Assessing the effect of clones on changeability

Angela Lozano, Michel Wermelinger
Computing Department and Centre for Research in Computing
The Open University, UK

Abstract

To prioritize software maintenance activities, it is important to identify which programming flaws impact most on an application's evolution. Recent empirical studies on such a flaw, code clones, have focused on one of the arguments to consider clones harmful, namely, that related clones are not updated consistently. We believe that a wider notion is needed to assess the effect of cloning on evolution. This paper compares measures of the maintenance effort on methods with clones against those without. Statistical and graphical analysis suggests that having a clone may increase the maintenance effort of changing a method. The effort seems to increase depending on the percentage of the system affected whenever the methods that share the clone are modified. We also found that some methods seem to increase significantly their maintenance effort when a clone was present. However, the characteristics analyzed in these methods did not reveal any systematic relation between cloning and such maintenance effort increase.

1. Introduction

A *clone* is a source code fragment whose structure is identical or very similar to the structure of another code fragment. Cloned code is a consequence of a frequent programming practice: copying a piece of functionality and pasting it in another context where it is adapted. A *clone family* (also called clone group or clone class) is a maximal set of source code fragments that are similar among themselves. There are many reasons to believe that clones are harmful for software maintenance, among others:

1. unawareness of clone families leads to incomplete updates that generate bugs [1];
2. clones increase the size of code, making it more complex and difficult to understand [1];
3. clones cause faulty behavior due to the lack of awareness of the different pre- and post-conditions of the source and target contexts of the copied code [2];
4. clones may indicate lack of inheritance or missing abstractions [1], which affects the flexibility of the design.

Most of the previous work [3-8] just tackles the issue of incomplete updates. These empirical experiments have shown that changes are propagated to the clone family in less than half of the cases [3, 4, 8], and that in some cases the lack of consistent changes indeed leads to bugs [6]. Nevertheless, these findings are not enough to grasp the extent of the harmfulness of clones w.r.t. maintainability. In this paper, we aim to account for the effect of clones as a whole by focusing on how clones affect the maintenance effort of the methods they belong to. Sanders and Curran [9] have defined changeability as the set of "attributes of software that bear on the effort needed for modification, fault removal or for environmental change". Our aim is hence to find whether the existence of clones is a changeability attribute of methods. Finding supporting evidence for this would allow us to conclude that, in general, eliminating clones is a good maintenance investment.

This paper presents four contributions. First, it introduces three measures to assess, in a holistic way, the effect of cloning on a method's maintenance effort. Second, it presents a new approach to perform origin analysis. Third, it presents a methodology to analyze the effect of a programming flaw in a method on its changeability. Fourth, it shows that when methods have clones the change effort may increase, and that although that increase is not present in half of the cases, when it happens the effort increases significantly. The rest of the paper is organized as follows. Section 2 describes the hypothesis and the empirical data required to test it. Sections 3, 4 and 5 explain the experiment, its results, and its threats to validity. Section 6 compares this experiment with the related work, and the final section presents concluding remarks and points to future work.

2. Experiment

The harmful effect of clones on changeability could go beyond incomplete and inappropriate changes. In fact incomplete changes could be just separate evolution. An approach that measures cloning effects from a changeability point of view may reflect in a better way the four consequences of cloning mentioned earlier. The rationale is that each of those consequences leads indirectly, via incomplete updates, bugs or missing abstractions, to increased modifications to the code. The purpose of our experiment is hence to find more general evidence for the belief that cloning affects the maintenance of an application, by measuring the overall effect of clones on changeability, which has not been quantified before (see Section 6).

2.1 Hypothesis

The hypothesis is:

If a method has clones, the effort spent in changing it increases.

The null hypothesis is:

There is no difference on the maintenance effort spent on a method when it has clones and when it has none.

We chose two ways of testing the null hypothesis. One is to consider only those methods that had periods with and without clones as suitable for analysis. Comparing the same method in two different periods allows us: (1) to eliminate noise due to differences in frequency and type of maintenance that exist between methods, and (2) to measure accurately the maintenance change due to clones.

Another way is to assume that methods, in general, have a typical maintenance effort. This means that regardless of the nature of the method its maintenance would lay among some typical values. For testing the null hypothesis we just compare those methods that were either always cloned or never cloned.

To reject the null hypothesis, the maintenance measure must tend to differ between cloned periods (or methods) and not cloned periods (or methods), and to support the hypothesis the measure must be higher during the period with clones.

2.2. Data gathered

Like several other studies on software evolution [4, 6, 10], we see the evolution of a software system as a sequence $s_0\, c_1\, s_1\, c_2\, s_2\, \ldots\, s_n$, where s_i is a snapshot (e.g. a version) of the system's evolving code base and c_i is a set of changes, leading from one snapshot to the next.

A *snapshot* is a set of uniquely named methods, where uniqueness is achieved by taking into account the file, package and class to which the method belongs, and its signature. For each snapshot we gather not only the system's set of methods, but also whether the methods contain clones or not, as identified by an external clone detection tool (details in Section 3).

