Tracking clones' imprint

Tracking clones' imprint

Angela Lozano *
Université Catholique de Louvain
2 Place Sainte Barbe
Louvain La Neuve, Belgium
angela.lozano@uclouvain.be

Michel Wermelinger
Computing Department
The Open University
Milton Keynes MK7 6AA, UK
http://michel.wermelinger.ws

ABSTRACT

Cloning imprint is the lasting effect of cloning on applications. This paper aims to analyze the clone imprint over time, in terms of the extension of cloning, the persistence of clones in methods, and the stability of cloned methods. Such level of detail requires an improvement in the clone tracking algorithms previously proposed, which is also presented.

We found that cloned methods are cloned most of their lifetime, cloned methods have a higher density of changes, and that changes in cloned methods tend to be customizations to the clone environment.

Categories and Subject Descriptors

D.2.7.m. [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering and reengineering*

General Terms

Cloning, Mining Software Repositories, Empirical Software Engineering

Keywords

Clones, maintenance, changeability, impact, persistence, stability, extension.

1. INTRODUCTION

To assess the effect of cloning on the ease to change an application, we analyze the evolution of cloned methods. We refer to cloned methods to those methods that contain clones[1], regardless of the coverage of clones with respect to the size of the method. This means that referring to a method as cloned indicates that a portion of the method is similar to a portion of another method.

In previous work we measured changes and co-changes of methods while cloned vs. while not cloned [15], we compared the ease to change cloned methods vs. not-cloned-methods [13], and the relation between method characteristics and their ease to change [14]. Our previous work tried to find trends on the effect of clones *in methods* using measurements that depended on the periods cloned and not cloned of each method analyzed. In contrast, this paper provides a global overview of the effect of cloning *in the application* using measurements that capture commit by commit the evolution of cloning in the application. Furthermore, as opposed to other analyses on the extension of cloning we study the extension of cloning over *the history of the application*. Moreover, as opposed to other analyses on the persistence or stability of cloning, we analyze how the changes observed in clone instances *affect cloned methods*, providing an alternative view on the impact of cloning. The analysis of the imprint of cloning over time would show if its effects grow, shrink or remain stable, as well as the rapidity of this behavior. These imprint evolution trends would permit checking whether or not cloning is a degenerative issue whose consequences become unbearable over time, or if it is a stable phenomenon whose consequences are restricted.

Our hypothesis was that clones have a negligible imprint because previous results showed them as stable areas in the source code [10], that were eliminated soon after they are created [8], and that affected a small percentage of the application (less than 25%) [3, 16, 12]. Nevertheless, we have found that among methods *that change*, those cloned change more than those not cloned which apparently contradicts the findings published in [10]. Moreover, the changes in cloned methods tend to be in lines of code cloned. We also found that methods tend to be cloned most of their lifetime which also contradicts a previous result [8]. Furthermore, we found that cloned methods have a longer lifetime than methods not cloned. In addition, we found that, over time, the percentage of cloning inside methods reduces, and that this reduction might be due to the customization of the clone by adding code that affects the boundaries of cloned fragments. Finally, based on the imprint scale we proposed, it is possible to conclude that *clones that change* can have an important

* This work has been supported by The Open University graduate studentships, the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State, Belgian Science Policy.

[1]A ***clone*** is a fragment of code repeated one or more times in the source code of an application. We use CCFinder [7] to detect similar code fragments of at least 30 tokens.

effect on the changeability of an application.

2. OUR APPROACH

In this section we explain in detail the collected data, why it is relevant to the problem, and the process to collect it.

2.1 Metrics to evaluate the evolution of clones

This paper aims to assess the effect of cloning on the changeability of the application. We achieve this by defining three impact axes: extension, persistence, and stability.

2.1.1 Extension

The analysis of the extension permits to identify the magnitude of cloning in the application, and at the level of methods. This is a first approach to assess its potential harmfulness. The evolution of the extension permits to evaluate if the methods affected at the beginning are the only ones that are affected in the application, or if cloning expands to other methods over time.

1. **Extension per method** is the average ratio between the number of cloned tokens and the total number tokens of the cloned methods at a given commit transaction.
2. **Extension per application** is the ratio between the number of cloned methods and the total number of methods that compose the application at a given commit transaction.

2.1.2 Persistence

The analysis of the persistence permits to know if cloning is a long-term issue. If cloning tends to disappear rapidly, there is a chance that it does not affect the maintainability of the application.

