A Study of Consistent and Inconsistent Changes to Code Clones

Jens Krinke
FernUniversität in Hagen, Germany
krinke@acm.org

Abstract

Code Cloning is regarded as a threat to software maintenance, because it is generally assumed that a change to a code clone usually has to be applied to the other clones of the clone group as well. However, there exists little empirical data that supports this assumption. This paper presents a study on the changes applied to code clones in open source software systems based on the changes between versions of the system. It is analyzed if changes to code clones are consistent to all code clones of a clone group or not. The results show that usually half of the changes to code clone groups are inconsistent changes. Moreover, the study observes that when there are inconsistent changes to a code clone group in a near version, it is rarely the case that there are additional changes in later versions such that the code clone group then has only consistent changes.

1 Introduction

Duplicated code is common in all kind of software systems. Although cut-copy-paste (-and-adapt) techniques are considered bad practice, every programmer uses them.

Since these practices involve both duplication and modification, they are collectively called *code cloning*. While the duplicated code is called a *code clone*. A *clone group* consists of code clones that are clones of each other (sometimes this is also called a clone class). During the software development cycle, code cloning is both easy and inexpensive (in both cost and money). However, this practice can complicate software maintenence in the following ways:

- Errors may have been duplicated (cloned) in parallel with the cloned code.

- Modifications done to the original code must often be applied to the cloned code as well.

Because of these problems, research has developed many approaches to detect cloned code [5, 6, 9, 12, 16–18, 20]. In addition, some empirical work done has attempted to check whether or not the above mentioned problems are relevant in practice. Kim et al. [15] investigated the evolution of code clones and provided a classification for evolving code clones. Their work already showed that during the evolution of the code clones, consistent changes to the code clones of a group are fewer than anticipated. Aversano et al. [4] did a similar study and they state “that the majority of clone classes is always maintained consistently.” Geiger et al. [10] studied the relation of code clone groups and change couplings (files which are committed at the same time, by the same author, and with the same modification description), but could not find a (strong) relation. Therefore, this work will present an empirical study that verifies the following hypothesis:

> *During the evolution of a system, code clones of a clone group are changed consistently.*

Of course, a system may contain bugs where a change has been applied to some code clones, but has been forgotten for other code clones of the clone group. For stable systems it can be assumed that such bugs will be resolved at a later time. This results in a second hypothesis:

> *During the evolution of a system, if code clones of a clone group are not changed consistently, the missing changes will appear in a later version.*

This work will verify the two hypotheses by studying the changes that are applied to code clones during 200 weeks of evolution of five open source software systems. The contributions of this paper are:

- A large empirical study that examines the changes to code clones in evolving systems. This study involves both a greater number and diversity of systems than previous empirical studies.

- The study will show that both hypotheses are not generally valid for the five studied systems. In summary, clone groups are changed consistently in roughly half of the time, invalidating the first hypothesis. The second hypothesis is only partially valid. This is because only a small amount of the missing changes can be observed, while the inconsistent changes appear much more often.

The next section presents the theoretical framework that defines changes, code clones, and clone groups. The setup of the empirical study is presented in Section 3 and its results in Section 4. After Section 5 discusses related work, the last section will conclude and present ongoing activities.

2 A Framework for Changes to Clones

This section will present the framework in which code clones, groups of code clones, and changes to code clones are defined and related to the evolution of software systems in terms of the versions of the system.

2.1 Code Clones

Code clones are usually described as source code ranges (or fragments) that are identical or very similar. They are grouped into *clone groups* (sometime called *clone classes*) which are sets of identical or very similar code clones. A code clone $c = (s, l, f)$ is the source code range starting at line s with the following l lines of code in file f, thus the last line of the code clone is $s + l - 1$. A clone group $G = \{c_1, \ldots, c_n\}$ is a set of n code clones $c_1, \ldots, c_n$, where each of the code clones is a clone of the others. For the purpose of this study, the effects of *split* or *fragmented* code clones are ignored. Such clones would consist of multiple source code ranges in the same file. An example of such a code clone is a source code range that is copied and afterward additional source code is inserted into the cloned code.

The code clones don't have to be disjunct: it is possible for two code clones $c_1 = (s_1, l_1, f)$ and $c_2 = (s_2, l_2, f)$ that they share a common source range ($min(s_1 + l_1, s_2 + l_2) > max(s_1, s_2)$).