As for defining what exactly constitutes a set of changes c_i, we use an approach commonly used in software evolution studies. We retrieve from the system's CVS repository the *commit transactions*, i.e. those change commits that were done by the same developer, described by the same message, and within an interval of 3 minutes [10]. Since CVS repositories record changes at the file level, we map the lines changed per file to the set of methods changed.

With these concepts we now define measures to describe changeability.

2.3. Measures

The measures to be computed are the *likelihood* and *impact* of change: together, they represent the *work* required for maintaining a method. Hence, the independent variable is the fact of having a code clone or not, while the dependent variable is the maintenance work spent on a method. Our measures were inspired by the work of van Belle [11], but ours are normalized values, taking into account the change rate of the whole application.

Effort is a common way to assess maintainability. Usually, it is measured in number of hours invested in a particular task. Given that we cannot obtain that value from a CVS repository, our measures approximate the effort of changing a method by indicating how much change is required to maintain a method during a period of time.

To formally define the metrics, we need a predicate *changed(m, c)* that indicates if commit transaction *c* changed method *m*. We do not consider method creation and removal as changes. Finally, a *period* is a set of not necessarily consecutive commit transactions.

2.3.1. Likelihood. The likelihood of change of method *m* during period *P* is the ratio between the number of changes to *m* and the overall number of changes in the system, during *P*. A concise definition of likelihood is presented in table 1, where $CommitsChanged(m,P) = \{c \in P \mid changed(c, m)\}$ is the set of commit transactions that changed *m* during *P*. The numerator of the above fraction is the number of changes to *m* during *P*, while the denominator counts the total number of changes, at method level, during *P*.

2.3.2. Impact. The impact of changes to method *m* represents the percentage of the system that, on average, is changed whenever *m* changes during the period analyzed, i.e. it is the average percentage of methods that are changed by the same commit transactions that change *m*. The formula of impact is shown in table 1, where *CochangedMethods(m, c_i)* returns the empty set if *changed(m, c_i)* is false, otherwise returning $\{m' \in c_i \mid changed(m', c_i)\}$, and $|s_i|$ is the amount of methods that compose the snapshot *i*.

2.3.3. Work. Having computed the likelihood and impact of change of a method *m*, during the same period *P*, we define the effort of maintaining *m* during *P* as the product of those two values (see table 1).

According to this, the *work* increases whenever the method requires to be changed more frequently (*likelihood*) or when its changes require propagation to a larger proportion of the system (*impact*).

Table 1. Measures used

$likelihood(m, P) = \dfrac{\lvert \text{Commits Changed}(m, P) \rvert}{\sum_{c_i \in P} \left(\sum_{m \in s_i} \begin{cases} 1, \text{if changed}(m, c_i) \\ 0, \text{otherwise} \end{cases} \right)}$
$impact(m, P) = \dfrac{\sum_{c_i \in P} \dfrac{\lvert \text{CochangedMethods}(m, c_i) \rvert}{\lvert s_i \rvert}}{\lvert \text{Commits Changed}(m, P) \rvert}$
$work(m, P) = impact(m, P) \times likelihood(m, P)$

2.4. Experiment definition

To define the periods of interest for the previous measures, we consider the sets of commit transactions when the method has a clone *(P_C)* and when it does not have a clone *(P_{NC})*:

$$P_C(m) = \{c_i \mid m \in s_i \land cloned(m, c_i)\}$$

$$P_{NC}(m) = \{c_i \mid m \in s_i \land \neg cloned(m, c_i)\}$$

where the predicate *cloned(m,c_i)* indicates that the method *m* has a clone after the commit transaction c_i.

There are three disjoint sets to which a method can belong, depending on the length of its cloned and not cloned periods: it always had a clone (AC-methods), it never had a clone (NC-methods), or sometimes it had a clone and sometimes it did not (SC-methods). Formally:

$$m \in AC \leftrightarrow P_C(m) = lifetime(m)$$

$$m \in NC \leftrightarrow P_{NC}(m) = lifetime(m)$$

$$m \in SC \leftrightarrow P_C(m) \neq \varnothing \land P_{NC}(m) \neq \varnothing$$

$$where, lifetime(m) = \{c_i \mid m \in s_i\}$$

To reject the null hypothesis, we compare the average behavior of the changeability measures between AC-methods and NC-methods, and for each SC-method we compare the measures between its two periods. To validate the hypothesis, we check if the measures of SC-methods increase when cloned.

Note that the measures should be affected by the fact of being cloned. The likelihood should increase because (1) cloning affects understandability which may lead to more changes; and (2) incomplete updates in clone families may require to be fixed. The impact should increase because (1) changes to clones generate ripple effects to their clone families, and (2) cloning may indicate lack of abstractions, which requires more methods to be modified than usual to achieve a logical change.

2.5. Case studies

We expect that cloning behaves in the same way regardless of the application domain. Therefore, we selected four open source Java projects from SourceForge, with varying age, size, number of developers and activity rate (commit transactions per month), as Table 2 shows. The number of LOCs and methods reported are for the last version we analyzed.