1. **Persistence per method** measures the percentage of the lifetime of methods affected by cloning up to the given commit transaction, i.e. it is the ratio between the number of commit transactions in which the method had clones over the number of commit transactions in which the method has been part of the application.
2. **Persistence per application** is a measure of the average persistence of methods affected by cloning up to the given commit transaction, i.e. it is the ratio between the persistence of the clones in the methods that had clones up to that commit transaction, over the number of methods that had clones up to that commit transaction.

2.1.3 Stability

The analysis of the stability of cloned methods in the application in comparison with those that do not have clones reveals whether or not clones may affect the methods that host them, regardless of whether the changes are inside clones. In addition, the analysis of stability inside the methods permits to assess accurately to what extent the stability measured in cloned methods is indeed due to clones.

1. **Stability per method** is a measure of the instability of a method that can be attributed to its clones. It is calculated as the average ratio between the *number of changes*[2] in the cloned tokens, over the number of changes in the method up to the given commit transaction.
2. **Stability per application** is a measure of the average instability that can be linked to cloned methods at the given commit transaction, i.e. it is the ratio between the *likelihood of change* of the methods that had clones, over the number of methods that had the clones up to that commit transaction. The likelihood of change is calculated as the number of changes in a method over the number of commit transactions that the method has been alive, up to that commit transaction.

Stab.	**Persist.**	**Ext.**	**Impact Level**
High	Low	Low	0. No effect
High	Low	High	1. Cloning occurs in a large percentage of the application but is a volatile phenomenon that does not affect negatively the stability of methods
High	High	Low	2. Cloning is a persistent phenomenon that affects a small percentage of the application, and does not affect negatively the stability of methods
High	High	High	3. Cloning is a persistent phenomenon that affects a large percentage of the application, but does not affect negatively the stability of methods
Low	Low	Low	4. Although cloning reduces the stability of methods, it is a small phenomenon that disappears quickly
Low	Low	High	5. Cloning reduces the stability of a large percentage of methods, but it is a volatile phenomenon
Low	High	Low	6. Cloning reduces the stability of methods for a large amount of time, but it only affects a small percentage of methods
Low	High	High	7. Cloning is a large and persistent phenomenon that reduces the stability of methods

Table 1: Scale (or levels) of cloning imprint

There are eight possible levels of cloning imprint (Table 1), which indicate the level of priority in which cloning should be dealt with. In table 1 we propose to subordinate extension to persistence and stability because the extension just acts as a multiplier of the persistence and of the stability. Furthermore, a high extension of cloning is irrelevant if the persistence is low or the stability is high, because a low persistence or a high stability would indicate that clones do not have a long term impact. Similarly, we subordinate the persistence to the stability because a high persistence of clones is irrelevant if the stability of clones is high: if the stability is low there is a higher chance that cloning affects the changeability of the application regardless of its persistence.

2.2 Tracking the evolution of clones

The measurements we gather define requirements on the level of detail in the data collection, in particular we need to be able to detect:

[2]Number of times that the text of a method or of a clone has been modified from one version to the next.

1. methods that are part of the application at a given commit transaction
2. methods that contain clones at a given commit transaction
3. tokens that compose the method at a given commit transaction
4. tokens of the method that are cloned at a given commit transaction
5. changes in methods between commit transactions
6. changes in clones between commit transactions

Tracking clones only by their location can result in inaccurate data given that methods are renamed and moved during their lifetime, and that methods may have several clones. For instance, when deciding if the clone was changed, or in analyses that take into account the characteristics of the clone it is necessary to distinguish which clone is being analyzed. Therefore, given that the identity of clone instances does not only depend on the method (or source code entity) that hosts them, it is necessary to track also their text.

Kim et al. [8] proposed to track clone instances across changes by comparing the text and location of the clone instances of version *i* and *i-1.* There are three cases to decide if a clone instance is the new version of another clone instance: if their location and their text are exact; if their location is exact and their text is very similar; and if their text is exact and their location is very similar. However, the approach is incapable of detecting late propagations because it does not compare the current clone instances with those that belonged to the family several changes ago. Furthermore, it does not take into account the difference between late propagations and independent evolutions, which is a key issue when analyzing clones because they indicate to what extent the lack of clone management causes bugs that are fixed by late propagations. Furthermore, the subtraction of a clone instance from its *clone family*[3] may happen by chance in two cases: if changes in the clone instance result in a similarity below the threshold with the rest of the fragments in the family, or if the method that has the cloned fragment is deleted, the fragment would be identified as an element removed from the cloned family. If the removal of cloned fragments occurs in a clone family with only two fragments, the clone family would be identified as deleted. This means that with Kim's approach for tracking clones over time it is impossible to check if clones were removed by refactorings or not. Some of these observations coincide with the issues identified for tracking clones over time [5].