Most of the available tools for code clone detection generate a list of clone groups. Usually, a minimal size k of an identified code clone $c = (s, l, f)$ can be specified, i.e. $size(c) > k$. In the following it is assumed that $size(c) = l$. However, many tools use the number of lexical tokens that is covered by a code clone as the size of the code clone.

2.2 Changes

Changes to a software are usually described as a source code range that has been replaced by other source code. A change $d = (s, l, f, n, t)$ is the source code range starting at line s in file f with a size $l \geq 0$ (number of lines) that will be replaced by the text t with a size of $n \geq 0$ (number of lines). Note that a change is not specified with the last line number because that would not allow the specification of an empty range. The kind of change is usually distinguished between *deletion*, *addition*, or *change*. Because addition and deletion are special cases with $l = 0$ or $n = 0$ resp. this distinction will not be made in the following. The changes to a software can be expressed by a set of changes $D = \{d_1, \ldots, d_k\}$ where the d_i don't overlap.

2.3 Software Systems and their Version History

For the purpose of the study, a software system $S = \{f_1, \ldots, f_n\}$ consists of a set of n source code files $f_i, 1 \leq i \leq n$. Independent of the specific kind of version, it is assumed that a system exists in multiple versions v where the complete system can be retrieved for every version v: $S(v) = \{f_1^v, \ldots, f_n^v\}$ is the system in version v. In a versioning system like CVS, the version v can be specified in a very flexible way, for example it can be specified as a time or by a name given to a specific version (branching will be ignored in this work). The differences between two versions x and y of a system can be identified by a set of changes as described above: Let $D(v, w)$ be the set of changes $\{d_1, \ldots, d_k\}$ between $S(v)$ and $S(w)$.

2.4 Consistent and Inconsistent Changes

In this study, the changes to a system S between two versions v and w will be analyzed with respect to the clones in the system. Let $c(v)$ be the set of code clones $\{c_1, \ldots, c_n\}$ in $S(v)$, and let $G(v)$ be the set of clone groups $\{G_1, \ldots, G_m\}$ in system $S(v)$. The first step is to identify the changes that overlap with a code clone: A change $d_i = (s_i, l_i, f_i, n_i, t_i)$ is relevant to a code clone $c_j = (s_j, l_j, f_j)$ if the source code ranges overlap, i.e.

$$\begin{aligned} f_i = f_j \quad \wedge \quad & (max(s_i, s_j) < min(s_i + l_i, s_j + l_j) \\ & \vee (l_i = 0 \wedge s_j < s_i < s_j + l_j - 1)) \end{aligned}$$

Note that additions ($l_i = 0$) have to be handled explicitly and are ignored if they appear at the beginning or end of the code clone.

The next step is to map the changes that are relevant to a code clone on the source range of the code clone such that the start and size of the change is expressed in respect to the start of the code clone. The set of (mapped) changes between version v and w that are relevant to a code clone c_j is then the set

$$\begin{aligned} \delta(c_j, v, w) \quad = \quad & \{(s', l_i, n_i, t_i) \\ | \quad & c_j = (s_j, l_j, f) \\ \wedge \quad & d_i = (s_i, l_i, f, n_i, t_i) \in D(v, w) \\ \wedge \quad & (max(s_i, s_j) < min(s_i + l_i, s_j + l_j) \\ & \vee (l_i = 0 \wedge s_j < s_i < s_j + l_j - 1)) \\ \wedge \quad & s' = s_i - s_j\} \end{aligned}$$

The set of mapped changes will contain two different kinds of mapped changes: the change may be completely embedded in the code clone ($0 \leq s' \leq s' + l_i \leq s_j + l_j$) or it starts before ($s' < 0$) or ends after the code clone ($s' + l_i > s_j + l_j$).