Table 2. Case studies and their characteristics

Project	KLOC	methods	commits	developers	Start-End months : months
ganttProj.	44	14469	2701	20	May 03-Dec 06: 45
jEdit	92	7868	1381	13	Sep 01-Jul 06: 58
freecol	54	3928	1087	14	Apr 04-Mar 07: 35
jboss/jboss	10	18781	5225	50	Apr 00-Jul 06: 63

GanttProject is a scheduling application with facilities for doing Gantt charts, resource management, calendars, etc. JEdit is a text editor for programmers that can be configured as an IDE through its plug-in architecture. FreeCol is a game in which players have to conquer and colonize new worlds. JBoss is a J2EE based application server, from which we analyzed the jboss module.

3. Data collected

We have built a tool [12] that gathers the required data (see section 2.2) and stores it in a MySQL database. That is, for each snapshot, the tool extracts from the CVS repository the files that have changed and, from the lines changed, computes the methods that were created, deleted and modified. After that, the tool identifies which of the methods existing in that snapshot had clones.

3.1. Elimination of noise from data

To be able to draw accurate conclusions from the statistical analysis it is necessary to ensure that the data collected complies with our assumptions: whenever there is a clone it can be tracked along the versions of the method where it is located; each method is identified uniquely along its lifetime; all the functionality changes are taken into account. In the following sections we argue the steps required to ensure each of these assumptions.

3.1.1. Clone identification. Our tool uses CCFinder [13] to detect the clones of 30 or more consecutive tokens. Setting 30 tokens as the threshold for the minimum clone length can increase the number of false positives because small clone fragments that are accidentally similar might be identified as clones. However, choosing that threshold can also increase the chances of identifying *fragmented clones*, i.e. those that are interrupted by a few lines of non-cloned code, which CCFinder does not find.

We chose CCFinder because, compared with other clone detection tools, CCFinder has better performance, scalability, and recall, but a lower precision [14]. Moreover, CCFinder does not depend on the correctness of the source code analyzed. This is an important characteristic given that some snapshots have syntax errors. Furthermore, CCFinder does not rely on the syntax of the language, as it is the case with AST based detection tools. This makes CCFinder immune to the evolutions of the programming languages, like those that Java went through in the timeframe of the selected case studies.

3.1.2. Method identification. Given that we identify methods by their signature and location (section 2.2), whenever a method (or its enclosing class and package) is renamed or moved, the tool assumes it has been deleted and another one has been created. This makes the results inaccurate. We therefore defined an algorithm to perform what is known as origin analysis [15]. That is, finding if $m \in s_i$ is the same as method $n \in s_{i+1}$. Our algorithm works as follows:

1. For each method detected as deleted by a commit transaction, the candidate set of methods contains those created by the same transaction.
2. The set of candidates is divided into those that changed the signature but are in the same location (i.e. same file, package, and class), w.r.t. the original method that supposedly was deleted, and those that are in a different location but the method name and the parameters are the same.
3. The candidates are compared line by line with the original method. If the similarity between the best candidate and the original method is above a certain threshold, then the candidate is identified as the new version of the original method.
4. Once the whole system evolution has been processed, we have a set of pairs *a-- X -->b* stating that method *b* seems to be a new version of method *a*, and that they have a similarity *X*. Nevertheless, several methods may have the same possible evolution, i.e. *a-- X -->b* and *p-- Y -->b*. Those cases are resolved by choosing the method that has the highest similarity with *b*. If all the possible previous versions of a *b* have the same similarity, we assume that *b* is a new method and all its possible previous versions are deleted methods.
5. We have now a set of pairs *a->b* stating that the unique method names *a* and *b* denote in fact *the same method*. We then put together pairs to obtain chains *a->b->c->...* that show rename and move operations on the same method. This step also ensures that the renaming of methods is stored as a change to the method.
6. Finally, the information about the P_C and P_{NC} periods of *a, b, c, ...* is merged together.

The similarity (i.e., step 3) between any two methods *a* and *b* is computed as follows, where *a* designates the shorter method, i.e. the one with less lines. First, we remove all layout characters from *a* and *b*. Then, for each line of *a*, we compute its similarity with every line of *b* using the Strike A Match algorithm[1] and we keep the highest similarity value. Next we compute the average value, over all lines of *a*, and this will be the similarity between methods *a* and *b*. Of all candidate methods in the "changed signature", the one with the highest similarity value is kept. If this value is above 70%, we consider to have found the new version of the original method, and proceed with step 4. Otherwise, we go through the same process with the candidate methods in the "moved" set. If no method there has a similarity value above the 70% threshold, we consider the original method to be effectively deleted from the system.

It might be tempting to use CCFinder to perform origin analysis, by checking which methods created in one commit transaction are clones of those deleted in the same transaction. However, CCFinder cannot find fragmented clones, i.e. those interrupted by a few lines of code. That is also the reason to compare the original and candidate methods line by line: the algorithm

[1] www.catalysoft.com/articles/StrikeAMatch.html

works even if there are new scattered lines from one version to the next one. In order for CCFinder to detect such cases, one would have to set a very low minimum number of similar sequential tokens, but this in turn would increase the possibility of false positives for the origin analysis.

Results of the origin analysis filtering are summarized in table 3. The third row indicates how many methods were renamed or moved at least once in their lifetime.