In order to tackle these issues we decided to track clones by their belonging to a clone family (i.e. if they match with the template established by the common code shared by the family members), and their exact location (method, lines of code, and tokens). Our procedure is as follows.

a. Collection of commit transactions Group the log records of the trunk by author, message, and time. Order the commit transactions chronologically, and save them in a database.

[3]Set of code fragments that share the same clone, also known as clone class or clone cluster.

b. For each commit transaction :

b1-Extract snapshot from CVS Get a snapshot of the repository by the time when the commit transaction finishes.

b2-Find methods For each downloaded file, detect the methods, and their boundaries in terms of lines of code where the method is implemented.

b3-Find changes in methods Translate the lines changed, added or deleted per file to methods changed, and lines modified per method. Save the methods that were modified in that commit transaction in the database.

b4-Find clones Run the cloning detection algorithm for all files that belong to the application code base at that commit transaction.

b5-Find clones in methods Translate the clone detection results into clone families, clone instances, and cloned methods. If it is the first time, it requires a translation of all results, otherwise it is enough to translate the results related to the file changes and propagate such translation, i.e. detect whenever changes in a file could affect the clones of unchanged files. Notice that there could be clones added, modified and deleted in unchanged files, because changing any file with which a clone instance is cloned may be enough to change the clone instance. If the file changed acquires, modifies, or deletes a code fragment that is similar to a fragment in unchanged files, it is necessary to update the clones in unchanged files.
Save the information collected for the corresponding commit transaction into the database. In particular, for each clone instance store its clone family, the parameters for the clone (the parts of the cloned fragment that are not common to the rest of the family), the method that hosts it, and the tokens cloned in the host method. For each clone family store the clone (template), and the location of its instances. For each cloned method store the families of its instances, and the clone pairs that it forms with other methods.

b6-Find changes inside clones Translate the lines changed per file to the clone instances and families changed. A clone instance is considered changed if the lines of code corresponding to its tokens were modified. A family is modified if any of its instances is changed, i.e. the instance still shares the same common clone but its corresponding lines were modified. Store in the database the modifications done in that commit transaction: the clone instances changed, the clone instances added/deleted/or modified per clone family, and update the tokens cloned per clone instance.

c. Find the origin of methods All methods that seem new at each commit transaction are compared against all methods that seem deleted in the same commit transaction in order to detect renamings, or movement of methods to other classes or packages (see [13]).

	Freecol	JEdit	Gantt project	Columba	JBoss module
Purpose	Game	Text editor	Planning tool	Mail client	J2EE app. server
Methods	4050	8004	14616	28376	12132
Months	35	58	43	52	30
Changes	1087	1381	2701	3108	3346
Developers	14	13	20	16	86

Table 2: Characteristics of the analyzed applications. Number of methods is over the whole history, after merging the lifetimes of renamed and moved methods)

3. RESULTS

Five open source projects were analyzed over at least thirty months (see Table 2). These projects vary in age, size, and number of developers. The projects are different in purpose. If we obtain different results, the differences among applications would give us hints on factors that could alter the effect of cloning. In case the effect is the same, we have evidence to believe that domain factors are irrelevant when evaluating the effect of cloning in the changeability of an application.

3.1 Results of our tracking approach

As we explained previously, clone tracking techniques require improvements to be able to locate automatically late propagations and clones whose method host has been renamed. Table 3 shows that our technique is capable of detecting both renaming and movement of the location of a clone, and late propagations. This improved tracking permits more accurate analyses of the effect of clones in methods. The changes on the location are detected using our implementation of origin analysis. Late propagations (i.e. clone instances that were not consistently changed) are detected by checking if a method that is added to a clone family has been part of that family before. In other words, late propagations are located thanks to the separation between the clone family and the clone instance location. Independent evolutions are detected by locating methods whose clone instances stop belonging to their original family and start belonging to a new family at a commit transaction, but never go back to be part of the original family. Notice that becoming again part of a family (i.e. fitting the template established by the common code) is enough to detect late propagations, and avoids checking of consistent changes after every commit. Finally, changes in clone families are calculated as additions, deletions, or changes in a method with respect to its clone families. This automatic approach indicates that the percentage of late propagations is much lower than previously reported using manual detection [1], but the percentage of independent evolutions found seems much higher.

