltInInputFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.values().iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (!format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

```
public SortedSet getBuiltInInputFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (!format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

```
public SortedSet getCustomImportFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.values().iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

```
public SortedSet getCustomImportFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

Figure 8: The figure shows a similarity preserving co-change of two clone fragments (Clone Fragment 1, and Clone Fragment 2) in the commit operation which was applied on revision 1349 of our subject system Jabref. The commit operation that was applied on revision 1349 is a bug-fix commit. The commit message says ‘JabRef 2.0: fixed some Bugs’. We provide the snapshots of the two clone fragments in two revisions, 1349 and 1350, and highlight the differences between the snapshots. From the figure it is clear that the two clone fragments contained the same bug, and the fragments were changed in the same way for fixing the bug. This bug is a propagated bug, because the clone fragments evolved by following the first bug-propagation pattern defined in Section 4. We see that the difference between the two clone fragments lies in the condition part of the *if-statement*. Clone Fragment 1 contains a NOT operator (!) which is absent in Clone Fragment 2.

(i.e., both of the clone fragments in the pair were changed) in this commit for fixing a bug. We first determine whether the clone-pair experienced a similarity preserving co-change in the bug-fix commit BFC (i.e., we check **Condition 3** in each of the two patterns we have just described). If it did, then we analyze the genealogies of the two clone fragments in the pair. We previously mentioned that we use the tool SPCP-Miner [27] for detecting clone genealogies. By automatically examining the genealogies of the clone fragments we determine how they evolved in the past (i.e., how they evolved before the occurrence of the bug-fix commit BFC). We check each of the first two conditions of each of the two patterns by analyzing their genealogies. If no clone-pair experienced a similarity preserving co-change in the bug-fix commit BFC, then we consider that the clone related bug that was fixed in BFC is not a propagated bug.

4.4. An Example of Bug Propagation

In Fig. 8 we provide an example of fixing a propagated bug. The figure shows the similarity preserving co-change of two clone fragments (Clone Fragment 1, and Clone Fragment 2) in a bug-fix commit operation which was applied on revision 1349 of our candidate system Jabref. Our implemented prototype tool detects the evolution pattern of these two clone fragments as a bug-propagation pattern of the first category (i.e., defined in Section 4.1). Each of these two clone fragments were created in revision 1344. We see that the clone fragments have almost the same implementation except the condition parts of their *if-statements*. Clone Fragment 1 contains a NOT operator (!) which is not present in Clone Fragment 2. In such a phenomenon it is likely that one of these fragments was created from the other (possibly by copy/pasting), and thus the bug that was introduced in one fragment propagated to the other one. After being created in revision 1344, each of the fragments remained unchanged up to revision 1349. The commit operation which was applied on revision 1349 fixed the bug in these two clone fragments. From Fig. 8 and its caption it is clear that each of the fragments contained the same bug, and the fragments experienced a similarity preserving co-change for the purpose of bug-fixing. The changes that occurred to the clone fragments are also highlighted in the figure.

5. Experimental Results and Analysis

We apply our experimental steps on each of our subject systems and identify bug propagation patterns in each of the three types of code clones (Type 1, Type 2, and Type 3) separately.

Comparing the bug-proneness of clone and non-clone code. Before investigating bug-propagation, it is important to analyze whether bug-proneness of code clones is at all important for investigation while the major portion of a code-base generally contains non-clone code and non-clone code can also experience bug-fixes. We analyze by comparing the bug-proneness of non-clone code and different types of clone code. In particular, we analyze whether code clones are more likely to experience bug-fixes compared to non-clone code. We perform our analysis in the following way.

For a particular subject system, we first identify the bug-fix commits. Let us assume that BFC is such a bug-fix commit which was applied on a particular revision of the subject system. We detect code clones of all three types (Type 1, 2, and 3) from that revision. From the detected clones, we determine which lines in that revision are clone lines. We determine the total number of clone lines in the revision. By excluding the clone lines, we get all the non-clone lines. We also detect the changes that occurred to the code-base of the revision in the bug-fix commit (BFC). If one or more lines belonging to code clones were changed in BFC, we consider this commit as a *clone bug-fix commit*. In a similar way, we determine whether BFC can also be considered as a *non-clone bug-fix commit*. For fixing a bug, both clone and non-clone code might need to be changed. We examine all the bug-fix commits and determine the following measures:

- M1: Total number of bug-fix commits experienced by clone code.
- M2: Total number of bug-fix commits experienced by non-clone code.
- M3: Average number of clone lines in the revisions that experienced the bug-fix commits.
- M4: Average number of non-clone lines in the revisions that experienced the bug-fix commits.

Table 3 shows these measures for each of our subject systems. From these four measures, we determine the bug-proneness of clone and non-clone code using the following equations.

Table 3: Bug-proneness data for non-clone code and different types of clone code

	Type 1 clones		Type 2 clones		Type 3 clones		Type clones	
Systems	BC	ACL	BC	ACL	BC	ACL	BC	ANCL
Carol	11	503.13	9	454.92	31	1633.19	114	30779.74
Freecol	11	594.43	14	429.21	58	2767.33	317	50553.18
jEdit	38	83554.74	10	4415.81	44	31065.74	54	179651.86
Jabref	10	829.29	8	709.77	28	2847.19	202	49115.48
GLGraphics	7	538.80	0	217.03	15	994.28	56	11604.17
Ctags	4	107.45	7	167.67	25	554.30	218	16035.42
MonoOSC	3	445.14	2	219.35	8	1331.14	42	20916.63

BC = Number of bug-fix commits experienced by non-clone code or a particular type of code clones
ACL = Average number of clone lines in the revisions that experienced bug-fix commits
ANCL = Average number of non-clone lines in the revisions that experienced bug-fix commits

$$BCC = (M1 \times 100)/M3 \tag{1}$$

$$BNC = (M2 \times 100)/M4 \tag{2}$$

Here, BCC and BNC are the bug-proneness of clone and non-clone code respectively. We determine the bug-proneness per 100 lines of clone and non-clone code. By looking at the equations we realize that the bug-proneness measures are normalized by code-size. In a previous study [29] on comparing the bug-proneness of clone and non-clone code, the bug-proneness measures were not normalized by code size. Such a normalization is important because the majority of a code-base is generally non-clone code. As the size of non-clone code is generally bigger than that of clone code, it is expected that non-clone code will experience a higher number of bug-fixes during evolution. Without normalization by code size, we cannot make a fair comparison of the bug-proneness of clone and non-clone code. Fig. 9 compares the bug-proneness of non-clone code and different types of clone code. We again would like to note that when measuring the bug-proneness of non-clone code we exclude clone lines of all three types (Type 1, 2, and 3) from our consideration. From the figure we understand that the bug-proneness of each type of code clones is higher than the bug-proneness of non-clone code except for Type 2 clones of GLGraphics. From Table 3 we realize that Type 2 clones of GLGraphics did not experience any bug-fix commit operation during the entire period of evolution. We finally understand that bug-proneness of code clones is generally higher than the bug-proneness of non-clone code. In the following subsections we answer the research questions by presenting and

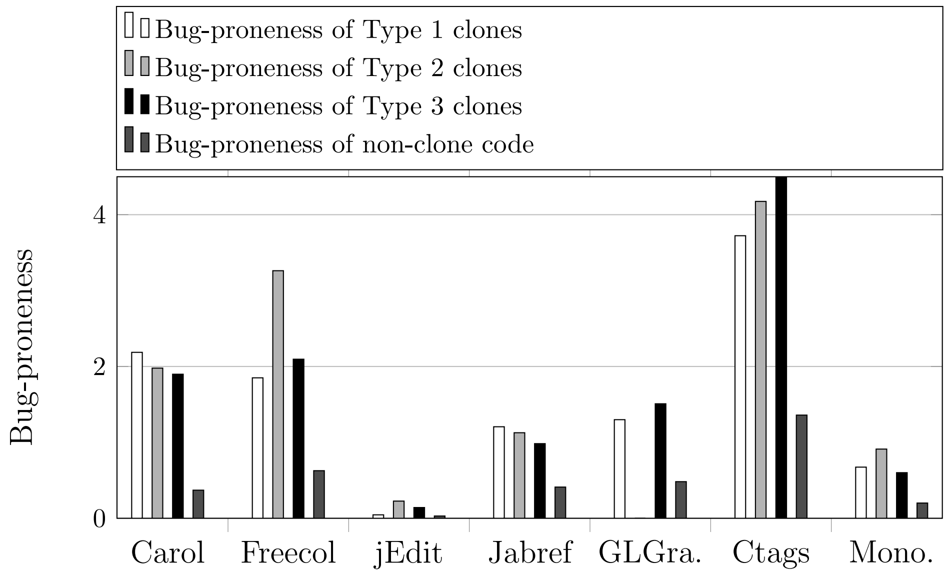


Figure 9: Comparing bug-proneness of non-clone code and different types of clone code

analyzing our experimental results.

5.1. Answering RQ 1

RQ 1: *What percentage of code clones in different clone-types can be involved with bug propagation?*

Rationale. Answering RQ 1 is important. Bug propagation has always been considered a negative impact of code cloning. However, none of the existing studies investigated the intensity of bug propagation in code clones. Without investigating bug propagation intensities in different clone-types, we cannot properly realize the impact of code cloning on software evolution and maintenance. In RQ 1 we determine bug propagation intensity in each of the three major clone-types (Type 1, Type 2, and Type 3), and then make a comparative analysis of the intensities to determine which clone-type exhibits the highest intensity. We perform our investigation in the following way.

Methodology: For a particular subject system we determine all the bug-fix commits by applying the procedure described in Section 3.2. Considering each clone-type we select those bug-fix commits where code clones of that particular type were changed for fixing bugs. For such a bug-fix commit for a particular clone-type, we determine whether a clone-pair has been changed (i.e., whether the two clone fragments in a clone-pair have been changed together or co-changed) in the commit. Considering each of the clone-pairs that have been changed in the bug-fix commit we automatically determine whether the two clone fragments in the pair evolved by following a bug propagation pattern defined in Section 4. In Section 4.3 we described an automatic procedure for detecting a bug propagation pattern. If a clone-pair evolved by following a bug propagation pattern, we call this pair a *bug propagation clone pair*. Considering all the bug-fix commits affecting a particular clone-type we determine all the bug propagation clone pairs. For a particular clone-type of a subject system we determine the following measures, and report these in Table 4.

- **CF (Clone Fragment):** The total number of distinct clone fragments (i.e., of the particular clone-type) that were created during the whole period of evolution of the subject system. This number is actually the total number of clone genealogies during system evolution.
- **BC (Bug-fix Commit):** The total number of bug-fix commits that affected clone fragments of that particular clone-type.
- **BCF (Bug-fix Clone Fragment):** The total number of clone fragments that experienced changes in the bug-fix commits. A particular clone fragment might experience changes in more than one bug-fix commit. We determine the number of distinct clone fragments that experienced bug-fix changes. By analyzing the genealogy of a clone fragment we can identify which bug-fix commits it was changed in.
- **BPCP (Bug Propagation Clone Pair):** The number of distinct bug propagation clone pairs (i.e., the clone pairs that evolved following a bug propagation pattern defined in Section 4).
- **BPCF (Bug Propagation Clone Fragment):** The number of distinct clone fragments involved in the bug propagation clone pairs. We can easily understand that this number is the total number of clone fragments that were involved with bug propagation.

For each clone-type of each of the subject systems, we also determine the following two percentages considering the above measures.

Table 4: Bug propagation in different clone-types

	Type 1					Type 2					Type 3				
Systems	**CF**	**BC**	**BCF**	**BPCP**	**BPCF**	**CF**	**BC**	**BCF**	**BPCP**	**BPCF**	**CF**	**BC**	**BCF**	**BPCP**	**BPCF**
Carol	415	11	41	1	2	211	9	38	3	6	682	31	201	141	50
Freecol	239	11	25	2	4	162	14	13	3	6	752	58	151	29	44
jEdit	7398	38	77	0	0	399	10	20	1	2	2688	44	185	4	8
Jabref	483	10	9	1	2	228	8	14	0	0	1363	28	31	2	4
GLGraphics	77	7	11	3	6	80	0	0	0	0	209	15	33	19	19
Ctags	53	4	4	0	0	89	7	0	0	0	156	25	17	2	4
MonoOSC	153	3	2	1	2	41	2	0	0	0	184	8	9	1	2

CF = Total number of distinct clone fragments (i.e., clone genealogies) of a particular clone-type that were created during the whole period of evolution.
BC = Total number of bug-fix commits that affected clone fragments of a particular clone-type.
BCF = Total number of distinct clone fragments of a particular clone-type that experienced bug-fix commits (i.e., that experienced bug-fixes).
BPCP = Total number of distinct clone-pairs of a particular clone-type that evolved following a bug propagation pattern.
BPCF = Total number of distinct clone fragments of a particular clone-type that were evolved in the bug-propagation clone-pairs.

