A Tool for Detection of Similarities in Large Software Systems (preliminary report)

T. Kowaltowski and Ranieri C. Pereira Filho

Technical Report - IC-09-26 - Relatório Técnico

August - 2009 - Agosto

A Tool for Detection of Similarities in Large Software Systems (preliminary report)

Tomasz Kowaltowski[*‡] Ranieri C. Pereira Filho[*†‡]

July 2009

Abstract

Reuse of program parts through cloning and adaptation is a common practice in software development. However it creates later problems in system maintenance. This report describes the main ideas behind a tool developed to assist in identification of software clones based on some techniques used in detecting program plagiarism.

1 Introduction

Cloning of program parts occurs quite often when software developers try to reuse components without due attention to future maintenance and modification problems. In principle the problem should be avoided by strict adherence to software development rules and support for component based development. However in practice this problem is very serious and its consequences are quite costly. There are several references in the literature addressing this issue. For instance Basit and Jarzabek [2] offer a broad view of the problem and an extensive bibliography related to this subject.

We describe in this report an experimental tool for detecting software clones in large programming systems. Our tool is specialized for Java because of the client requirements but its general principles are applicable to any programming language. Its main ideas are based on techniques used in detecting program plagiarism. In its essence both cloning and plagiarism are very similar, except that, in most cases, cloning is socially acceptable whereas plagiarism is obviously not. As a consequence a plagiarist tries to hide his actions by applying several transformations to the program: reformatting, changing identifiers, reordering declarations, and so on. On the other hand, a programmer who clones a software component usually makes minor adaptations without need to hide this fact. Even so the techniques used in detecting plagiarism turn out to be very useful for cloning detection. Some of the ideas we borrowed are described briefly in Kleiman and Kowaltowski [8].

[*]Institute of Computing, University of Campinas, Caixa Postal 6176, 13081-970 Campinas, SP, Brazil.

[†]Sensedia, Polis de Tecnologia, Rodovia Campinas-Mogi Mirim, Km 118,5, 13086-902 Campinas, SP, Brazil.

[‡]The project was partially supported by the grant 2006/52132-4 from the Fundação de Amparo à Pesqisa do Estado de São Paulo (FAPESP) and by Sensedia.

It should be noticed that our tool uses in its first stage the concept of tokenization which was also proposed by other authors; see for instance Basit et al. [3]. However we use simpler, even though theoretically less efficient, algorithms to manipulate resulting code description strings. In the following sections we describe the main ideas behind our tool and present some experimental results. Our description is based on the assumption that the system to be analyzed was developed in Java.

2 General overview

The starting point for our tool is the directory which contains the package or packages to be analyzed. There are several parameters which can be specified in order to allow for partial inclusion or exclusion of some subdirectories. In addition packages, classes and/or methods may be excluded by specifying their names through regular expressions. These features are useful for instance when parts of the system are built by automatic program generators or when common methods such as *getters* and *setters* should not be considered. Several other parameters may be set such as minimal signature length, string comparison algorithm to be used, more detailed treatment of expressions, inclusion or not of certain constructs like interfaces, constructors and so on. Some of these parameters will be commented further on.

The main choices for program analyses are:

- class similarity analysis
- method similarity analysis within classes
- method similarity analysis across classes

There are some additional operations which help in debugging and tuning the system such as simple signatures generation and string algorithm comparison. As a side product we also implemented method cyclomatic numbers calculation which can be easily computed from our token strings.

The first step in all of our analyses is the construction of token strings, called *signatures*, for all classes and/or methods selected. The next stage is the construction of *similarity clusters* which represent sets of classes or methods which are suspect to be clones. The final step is the exhibition of the results. The same way as it happens with plagiarism, the final decision about similarity cannot be fully automated and must be taken by a human user.

3 Signature generation

In this phase our tool generates, for each selected unit (class, interface, method, constructor, static block), a string of characters which is its *signature*. Each character of the signature represents a language lexical token such as `while`, `if`, {, etc. Comments and formatting are ignored, and identifiers are treated as a unique token except for units' naming.

The inclusion or not of certain tokens is based on a careful analysis of their meaning, and is language dependent. For instance, we do not include type tokens such as `int` and `String`. Treatment of expressions depends on execution options, but explicit method invocations are always included. It is interesting to notice that in plagiarism detection expression details are frequently ignored because they are very easy to modify without changing their meaning. Also we do not include many qualifiers such as `public` and `private`.