Table 3. Methods filtered with origin analysis

	gantt	jEdit	freec	jboss
methods identified initially	14895	8434	4099	18942
number of unique methods	11805	7392	3901	17784
from those were renamed	1743	858	184	1001

3.1.3. Eliminating atypical changes. The comparison of metrics can be greatly affected if one of the periods had atypical changes. An atypical change is a change that does not aim to modify a single functionality feature. Examples of atypical changes are those changes that make several improvements to functionality, or that restructure the application. When there is an atypical change in a period, the impact measure increases greatly, making it difficult to compare with a 'normal' period. We therefore discarded for each project the 2.5% largest commit transactions.

3.1.4. Eliminating volatile periods. A volatile period is a short period and can affect significantly the comparison of the likelihood. Given that the likelihood is calculated on the number of overall methods changed per period, having periods with very different lengths increases the chances of having very different denominators in the likelihood of each period. This results in significantly different likelihoods even if the ratio of changes per commit is the same in both periods.

We defined a volatile period as one that lasts less than 15% of the method's lifetime. Any method with a volatile period (whether with or without clones) was not analyzed.

3.2. Data analyzed

Table 4 has the total number of methods after origin analysis, divided into the three groups, and the number of methods actually analyzed. Analyzable methods are those that have at least one change, so that the impact measure can be calculated. Note that AC- and NC-methods require at least one change in their lifetime to be analyzable, but SC-methods require at least one change in each period.

As table 4 shows if comparing the NC-methods to the sum of AC- and SC-methods, there are far more methods without clones than with clones. As the drop to analyzable methods indicates, most methods are never changed. Note also that atypical changes, by definition, change methods across the whole system and hence usually affect all sets of methods (NC, AC, SC).

Table 4. Methods eliminated from the analysis

	gantt	jEdit	freec.	jboss
NC	10428	6291	2962	13962
NC analyzable	3681	3659	2578	6177
NC after atypical filter	3312	2263	1310	4558
AC	790	418	363	1841
AC analyzable	91	206	213	802
AC after atypical filter	91	202	169	743
SC	587	683	576	1981
SC analyzable	210	332	330	613
SC after atypical filter	194	329	262	558
SC after volatile filter	130	182	176	352

4. Results

In this section we present the results of the experiment.

4.1. Analysis of null hypotheses

This section presents the distributions of the measures to test the null hypothesis. Tables 5 and 6 summarize the p-values obtained from the statistical tests. Table 5 shows the p-value obtained by the Wilcoxon rank sum test (i.e. paired test) from comparing the distributions when cloned vs. when not cloned for the SC-methods. Table 6 shows the p-value obtained by the Mann Whitney test from comparing the distributions between AC- and NC-methods. The p-value indicates the probability that the null hypothesis was discarded by chance in the statistical test. Having a p-value under 5% is considered to be enough to reject confidently the null hypothesis, i.e. to say that the distributions compared are different. The p-values that do not allow to reject the null hypothesis (>0.05), are highlighted in bold. These p-values could mean that the distributions are indeed very similar or that they are different but the statistical approximation does not show it. As several p-values were not small enough to reject the null hypotheses, we decided to check their results graphically to see if the distributions are indeed similar or if the statistical test just could not achieve enough confidence with the data provided. Statistical tests may produce misleading results if the distributions vary a lot. The statistical tests work either by obtaining the