- **PCFBP (Percentage of Clone Fragments involved with Bug Propagation):** This is the percentage of clone fragments that are involved with bug propagation with respect to all clone fragments in the system. We determine this using the following equation.

  $$PCFBP = \frac{BPCF \times 100}{CF} \tag{3}$$

  The bar graph in Fig. 10 shows this percentage for each clone-type of each of the subject systems.

- **PBCFBP (Percentage of Bug-fix Clone Fragments involved with Bug Propagation):** This is the percentage of clone fragments that are involved with bug propagation with respect to all bug-fix clone fragments. We calculate this in the following way.

  $$PBCFBP = \frac{BPCF \times 100}{BCF} \tag{4}$$

  Fig. 11 shows this percentage for each clone-type of each subject system.

From Fig. 10 we realize that Type 3 clones are primarily involved with bug-propagation. We see that for two subject systems, jEdit and Ctags, Type 1 clones were not related with bug-propagation at all. The same is true for the Type 2 clones of Jabref, GLGraphics, Ctags, and MonoOSC. However, Type

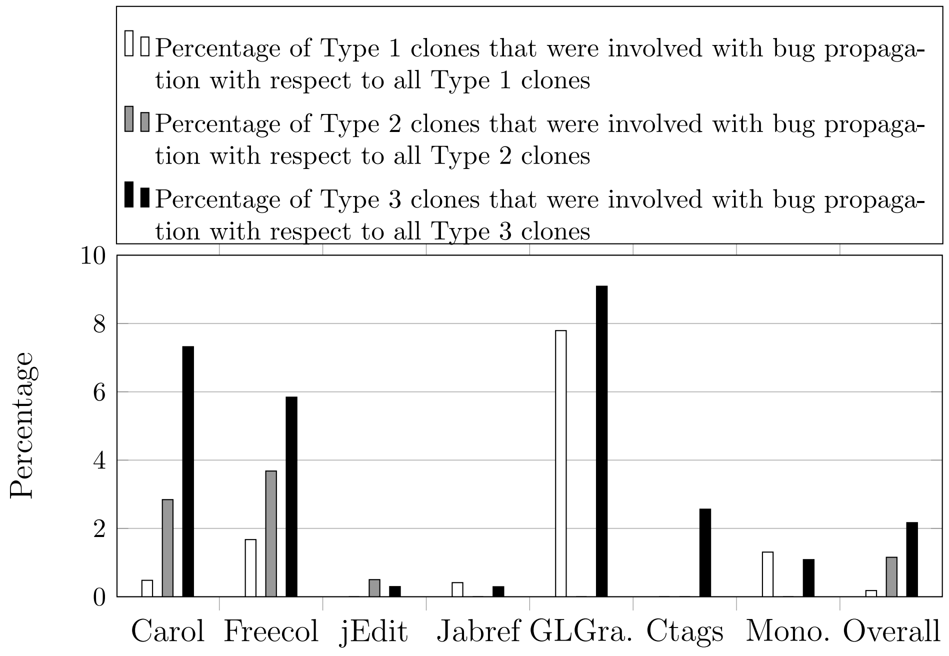


Figure 10: Percentage of clone fragments that were involved with bug propagation with respect to all clone fragments considering each clone-type.

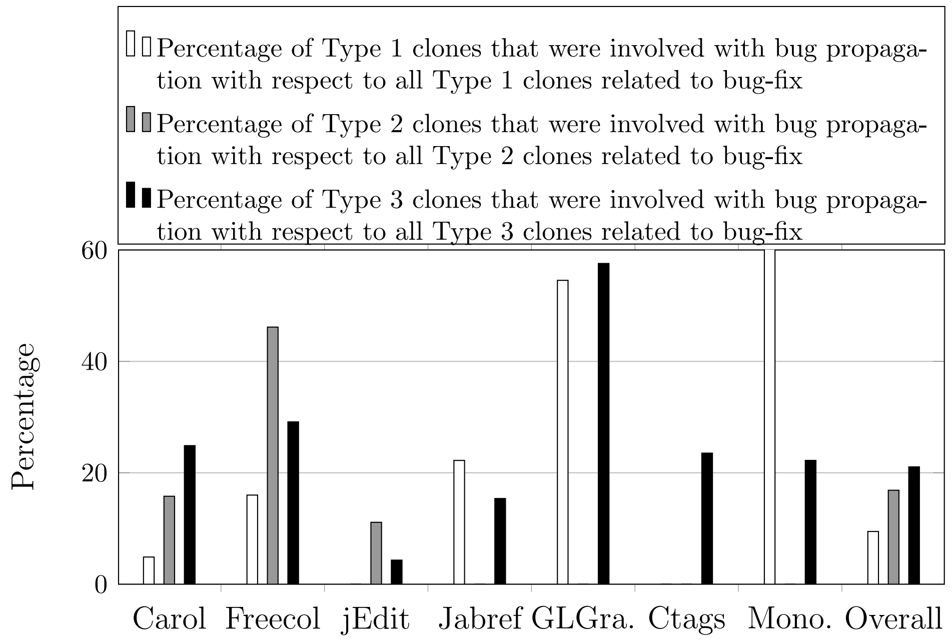


Figure 11: Percentage of clone fragments that were involved with bug propagation with respect to all bug-fix clone fragments considering each clone-type.

3 clones in each of the subject systems are related with bug-propagation. For four subject systems (Carol, Freecol, GLGraphics, and Ctags), the percentage of Type 3 clones that are involved with bug-propagation is the highest among the three clone-types. The overall percentages also indicate a similar scenario. We see that the overall percentage regarding Type 3 clones is the highest. We should note that the percentages plotted in Fig. 10 are generally very low. The reason behind this is that we calculate these percentages with respect to all clone fragments that were created during evolution. We know that a significant proportion of the code clones do not get changed at all during evolution [17]. The percentage of code clones that experience bug-fixes is generally very low (up to 19% according to a previous study [28]). The percentage of clone fragments that are involved with bug propagation should be even lower, because such clone fragments must be bug-fix clones and evolve following particular evolution patterns (defined in Section 4).

Fig. 11 shows the percentage of bug propagation clone fragments with respect to all bug-fix clones. We see that the percentages plotted in this graph are higher compared to the percentages plotted in the graph of Fig.

10. The overall comparative scenario of bug propagation in three clone-types presented in Fig. 11 is similar to that in Fig. 10. The overall percentages of the bug-propagation clone fragments with respect to the bug-fix clones are 9.46%, 16.86%, and 21.06% for Type 1, Type 2, and Type 3 cases respectively. If we consider all clone-types of all of our subject systems, then this percentage becomes 18.42% (161 clone fragments are involved with bug-propagation out of 874 bug-fix clone fragments).

Answer to RQ 1. From our experimental results and analysis we can state that *overall up to 21.06% of the clone fragments that experience bug-fixes can be involved with bug-propagation. Type 1 clones exhibit the lowest possibility of being involved with bug propagation. Bug propagation is mainly observed in Type 2 and Type 3 clones with Type 3 clones showing the highest intensity of propagation.*

Our findings imply that the possibility of bug-propagation through exact copy/paste activities is very low, because identical clones have a very low possibility of containing propagated bugs. However, a considerable proportion of the near-miss clones can be involved with bug-propagation. Thus, near-miss clones should be considered more important for management (such as refactoring or tracking) than the identical clones from the perspective of bug-propagation. The prototype tool that we have implemented for our research can be customized for identifying code clones that are likely to contain hidden but propagated bugs. We believe that the comparative intensities of bug-propagation in different clone-types should be taken into proper consideration when making clone management decisions. Our prototype tool can help us make such decisions.

Manual Analysis of the Bug Propagation Patterns: We manually analyzed the evolution patterns of all the 145 bug propagation clone pairs (1 pair from Type 1 case + 3 pairs from Type 2 case + 141 pairs from Type 3 case) from our subject system Carol. We have the following observations from our manual analysis.

(1) For most of the bug propagation clone pairs (for 141 out 145), the two clone fragments in the pair did not get any change before experiencing the bug-fix change in the bug-fix commit. In other words, after getting created the first change that the two clone fragments experienced was the bug-fix change for most of the pairs. From the two patterns defined in Section 4

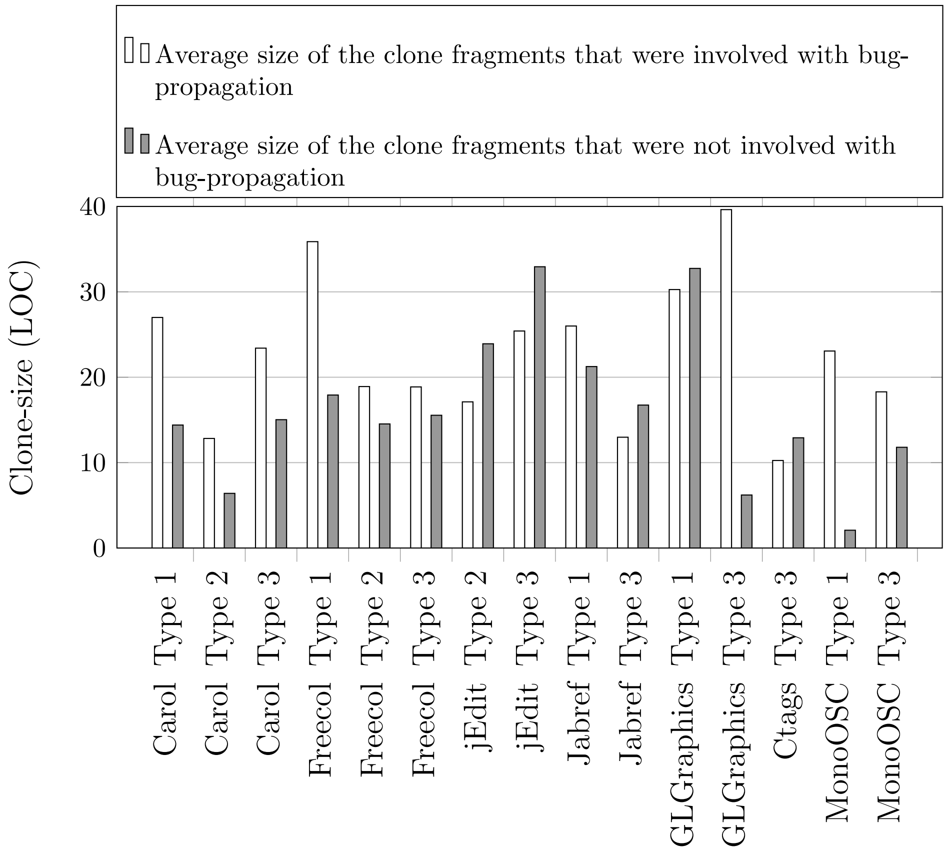


Figure 12: Average sizes of the clone fragments that were or were not involved with bug-propagation

we see that we allow only one of the two fragments to be changed before experiencing the bug-fix change. We found only four pairs where one of the two fragments got changed before both experienced the bug-fix change.

(2) For most of the bug propagation clone pairs (for 143 out 145), the two clone fragments are full methods. It seems that bug propagation mainly occurs in method clones. For the remaining two pairs, the clone fragments were if-blocks and try-catch blocks respectively. Fig. 8 shows fixing of a propagated bug in two method clones. We provide this example from our subject system Jabref.

5.2. Answering RQ 2

RQ 2: *Does clone-size have an effect on the bug-propagation possibility of code clones?*

Rationale. As code clones have a tendency of being related with bug-propagation, it is important to analyze which factor(s) affects the bug-propagation tendency. In this investigation we analyze whether clone-size has an effect on the bug-propagation tendency of code clones. We perform our investigation in the following way.

Investigation Procedure. We first identify those cases where code clones were involved with bug-propagation. A particular case consists of a subject system and a clone-type. If we look at Table 4, we realize that the number of cases where there were occurrences of bug-propagation is 15. An example of such case is Type 1 clones of Jabref. Table 4 shows that two Type 1 clone fragments of Jabref took part in bug-propagation. However, Type 2 clones of Jabref did not propagate bugs. Thus, we do not consider this case in this investigation. For each of the 15 cases (i.e., cases with occurrences of bug-propagation), we identify two sets of code clones. While one set contains those clone fragments that were involved with bug-propagation, the other set consists of those clone fragments that did not take part in bug-propagation. We determine the size (in LOC) of each of the clone fragments in these two sets. We finally determine the average size per clone fragment for each set. Thus, for each of the two sets of a particular case, we get an average size. Finally, from the 15 cases, we get 15 average sizes for the clone fragments that were involved with bug-propagation and the corresponding 15 average sizes for the clone fragments that were not involved with bug-propagation. The average sizes of the bug-propagation and non-bug-propagation clones for the 15 cases are shown in Fig. 12.