3.2 Evolution of clones

The ratio between the number of pairs of methods cloned and the number of clone families is between 1.12 and 1.48 (Table 4). Given that clone families of three elements require three pairwise cloning relations, the results on Table 4 mean that the majority of clone families analyzed have only 2 clone instances. However, notice that if the majority of families

	Methods cloned & renamed	Late propagations	Independent evolutions	Changes clone families
Freecol	25	9	1512	3187
JEdit	191	37	1538	3901
Ganttproject	97	7	1538	4449
Columba	641	0	12514	23264
JBoss mod.	83	37	5488	8110

Table 3: Usefulness of proposed tracking

have two instances, changes in a single method could affect (even add or delete) several families, which makes clone families very sensitive to changes.

Application	clone families	pairs of methods cloned
Freecol	1159	1407
JEdit	1289	1454
Ganttproject	1728	2100
Columba	7340	10870
JBoss mod.	2649	3787

Table 4: Summary of identified clones

3.2.1 *Extension per method*

Figure 1 presents in the x-axis the commit transactions in sequential order, and in the y-axis their corresponding *extension per method*. That is, each dot is the average ratio between the cloned tokens and the number of tokens in cloned methods for an application at a commit transaction.

We have found that whenever methods are cloned, they tend to be *highly cloned* (i.e. the extension is close to 100%), in particular for large applications. However, over time the percentage of cloning decreases. Note that until commit 1050 there is a rapid decrease in the percentage of cloning per method in Ganttproject. From this commit onward, several test classes are added, which could explain the slow increase of the average of cloning inside methods given that test methods tend to have clones.

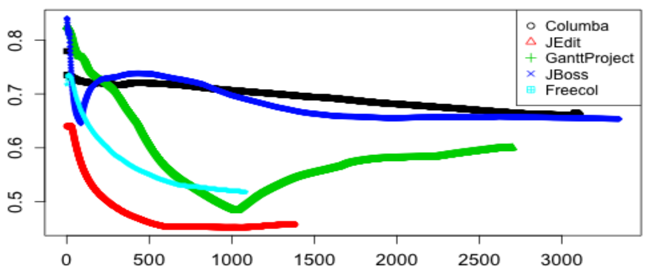

Figure 1: Extension of cloning per method: *avg(tokens cloned / tokens in method)*

3.2.2 *Extension per application*

Figure 2 presents the *extension per application* at each commit transaction. We have found that the percentage of cloning in the application is between 8% and 45% of the application, but it tends to remain stable over time, between 10% and 20 %.

The curves tend to be continuous, but there are some discontinuities in Ganttproject, JBoss, and Columba which

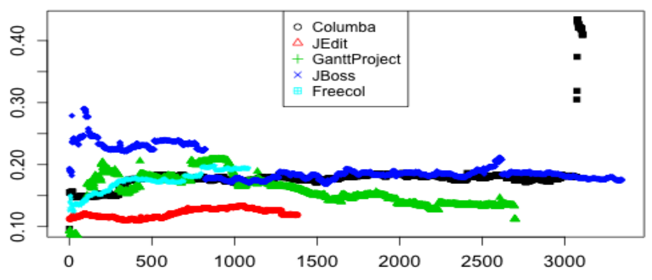

Figure 2: Extension of cloning in the applications: *methods cloned / methods in application*

indicate exceptional events, related with deletion or addition of a large number of methods. For instance, all discontinuities in Ganttproject are due to method deletion. When the majority of the deleted methods were cloned, the curve fell; when the majority of the deleted methods were not cloned, the curve raised. The discontinuities in JBoss are related with a package that handles pools of objects in which 38% of methods are cloned. When that package was added (commit 87) the curve jumped up; when that package was replaced by a jar maintained by a third party (in commit 825) the curve fell. The largest discontinuity is found in Columba, by the end of the analyzed history, after commit 3072. In these last commits, the developers are restructuring the application to migrate from CVS to SVN. Judging by the messages of these commits, they copied most of the code to a different directory structure, while at the same time they fixed warnings, updated the documentation, standardized the format of the code, and added some key functionality.