For better understanding, first assume that all mapped changes are completely embedded in their code clone. In this case it is easy to distinguish if a clone group only has consistent changes to its code clones or not: Consider a clone group $G = \{c_1, \ldots, c_k\} \in G(v)$. The changes between versions v and w to the clone group's code clones are consistent changes, if each code clone has the same set of mapped changes:

$$\forall c_i \in G : \forall c_j \in G : \delta(c_i, v, w) = \delta(c_j, v, w)$$

Things are more complicated with changes that are not completely embedded in their code clone. In this case, it is better not only to map, but also to reduce the change to the part that is completely embedded in the code clone. This will cut the part of the change that starts before the code clone and the part that ends after the code clone. The cutting will be performed by the cutting function $\gamma(c, d)$ which maps or reduces a change $d = (s, l, f, n, t)$ into a code clone c. The reduction of the source range is a simple mathematical operation, however special care has to be take of the added text specified by t for $n > 0$:

- If the change starts before the code clone, it could represent that the beginning of the code clone is deleted and the added text is added to the source code before the clone.
- If the change ends after the code clone, it could represent that the end of the code clone is deleted and the added text is added to the source code after the clone.
- If the change starts before the code clone and ends after it, it could represent that the code clone is deleted completely and replaced by different source code.

Because the kind of the change is not obvious, it is assumed that the above cases always apply. Therefore, the cutting function will replace the change with a deletion. The cutting function is now defined as ($c = (s_j, l_j, f)$ and $d = (s_i, l_i, f_i, n_i, t_i)$):

$$\gamma(c, d) = \begin{cases} (s_i - s_j, l_i, n_i, t_i) \\ \quad \text{if } (s_j < s_i) \wedge (s_j + l_j > s_i + l_i) \\ (max(s_i - s_j, 0), \\ min(s_i + l_i, s_j + l_j) - s_j, \\ 0, \emptyset) \\ \quad \text{otherwise} \end{cases}$$

This function maps the source range of the change into the code clone if the change is completely embedded in the code clone. Otherwise, the source range is reduced and the change operation is replaced with a deletion.

The set of mapped and reduced changes between versions v and w that are relevant to a code clone c_j is now the set

$$\begin{array}{rcl} \Delta(c_j, v, w) & = & \{\gamma(c_j, d_i) \\ & | & c_j = (s_j, l_j, f) \\ & \wedge & d_i = (s_i, l_i, f, n_i, t_i) \in D(v, w) \\ & \wedge & (max(s_i, s_j) < min(s_i + l_i, s_j + l_j) \\ & & \vee\ (l_i = 0 \wedge s_j < s_i < s_j + l_j - 1))\} \end{array}$$

Again, consider a clone group $G = \{c_1, \ldots, c_k\} \in G(v)$. The changes between version v and w to the clone group's code clones are consistent changes, if each code clone of the group has the same set of mapped and reduced changes:

$$\forall c_i \in G : \forall c_j \in G : \Delta(c_i, v, w) = \Delta(c_j, v, w)$$

The set of clone groups for a system $S(v)$ with consistent changes between version v and w is now

$$\begin{array}{rcl} GC(v, w) = \{G \in G(v) & | & \forall c_i \in G : \forall c_j \in G : \\ & & \Delta(c_i, v, w) = \Delta(c_j, v, w) \\ & & \wedge\ \Delta(c_i, v, w) \neq \emptyset\} \end{array}$$

The set of clone groups with inconsistent changes between version v and w is

$$\begin{array}{rcl} GI(v, w) = \{G \in G(v) & | & \exists c_i \in G : \exists c_j \in G : \\ & & \Delta(c_i, v, w) \neq \Delta(c_j, v, w)\} \end{array}$$

And the set of clone groups with changes between version v and w is simply

$$GD(v, w) = \{G \in G(v) \mid \exists c_i \in G : \Delta(c_i, v, w) \neq \emptyset\}$$

With the now defined framework it is possible to analyze the version history of a system and identify the sets of clone groups with consistent and inconsistent changes from version to version.

This framework is general enough for the usual line based clone detection, change identification, and version control tools. However, because of the matching of changes to clones and the comparison of changes, the framework is to be used only with tools that identify identical or parameterized clones [7]. The advantage of this restriction is that the distinction between consistently and inconsistently changed clone groups is automated.

Although the framework is defined in terms of textual properties, it can be used with clone detectors based on syntax or semantics. Defining the framework in syntactic or semantic notions would prevent its usage with source code that is not parseable.