From Fig. 12 we see that for most of the cases (i.e., 10 out of 15 cases), the average size of the clone fragments that were involved with bug-propagation is higher than the average size of the clone fragments that were not involved with bug-propagation. We conduct Wilcoxon Signed Rank test [51, 52] to determine whether the average sizes of the bug-propagation clones are significantly different than the average sizes of the non-bug-propagation clones. We should note that Wilcoxon Signed Rank test is suitable for paired samples. This test is non-parametric, and thus, it does not require the samples to be normally distributed [52]. We conduct the test considering a significance level of 5%. The p-value regarding the test is 0.057 which is greater than 0.05. Thus, according to our significance test, the difference between the sizes of the bug-propagation and non-bug-propagation clones is not statistically significant.

Answer to RQ 2: According to our experiment, the clone fragments that are involved with bug-propagation are mostly bigger than the clone fragments that are not involved with bug-propagation. However, the difference between the average sizes of the bug-propagation and non-bug-propagation clones is not statistically significant.

5.3. Answering RQ 3

RQ 3: *What percentage of the bugs that are experienced by different clone-types can be propagated bugs?*

Rationale. From our answer to RQ 1 we realize what proportions of the clone fragments in different clone-types can be involved with bug propagation. However, we still do not know what percentage of the bugs experienced by code clones can be propagated bugs. Without this information we cannot fully realize the bug propagation scenarios in different clone-types. In RQ 3 we first determine what percentage of the bugs experienced by each clone-type can be propagated bugs, and then make a comparison considering the percentages regarding three clone-types of each of the subject systems. We investigate in the following way.

Methodology. We first identify the bug-fix commit operations for a subject system following the procedure described in Section 3.2. Considering a particular clone-type we select those commits where code clones of that particular type were changed. For such a commit we identify whether a clone-pair was changed (i.e., whether both of the clone fragments in the pair were co-changed) in it. Considering each such pair we determine whether the participating clone fragments evolved following a bug propagation pattern. The procedure for determining whether a clone-pair followed a bug propagation pattern has been discussed in Section 4.3. If a bug-fix commit modifies a clone-pair that evolved following a bug propagation pattern, we consider that the bug that was fixed in that commit is a propagated bug. We analyze each of the bug-fix commits that affected code clones of a particular clone-type and determine which bugs were propagated bugs. Considering each clone-type of each of the subject systems we determine the following two measures, and report these in Table 5:

Table 5: Number of propagated bugs in different clone-types

	Type 1		**Type 2**		**Type 3**	
Systems	**BC**	**BCPB**	**BC**	**BCPB**	**BC**	**BCPB**
Carol	11	1	9	3	31	7
Freecol	11	2	14	3	58	17
jEdit	38	0	10	1	44	3
Jabref	10	1	8	0	28	2
GLGraphics	7	2	0	0	15	3
Ctags	4	0	7	0	25	2
MonoOSC	3	1	2	0	8	1

BC = Total number of bug-fix commits that affected clone fragments of a particular clone-type.

BCPB = Number of bug-fix commits that indicate fixing of a propagated bug.

- **BC (Bug-fix Commit):** The total number of bug-fix commits (i.e., the number of bugs) experienced by the code clones of that particular clone-type.
- **BCPB (Bug-fix Commit indicating fixing of a Propagated Bug):** The number of bug-fix commits that indicate fixing of propagated bugs.

For a particular clone-type of a particular subject system, we also determine the percentage of bug-fix commits that indicate fixing of a propagated bug (this percentage = $(BCPB \times 100)/BC$), and show this percentage in the bar graph of Fig. 13.

From Fig. 13 we see that for two subject systems, jEdit and Ctags, none of the bug-fix commits affecting Type 1 clones indicate fixing of a propagated bug. The same is true for the Type 2 clones of Jabref, GLGraphics, Ctags, and MonoOSC. Such a scenario is similar to the scenarios in Fig. 10 and 11. From the overall percentages in Fig. 13 we realize that the percentage of commits that fixed propagated bugs is the highest in Type 3 clones among the three clone-types.

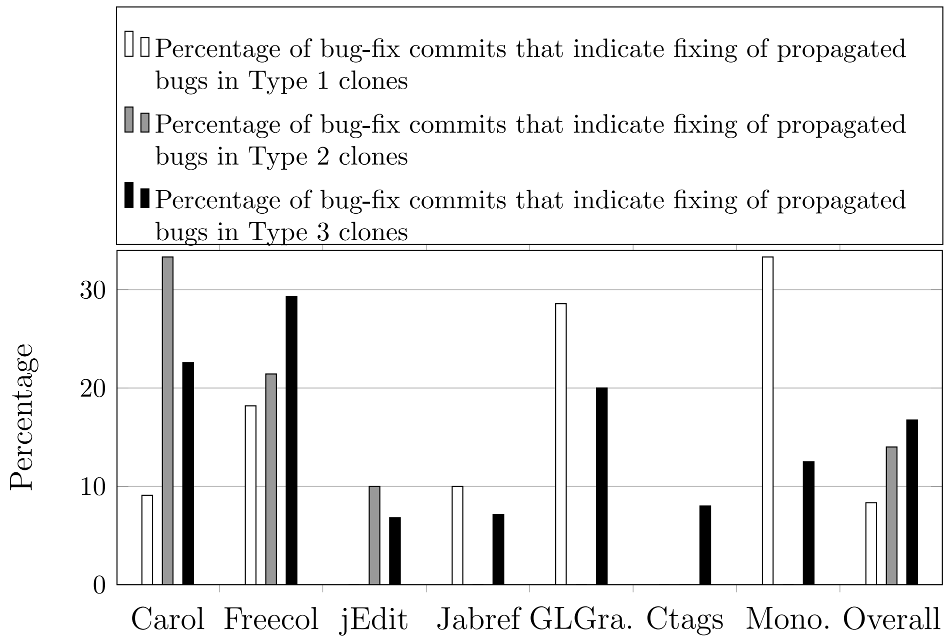


Figure 13: Percentage of bug-fix commits that indicate fixing of propagated bugs in different clone-types

Answer to RQ 3: According to our experimental results and analysis, *overall up to 16.74% of the bugs experienced by code clones can be propagated bugs. The overall percentage of propagated bugs is the highest in Type 3 case, and the lowest in Type 1 case.*

Our findings from RQ 3 are similar to those from RQ 1. We find that near-miss clones (Type 2 and Type 3) have higher possibilities of containing propagated bugs compared to Type 1 clones. Thus, near-miss clones should be given a higher priority for management.

5.4. Answering RQ 4

RQ 4: *Which pattern of bug-propagation is more intense during evolution?*

Rationale. We have defined two bug propagation patterns in Section 4. It is important to investigate which pattern is more intense during evolution. If a particular pattern appears to be more intense than the other one, then we can prioritize refactoring of clone fragments that have evolved as well as that have the possibility of evolving following that pattern. We perform our investigation in the following way.

Methodology. Considering each clone-type of each of the subject systems we identify the bug-propagation clone-pairs as we did for answering our previous research questions. Then, we determine the following two measures: (i) the number of clone-pairs that followed the first bug propagation pattern defined in Section 4.1, and (ii) the number of clone-pairs that followed the second bug propagation pattern defined in Section 4.2. These two measures for each clone-type of each of the subject systems have been reported in Table 6. Using the data in Table 6 we also draw a stacked bar graph in Fig. 14 showing the percentage of bug propagation clone-pairs following each pattern.

From both Table 6 and Fig. 14 it is clear that bug propagation of the first category (defined in Section 4.1) is more likely to occur compared to the second one (defined in Section 4.2) during evolution. From the overall scenario (i.e., considering all subject systems) we realize that around 87.5%, 71.42%, and 87.9% of the bug-propagation clone-pairs in Type 1, Type 2, and Type 3 case respectively followed the first bug-propagation pattern.

Table 6: Number of clone-pairs following different bug propagation patterns

	Type 1		**Type 2**		**Type 3**	
Systems	**PF**	**PS**	**PF**	**PS**	**PF**	**PS**
Carol	1	0	2	1	138	3
Freecol	2	0	2	1	20	9
jEdit	0	0	1	0	2	2
Jabref	0	1	0	0	2	0
GLGraphics	3	0	0	0	10	9
Ctags	0	0	0	0	2	0
MonoOSC	1	0	0	0	0	1

PF = Number of clone pairs that followed the first pattern

PS = Number of clone pairs that followed the second pattern

Answer to RQ 4: According to our investigation, *the first bug propagation pattern where the two clone fragments in the bug propagation clone pair were created in the same revision is more likely to occur compared to the second pattern where the two clone fragments were created in two different revisions.*

Our finding implies that clone fragments that are created together in the same commit operation have higher possibilities of containing propagated bugs compared to the clone fragments that were created in different revisions. Thus, for minimizing bug propagation we prioritize refactoring of clone fragments that were created together.

5.5. Answering RQ 5

RQ 5: *How often does a propagated bug residing in two clone fragments of a clone-pair get fixed in two different commit operations?*

Rationale. In the bug-propagation patterns described in Section 4, a propagated bug residing in the two clone fragments of a clone-pair always gets fixed in both fragments in the same commit operation. However, the two

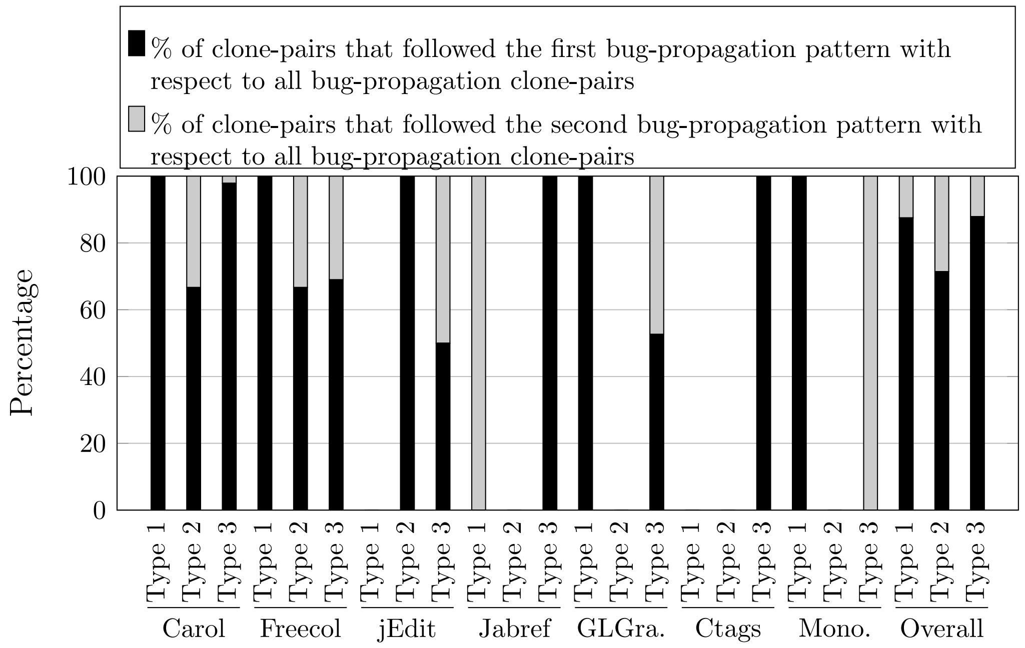


Figure 14: Comparing the likeliness of occurrence of two bug propagation patterns.

Table 7: Number of bug-propagation clone-pairs having clone fragments from the same or different files

	Type 1		**Type 2**		**Type 3**	
Systems	**SF**	**DF**	**SF**	**DF**	**SF**	**DF**
Carol	0	1	3	0	136	5
Freecol	0	2	3	0	20	9
jEdit	0	0	1	0	4	0
Jabref	0	1	0	0	2	0
GLGraphics	0	3	0	0	6	13
Ctags	0	0	0	0	2	0
MonoOSC	0	1	0	0	1	0

SF = Number of clone pairs each having clone fragments from the same file
DF = Number of clone pairs each having clone fragments from two different files

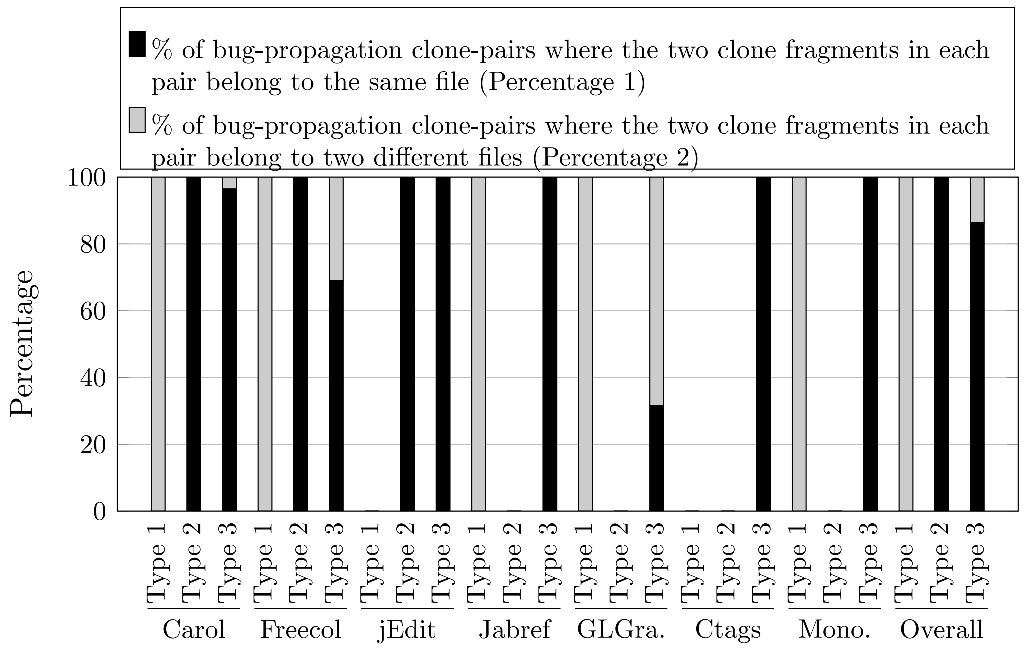


Figure 15: Percentages of bug-propagation clone-pairs having clone fragments from the same or different source code files

Table 8: Number of potential clone-pairs of different clone-types

Systems	**Type 1**	**Type 2**	**Type 3**
Carol	2	1	8
Freecol	0	3	11
jEdit	0	0	2
Jabref	0	0	0
GLGraphics	1	0	4
Ctags	1	1	8
MonoOSC	0	0	1

fragments might also experience the fix in two different commit operations. In this experiment, we investigate whether such cases really exist.