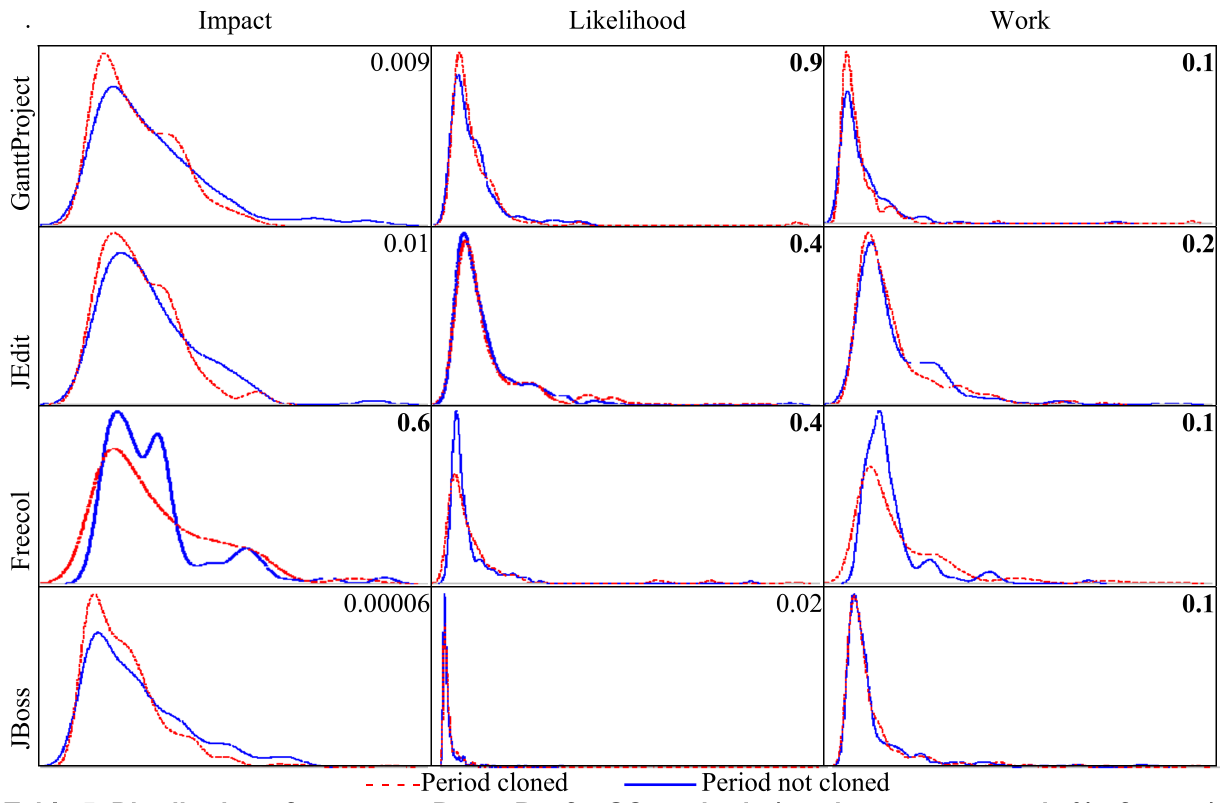

Table 5. Distribution of measures P_C vs. P_{NC} for SC-methods (x-axis measure, y-axis % of cases)

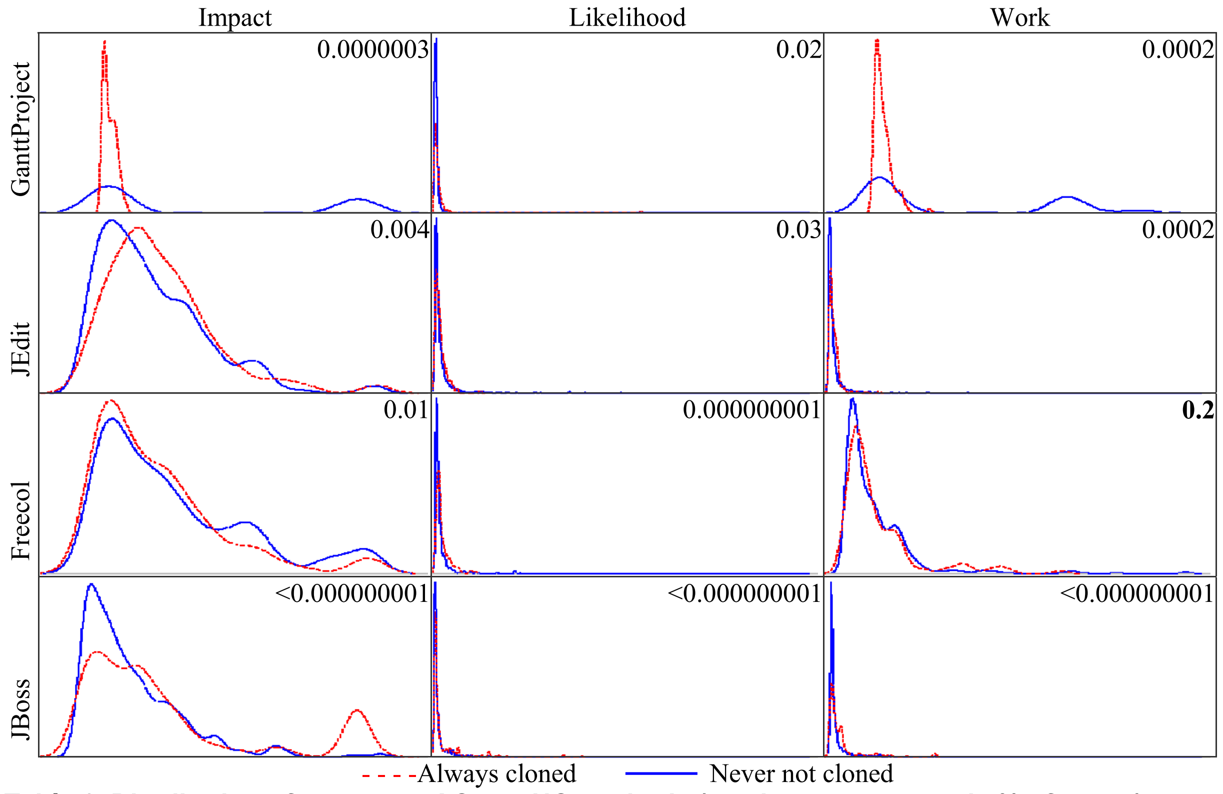

Table 6. Distribution of measures AC- vs. NC-methods (x-axis measure, y-axis % of cases)

median of differences between the distributions (t-test and Wilkinson test) or by computing the difference of the areas when one curve is above or below the other (MannWhitney test). However, it can happen by chance that the sum of the areas in which one curve is above is the same as the sum of the areas in which that curve is below the other.