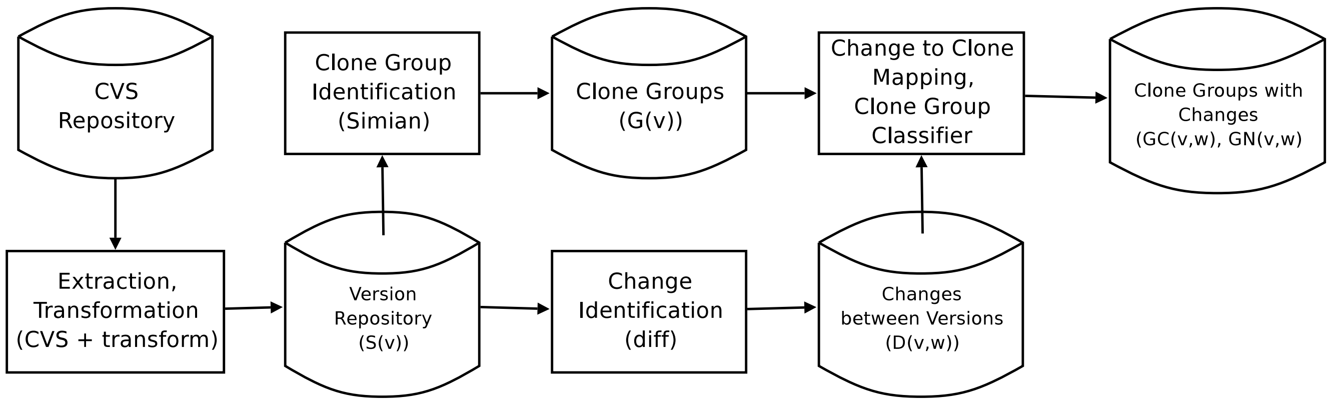
CVS Repository
Clone Group Identification (Simian)
Clone Groups (G(v))
Change to Clone Mapping, Clone Group Classifier
Clone Groups with Changes (GC(v,w), GN(v,w)
Extraction, Transformation (CVS + transform)
Version Repository (S(v))
Change Identification (diff)
Changes between Versions (D(v,w))

Figure 1. General Setup of the Study

3 Experiment Setup

For the study the version histories of five open source systems have been retrieved, which have a sufficiently long development history:

- The first system is ArgoUML[1]. It is a UML modeling tool that includes support for standard UML diagrams. It is written in Java and its version archive is available via subversion.
- The second system is CAROL[2], a library allowing to use different RMI implementations. Its version archive is also available via CVS and the `carol` CVS-module has been used.
- The third system is the JDT core subsystem of Eclipse[3]. From Eclipse's version archive the `org.eclipse.jdt.core` CVS-module has been used.
- The fourth system is GNU Emacs[4], the famous text editor. It is written in C.
- The fifth system is FileZilla[5], a FTP client with a graphical user interface for Windows, written in C++.

All five systems are large enough and have enough changes in their version archive within the 200 observed weeks. Moreover, they cover different applications, different platforms, and different programming languages. Two of the systems, ArgoUML and CAROL, have been used in previous studies by Kim et al. [15] and Aversano et al. [4]. A sixth system has been prepared for the study because both previous studies have analyzed it: DNSJava, an implementation of DNS in Java. However, this system had too few and too small changes affecting clones and thus has been rejected.

The sources of all five systems have been retrieved based on their status in the version archive on 200 different dates, such that each version is exactly one week later or earlier than the next or previous version. A one week cycle has been chosen because CVS activity is usually dependent on the weekday [22] and projects often use a week oriented process (e.g. within Eclipse). For all systems, the first version was from 2002-08-08 and the last version was from 2006-06-01.

In all systems, only the Java, C, and C++ source and header files have been analyzed. Also, the source files have been transformed to eliminate spurious changes between versions: Comments have been removed from the sources and afterward the source files have been reformatted with the pretty printer *Artistic Style*[6]. The transformed sources are saved to a repository. With this repository, all $S(v), 0 \leq v < 200$ can be accessed.

The changes between the versions of the systems have been identified by the standard *diff* tool. For each version v of the analyzed system, the changes between version v and $v + 1$ (the version of the next week) have been identified, generating $D(v, v + 1)$.