Investigation Procedure. We perform our investigation in the following two steps. While the first part (Step 1) is automatic, the second part (Step 2) involves manual investigation.

Step 1: In the first step, we identify bug-fix commit operations where code clones were modified. We analyze each of these commit operations for our investigation. Let us assume that BFC is such a bug-fix commit operation which was applied on a particular revision r of a subject system. We identify clone-pairs in revision r such that one fragment in the pair experienced the bug-fix (i.e., was modified in the bug-fix commit BFC) but the other fragment did not. From these identified pairs, we select each pair such that the fragment (of the pair) that did not have the bug-fix change in BFC experienced a bug-fix later. We call these selected pairs as the *potential pairs*. From all our subject systems, we obtained 43 potential pairs in total. Table 8 shows the number of potential pairs for different clone-types of different subject systems.

Step 2: In this step, we manually analyze each of the potential pairs obtained from Step 1. From our discussion in Step 1 it is clear that the two clone fragments in a potential pair experienced bug-fixes in two different commit operations. We manually analyze these bug-fix changes that occurred to the two fragments in order to realize whether the same fix was experienced by both fragments. For each clone-type of each of the subject systems, Table

Table 9: Number of potential clone-pairs where the two clone fragments experienced the same bug-fix in two different commits

Systems	**Type 1**	**Type 2**	**Type 3**
Carol	0	0	1
Freecol	0	0	0
jEdit	0	0	0
Jabref	0	0	0
GLGraphics	0	0	0
Ctags	1	0	0
MonoOSC	0	0	1

9 shows the number of potential clone-pairs where the two clone fragments experienced the same bug-fix in two different commits. As we can see in Table 9, we obtained three potential clone-pairs (one pair from Type 3 case of Carol, one pair from Type 1 case of Ctags, and the remaining pair from Type 3 case of MonoOSC) where the two clone fragments in the pair experienced the same bug-fix at different commit operations. We discuss these examples as follows.

In the case of Ctags, we found one Type 1 clone-pair residing in revision 633. One of the two clone fragments in the pair was located in a file named 'jscript.c' and the other one was located in a file named 'sql.c'. The fragment located in 'jscript.c' was changed for fixing a bug in the commit operation which was applied on revision 633. The commit message says, *jscript.c was not properly handling escaped quotes.* The clone fragment in file 'sql.c' contained the same bug which was fixed in the same way in the commit operation which was applied on revision 636. This commit message says, *Ported the same change made to handle escaped strings in jscript.c into sql.c.* In the case of MonoOSC, we found a Type 3 clone pair where one fragment in the pair experienced a bug-fix in the commit operation on revision 307, and the other one experienced the same fix in the commit operation on revision 312. From the Type 3 clones of Carol, we found a clone pair where one fragment in the pair experienced a bug-fix in the commit on revision 396. The other fragment experienced a similar fix in the immediate next commit operation.

While performing manual analysis, we ensured that these three bugs (each fixed in two different commits) were propagated bugs.

Answer to RQ 5. According our experiment, a propagated bug that resides in two or more clone fragments mostly gets fixed in all the container fragments in the same commit operation. Fixing of a propagated bug in different commits is a rare phenomenon according to our analysis.

5.6. Answering RQ 6

RQ 6: *Does bug propagation occur in the same file or across different files?*

Rationale. It is important to know whether bug-propagation mainly occurs within the same file or across different files. If it is found that the clone fragments that are involved with bug-propagation have a high tendency of residing in the same file or in different files, then such clone fragments can be prioritized for refactoring. Refactoring/merging of code clones having high tendencies of being involved with bug-propagation, can help us minimize efforts for fixing propagated bugs in multiple clone fragments. We perform our investigation for answering RQ 6 in the following way.

Investigation Procedure. For each clone-type of each of our subject systems, we identify the clone-pairs that were involved with bug-propagation as we did for answering our previous research questions. Considering each of these bug-propagation clone-pairs we determine whether the fragments in the pair belong to the same file or in different files. Table 7 shows the results from our investigation. From the data recorded in Table 7 we determine the following two percentages for each clone-type of each of our subject systems.

- **Percentage 1:** Percentage of bug-propagation clone-pairs where the two clone fragments in each pair belong to the same source code file with respect to all bug-propagation clone-pairs.

- **Percentage 2:** Percentage of bug-propagation clone-pairs where the two clone fragments in each pair belong to two different source code files with respect to all bug-propagation clone-pairs.

The stacked bar graph in Fig. 15 shows these two percentages for our subject systems. From the figure we see that for each of the bug-propagation clone-pairs in Type 1 case, the two clone fragments in the pair belong to two different source code files. The opposite is true for Type 2 case. Each Type 2 bug-propagation clone-pair consists of clone fragments belonging to the same file. For most of the Type 3 bug-propagation clone pairs of each of our subject systems, **Percentage 1** is greater than **Percentage 2**. The overall percentages regarding the three clone-types indicate a similar scenario.

Answer to RQ 6. According to our experiment results and analysis, while each bug-propagation clone-pair in Type 1 case contains clone fragments from two different source code files, most of the bug-propagations in near-miss code clones (Type 2 and 3) occur in the same file.

Our finding is important for making clone management decisions from the perspective of bug-propagation. When identifying Type 1 clone-pairs having the possibility of containing propagated bugs, we can mainly focus on clone-pairs each containing clone fragments from different source code files. However, when working with near-miss clones, we should primarily focus on the clone-pairs each having clone fragments from the same source code file.

5.7. Answering RQ 7

RQ 7: *Do severe bugs get propagated through code cloning?*

Rationale. From our answer to RQ 3 we realize that a considerable proportion of the bugs contained by code clones can be propagated bugs. We wanted to do further investigations in order to understand whether bugs that get propagated through code cloning can be severe bugs. If we see that propagated bugs can often be severe, then it is important to identify clone pairs that have a high possibility of containing propagated bugs so that we can refactor them with high priority. We perform our investigation in the following way.

Investigation Procedure. We conduct our investigation in two steps: (1) identifying the bug-fixing commits that occurred for fixing severe bugs, and (2) investigating propagation of severe bugs. We discuss these two steps in the following paragraphs.

(1) Identifying the bug-fixing commits that occurred for fixing severe bugs. For each of our subject systems, we first identify the commit log that contains the commit messages of all the commit operations. We automatically identify the message for each commit separately and determine whether this is a bug-fix commit by applying the technique proposed by Mockus and Votta [26]. If a particular commit is a bug-fix commit, then we identify whether this commit occurred for fixing a severe bug. For identifying whether a commit operation occurred for fixing a severe bug, we apply the technique proposed by Lamkanfi et al. [21]. They suggested a list of keywords [21] from their investigation such that if the commit message of a bug-fix commit operation contains any of those keywords, we can consider that commit as a severe bug-fix commit. We identify all the severe bug-fix commits from each of our subject systems. Lamkanfi et al. [21] reported that the precision and recall of their bug severity detection technique can vary within the range 65% to 75%.

(2) Investigating propagation of severe bugs. In Section 4.3, we discussed how we identify the bug-propagation patterns by analyzing the bug-fix commit operations. We follow the same procedure of identifying bug-propagation patterns by considering only the severe bug-fix commits. From all the bug-propagation clone-pairs of a subject system, we determine how many of those were involved in propagating severe bugs. Table 10 shows the number of clone-pairs that were involved in propagating severe bugs.

From Table 10 we realize that propagation of severe bugs through code cloning is not a common phenomenon during software evolution. One Type 1 clone-pair of Freecol and respectively ten and one Type 3 clone-pairs of Freecol and jEdit were involved in propagating severe bugs. We answer RQ 7 in the following way.

Answer to RQ 7. According to our investigation result, severe bugs can sometimes get propagated through code cloning. However, the possibility of this propagation is very low. Only 5.16% of all bug-propagation clone-pairs (11 out of 213 bug propagation clone pairs from all clone-types of all subject systems) were involved with propagating severe bugs.

Although code cloning has a very low probability of propagating severe bugs, we see that propagation of severe bugs can sometimes occur to a considerable extent. Around 34.48% (10 out of 29) of the Type 3 bug-propagation clone-pairs of Freecol were involved in propagating severe bugs. Thus, bug-propagation through code cloning should not be ignored. Code clones having high possibilities of propagating bugs should be identified and refactored with

Table 10: The number of bug-propagation clone-pairs that are involved with propagating severe bugs

Systems	**PSB (Type 1)**	**PSB (Type 2)**	**PSB (Type 3)**
Carol	0	0	0
Freecol	1	0	10
jEdit	n/a	0	1
Jabref	0	n/a	0
GLGraphics	0	n/a	0
Ctags	n/a	n/a	0
MonoOSC	0	n/a	0

PSB = Number of clone pairs that propagated severe bugs.
n/a = Analysis regarding propagating severe bugs is not applicable for the case because no clone pair for that case was involved with bug-propagation.

high priority.

6. Implications from Our Findings

Although existing studies [33, 28, 37, 38] on code clone detection and analysis suspect that code cloning can be responsible for bug-propagation, none of the existing studies investigated it. Our study is the first one to show that bug-propagation is a fact. As bug-propagation occurs through code cloning, it is important to know which types of clones are more involved with bug-propagation so that programmers can avoid making such clones during programming. Also, if such clones are already existing in the code-base, we should consider refactoring or tracking them with high priority. In the following subsections, we first summarize our findings, and then discuss how these findings can be important from two different perspectives: (1) clone management perspective and (2) programming perspective.

6.1. Summary of our findings

We summarize our findings in the following points.

- **Finding 1:** According to our answer to RQ 1, near-miss clones are more involved with bug-propagation compared to exact clones. We should consider managing (refactoring or tracking) near-miss clones with higher priorities compared to exact clones.

- **Finding 2:** According to our manual analysis, method clones have high possibilities of being involved with bug-propagation. Thus, programmers should assume higher priorities for refactoring or tracking method clones than block clones in order to minimize bug-propagation.

- **Finding 3:** According to our answer to RQ 6, near-miss code clones residing in the same file have a higher possibility of being related to bug-propagation compared to the ones residing in different files. The opposite is true for the identical clones. These findings can help us develop a clone-type sensitive automatic tool for identifying code clones having high possibilities of containing propagated bugs.

- **Finding 4:** According to our answer to RQ 4, the first bug-propagation pattern is more frequent than the second one. Thus, making clones from a previously committed code fragment is safer than making clones from a newly created (uncommitted) code fragment.

- **Finding 5:** According to our answer to RQ 7, severe bugs can sometimes get propagated through code cloning. Thus, bug-propagation through code cloning should not be ignored.

Our findings are important from two perspectives: (1) managerial perspective and (2) programming perspective as discussed below.

6.2. Importance of our findings from clone management perspective

Our first three findings (i.e., Finding 1, 2, and 3) in Section 6.1 are important from the perspective of clone management (i.e., clone refactoring and tracking). A software system generally contains a huge number of code clones. It is impractical to aggressively refactor or track all these code clones [17]. Clone refactoring often requires a lot of time and effort from the programmers. Clone tracking is resource intensive. In such a situation, it is necessary to identify code clones that should be considered important for refactoring or tracking. Our first three findings can help us pin-point the important clones. If we combine the first three findings, we realize that near-miss method clones residing in the same file should be prioritized for management. The percentage of such clones in the entire code-base of each of our subject systems is shown in Fig. 16. From Fig. 16 we realize that the percentage of code clones that can be considered important (i.e., that can be prioritized) for refactoring or tracking is very low for each system compared to the percentage of the remaining clones in that system.