3.2.3 *Persistence per application*

Figure 3 shows the *persistence per application* for each commit transaction, i.e. the average percentage of the methods' lifetime in which they have clones.

The results show that, in average, methods are cloned at least 85% of their lifetime. For all applications the persistence is in the same range, presenting some exceptional events (sudden increase or sudden reduction), but tending to stabilize between 90% and 95%. The largest discontinuities are in Freecol, Ganttproject, and Columba. The largest discontinuity in Freecol (in commit 803) happened because 27 preexistent methods became cloned. The one in Ganttproject (in commit 427) happened because 43 methods were created cloned. As for Columba, the jump at the end occurs because the application is being restructured by copying and pasting methods from the current code-base into other directories.

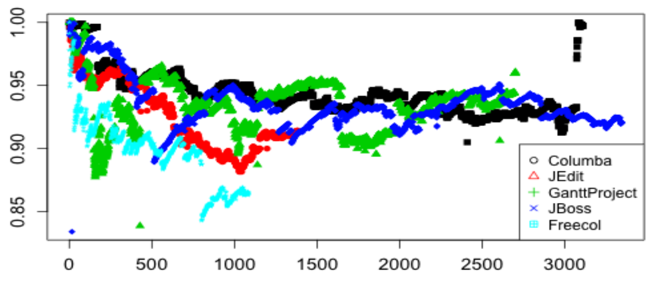

Figure 3: Persistence of cloning in methods that had clones: *avg(commits cloned / commits alive)*

3.2.4 *Stability per method*

Figure 4 presents the *stability per method* for all the commit transactions analyzed. The y-axis is the ratio between the changes inside the cloned fragments and the overall changes, for cloned methods up to each commit transaction. Figure 4 shows that most of the changes when a method is cloned occur inside the cloned fragment. This result is consistent with the fact that cloned fragments cover the majority of the method (see Fig. 1), making changes more likely to occur in the cloned area. Notice that for all applications the stability is low in the first commits, indicating that the cloned areas are stable while an initial code-base is established.

3.2.5 *Stability per application*

Figure 5 shows the *stability per application* i.e. the changes due to cloned methods in the application, up to that commit transaction. Figure 5 shows that from all the changes that occur in the application between 20% and 40% occur when methods are cloned. Note that the majority of changes in cloned methods occur at the beginning of the application's history, and that the stability varies significantly in the first 300 commits. This could indicate that cloned methods are very sensitive to the maturity of the application's code-base.

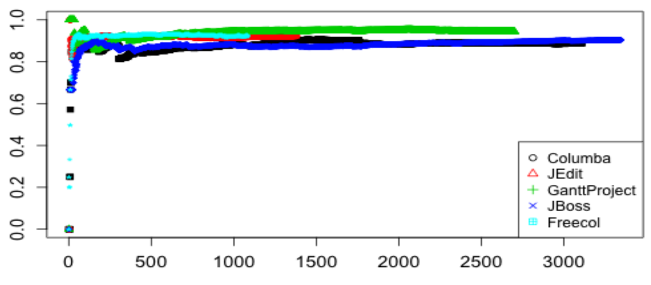

Figure 4: Stability of cloned methods: *avg(changes inside clones / changes while cloned)*

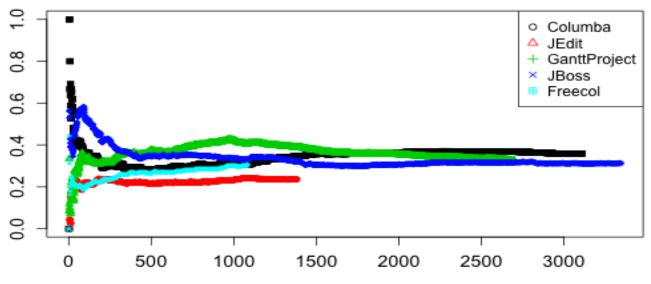

Figure 5: Stability of the applications due to cloned methods: *avg(changes in cloned methods / changes in the application)*

3.3 Analysis of results

3.3.1 *Extension per method*

There are three possible explanations for the decrease of the clone extension inside methods:

- extension of methods that do not touch the clones
- volatile and intermittent clones
- fragmentation of clones over time

In order to analyze these possibilities we checked:

- Whether or not the methods grow over time
- The lifetime of clones vs. the lifetime of methods
- Whether or not the clones shrink over time