For each of the 200 versions, the clone groups $G(v)$ have been identified by the use of the clone detection tool *Simian*[7] from RedHill Consulting Pty. Ltd. It is a text-based clone detector that detects almost identical clones. The possibility to relax the identification by assuming that all literals are identical has not been used. The decision to use *Simian* was based on the following requirements for the clone detector:

[1]`http://argouml.tigris.org/`
[2]`http://carol.objectweb.org/`
[3]`http://www.eclipse.org/`
[4]`http://www.gnu.org/software/emacs/`
[5]`http://filezilla.sourceforge.net/`

[6]`http://astyle.sourceforge.net/`
[7]Available at `http://www.redhillconsulting.com.au/products/simian/index.html`

System	Source LOC	Changes LOC	Clones LOC		Groups
ArgoUML	118366	2816	14862	13%	313
CAROL	9824	248	601	6%	17
jdt.core	192930	2478	29438	15%	644
Emacs	227964	578	22966	10%	528
FileZilla	90138	698	14362	16%	210

Table 1. Analyzed Systems

1. It has to analyze Java, C, and C++ source files.
2. It has to be freely available.
3. It has to be fast.
4. It has to have textual output of the results for further processing.
5. It has to be usable with a batch processor.
6. It has to match the above defined framework.

Most of the other (freely) available clone detectors cannot be used because they require the use of a GUI or are restricted to Java source files. *Simian* has been instructed to identify clones with a size of at least 11 source code lines (the reason for this choice will be discussed below).

The framework described in the previous section has been implemented in a tool that takes a list of clone groups $G(v)$ as detected by *Simian* and a list of changes $D(v, w)$ as produced by *diff* that are then mapped and reduced on the code clones. The tool will produce a list of clone groups that only have consistent changes ($GC(v, w)$) and a list of clone groups with inconsistent changes ($GI(v, w)$). Figure 1 shows the setup of the study and the arrangement of the tools.

The analysis has been done on 200 versions. For each week w, $1 \leq w \leq 200$, the tool has generated the list of clone groups with consistent and inconsistent changes ($GC(w-1, w)$, $GI(w-1)$), based on the clone groups of the analyzed system in week $w-1$ and the changes from week $w-1$ to week w.

Table 1 shows some properties of the analyzed systems: The second column contains the average size of the analyzed source base (in LOC) for a week. The next column contains the average number of added and deleted lines from one week to the next. The last three columns contain the average size of cloned source code (in LOC and as an percentage of the source code) and the number of clone groups for a week. For example, GNU Emacs is the largest system with 228 KLOC on average, from which 10% (23 KLOC) is cloned code in 528 clone groups. It is the least active system, as only 0.13% of its source code is changed within a week (578 added or deleted lines).

	$\lvert GC\rvert$	$\lvert GI\rvert$	$\frac{\lvert GI\rvert}{\lvert GD\rvert}$
ArgoUML	1049	1050	50%
CAROL	66	69	49%
jdt.core	1375	1124	55%
Emacs	440	543	45%
FileZilla	246	204	55%

Table 2. Number of Changed Clone Groups

4 Results

This section presents the results of the study as described in the previous section. Table 2 shows the number of clone groups with consistent ($|GC|$) and inconsistent ($|GI|$) changes and the percentage of clone groups that only have consistent changes ($\frac{|GI|}{|GD|}$). For example, ArgoUML has 2099 changed clone groups during the 200 weeks time period, where 1049 are changed consistently and 1050 inconsistently (roughly 3% of all 62600 clone groups are changed). For all five systems, only 45%–55% of the clones groups are changed consistently (roughly half of the clone groups).

4.1 Impact of Minimum Clone Size

These numbers are smaller than expected and it might be the case that the numbers are low because of the study's parameters. First of all, the minimum size of the detected clones might be a reason for the small percentage of consistent changes. It is known that small detected clones are often only incidental clones. Therefore, the study has been repeated for different minimum sizes of detected clones (in a range between 6 and 30 lines). The observed impact of increasing the minimum clone size has been:

- ArgoUML increases, but not monotonously, from 50% to a maximum of 55% (at a minimum clone size of 28 lines).
- CAROL decreases, but not monotonously, from 49% (maximum of 51% at 8).
- jdt.core increases for a short time from 55% to 57% (at 15), but then decreases (not monotonously).
- GNU Emacs increases for a short time from 45% to 46% (at 19), but then decreases (not monotonously).
- FileZilla increases, but not monotonously, from 55% to a maximum of 64% (at 28).

In four of the five systems the percentage of consistent changes increases with an increased minimum size. The different behavior for CAROL can be explained by the small

$n =$	0	1	2	3	4	5
ArgoUML	50%	52%	52%	52%	52%	52%
CAROL	49%	52%	47%	47%	47%	46%
jdt.core	55%	57%	56%	56%	55%	53%
Emacs	45%	47%	48%	49%	50%	50%
FileZilla	55%	58%	58%	59%	59%	61%

Table 3. Impact of Removing Lines

size of the system and the small number of changed clone groups. However, if the minimum limit is very large, all systems have a smaller percentage of consistent changes. If the minimum size is decreased, the percentage of consistently changed groups decreases for all systems except CAROL, for which the percentage stays within 49%–51%. In spite of everything, the impact of the minimum size is not dramatic.