Tables 5 and 6 hence also show graphically the distribution of the changeability measures when cloned (dashed line) and when not cloned (solid line). Each row in tables 5 and 6 represents a case study and each column represents a changeability measure. The x-axis of each distribution graph represents the values of the measurement for that column, and the y-axis represents the value of the probability density function, i.e. it is a smooth curve that represents the proportion of methods that had the measurement value given by x.

Several graphs show differences that statistical tests miss (bold numbers vs. curves that are different). In cases like the impact in freecol in table 5, although the test does not give certainty that the distributions are different when cloned and not cloned, the graphs show that there are differences.

According to the p-values of table 5, the *likelihood* did not change much for the methods sometimes cloned. When a method has a clone it is expected to undergo extra changes, namely those that come from changes to the clone family. However, the total number of changes in the system will increase as well. Given that the likelihood is a ratio between the changes on the method over the changes of the system, the fact that the likelihood remains unchanged when cloned and not cloned means that, in general, the number of extra changes is proportional in the numerator and denominator.

According to the information in tables 5 and 6, *impact* behaves differently between cloned and not cloned. However, table 5 may be more accurate than table 6 because it compares the same methods when cloned and not cloned, which leaves out inherent differences on the changeability of each method. In all cases the main peak of the impact distribution is on the left end of the x axis: this means that most of the methods with clones have a low impact value. Given that the peak of the distribution when cloned is higher than the peak distribution when not cloned, for most of the cases, one can say that most of the methods seem to have a similar impact when cloned. So, most of the methods with clones have a similar low impact. Therefore, being cloned seems to standardize the value of a method's impact.

Not all impact distributions when cloned follow the pattern described before i.e. a higher peak than when not cloned, on the left end of the x-axis. Some of these distributions have (1) their highest peak slightly to the right of the peak of the not cloned distribution, or (2) a wider distribution. The first pattern can be seen in JEdit in table 6. This means that in general the impact of those always cloned is higher than those never cloned in this case study, which supports the hypothesis. The second pattern appears in freecol in table 5, and in jBoss in table 6. This pattern means that there is a significant number of methods with a higher value in their impact when they have clones, supporting as well the hypothesis. Besides, jBoss in table 6 also shows a second peak at the right end of the x-axis: a significant percentage of always cloned methods have a much higher impact than those never cloned.

Summarizing, we have found evidence suggesting that, in general, cloning does not affect much the likelihood of changes, although it may increase the number of changes in a method. While in most of the cases cloning standardizes the impact of changing a method to the worst impact of the clone family, in a few others the impact seems to confirm the hypothesis.

4.2. Support for the hypothesis

We also calculate for each SC-method and for each measure the effect of being cloned. The effect is defined as the ratio of the increase or decrease of the measure between periods w.r.t. to the value of the measure during the not cloned period:

$$\mathit{increase}(M, m) = \frac{M(m, P_C(m)) - M(m, P_{NC}(m))}{M(m, P_{NC}(m))}$$

where M is the measure, and m the method analyzed.

Figure 1 shows the increase of work in the cloned periods. The y axis shows the level of increase of the measure and the x axis shows the *cumulative* percentage of methods in increasing order of y (work increase). The figure shows that it is inadequate to assume that all clones are harmful because in all case studies just half of the analyzed methods lead to an increase in maintenance effort. However, figure 4 seems to confirm the intuition that clones *can be* very harmful for changeability. When the measures are lower when cloned, and hence the difference between the measures is negative, the measure decreased by at most 100% (lowest negative value on the y axis). But when the measures are higher when cloned, the difference could be up to 72479% (outside the y-axis range shown in the figure). Besides, there is a rapid growth in the difference of the measures as soon as it becomes positive.

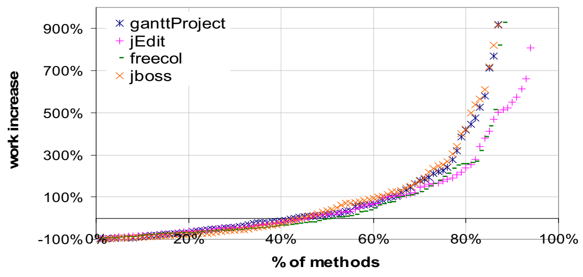

Figure 1. Effect of clones on the work

We decided to evaluate those methods that presented the highest increase of work to validate that there are clones that are worse for maintenance. We selected for each application those SC-methods that were in the top 10% of work increase when cloned. We expected to find any common characteristics in these harmful methods. The characteristics analyzed were size of the clone, size of the family, similarity in the name of the methods cloned, and the distance between methods cloned, i.e. the distance of the closest common ancestor between two methods in a tree defined by the package and class hierarchy. If the cloned methods are in the same class, their distance is zero, if they are in different classes of the same package, their distance is one. The distance is inspired on the clone classification by location proposed by Kapser [16].

The evaluated characteristics aim to grasp some of the reasons to consider clones harmful. The size of the clone could account for the effect on understandability of the method. The size of the family was looked at because the larger the family is the more likely it is that the method will change. The similarity of the method names show if the methods aim to provide similar functionality and therefore could be an example of lack of abstraction. Finally the distance of the methods aims to grasp the difficulty of modifying a clone family that does not have its clones close to each other.