We also perform Wilcoxon Signed Rank (WSR) test [51, 52] to determine whether the percentage of code clones that we recommend as important for management is significantly lower than the percentage of the remaining clones in a software system. We perform WSR test, because it is suitable for paired samples. Moreover, WSR test is non-parametric [52], and thus, the samples do not need to be normally distributed for applying this test. This test can be applied for both small and large data samples [52]. We conduct the test considering a significance level of 5%. The p-value regarding our test is 0.001 which is smaller than 0.05. Thus, according to our test, the percentage of code clones that we recommend as important is significantly smaller than the percentage of the remaining code clones in a software system. We also determine the Cohen's d effect size [53, 54] for our test and find that the effect size is 0.632. We finally realize that prioritizing the code clones that we recommend as important for management can save a considerable amount of clone refactoring and tracking effort and time.

6.3. Importance of our findings from programming perspective

Our fourth finding in Section 6.1 is important from programming perspective. According to this finding, copying from uncommitted code is riskier than copying from a previously committed code. Thus, we strongly recommend programmers to stop copying from newly written or uncommitted code. In general, copy/pasting cannot be completely stopped during programming. However, if our recommended copying discipline is followed, it will minimize the risk of bug-propagation through code cloning. Our fifth finding raises the awareness that code cloning can be responsible for propagating severe bugs too. The programmers can minimize bug-propagation through applying our recommended copying discipline.

7. Related Work

A great many studies have investigate code clones from different perspectives such as: clone detection [37, 36, 45, 42], impact analysis [1, 2, 7, 8, 9,

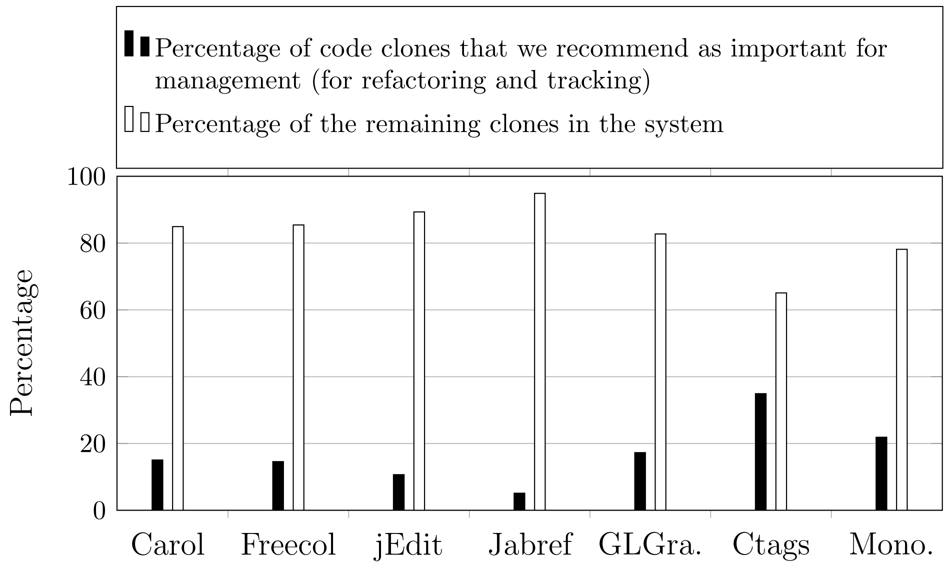


Figure 16: Percentage of code clones that we recommend as important for management (refactoring and tracking)

15, 16, 18, 19, 20, 24, 25, 32, 33, 22, 44, 13], maintenance [49, 46, 27, 31], and bug-proneness [22, 23, 7, 48, 28]. Our study in this paper focuses on the bug-proneness of code clones.

Bug-proneness of code clones has been investigated by a number of studies. Li and Ernst [22] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [23] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [44] investigated on finding instances of incompletely fixed bugs in near-miss code clones by investigating a broad range of features of such clones involving machine learning. Göde and Koschke [7] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes occurred to the code clones are unintentionally inconsistent. Chatterji et al.[4] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al.[13] performed a study on the context based inconsistencies related to clones. They developed an algorithm to mine such inconsistencies for the purpose of locating bugs. Using their algorithm they could detect previously unknown bugs from two open-source subject systems. Inoue et al.[10] developed a tool called 'CloneInspector' in order to identify bugs related to inconsistent changes to the identifiers in the clone fragments. They applied their tool on a mobile software system and found a number of instances of such bugs. Xie et al.[48] investigated fault-proneness of Type 3 clones in three open-source software systems. They investigated two evolutionary phenomena on clones: (1) mutation of the type of a clone fragment during evolution, and (2) migration of clone fragments across repositories and found that mutation of clone fragments to Type 2 or Type 3 clones is risky. We see that a number of studies investigated bug-proneness in code clones. However, none of these studies focus on bug propagation through code cloning.

Islam et al. [11] investigated bug-replication in code clones. They identified which of the clone fragments in a clone class contains the same bug. If more than one clone fragment in a clone class contain the same bug, they considered that the bug is a replicated one. However, it does not imply that this bug is a propagated bug. It might be the case that similar buggy changes occurred to the clone fragments in a clone class during evolution. Such a bug will be considered as a replicated bug according to Islam et al.'s [11] consideration. However, this is not a propagated bug. Bug propagation only occurs when a particular code fragment contains a bug, and this code fragment is copied to several other places in the code-base being unaware of the presence of the bug. In our study, we define two bug-propagation patterns and propose an automatic mechanism for identifying propagated bugs in code clones. Thus, our bug propagation study is significantly different from Islam et al.'s study [11].

In another study, Mondal et al. [28] compared the bug-proneness of three types of code clones. They investigated which types of code clones experience bug-fixes more frequently. However, they did not investigate bug-propagation in their study.

Rahman et al. [35] made a comparison of the bug-proneness of clone and non-clone code and found that clone code is less bug-prone than non-clone code. They performed their investigation on the evolution history of four subject systems using DECKARD [14] clone detector. However, they did not investigate bug-propagation through code cloning. Thus, our study on the intensity of bug-propagation in different types of code clones is different from their study.

Selim et al. [43] used Cox hazard models in order to assess the impacts of cloned code on software defects. They found that defect-proneness of code clones is system dependent. However, they considered only method clones in their study. We consider block clones in our study. While they investigated only two subject systems, we consider four subject systems in our investigation. Also, we investigate bug propagation in different types of code clones. Selim et al. [43] did not perform such an investigation.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al.[1] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al.[2, 3] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies.

We see that different studies have investigated clone related bugs in different ways and have developed different bug detection tools. However, none of these studies investigate the intensity of bug-propagation through code cloning. Investigating bug-propagation through code cloning is important. Without such an investigation we cannot properly realize the impacts of code cloning on software maintenance and evolution. Focusing on this issue we define and investigate two bug propagation patterns in code clones in our study. Our investigation involving manual analysis of the clone fragments that evolved following the bug-propagation patterns results interesting findings which are important for better management of code clones.

Our study presented in this paper is a significant improvement of our previous work [12] on bug-propagation. In this extended work, we answer four new research questions (RQ 2, RQ 5, RQ 6, and RQ 7) involving file-proximity of the bug-propagation clones, and propagation of severe bugs. We did not investigate these in our previous study [12]. We also investigate three additional subject systems written in Java, C, and C# in this extended study. In our previous study, we only studied four Java systems. Moreover, we considered a case sensitive search of the bug-fix commits in our previous study [12]. However, this search has the possibility of missing some bug-fix commits. Focusing on this, we made a case insensitive search of the bug-fix commits in this extended study. Thus, all the tables and figures in this extended study beside the newly added ones have been refined with updated data. One of the major findings from our extended research is that severe bugs can sometimes get propagated through code cloning. Thus, bug-propagation should be taken into proper consideration when making clone management decisions.

8. Threats to Validity

We used the NiCad clone detector [5] for detecting clones. For different settings of NiCad, the experimental results that we present in this paper might be different. Wang et al. [47] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [40] and with these settings NiCad can detect clones with high precision and recall [41, 42, 45]. Thus, we believe that our findings on bug propagation through code cloning are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al.[3]. Such a technique proposed by Mocus and Votta [26] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al.[3] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the comparative bug-proneness of clone-types. However, we selected our candidate systems emphasizing their diversity in sizes and revision history lengths. Thus, we believe that our findings cannot be attributed to a chance. Our findings are important from the perspectives of clone management and can help us in better ranking of code clones for refactoring and tracking.

Identifying severity from the commit log is not necessarily precise. Our implementation for identifying severe bugs was based on the keywords suggested by Lamkanfi et al. [21]. While our analysis could be based on the severity values assigned to the bugs in the bug database, the assigned severity values often do not indicate the extent of severity of the bugs. In most of the cases, bugs are given a default severity value. Thus, the way [21] we perform bug severity analysis is reasonable. Lamkanfi et al. [21] reported that their technique had a precision and a recall in the range 65% to 75% in detecting severe bugs.

In our last research question (RQ 8), we have shown the time duration between propagation and fixing of the propagated bugs. This would be good to determine fixing time of the bugs in non-clone code, and then make a comparison between the fixing time of the bugs in non-clone code and the propagation time of the propagated bugs in clone code. However, our bug detection technique does not support identifying when a particular bug was introduced to the code-base. As a result, we cannot determine the fixing time of the bugs occurred in non-clone code, and we could not make such a comparison in our research. However, our technique supports identifying when a particular bug was propagated through code cloning. Thus, in RQ 8, we could report the time duration between propagation and fixing of propagated bugs. Our finding from our RQ 8 indicates that bugs generally take a long time to get fixed after being propagated through code cloning. Thus, bug-propagation should be taken into proper consideration.

9. Conclusion

In this study we investigate the intensity of bug-propagation through code cloning. We define two bug-propagation patterns, and automatically mine these patterns by analyzing the entire evolution history of our subject systems. We perform our investigation on thousands of revisions of seven subject systems and answered seven important research questions. According to our analysis, overall 18.42% of the code clones that experience bug-fix changes can be related with bug-propagation. Near-miss clones (Type 2, and Type 3 cones) have a higher tendency of being involved with bug-propagation compared to identical clones (Type 1 clones). Thus, near-miss clones should be given a higher priority for management from the perspective of bug-propagation. We also observe that overall 16.74% of the bug-fix changes in code clones can occur for fixing propagated bugs. We manually investigate the occurrences of bug-propagation in code clones and discover that method clones are mostly involved with bug-propagation. Bug-propagation primarily occurs in the clone fragments that got created together in the same commit operation. We also realize that near-miss code clones belonging to the same source code file have a higher possibility of propagating bugs compared to those that reside in different source code files. Code cloning can sometimes propagate severe bugs according to our analysis. An automatic tool for immediate detection of the occurrences of bug-propagation through code cloning can be beneficial for software maintenance and evolution. Our prototype tool implemented for this study can assist programmers in identifying code clones that are likely to contain propagated bugs. Thus, our tool can be helpful for prioritizing code clones considering their likeliness of being involved with bug-propagation.

[1] L. Aversano, L. Cerulo, and M. D. Penta, “How clones are maintained: An empirical study”, Proc. *CSMR*, 2007, pp. 81 – 90.

[2] L. Barbour, F. Khomh, Y. Zou, “Late Propagation in Software Clones”, Proc. *ICSM*, 2011, pp. 273 – 282.

[3] L. Barbour, F. Khomh, Y. Zou, “An empirical study of faults in late propagation clone genealogies”, *Journal of Software: Evolution and Process*, 2013, 25(11):1139 – 1165.

[4] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, N. A. Kraft, “Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study”, Proc. *ESEM*, 2011, pp. 20 – 29.

[5] J. R. Cordy, C. K. Roy, “The NiCad Clone Detector”, Proc. *ICPC Tool Demo*, 2011, pp. 219 – 220.

[6] CTAGS: `http://ctags.sourceforge.net/`

[7] N. Göde, Rainer Koschke, “Frequency and risks of changes to clones”, Proc. *ICSE*, 2011, pp. 311 – 320.

[8] N. Göde, J. Harder, “Clone Stability”, Proc. *CSMR*, 2011, pp. 65 – 74.

[9] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, “Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software”, Proc. *EVOL/IWPSE*, 2010, pp. 73 – 82.

[10] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, E. Lee, “Experience of Finding Inconsistently-Changed Bugs in Code Clones of Mobile Software”, Proc. *IWSC*, 2012, pp. 94 – 95.

[11] J. F. Islam, M. Mondal, C. K. Roy, “Bug Replication in Code Clones: An Empirical Study”, Proc. *SANER*, 2016, pp. 68 - 78.

[12] M. Mondal, C. K. Roy, K. A. Schneider, “Bug Propagation through Code Cloning: An Empirical Study”, Proc. *ICSME*, 2017, pp. 227 – 237.

[13] L. Jiang, Z. Su, E. Chiu, “Context-Based Detection of Clone-Related Bugs”, Proc. *ESEC-FSE*, 2007, pp. 55 – 64.

