Evaluating the harmfulness of cloning: a change based experiment

Angela Lozano, Michel Wermelinger, Bashar Nuseibeh
Computing Department, The Open University, UK
{a.lozano-rodriguez, m.a.wermelinger, b.nuseibeh}@open.ac.uk

Abstract

Cloning is considered a harmful practice for software maintenance because it requires consistent changes of the entities that share a cloned fragment. However this claim has not been refuted or confirmed empirically. Therefore, we have developed a prototype tool, CloneTracker, in order to study the rate of change of applications containing clones. This paper describes CloneTracker and illustrates its preliminary application on a case study.

1. Introduction

Code clones are identical or nearly identical fragments of code [5]. Cloning code is employed as a fast way of reusing reliable semantic or syntactic constructs, and as a way to implement crosscutting concerns [7].

Code clones are generally considered harmful. They indicate lack of abstraction, and they increase complexity by increasing the code size and by introducing hidden relations [2]. More importantly they are believed to have a negative impact on evolution [3]. This negative impact is due to an increment of the maintenance effort because a change in any of the cloned fragments may require a change in all fragments, of which the developer may not be aware.

However, it has been argued that code clones are beneficial in certain situations [6]; e.g., being aware of copied code can help programmers identify recurring patterns that are then encapsulated in a layer of abstraction, thus eliminating the clones [7].

We have designed a tool to gather evidence to either confirm or refute the belief that clones are harmful. Our tool looks for methods that had a cloned fragment at some point in time, and counts how often the method was changed both when it contained a clone and when it did not. The tool also looks for method pairs containing the same code fragment, and counts how often they are changed in the same transaction, again distinguishing periods when they were clones from those when they were not. This paper describes what data the tool gathers, how it gathers it, and the preliminary results of gathering data from a java application.

2. Related work

Although there is a significant body of work on cloning, many issues are still open [10]. For example, there is little work on cloning and evolution. In one of the first papers in this area, by Lague et al. [9], the authors examined which clones exist and which ones are added, deleted, or modified in six versions of a large telecom system written in a Pascal-like language. They found that most clones remain stable and that clone coverage, i.e. the percentage of cloned code in the system, does not degrade substantially over the evolution lifetime. They also reported that half of the changes to a clone were propagated to the other clone instances. However, their notion of clone is limited to nearly exact function copies. Antoniol and colleagues also found that clone coverage does not degrade over time; their case study was the Linux kernel [1].

Kim and colleagues presented a finer-grained analysis of how clones evolve [8]. They defined a *clone group* as a set of clones with the same text snippet within the same version, and a *clone genealogy* as a directed graph where nodes are clone groups and arcs show how the source clone group was transformed into the target clone group, e.g. by adding a clone or by consistently modifying all clones within the group. The authors disregarded any version in which the number of cloned lines had not changed with respect to the previous version. This means they were not attempting to relate the clones' evolution to maintenance effort, because they disregard part of the system's history. By providing statistics only at the level of complete genealogies, their data is in our opinion too coarse grained. The authors also observed that over 50% of the genealogies cannot be eliminated by refactoring the clones; e.g., by encapsulating the cloned code into a new method. In other words, one of the main reasons

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

why clones persist over various versions is because they simply cannot be removed (at least not easily), and not because they are an indication of poor abstraction or coding.

Geiger and colleagues [3] attempted to correlate the occurrence of clones to the number of co-changes, i.e. simultaneous changes, to the files containing the clones. They concluded that although there is a reasonable amount of cases where the relation exists, it was statistically unverifiable. We think one cause for this might be that change frequency was measured at a coarse level of granularity, namely at file level. It is therefore hard to argue that a change to a file was caused by an update to a clone contained in that file.

3. Data gathered: a small example

Previous experiments were done at a very high level of granularity affecting the accuracy of measurements [3] or neglected part of historical information by eliminating all changes that did not affect any cloned fragment [8]. Therefore our aim is to analyze *all* changes and relate them to cloning at a finer level of granularity. We analyze clones at the method level because they are a functional and syntactic unit, and because 98% of clones are produced at that level [7].

Our tool, CloneTracker, measures the ***number*** (i.e. amount) and ***density*** (i.e. amount per time unit) ***of changes*** in methods while they have a cloned code snippet versus while they do not. The tool also measures the ***number and density of co-change*** in pairs of methods while they share a cloned code snippet versus while they do not. A ***change*** occurs whenever there is a difference in the method's code between two consecutive file versions. A ***co-change*** occurs whenever two methods are changed by in the same ***transaction***. A transaction is a commit operation done by a single author in given time frame. CVS repositories do not store this information [11] but it can be extracted by grouping all files that were changed with the same message, by the same author, and within a certain timestamp range – we use a sliding window of three minutes, as in[8]. The ***density of (co-)changes*** is the number of (co-)changes in a period over the length of the period, measured in days. A ***period*** is the set of not necessarily contiguous days when there was (not) a cloned fragment.

CloneTracker can be downloaded from http://mcs.open.ac.uk/alr242 and analyzes Java applications with their history available in a CVS repository. The tool uses third party tools: CCFinder to detect clones, CTAGS to detect where methods start in a source code file, and CVS commands to extract information from the source code repository.