4.2 Impact of Maximization of Clones

Another reason might be related to the way how clone detector tools work (in comparison to human oracles). Like any other clone detection tool, *Simian* tries to detect maximal sized code clones. This can result in code clones that are actually larger than the 'intended' or 'real' code clone. For example, the code clones may include source code lines at the beginning or end of the clone that are only identical by incident. If a change is now applied in such regions, it may be detected as an inconsistent change, while the 'intended' or 'real' code clone is changed consistently or not at all. Such a change is classified as irrelevant for the study. To eliminate this kind of problems, the code clones as listed by *Simian* can be reduced before further analysis: a number of source code lines at the beginning and the end of the detected code clone can be removed. In the presented study, five additional experiments have been done: At the beginning and at the end of the detected cloned between one and five lines are removed automatically (with a minimal clone size of 11 and five removed lines at the beginning and the end of a clone, only a single line is left of a minimal sized clone).

The observed impact is shown in Table 3, where each of the column 0–5 show the percentage of consistently changed clone groups when n number of lines at the beginning and the end of the clone are removed. If only one line is removed, the percentage of consistently changed groups increase as expected. However, for larger numbers of removed lines, there is no common trend:

- ArgoUML does not change and stays at 52%
- CAROL decreases to 46% with 5 removed lines
- jdt.core decreases to 53% with 5 removed lines
- GNU Emacs increases to 50% with 5 removed lines
- FileZilla increases to 61% with 5 removed lines

Again, the changes in percentage is not dramatic and the general observation is that a little bit more than half of clone groups with changes are consistently changed.

4.3 Impact of Change Detection

Another reason of the lower than expected percentages might be related to the way how changes are compared. In the presented setup, the changes are first computed by the *diff* tool. The risk that two changes are considered to be different changes while they are identical changes is decreased by a combination of techniques:

- The original sources are transformed as described above in the Introduction. Thus, changes cannot be different due to comments, indentation or formatting.
- Whitespace like linebreaks, tabulators, etc. are removed before changes are compared.

This reduces the risk, but does not eliminate it. Therefore, a manual inspection for the systems CAROL, GNU Emacs, and FileZilla has been done to examine the 332 critical changes where the same source lines of every clone in a group are replaced by different source text. Almost all of the changes are very similar, but most of them cause subtle semantic differences. Typical examples are different parameters for the same method call, differences in thrown exceptions, or differences in predicates in otherwise identical if- or while-statements.

These observations seems to indicate that a large part of clones start as identical clones and evolve independently, often with just subtle differences.

4.4 Effects of Comments and Indentation

As mentioned in the previous section, the source code has been transformed by comment removal and reformatting. The intention was to get rid of spurious changes. The same analysis has been applied to the original sources to see if the transformation has the intended effect.

Table 4 shows the overall numbers for the original and the transformed source code. It can be seen that the number of changed clone groups increases, while the percentage of consistently changed clone groups drops. The most dramatic change can be seen for ArgoUML, where the number of changed groups doubles while the percentage of consistently changed clone groups drops from 50% to 30%. Overall, the numbers show that the transformation of the

	Transformed			Original		
	$\vert GC\vert$	$\vert GI\vert$	$\frac{\vert GI\vert}{\vert GD\vert}$	$\vert GC\vert$	$\vert GI\vert$	$\frac{\vert GI\vert}{\vert GD\vert}$
ArgoUML	1049	1050	50%	1266	2988	30%
CAROL	66	69	49%	77	170	31%
jdt.core	1375	1124	55%	1416	2194	39%
Emacs	440	543	45%	480	1006	32%
FileZilla	246	204	55%	270	316	46%

Table 4. Comparison of Results for Original and Transformed Source Code

source code will result in fewer detected changes within clone groups, thus the transformation achieves the desired improvement.