The results of these checks are summarized in Table 5, which shows that method growth could explain the decrease of cloning extension. In order for the extension of cloning to decrease, the new code cannot be cloned. Moreover, given that most of the changes while cloned occur inside the cloned fragments, the new code would probably affect the clones inside the method. These extra lines in methods support the hypothesis of method growth, that does not fragment or eliminate clones. Table 5 also shows that the average lifetime of a clone instance tends to be longer than the average lifetime of a method, and therefore the hypothesis of volatile clones should be discarded. Finally, Table 5 shows that clones hardly grow or shrink over time, rejecting the fragmentation hypothesis. By analyzing the changes in some cloned methods (randomly chosen) we have found that changes to cloned methods tend to affect the boundaries of the cloned fragments. Some changes make the clone fragment to expand several tokens, while other changes make the clone fragment to contract. However, the center of the clone fragment remains intact. This behavior indicates that changes in cloned methods customize the environment of the cloned fragment.

Application	Method growth (LOC)	Clone pair growth (tokens)	Method life (commits)	Clone inst. life (commits)
Freecol	+9.79	+0.56	466.5	536.3
JEdit	-1.17	+0.65	519.6	607.0
Ganttpr.	+1.88	+1.15	270.2	493.2
Columba	+0.04	+0.37	580.8	590.9
JBoss m.	+4.34	+0.52	787.5	863.5

Table 5: Possible explanations for the decrease of cloning extension inside methods

Results also show that the decrease of the cloning extension inside methods is slower in large applications. A possible explanation for this difference between applications could be that clones of large applications are changed less than the clones of smaller applications. We have found that the average number of changes in the clone per pair of methods was 9.5, 8.5, and 8.4, for Freecol, JEdit and Ganttproject respectively. However, the average number of changes in the clone per pair of methods was just 4.1 for Columba and 5 for JBoss. The fact that Columba and JBoss show fewer changes per clone relation cannot be explained by less change in these applications, because all applications have the same average of two changes per method.

3.3.2 *Extension per application*

Given that the size of the applications increased over time and that the extension of cloning tends to be stable, we conclude that cloning increases in a proportional rate with the size of the application. Furthermore, there does not seem to be a policy of clone removal in any of the applications analyzed, because most of the changes in the extension of cloning are due to accidental modification of the ratio between cloned and not cloned methods.

3.3.3 *Persistence per application*

We have found that cloned methods tend to be cloned most of their lifetime. This indicates that clones are not refactored but eliminated when its host method is eliminated, which is consistent with the sensitivity of small clone families to changes. Notice that the fact that cloning is persistent inside methods does not necessarily contradict the fact that clone instances are volatile [8]. It could be that the similarity between cloned methods of a family persists after changes but is not enough to be considered the same family, so that the methods keep being cloned but as different instances that belong to different families. In fact, our definition of persistence is relative to the lifetime of the method, while the one analyzed in [8] is absolute (in terms of number of commits). However, in Table 5 we have shown that clone instances have long lifetimes. We think that the difference of results is due to the choice of applications: we have previously [13] warned of the issues of using DnsJava to analyze clone behaviour, as it is an application that is continually restructured by cloning chunks of functionality in other directories.

3.3.4 *Stability per method*

By analyzing the instability due to clones we have found that the majority of changes in cloned methods occur inside the clones. Although one might think that this result is not significant as the majority of cloned methods is covered by clones, notice that even when the extension of cloning inside methods is low (at the final of the history analyzed) the majority of changes in cloned methods are inside its clones. This behavior make us believe that cloning is indeed a driving factor for changes at the level of methods.

3.3.5 *Stability per application*

Figure 2 shows that the percentage of cloned methods in the applications analyzed is between 10% and 20% of the application. Figure 5 shows that of all the changes that occur in the application between 20% and 40% occur when methods are cloned. That is, 10–20% of the methods (i.e. those cloned) cause 20–40% of the changes. Hence, we hypothesized that cloned methods tend to have a larger number of changes than methods without clones. Such hypothesis is supported by the fact that cloned methods are more likely to change than methods without clones (Table 6). Note that the percentage of clone methods that changed is larger than the percentage of methods not cloned that change in every application, and that the difference is considerably larger for cloned methods. Again, note that a larger number of changes when cloned might be due just to the fact that cloned periods are larger than not cloned periods (as cloned instances tend to have larger lifetimes than the average method). It seems that our results contradict those presented in [10]. However, note that the stability can be calculated only for those methods that change at least once in their lifetime. Therefore, it is possible that both results are correct. That is, the lines of cloned code change less than those not cloned, but *cloned code that changes* is highly concentrated in certain methods.