CloneTracker generates three intermediate files (see Figures 1, 2 and 3) to calculate the number and density of (co-)changes for the (pairs of) methods that have had cloned fragments.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	
–	+	+	–	+	–	–			m1()
				–	–	+	+	+	m2()
	–	+	–	+	–	–	+	–	m3()

Figure 1. Record of changes

–	–	1.0	1.0	1.0	1.0	1.0	–	–	m1()
–	–	–	–	0.7	0.7	0.7	0.7	0.7	m2()
–	–	0.3	0.3	0.3	0.3	0.3	0.3	0.3	m3()

Figure 2. Record of cloning percentage

–	–	30	30	30	30	30	–	–	m1()m3()
–	–	–	–	40	40	40	40	40	m2()m3()

Figure 3. Record of sizes of cloned fragments

Fig. 1 illustrates the record of changes; each row is a method and each column a transaction. If the method did not exist after a transaction its corresponding cell is a blank character; otherwise it is either a plus character, if it changed, or a minus character, if it did not change. For instance, method m1() wasn't changed by the first transaction but it was by the second one, and it was removed by the eighth transaction.

Fig. 2 illustrates the record of cloning percentages. Each row is for a method that has had cloned fragments. Each cell shows the percentage of the method's code that is cloned – number of lexical tokens cloned over number of tokens. The table just records the total amount of a method's code that is cloned; it doesn't matter whether different parts of the method are cloned from different other methods. If the method is not cloned by that transaction there is a minus character.

Fig. 3 illustrates the record of the sizes of common code snippets between method pairs. Each row is a pair of methods that have shared cloned fragments. Each cell shows the number of tokens that compose the cloned fragment. If the two methods do not share any cloned fragment there is a minus character.

From the intermediate data, our tool generates four text files that contain the number (Fig. 4 left) and density of changes (Fig. 4 right) for methods that had a cloned fragment at some time during their lifetime, and their equivalent for pairs of methods, that is number (Fig. 5 left) and density of co-changes (Fig. 5 right). For all these files, the second column shows the values for the period where the method (pair) was cloned,

while the third column corresponds to the period without cloning.

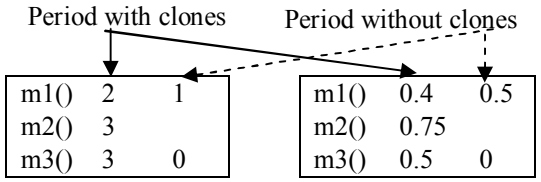

Fig. 4. Change number (left) and density(right)

If a method always had a cloned fragment, there are no two different periods to compare. In such cases, the number of changes is omitted, which is different from a zero result. For example, method m2() was always cloned (compare Figures 1 and 2) and therefore the third column in Fig. 4 left is empty. By contrast, method m3() was not cloned for some time (transaction 2), but never changed in that period, hence the zero the third column.

The density of changes is determined by dividing the number of changes of a period over the number of days on that period. Supposing that the transactions occurred one day after the other, the densities would be as Fig. 4 right shows. For example, method m1() changed once (transaction 2) in its interval without clones (transactions 1 and 2). Hence, the density is 0.5 (1 change in 2 days). If the method did not have a period without cloned fragments, the corresponding density is not computed.

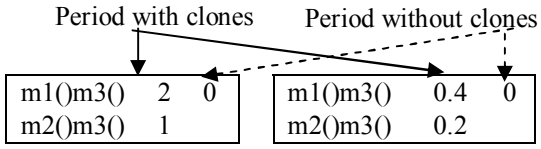

Fig. 5. Co-change number and density

Given the output files, we count how many methods (and pairs) increase, decrease, and maintain the amount and density of (co-)changes when cloned versus when not cloned. The methods and pairs that do not have two distinct periods (i.e. are always cloned) are not taken into account as they do not have a counterpart to compare with. In this example, 33% of the methods, namely method m2(), is eliminated due to lack of distinct periods.

4. Case study

We selected DnsJava, an implementation of a domain name system that has already been used for similar experiments [8]. It is still an active project, with two developers, that has evolved over 99 months and has currently over 21KLOC. DnsJava has an average activity level of 13.3 transactions per month. The data for this paper took less than 15 hours to gather on a computer with an Athlon 64 processor at 2.4 GHz and 1GB of RAM. All the output files (Figures 1 to 5) are publicly available to support other researchers in their own studies.

Table 1 has a classification of methods and method pairs according to their change and cloning characteristics. It shows, for example, that 68% of methods never changed (1st column) and that only 26% of methods had two periods (2nd row).

Table 1. Change and cloning characteristics

Methods (4890)	Never changed	Sometimes changed
never cloned	2232 (46%)	1041 (21%)
sometimes cloned	761 (16%)	483 (10%)
always cloned	309 (6%)	64 (1%)
Method pairs (4890 * 4889)	**Never co-changed**	**Sometimes co-changed**
never shared a clone	23,881,024	23,723
sometimes shared a clone	145	103
always shared a clone	1938	277

It only makes sense to compare the (pairs of) methods that had two periods and (co-)changed at least once. For the case, that means only 483 of all methods and just 103 of the method pairs can be analysed.