4.5 Evolution of Changed Clone Groups

Up to now, the presented results only gave evidence for the evaluation of the first hypothesis. The second hypothesis "*During the evolution of a system, if code clones of a clone group are not changed consistently, the missing changes will appear in a later version*" has also been evaluated within this study. If this hypothesis was true, changes to a clone group that are not applied to all code clones of a group would appear in a later version. Until now, the presented study only considered the changes to a system that appear within one week. If the hypothesis would be true, there have to be more consistent changes if a longer time period is considered. To verify this, the study has been repeated with a different time distance between one and 10 weeks.

Table 5 shows the numbers of clone groups changed within one week (same numbers as in Table 4) and the number of clone groups changed within 2, 4, 7, and 10 weeks.

The numbers show that during larger periods much more changes occur than within only the first week. However, the percentage of consistent changes is very stable and does not change much. This indicates that the second hypothesis is not valid. If it was valid, the percentage of consistent changes would increase. There are exceptions to this general observation as a manual inspection revealed. However, the number of these exceptions is too low to be observable in general.

It can be concluded that both hypotheses cannot be generally observed in practice. The study showed that clone groups in general have roughly the same number of inconsistent changes or consistent changes, invalidating the first hypothesis. The study also showed that the second hypothesis is also not valid in general, because inconsistently changed clone groups that become consistently changed clone groups later can be found rarely.

4.6 Threats to Validity

There are some potential threats to validity in the presented study. First of all, there is no clear definition of a clone. Moreover, a clone detected by a clone detector may not be a clone in reality (false positive) or a clone in a system may be missed by a clone detector (false negative). To reduce the number of false positives, we have used *Simian* with strict settings such that only identical clones are detected. Moreover, the analyzed systems have been transformed by removing comments and pretty printing. It is known that clone detectors have a low recall [7], so the false negative cause a threat to validity which cannot be estimated. Another potential threat to validity is caused by the technique to detect changes with *diff*, however, the risk is reduced by transformation of the analyzed systems and by ignoring whitespace in changes.

The experiment is also influenced by the analyzed systems. To be able to draw general conclusions, five systems have been chosen that are of different application types, written in different programming languages, are of sufficient size, and went through enough changes.

Other parameters that influence the experiment, like the choice of a minimum clone size, have been studied and are shown to have only a small impact.

5 Related Work

There are only a few empirical studies that analyze the effect of changes on the code clones of a system. Geiger et al. [10] studied the relation of code clones and change couplings (files which are committed at the same time, by the same author, and with the same modification description), but could not find a (strong) relation. The results of the previous section give reasons why Geiger et al. could not find such a relation: the amount of consistent changes is too low.

Kim et al. [15] investigated the evolution of code clones and provided a classification for evolving code clones. Their work already showed that during the evolution of the code clones, consistent changes are fewer than anticipated. However, the study analyzed the evolution of two very small systems, DNSJava and CAROL, both written in Java, and both are a similar type of application.

Aversano et al. [4] did a similar empirical study with a slightly refined framework. Similar to Kim et al., they analyze so called co-changes that are changes committed by the same author, with the same notes, and within 200 seconds. They used a Java-only clone detector that compares subtrees in the abstract syntax tree. The analyzed systems were DNSJava and ArgoUML. Although Aversano et al. state "that the majority of clone classes is always maintained consistently", the numbers they present contradict this statement: For ArgoUML, they found that 45% of the

	1 week			2 weeks			4 weeks			7 weeks			10 weeks		
	$\|GC\|$	$\|GI\|$	$\frac{\|GI\|}{\|GD\|}$	$\|GC\|$	$\|GI\|$	$\frac{\|GI\|}{\|GD\|}$	$\|GC\|$	$\|GI\|$	$\frac{\|GI\|}{\|GD\|}$	$\|GC\|$	$\|GI\|$	$\frac{\|GI\|}{\|GD\|}$	$\|GC\|$	$\|GI\|$	$\frac{\|GI\|}{\|GD\|}$
ArgoUML	1049	1050	50%	1998	1981	50%	3703	3668	50%	6002	5869	51%	8059	7801	51%
CAROL	66	69	49%	115	124	48%	211	221	49%	338	350	49%	447	461	49%
jdt.core	1375	1124	55%	2629	2114	55%	4842	3996	55%	7706	6565	54%	10097	8911	53%
Emacs	440	543	45%	854	1074	44%	1680	2041	45%	2802	3553	44%	3699	5204	42%
FileZilla	246	204	55%	464	381	55%	819	697	54%	1277	1111	53%	1670	1477	53%

Table 5. Comparison for inconsistent changes during different weeks

clone groups underwent consistent changes – similar to the 50% we found above. It is worth mentioning that Aversano only had to inspect 237 clone groups in a similar time period, while we found 2099 changed clone groups. The reasons for this huge difference is not clear.