Finally, it seems that the average extension per method and the stability of the application are inversely related. In the first commits, when the extension of cloning inside methods is the highest, the stability is the lowest. As the the extension of cloning inside methods decreases, the number

	Methods with clones that change	Methods without clones that change	Increase of chance of changing for cloned methods
Freecol	65%	31%	109%
JEdit	65%	44%	47%
Ganttproject	39%	11%	254%
Columba	38%	22%	72%
JBoss	45%	22%	104%

Table 6: Average percentage of methods that change, cloned vs. not cloned

of changes in cloned methods increase w.r.t. the number of changes in methods not cloned. This relation is consistent with the hypothesis that the lines of code added to methods are not cloned but modify the boundaries of clone instances without fragmenting them.

4. THREATS TO VALIDITY

The experiment seems internally valid as the dependent variables (i.e. impact of cloning measured as its persistence, stability, and extension) can be caused by the independent variables (i.e. methods at the commit transactions analyzed). Given that our approach to track clone instances distinguishes late propagations from independent evolutions, detects changes in the identification of the clone location, and distinguishes clone elimination from method elimination there is a higher confidence in the reliability of the data. Besides, the risk of confounding factors is reduced in two ways:

- by increasing the level of detail of analysis (extension of clones inside methods, and the changes inside clones);
- by gathering additional data (method and clone growth, method and clone lifetime, methods that change) to validate or to discard hypotheses obtained from the analysis of the results.

We developed tools for the data collection and analysis. Although using tools increases the reliability, by following the same protocol in all case studies and reducing possible bias, it also increases the chances of undetected problems in the data collection and analysis. However, these tools are the result of several iterations analyzing the evolution of clones [15, 13, 14] and therefore the number of undetected bugs should be low.

Notice that choices in the experiment setup could affect significantly the types of clones found, and therefore their evolution. For instance, the minimum size of a clone, the programming language and programming paradigm of the applications analyzed. For that reason, we chose a common clone detection setup[4], and we selected several applications with two pre-established conditions: being written in Java, and having their code repository in CVS. Moreover, we decided to analyze diverse applications in order to find commonalities in the evolution of cloning regardless of the application domain. Our setup was successful in the sense that we got similar results for all applications analyzed. Nevertheless, the conclusions are limited to open source Java applications.

[4]Note that although CCFinder is designed to find type I (exact) and type II (parameterized) clones, the choice of a small set of tokens to consider a fragment as cloned could help to detect type III (fragmented) clones.

5. RELATED WORK

The first attempt to track the evolution of a clone was done by locating the function that hosted it from a version to the next, and consistent and inconsistent changes were documented as the application analyzed evolved [11]. Afterwards, clone instances were tracked across versions by comparing previous vs. current location and lines of code [8], which allowed to model the changes that a clone family could undergo. This initial model of clone family evolution was extended by introducing the concept of late propagations [1], but they were not detected automatically. Therefore, current approaches to track the evolution of clones are not accurate enough to distinguish late propagations from independent evolutions [5]. This paper proposes an alternative to overcome these issues, and to assess the impact (imprint) of cloning in an application.

The analysis of the impact of clones has focused on measuring to what extent cloned code requires consistent changes. Most of previous work found that the chance of requiring a consistent change is at most 50% [11, 15, 9, 8]. In fact, no statistical relationship could be found between being cloned and changing at the same time [4]. Nevertheless, there is no agreement on the need of consistent changes in cloned code. While some authors point out that inconsistent customization of clones results in many bugs [12], others indicate that inconsistencies among clones of the same family tend to last less than 24 hours [1], and that just 7-8% of clones generate faults due to unintentional inconsistent changes [6].

Regarding the effect of clones on the quality of the code, there is some empirical evidence that indicates that many faults in Operating Systems can be linked to clones [2].

There have been also studies that have analyzed the extension, persistence, and stability of cloned code. Several authors have reported the percentage of the application affected by cloning. Although the results vary a lot among different publications, many of them have found that cloning in an application is below 25% of the source code entities analyzed [3, 16, 12], which is consistent with our results. Kim et al. found that the majority of clone instances lasted less than 8 commits [8]. A reason that could explain the divergence of results is that the application analyzed in [8] (dnsJava) has an unusual cloning evolution [13]. However, note that we obtained similar results for all the applications analyzed, which indicates that a cloning evolution such as the one found in dnsJava is more the exception than the rule. Furthermore, this divergence highlights the importance of analyzing typical clones and typical evolution of cloning.