Table 2 shows that 75% of the considered methods are changed more times and 82% are changed more frequently when they are cloned. A possible explanation is that the methods were subject to extra changes due to co-changes of their cloned snippets.

The results in Table 2 show that 56% (resp. 69%) of the considered method pairs are co-changed more often (resp. more frequently) when they do *not* share any cloned fragment. This seems to indicate that the programmer is not aware of the pairs of methods that share the same snippet. However, this explanation is not very convincing: with only two developers on the project throughout its 8 year lifetime, each one should know the code relatively well.

Table 2. Comparison between periods

	for the period with a cloned fragment		
	is greater	is lower	is equal
Number of changes	75%	20%	5%
Density of changes	82%	18%	0%
Number of co-changes	34%	56%	10%
Density of co-changes	31%	69%	0%

Among the methods that have changed at least once (Table 1 right hand column), we have compared the average amount and density of change between methods that were never cloned and those that were cloned at least once in order to see if cloning

0-7695-2950-X/07 $20.00 © 2007 IEEE

introduced any substantial difference to the base rate of change.

As one can see on Table 3, both the average amount and density increased, by 89% and 68%, respectively. All comparison metrics indicate that methods that have been cloned tend to change more and more often than those that have not had any clone.

Table 3. Number and density of changes

	Methods always or sometimes cloned		Methods never cloned	
	Amount of changes	Density of changes	Amount of changes	Density of changes
Avg.	2.75	1 / 746	1.86	1 / 1258
Median	2	1 / 1400	1	1 / 2083
Mode	1	1 / 3012	1	1 / 3030
Min.	1	1 / 3012	1	1/ 5556
Max.	30	1 / 74	17	1 / 100

5. Concluding remarks

Relating evolution and cloning is necessary to refute or prove the claim that clones lead to an increased maintenance effort. Previous work either relates cloning to changes at file level or disregards part of the system's history. As far as we know, there was no work comparing the change frequency of cloned vs. non-cloned code at a fine level of granularity.

We developed CloneTracker, a tool that computes which methods are cloned and which methods are changed in each transaction. Then we compared the amount and density of changes in the cloning period against the non-cloning period. We used a small preliminary case study to test our tool.

Only a small fraction of the (pairs of) methods had both cloning and non-cloning periods. In those cases, we observed that the vast majority of methods indeed changed more, and more frequently, when they have cloned code. However, the amount and density of co-changes between method pairs decreased when they were cloned. These results *seem to support* the belief that cloned code leads to more changes, because the various copied fragments have to be changed consistently, but that programmers are not aware of the clone's existence and therefore the necessary changes are made in a delayed way, not simultaneously. However, this explanation does not seem very plausible for this case study, given that all the code was programmed by just two developers.

It is too early to make any be conclusive at this stage. For future work we plan to conduct the same experiments for more and larger applications. To improve the accuracy of results we plan to track method renaming using the approach proposed by Godfrey and Zou in [4].

6. References

[1] Antoniol, G., Di Penta, M., Merlo, E.: An automatic approach to identify class evolution discontinuities. In *Proc. of the Int'l Workshop on Principles of Software Evolution*, 2004, pp. 31-40.

[2] Ducasse, S., Rieger, M., Demeyer, S.: A Language Independent Approach for Detecting Duplicated Code. In *Proc. of the Int'l Conf. on Software Maintenance*: IEEE Computer Society, 1999, p 109.

[3] Geiger, R., Fluri, B., Gall, H.C., Pinzger, M.: Relation of Code Clones and Change Couplings. In *Proc. of the Int'l Conf. of Fundamental Approaches to Software Engineering*, Vienna, Austria: Springer, March 2006, pp. 411-425.

[4] Godfrey, M.W., Zou, L.: Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Trans. Softw. Eng.*, 31(2) (2005), 166-181.

[5] Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7) (2002), 654-670.

[6] Kapser, C., Godfrey, M.W.: 'Cloning considered harmful' considered harmful. In *Proc. of the Working Conf. on Reverse Engineering*, Benevento, Italy, 23-28 October 2006.

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

[8] Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In *Proc. of the European Software Engineering Conference*, Lisbon, Portugal: ACM Press, 2005, pp. 187-196.

[9] Lague, B., Proulx, D., Mayrand, J., Merlo, E.M., Hudepohl, J.: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proc. of the Int'l Conf. on Software Maintenance*: IEEE Computer Society, 1997, pp. 314-321.

[10] Walenstein, A., Lakhotia, A., Koschke, R.: The Second International Workshop on Detection of Software Clones: workshop report. *SIGSOFT Softw. Eng. Notes*, 29(2) (2004), 1-5.

[11] Zimmermann, T., Weibgerber, P.: Preprocessing CVS data for fine-grained analysis. In *Proc. of the 1st Int'l Workshop on Mining Software Repositories*, 2004, pp. 2-6.

0-7695-2950-X/07 $20.00 © 2007 IEEE