Our studies have found a little bit higher percentages of clone groups to be consistently changed. This may be related to the transformation of the sources. Besides the above mentioned empirical studies, there is some not directly related work that focuses on the evolution of systems and the contained code clones, without looking at the changes:

Antoniol et al. [3] have analyzed the cloning evolution in the Linux kernel for 19 releases. They found that the Linux system does not contain a relevant fraction of code duplication. Furthermore, they found that code duplication tends to remain stable across releases. Lagüe et al. [19] have analyzed the amount of clones for different versions of a large telecommunication switching software. An experience in applying time series to cloning ratio prediction was presented by Antoniol et al. [2]. Al-Ekram et al. [1] investigated the code cloning across software systems.

Kim et al. [14] studied why and how programmers introduce code clones into software systems. Lagüe et al. [19] show how software development could benefit from the inclusion of code clone detection tools into the development process. The relation of code clones to the reliability and maintainability of a system has been examined by Monden et al. [21].

Jarzabek and Li [11] found that at least 68% of the Java Buffer library's code was contained in cloned classes or class methods. Close analysis of program situations that led to cloning revealed difficulties in eliminating clones with conventional program design techniques. Kapser and Godfrey [13] list several patterns of cloning that are used in real software systems and argue that clones can be a reasonable design decision.

Many approaches and tool for the detection of code clones have been developed. They can be grouped in based on the underlying technique: lexical analysis [5, 9, 12], source code metrics [17, 20], and comparison of structures like the abstract syntax tree [6] or program dependence graph [16, 18]. Comparative studies have been done by Burd and Bailey [8], Bellon [7], and Van Rysselberghe and Demeyer [23, 24].

6 Conclusions and Future Work

The presented study verified two hypotheses that are usually stated as reasons why cloning of source code is considered a threat to software maintenance:

1. *During the evolution of a system, code clones of a clone group are changed consistently.*
2. *During the evolution of a system, if code clones of a clone group are not changed consistently, the missing changes will appear in a later version.*

The study observed five large open source systems based on an introduced framework that defines code clones, clone groups, changes, and how changes apply to the code clones of a clone group.

The study showed that the hypotheses are not valid generally. The study showed that clone groups are consistently changed in roughly half of the time—invalidating the first hypothesis. The study also showed that the second hypothesis is also not valid in general, because inconsistently changed clone groups that become consistently changed clone groups later can be found rarely.

Currently, the study is expanded with the analysis of more and larger systems. It is also planned to use other clone detection tools than *Simian* to achieve more general results.

References

[1] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *International Symposium on Empirical Software Engineering*, 2005.

[2] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proc. Int'l Conf. Software Maintenance (ICSM'01)*, pages 273–280, Nov. 2001.

[3] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, Oct. 2002.

[4] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proc. 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, 2007.

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings: Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings; International Conference on Software Maintenance*, pages 368–378, 1998.

[7] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diplomarbeit, Universität Stuttgart, 2002. (In German).

[8] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pages 36–43, 2002.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 109–118, 1999.

[10] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, number 3922 in LNCS, pages 411–425. Springer, Mar. 2006.

[11] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution*, 18(4):267–292, 2006.

[12] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[13] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 19–28, 2006.

[14] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[15] M. Kim, V. Sazawal, and D. Notkin. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pages 187–196, 2005.

[16] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Eigth International Static Analysis Symposium (SAS)*, volume 2126 of *LNCS*, 2001.

[17] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings Fourth Working Conference on Reverse Engineering*, pages 44–54, 1997.

[18] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. Eigth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[19] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. Int'l Conf. Software Maintenance (ICSM'97)*, pages 314–321, 1997.

[20] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–254, 1996.

[21] A. Monden, D. Nakae, T. Kamiya, S. ichi Sato, and K. ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Eighth IEEE International Symposium on Software Metrics (METRICS'02)*, 2002.

[22] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? on fridays. In *International Workshop on Mining Software Repositories (MSR)*, 2005.

[23] F. van Rysselberghe and S. Demeyer. Evaluating clone detection techniques. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Applications*, 2003.

[24] F. van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th International Conference on Automated Software Engineering*, pages 336–339, 2004.