[14] L. Jiang, G. Misherghi, Z. Su, S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones”, Proc. *ICSE*, 2007, pp. 96105.

[15] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, “Do Code Clones Matter?”, Proc. *ICSE*, 2009, pp. 485 – 495.

[16] C. Kapser, M. W. Godfrey, ““Cloning considered harmful” considered harmful: patterns of cloning in software”, *Empirical Software Engineering*, 2008, 13(6): 645 – 692.

[17] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy, “An empirical study of code clone genealogies”, Proc. *ESEC-FSE*, 2005, pp. 187 – 196.

[18] J. Krinke, “A study of consistent and inconsistent changes to code clones”, Proc. *WCRE*, 2007, pp. 170 – 178.

[19] J. Krinke, “Is cloned code more stable than non-cloned code?”, Proc. *SCAM*, 2008, pp. 57 – 66.

[20] J. Krinke, “Is Cloned Code older than Non-Cloned Code?”, Proc. *IWSC*, 2011, pp. 28 – 33 .

[21] A. Lamkanfi and S. Demeyer and E. Giger and B. Goethals, “Predicting the Severity of a Reported Bug”, Proc. *MSR*, 2010, pp. 1 – 10.

[22] J. Li, M. D. Ernst, “CBCD: Cloned Buggy Code Detector”, Proc. *ICSE*, 2012, pp. 310 – 320.

[23] Z. Li, S. Lu, S. Myagmar, Y. Zhou, “CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code”, Proc. *OSDI*, 2004, pp. 20 – 20.

[24] A. Lozano, M. Wermelinger, “Tracking clones’ imprint”, Proc. *IWSC*, 2010, pp. 65 – 72.

[25] A. Lozano, M. Wermelinger, “Assessing the effect of clones on changeability”, Proc. *ICSM*, 2008, pp. 227 – 236.

[26] A. Mockus, L. G. Votta, “Identifying Reasons for Software Changes using Historic Databases”, Proc. *ICSM*, 2000, pp. 120 – 130.

[27] M. Mondal, C. K. Roy, K. A. Schneider, “SPCP-Miner: A Tool for Mining Code Clones that are Important for Refactoring or Tracking”, Proc. *SANER*, 2015, 5pp. (to appear).

[28] M. Mondal, C. K. Roy, K. A. Schneider, “A Comparative Study on the Bug-proneness of Different Types of Code Clones”, Proc. *ICSME*, 2015, pp. 91 - 100.

[29] J. F. Islam, M. Mondal, C. K. Roy, K. A. Schneider, “A Comparative Study of Software Bugs in Clone and Non-Clone Code”, Proc. *SEKE*, 2017, pp. 1 - 8.

[30] M. Mondal, C. K. Roy, K. A. Schneider, “Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems”, Proc. *CASCON*, 2012, pp. 205 – 219.

[31] M. Mondal, C. K. Roy, K. A. Schneider, “Automatic Ranking of Clones for Refactoring through Mining Association Rules”, Proc. *CSMR-WCRE*, 2014, pp. 114 – 123.

[32] 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”, Proc. *SAC*, 2012, pp. 1227 – 1234.

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

[34] Online SVN repository: `http://sourceforge.net/`

[35] F. Rahman, C. Bird, P. Devanbu, “Clones: What is that Smell?”, Proc. *MSR*, 2010, pp. 72 – 81.

[36] D. Rattan, R. Bhatia, M. Singh, “Software Clone Detection: A Systematic Review”, *Information and Software Technology*, 2013, 55(7): 1165 – 1199.

[37] C. K. Roy, J. R. Cordy, “A Survey on Software Clone Detection Research”, *Technical Report No. 2007-541*, 2007, School of Computing Queens University, pp. 1 - 115.

[38] C. K. Roy, M. F. Zibran, R. Koschke, “The Vision of Software Clone Management: Past, Present, and Future (Keynote paper)”, Proc. *CSMR-WCRE*, 2014, pp. 18 – 33.

[39] C. K. Roy, “Detection and analysis of near-miss software clones”, Proc. *ICSM*, 2009, pp. 447 – 450.

[40] C. K. Roy, J. R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization”, Proc. *ICPC*, 2008, pp. 172 – 181.

[41] 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*, 2009, 74 (2009): 470 – 495.

[42] C. K. Roy, J. R. Cordy, “A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools”, Proc. *Mutation*, 2009, pp. 157 – 166.

[43] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, “Studying the Impact of Clones on Software Defects”, Proc. *WCRE*, 2010, pp. 13 - 21.

[44] D. Steidl, N. Göde, “Feature-Based Detection of Bugs in Clones”, Proc. *IWSC*, 2013, pp. 76 – 82.

[45] J. Svajlenko, C. K. Roy, “Evaluating Modern Clone Detection Tools”, Proc. *ICSME*, 2014, pp. 321 – 330.

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

[47] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation", Proc. *ESEC/SIGSOFT FSE*, 2013, pp. 455 – 465.

[48] S. Xie, F. Khomh, Y. Zou, "An Empirical Study of the Fault-Proneness of Clone Mutation and Clone Migration", Proc. *MSR*, 2013, pp. 149 – 158.

[49] M. F. Zibran, C. K. Roy, "Conflict-aware Optimal Scheduling of Code Clone Refactoring", *IET Software*, 2013, 7(3): 167 – 186.

[50] Last revisions of the subject systems: `https://drive.google.com/drive/folders/1xgmHCxjIDav9Q5A7arJNXjsYrm5OpjcD?usp=sharing`

[51] Wilcoxon Signed Rank Test Calculator: `http://www.statskingdom.com/175wilcoxon_signed_ranks.html`

[52] Wilcoxon Signed Rank Test: `https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test`

[53] Effect Size calculator: `https://www.ai-therapy.com/psychology-statistics/effect-size-calculator`

[54] Effect size: `http://staff.bath.ac.uk/pssiw/stats2/page2/page14/page14.html`

*Biography

Manishankar Mondal

Manishankar Mondal is an Assistant Professor in the Computer Science and Engineering Discipline of Khulna University, Bangladesh. He completed his M.Sc. in Software Engineering from the Computer Science Department of the University of Saskatchewan, Canada by working under the supervision of Dr. Chanchal K. Roy and Dr. Kevin A. Schneider. During M.Sc. studies, he received the Best Paper Award from the 27th Symposium On Applied Computing (ACM SAC 2012) in the Software Engineering Track. He also received his PhD in February, 2017 from the same department by working under the same advisors. His research interests are software maintenance and evolution including clone detection and analysis, program analysis, empirical software engineering and mining software repositories. He has been a reviewer of a number of software engineering conferences and journals. He has served as the web and publicity co-chair of ICPC 2014 and as a program committee member of IWSC 2016. He has also served as a research associate in the software research laboratory of the Computer Science Department of the University of Saskatchewan.

Banani Roy

Banani Roy is an Assistant Professor at the Department of Computer Science at the University of Saskatchewan. Dr. Roy closely works with the graduate, summer, and undergraduate students, and postdoctoral fellows in the Cloud-based Big Data Analytics for Crop Phenomics project (a.k.a P2IRC Project 3.1, a USask CFREF funded project) and Global Water Future Project (a.k.a. GWF project, second USask CFREF funded project). Led by Prof. Kevin Schneider, she was also able to secure a Compute Canada Resource Allocation Competition ($178,543.00) grant for the P2IRC project. She received her Ph.D. in Engineering Interactive Systems from the Queen's University in 2013 under the supervision of Prof. Nicholas Graham and Prof. Carl Gutwin, She worked as a faculty member at the Computer Science and Engineering Department at Khulna University of Engineering and Technology (KUET). She also received several awards including a Queen's Graduate Scholarship and the highly competitive Ontario Graduate Scholarship in Science and Technology (OGSST). Her research interests are Engineering Interactive Systems, Software Reengineering, Software Architecture, Big Data Analytics and Empirical Software Engineering.

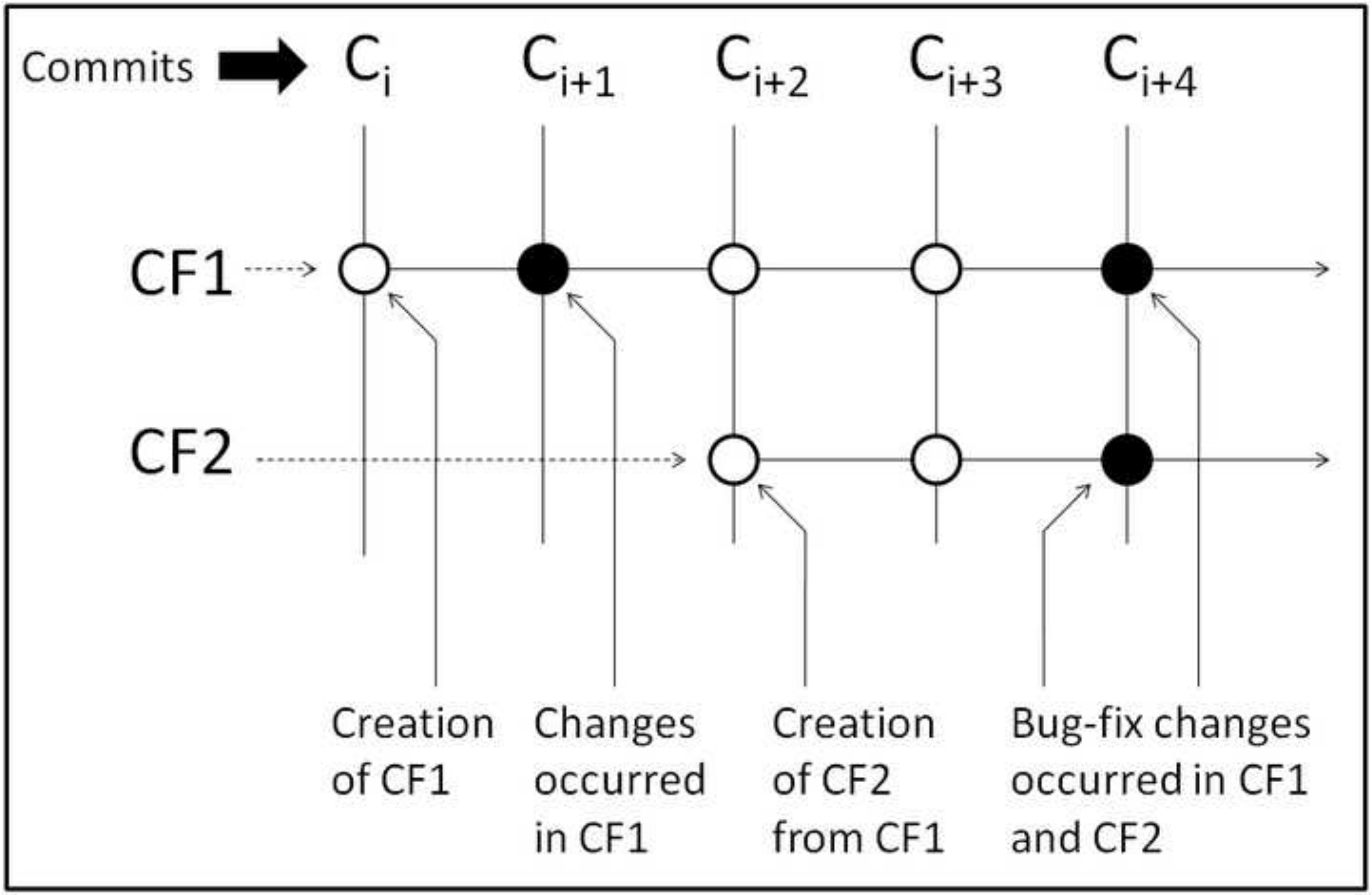
Chanchal K. Roy

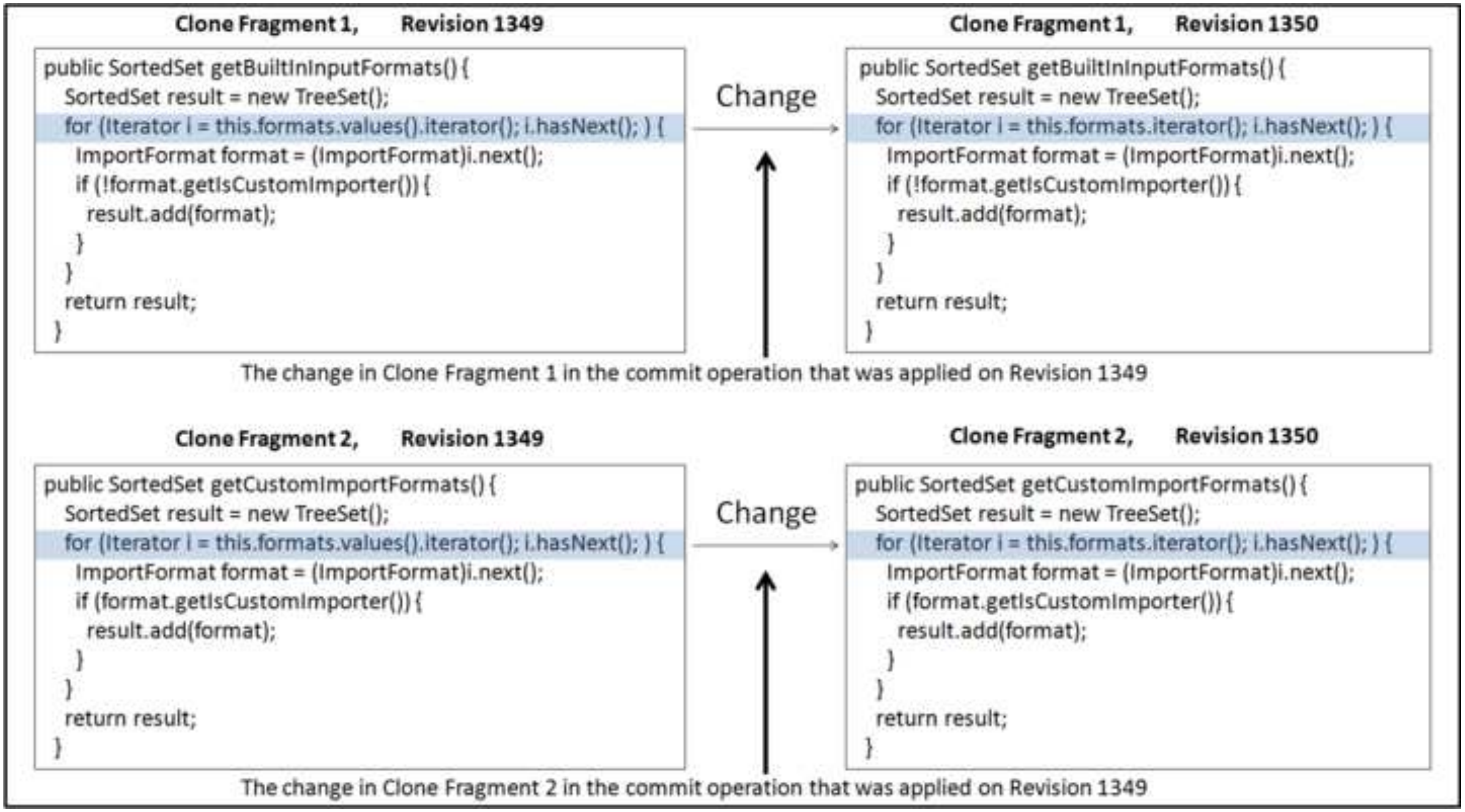
Chanchal Roy is an associate professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution, including clone detection and analysis, program analysis, reverse engineering, empirical software engineering and mining software repositories. He served or has been serving in the organizing and/or program committee of major software engineering conferences (e.g., ICSE, ICSME, SANER, ICPC, SCAM, CASCON, and IWSC). He has been a reviewer of major Computer Science journals including IEEE Transactions on Software Engineering, International Journal of Software Maintenance and Evolution, Science of Computer Programming, Journal of Information and Software Technology and so on. He received his Ph.D. at Queen's University, advised by James R. Cordy, in August 2009.