However, we could not find any trend in the characteristics among these clones. Sizes varied from 30 to 99 tokens. The size of the clone families varied from 2 to 18 methods. The distance among clones also varied, from 0 to 5. The similarity of the names of the methods also varied, with pairs like **`org.gjt.sp.jedit.textarea.FoldVisibilityManager.virtualToPhysical(int) - org.gjt.sp.jedit.textarea.FoldVisibilityManager.physicalToVirtual(int)`** and pairs like **`org.gjt.sp.jedit.browser.VFSFileNameField.doComplete() - org.gjt.sp.jedit.gui.PanelWindowContainer$ButtonLayout.layoutContainer(Container)`**.

Apparently the huge differences in work are due to atypically small changes that just modified the method analyzed in the whole application in the period not cloned, making the value of the work much lower when not cloned than when cloned. However more research is needed in order to identify if there are clones that regardless of the method they belong to have harmful effects, and what characterizes them.

5. Threats to validity

This section discusses issues that may affect the results (internal validity) or the boundaries in which such results apply (external validity).

First, CCFinder's lack of precision threats the comparisons of the distributions due to the size of the compared sets of methods. If one set is much larger than the other (e.g. NC- vs. AC-methods), false positives in the smaller set may affect the average behavior of the measures for that set (in this case the AC-methods). Therefore, comparing average measurements would not assess correctly the difference of the measurements in the sets. However, as previously mentioned, clone detection tools with higher precision, based on abstract syntax trees, have many restrictions to overcome for analyzing long-lived programs written in an evolving language. False positives will be tackled in future work.

Second, the selection of case studies could affect the results obtained. For example, dnsJava has unusual cloning patterns. There is a considerable amount of files repeated in several directories (e.g. `org/xbill/DNS/` and `org/xbill/DNS/utils/`), during periods varying from 1 commit to 95% of the application's lifetime. Moreover, these clones do not seem to be an experimental variation of the code, because the files change in the same way. Therefore, dnsJava might not be a suitable candidate for the analysis of the nature of cloning. This is a relevant observation given that dnsJava is a popular case study for the analysis of source code clones [4, 6]. We could not find any abnormal cloning patterns in the case studies we selected.

Third, the case studies selected may not represent typical applications: they are small open source systems. Small applications are usually developed by small groups of developers, so it is likely that they know very well the code base. In particular, they may know the location of clone families and consequently maintain them consistently. This awareness of cloning may not happen in larger applications. Nevertheless, our results on jBoss (50 developers) indicate although that effect can be seen (the impact in table 6 has a second peak to the right of the x-axis), most of the methods (i.e. the main peak of the distribution) behaved like the other case studies. As for being open source projects, we think that there is no limitation of our results given that it has not been proved yet that OSS applications are essentially different from closed source applications [17].

Fourth, another potential problem is the diversity of domains in the case studies because they may present different types of cloning. However, on one hand, comparing the changeability of the *same* method in two different periods of time makes the nature of the method irrelevant for the analysis. On the other hand, given that distribution graphs show general behavior of the measures, individual differences that may affect the comparison of AC- and NC-methods have less impact. Besides, the methods of each application were compared only against methods of the same application. Furthermore, the results suggest that even if the type of cloning among applications is different, their overall effect in changeability effort is similar (figure 1).

Fifth, the fact that all case studies are written in Java might affect the distribution of the types of cloned methods found, as some of the clones are introduced to cope with language limitations [18]. This issue might indeed lead to different results for applications written in other languages. So, in principle, our results apply only to Java applications and further research is needed for other languages.

Sixth, the definition of commit transaction makes the measurements sensitive to the developer's commit style. The assumption for the measurements is that commit transactions show a relation between methods that change together. However, if the developer commits in fixed time intervals regardless of the completeness of the changes, it is possible that the co-change relations obtained are by chance.

Seventh, given that the changes are identified at method level and not at clone level, it is possible that the changeability obtained has no relation with the fact of being cloned. However, as mentioned before, the idea was to assess if methods with clones were more difficult to change. That means that even if the change does not affect the cloned tokens it might be that the fact of being cloned was the cause of such change.

Eighth, the noise reduction was done after finding that most of the harmful clones either had a very short cloned period or they were cloned in restructuring periods. Therefore, in order to find the real harmful cloned methods, we introduced thresholds for atypical changes and volatile periods obtained by trial and error on reducing the amount of false harmful clones. A more systematic approach with statistical outliers may improve the results.

Finally, the fact that the NC methods are more affected by the removal of atypical changes may affect the comparison between AC and NC methods.

6. Related work

Several studies have addressed the impact of clones. Most of them have focused on the need for extra updates to keep the clone family consistent, and on the effects of not doing so [3-8]. Nevertheless, inconsistent changes may simply indicate separate evolution of the clones. These studies therefore do not address the whole effect of clones on changeability. In fact, some authors [18, 19] have found that not all cloning is harmful: it saves programming time, captures paired operations and crosscutting concerns, helps to identify which code is worth restructuring [18]. There are even cases where cloning has been argued a reasonable design decision [19].