Krinke found that lines of code cloned are more stable than those not cloned [10]. Our previous studies showed that cloned methods change more, and their changes affect a higher number of methods when they are cloned [15, 13]. However, our analyses have been restricted to methods that change, which could affect the results in comparison with those of Krinke[10]. It is expected to find that cloned code changes less because cloned code is scarce in comparison with code not cloned. However, our paper takes a relative view at changes in cloned code: starting from methods *that change*, we checked which ones were cloned and which ones were not. The relative measurement permits to conclude that cloned methods are more prone to change, even though code not cloned changes more. Another reason to explain the difference of conclusions is the type of clones analyzed, given that we detect clones of at least 30 tokens (approx. 3 LOCs) while Krinke [10] only analyzes clones of at least 15 LOCs. However, regardless of the likelihood of changes in lines of code cloned, we have shown that *cloned code that changes* is highly concentrated in certain methods. Moreover, that these cloned methods that change, change more than methods not cloned that change. To summarize, this paper contradicts two previous works: one that concluded that cloned code is more stable than code not cloned [10], and one that concluded that clones are volatile [8].

6. CONCLUSION

We proposed a new approach to track clone instances over time, which is resilient to events that current approaches cannot identify automatically: moves or renames of the source code entity that hosts the clone instance, late propagations, independent evolutions, and deletion of source code entities.

We also analyzed the cloning imprint by measuring the extension, persistence, and stability of clones in methods. Cloning extension remains stable in 10–20% of an application. Moreover, the events that affect cloning seriously are events of method creation and deletion, e.g. restructuring, adding packages, or integrating libraries. We have also found that cloned code covers the majority of cloned methods. This indicates that clone creation increases at the same rate as method creation. Cloning extension inside methods decreases slowly over time, probably due to customizations to the clone environment that only touch the boundaries of clone instances, and therefore do not fragment them.

Cloning is persistent as all applications presented cloning all their lifetime, and cloned methods had clones at least 85% of their lifetime.

Finally, cloned methods have a higher density of changes than methods not cloned. Furthermore, the majority of changes inside cloned methods occurred inside the cloned fragment, which is consistent with the high clone extension inside methods.

To sum up, cloning presents low stability, high persistence, and low extension, having a imprint level of 6 out of 7 (Table 1). This means a noticeable impact because, in spite of only affecting a small percentage of methods (cloned methods that change), cloning can reduce a lot the stability of methods for a large amount of time. Therefore, cloning affects the changeability of an application by shifting efforts from new requirements to clone tuning and maintenance. Nevertheless, it is important to analyze to what extent there is an increase in costs given that cloned methods tend to last longer than methods not cloned.

Future work will focus on describing typical clones, and the changes that affect cloned methods.

7. REFERENCES

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR'07)*, pages 81–90. 2007.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. of the eighteenth ACM symp. on Operating systems principles '01*, pages 73–88. 2001.

[3] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. Int'l Conf. on Software Maintenance (ICSM'99)*, pages 109–118. 1999.

[4] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proc. of the Int'l Conf. of Fundamental Approaches to Software Engineering (FASE'06)*, pages 411–425. 2006.

[5] J. Harder and N. GŽde. Modeling clone evolution. In *Proc. Int'l Workshop on Software Clones (IWSC'09)*.

[6] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the Int'l Conference on Software Engineering (ICSE'09)*, pages 485–495. 2009.

[7] 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, 2002.

[8] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of the European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng. (ESEC-FSE'05)*, pages 187–196. 2005.

[9] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. Working Conf. on Reverse Engineering (WCRE'07)*. 2007.

[10] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of the int'l workshop on Source Code Analysis and Manipulation (SCAM'08)*, pages 57–66. 2008.

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

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[13] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. Int'l Conf. on Software Maintenance (ICSM'08)*, pages 227–236. 2008.

[14] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the relation between changeability decay and the characteristics of clones and methods. In *Proc. Int'l Workshop on Software Evolution (Evol'08)*, pages 100–109.

[15] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Proc. of the int'l workshop on Mining Software Repositories (MSR'07)*, pages 18–21. 2007.

[16] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proc. int'l symp. on Software Metrics (METRICS'02)*, pages 87–94. 2002.