Kevin A. Schneider

Kevin Schneider is a Professor of Computer Science, Special Advisor ICT Research and Director of the Software Engineering Lab at the University of Saskatchewan. Dr. Schneider has previously been Department Head (Computer Science), Vice-Dean (Science) and Acting Chief Information Officer and Associate Vice-President Information and Communications Technology.

Before joining the University of Saskatchewan, Dr. Schneider was CEO and President of Legasys Corp., a software research and development company specializing in design recovery and automated software engineering. His research investigates models, notations and techniques that are designed to assist software project teams develop and evolve large, interactive and usable systems. He is particularly interested in approaches that encourage team creativity and collaboration.


Commits
C_i
C_{i+1}
C_{i+2}
C_{i+3}
C_{i+4}
CF1
CF2
Creation of CF1
Changes occurred in CF1
Creation of CF2 from CF1
Bug-fix changes occurred in CF1 and CF2


Clone Fragment 1, Revision 1349
Clone Fragment 1, Revision 1350
Change
Clone Fragment 2, Revision 1349
Clone Fragment 2, Revision 1350
Change

```
public SortedSet getBuiltInInputFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.values().iterator(); i.hasNext(); ) {
   ImportFormat format = (ImportFormat)i.next();
   if (!format.getIsCustomImporter()) {
    result.add(format);
   }
  }
  return result;
}
```

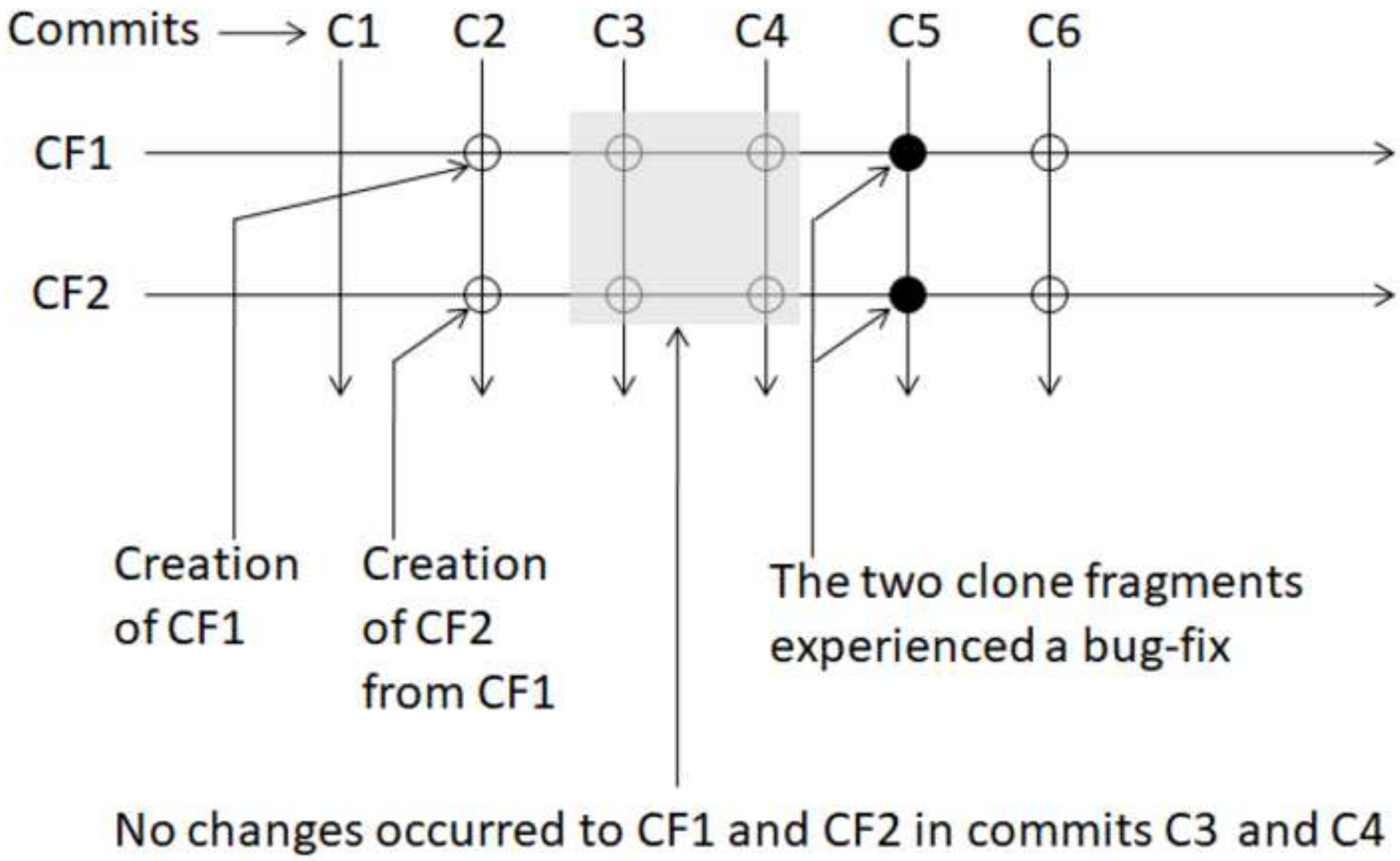
```
public SortedSet getBuiltInInputFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.iterator(); i.hasNext(); ) {
   ImportFormat format = (ImportFormat)i.next();
   if (!format.getIsCustomImporter()) {
    result.add(format);
   }
  }
  return result;
}
```

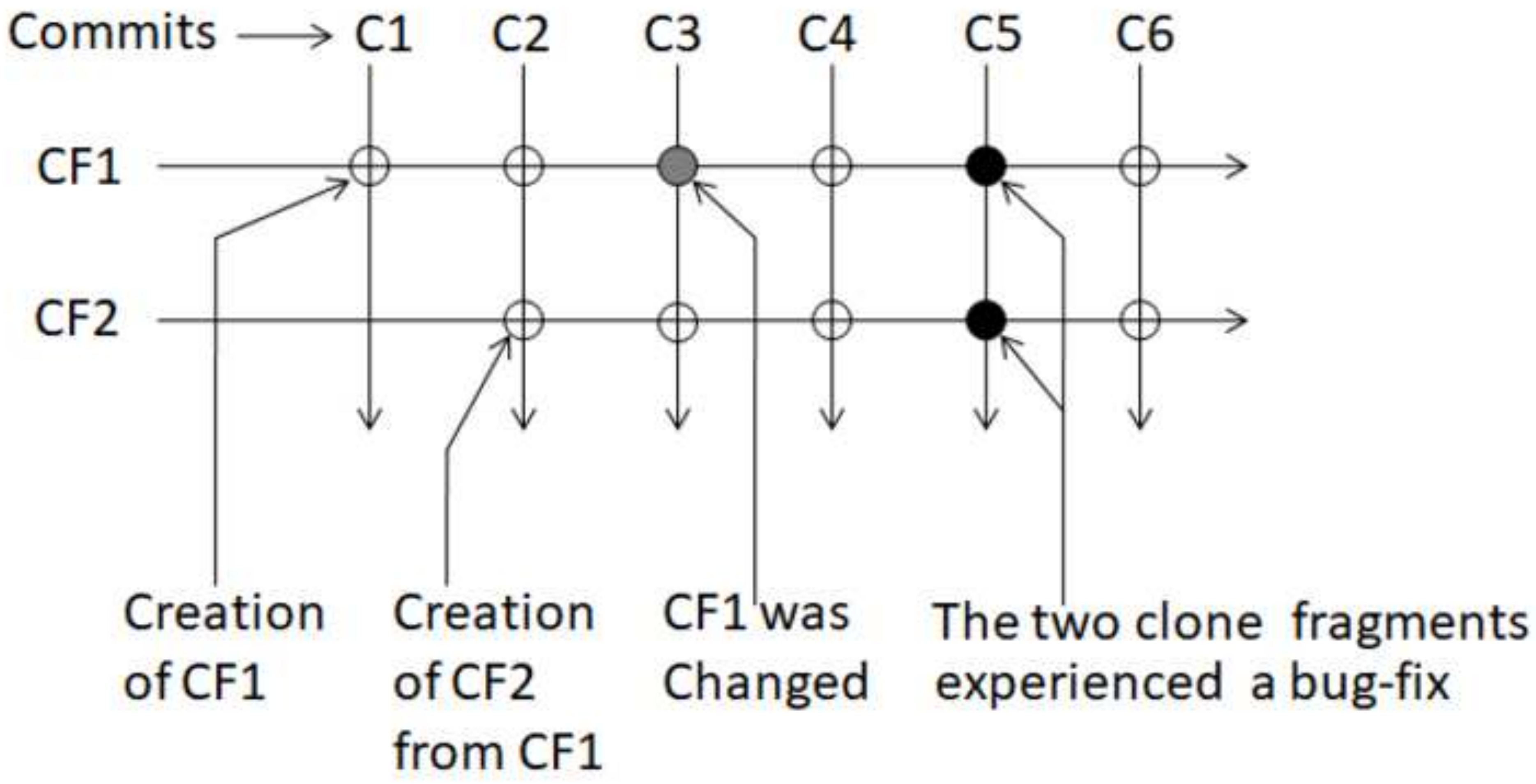
The change in Clone Fragment 1 in the commit operation that was applied on Revision 1349

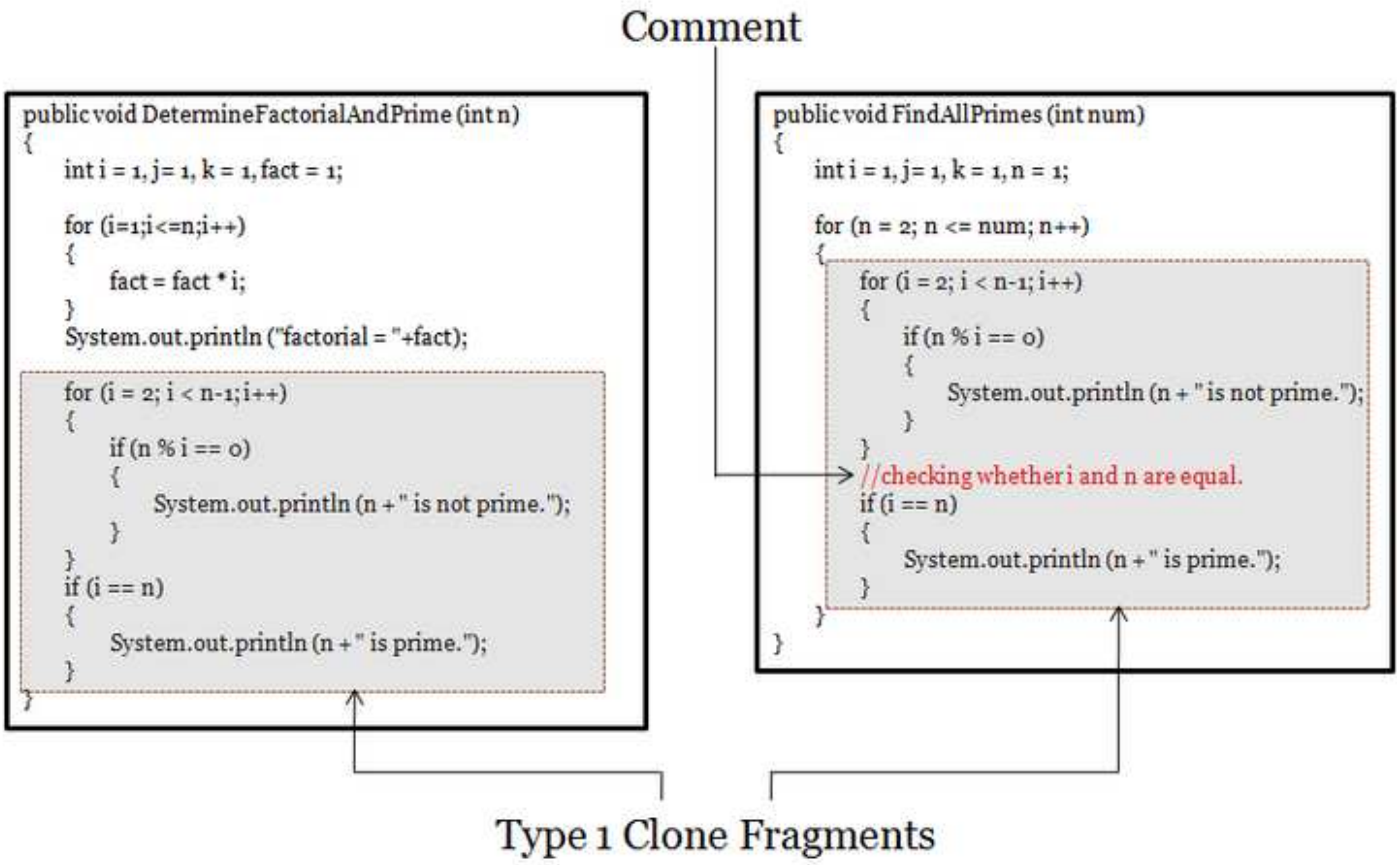
```
public SortedSet getCustomImportFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.values().iterator(); i.hasNext(); ) {
   ImportFormat format = (ImportFormat)i.next();
   if (format.getIsCustomImporter()) {