Our work is related to the study presented by Geiger et al. [5] as they tried to find if files that share a clone change together. Contrary to theirs, our study takes into account several details that could have affected their results, such as the granularity level (files vs. methods) and the fact that clones do not always span for the whole method's lifetime, which means that Geiger et al. do not compare different periods.

There are a couple of studies that assess the relation between clones and bugs [20, 21]. In contrast to our work, they do not analyze if the pasted clones required more maintenance, or maintenance that required more effort. Li et al. found that inconsistent renaming of identifiers where the clone is pasted produces bugs. They found that 13% of cloning produced errors, and 14% were potentially harmful [20]. Chou et al. analyzed code with errors, and found that bugs are correlated with developers that clone code without validating its pre- and post-conditions in the new context where the code is pasted, i.e. they are ignorant of the system's rules or of the system's interfaces [21].

The work closest to ours is the attempt by Monden et al. [22] to measure the impact of clones at module level. Maintenance is measured in absolute number of revisions. They found that modules are less maintainable if they have clones, in particular larger clones. However, our study is more precise: it is performed at a finer granularity level (i.e. methods instead of modules), and it has more accurate indicators of maintenance impact (measurements per period instead of number of revisions). Our measures are more precise since they are independent of the age of the methods, and of the overall level of activity (number of changes) on the system.

7. Conclusions and future work

The software engineering community has increased its interest in code cloning in recent years, in particular regarding untested myths about their harmfulness Nevertheless, most of the studies have focused on analyzing if related clones are consistently changed or not, and whether such inconsistent changes lead to more bugs. Apart from Monden et al., no one seems to have directly attempted to measure whether cloning reduces the ease of changing the code, independently of the reason.

We proposed a methodology to analyze source code flaws, by measuring the changeability of methods during periods with or without a particular characteristic of the method. The methodology is based on measures that capture the work required to keep the "consistency'' of the application after the change. Given that our measures are not absolute values like [10, 22], they can be compared regardless of the lifetime of methods. Furthermore, being ratio values, the measures have an intuitive meaning: the lower the measure, the better the changeability.

The results show that change effort may increase when the method has clones. In at least 50% of the cases, being cloned does not increase the changeability measures, but when it increases the difference can be significant. Such increase seems to be more related to the percentage of the system affected among the methods in the clone family, than to the type of clone that the method has.

In future work we will investigate if the maintenance effort still increases after cleaning the data from false positives and atypically small changes. We also plan to classify and describe the clones depending on their effect on changeability indicators.

References

[1] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proc. Int'l Conf. on Software Maintenance*, 1999, pp. 109-118.

[2] C. Kapser and M. Godfrey, "Aiding Comprehension of Cloning Through Categorization," in *Proc. Int'l Workshop on Principles of Software Evolution*, 2004, pp. 85-94.

[3] 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*, 1997, pp. 314-321.

[4] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proc. European Softw. Eng. Conf. and symp. on Foundations of Softw. Eng.*, 2005, pp. 187-196.

[5] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, "Relation of Code Clones and Change Couplings," in *Proc. Int'l Conf. of Fundamental Approaches to Software Engineering*, 2006, pp. 411-425.

[6] L. Aversano, L. Cerulo, and M. D. Penta, "How Clones are Maintained: An Empirical Study," in *Proc. European Conf. on Software Maintenance and Reengineering*, 2007, pp. 81-90.

[7] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone Smells in Software Evolution," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*: IEEE Computer Society, 2007.

[8] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones," in *Proc. Working Conf. on Reverse Engineering*, 2007, pp. 170-178.

[9] J. Sanders and E. Curran, *Software Quality, A framework for success in software development and support*: Addison-Wesley Professional, 1995.

[10] T. Zimmermann and P. Weibgerber, "Preprocessing CVS data for fine-grained analysis.," in *Proc. Int'l workshop on Mining Software Repositories*, 2004, pp. 2-6.

[11] T. B. V. Belle, "Modularity and the Evolution of Software Evolvability," The University of New Mexico, 2004.

[12] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Proc. Int'l Workshop On Principles of Software Evolution*, 2007, pp. 31 - 34.

[13] 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.*, vol. 28, pp. 654-670, 2002.

[14] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," in *Proc. Int'l workshop on Source Code Analysis and Manipulation*, 2002, pp. 36-43.

[15] L. Zou and M. W. Godfrey, "Detecting merging and splitting using origin analysis," in *Proc. Working Conf. on Reverse Engineering*, 2003, pp. 146-154.

[16] C. Kapser and M. Godfrey, "Improved Tool Support for the Investigation of Duplication in Software," in *Proc. Int'l Conf. on Software Maintenance*, 2005, pp. 305-314.

[17] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi, "Empirical Studies of Open Source Evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds.: Springer, 2008.

[18] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Proc. Int'l Symp. on Empirical Software Engineering*, 2004, pp. 83-92.

[19] C. Kapser and M. Godfrey, "'Cloning considered harmful' considered harmful," in *Proc. Working Conf. on Reverse Engineering*, 2006, pp. 19-28.

[20] 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.*, vol. 32, pp. 176-192, 2006.

[21] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. symp. on Operating systems principles*, 2001, pp. 73-88.

[22] 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*, 2002, pp. 87-94.