    result.add(format);
   }
  }
  return result;
}
```

```
public SortedSet getCustomImportFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.iterator(); i.hasNext(); ) {
   ImportFormat format = (ImportFormat)i.next();
   if (format.getIsCustomImporter()) {
    result.add(format);
   }
  }
  return result;
}
```

The change in Clone Fragment 2 in the commit operation that was applied on Revision 1349


Commits
C1
C2
C3
C4
C5
C6
CF1
CF2
Creation of CF1
Creation of CF2 from CF1
The two clone fragments experienced a bug-fix
No changes occurred to CF1 and CF2 in commits C3 and C4


Commits
C1
C2
C3
C4
C5
C6
CF1
CF2
Creation of CF1
Creation of CF2 from CF1
CF1 was Changed
The two clone fragments experienced a bug-fix



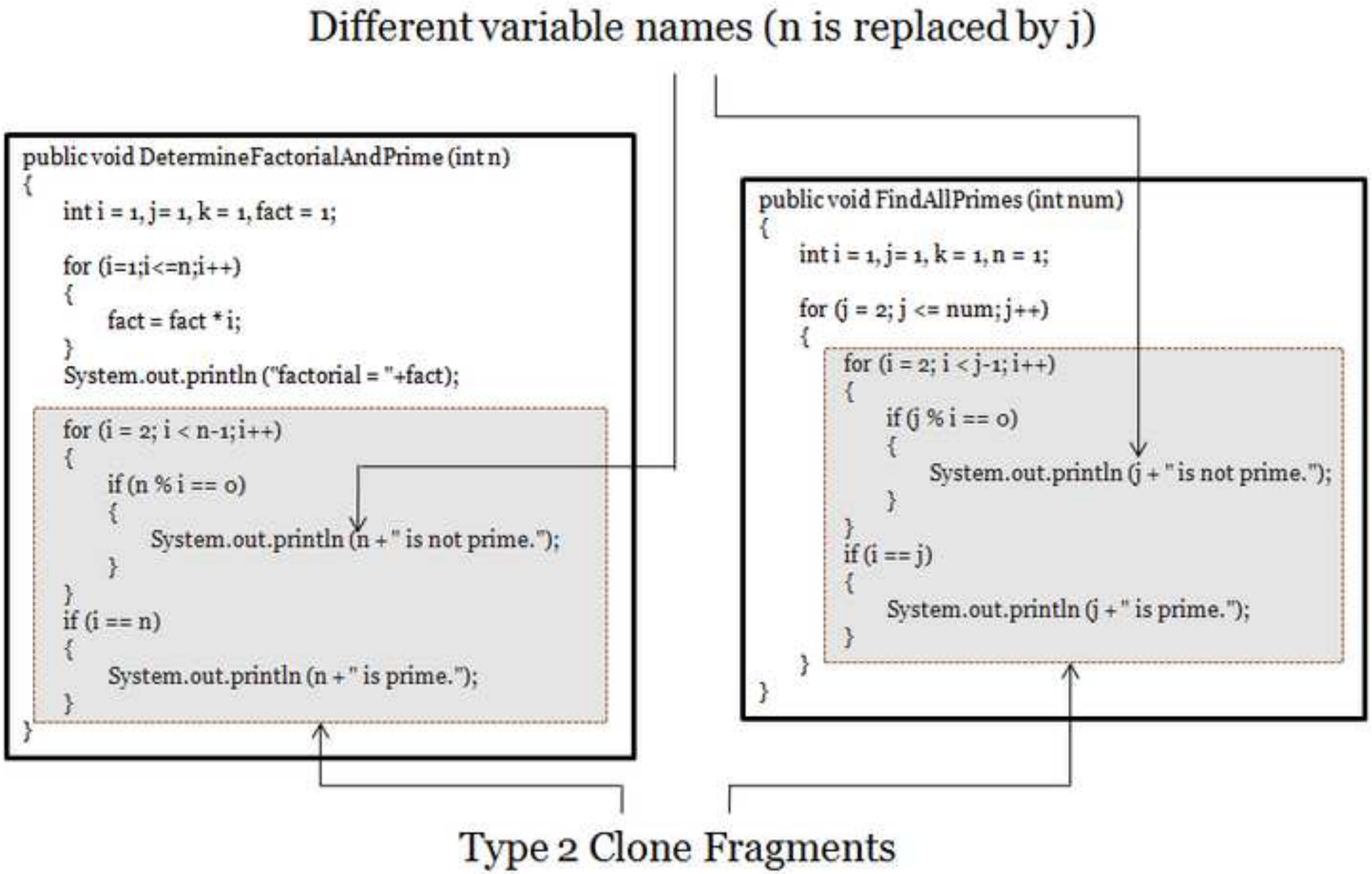
```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 1, n = 1;

    for (n = 2; n <= num; n++)
    {
        for (i = 2; i < n-1; i++)
        {
            if (n % i == 0)
            {
                System.out.println (n + " is not prime.");
            }
        }
        //checking whether i and n are equal.
        if (i == n)
        {
            System.out.println (n + " is prime.");
        }
    }
}
```



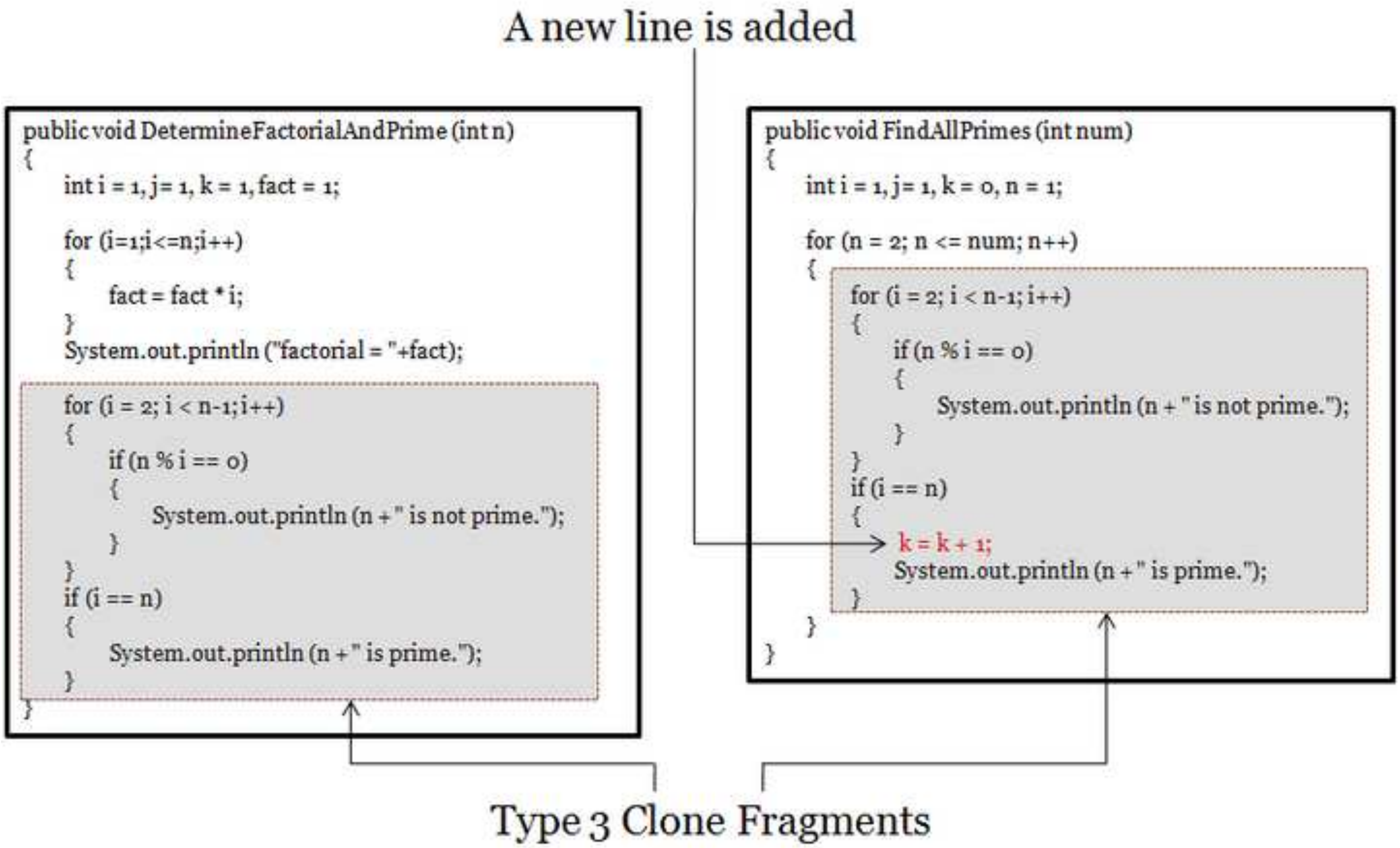
```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 1, n = 1;

    for (j = 2; j <= num; j++)
    {
        for (i = 2; i < j-1; i++)
        {
            if (j % i == 0)
            {
                System.out.println (j + " is not prime.");
            }
        }
        if (i == j)
        {
            System.out.println (j + " is prime.");
        }
    }
}
```



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 0, n = 1;

    for (n = 2; n <= num; n++)
    {
        for (i = 2; i < n-1; i++)
        {
            if (n % i == 0)
            {
                System.out.println (n + " is not prime.");
            }
        }
        if (i == n)
        {
            k = k + 1;
            System.out.println (n + " is prime.");
        }
    }
}
```

LaTeX Source Files

Click here to download LaTeX Source Files: commitmessages.tex