Signature generation is, in principle, a very straightforward and efficient task and could be performed easily by a simple lexical analyzer (*scanner*). However, depending on the language, it may be necessary to know certain facts about the surrounding context. In our system we decided to use a full syntax analyzer (*parser*) in order to simplify this task and also to make it easier for any future extensions. We chose the Java parser provided by Gesser [5] which uses the Java Compiler Compiler (JavaCC) [6] and generates an abstract syntax tree (*AST*). Our methods traverse recursively the resulting tree and produce the signatures. As shown further on this phase of our tool execution is very fast and has practically no bearing on the total execution time.

4 Computation of similarities

An important part of our similarity clusters construction is the algorithm used to compute the similarity of two signature strings s_1 and s_2. Given such signatures, the first step is to compute the total length q of a subset of the common substrings[1] of s_1 and s_2. The exact subset depends on the algorithm used as explained below. Once the value of q is computed, we have the choice of three natural ways of calculating the similarity S of s_1 and s_2:

- $S = \min(q/p_1, q/p_2)$
- $S = 2 * q/(p_1+p_2)$
- $S = \max(q/p_1, q/p_2)$

where p_1 and p_2 are the lengths of s_1 and s_2. Thus our similarity measure is always a value between 0 and 1. The exact choice might depend also on other options like minimum signature length and so on. The three measures are listed in order of decreasing strictness.

We implemented three algorithms to compute the total length of a relevant subset of the common substrings of two strings. All these algorithms were inspired by some techniques used in plagiarism detection:

- count of occurrences of common tokens (*CCT*)
- a modified longest common subsequence algorithm (*LCS*)
- greedy string tiling algorithm (*GST*)

The CCT algorithm is the simplest one. It consists of counting the number of occurrences of the same token in both strings and accumulating the minima of the two numbers for all tokens. One of the obvious drawbacks of this algorithm is that it depends only on the number of occurrences of the tokens; the structure of the program fragment being compared has no influence on the outcome. On the other hand it is extremely efficient.

[1]Here we use the term *substring* to denote a sequence of consecutive characters within a string; some authors use the term *segment*.

The classical LCS problem has a well known textbook solution based on *dynamic programming* (see for instance Cormen et al. [4]). Our modification of the algorithm consists of introducing an additional constraint which requires that each common substring, which we call *tile*, should have a minimal length $m \geq 1$; thus the usual LCS problem is a particular case with $m = 1$ (Kowaltowski [9]). One of the drawbacks of the LCS algorithm is that the tiles it determines appear in the same order in both strings and thus it is sensitive to reordering of program constructs such as declarations. However it is more efficient than the GST algorithm.

The tiling GST algorithm and its more efficient variant RKRGST are described by Wise [11] and are the basis of the JPlag plagiarism detection system [7, 10]. Given two strings s_1 and s_2, the algorithm determines a set of common tiles, such that their total length is maximal. These tiles do not appear necessarily in the same order in both strings which solves the reordering sensitivity problem, at some additional computational time cost. The algorithm finds an optimal solution when the minimal tile length m imposed is 1. Usually higher values are adopted but the practical results are very satisfactory. Another remark is that GST with $m = 1$ is obviously equivalent to CCT; this fact reinforces the idea that higher values of m should be used in order for GST (and LCS) to produce meaningful results.

The reordering sensitivity aspect is quite important in the context of plagiarism when reordering of declarations is a common hiding technique. In the context of clone detection it seems to be in general less important. Our system provides thus the three algorithms and the user can choose which one should be applied. Our experience shows that, under normal conditions, both algorithms LCS and GST produce very similar final results, but LCS is much faster. Since CCT always produces the highest similarity values, it is also used as a filter in order to avoid unnecessary LCS or GST calculations which would result in values below the similarity threshold (see the next section).

5 Cluster construction

Once the set of relevant signatures is computed as described in Section 3, our tool builds a collection of clusters. We define a *similarity cluster* as a set of signatures (i. e. classes or methods, depending on the analysis carried out) such that their pairwise similarities are above a certain prescribed threshold which can be set as a parameter. The typical value is above 0.8 (80%).

In order to build the collection of clusters we use a heuristic approach. We sort the signatures by their lengths and then process them sequentially starting with the longest one. For each signature s we have two possible cases:

- s fits an already existing cluster: s is then included in that cluster;
- s does not fit any of the already existing clusters: s becomes the first member of a new cluster.

We start with an empty collection of clusters. A signature s fits an existing cluster if its similarity to each cluster member is above the threshold. One of the options of our tool is to allow or not for a signature to belong to more than one cluster.

Before we apply the similarity algorithm we compare the length of s with that of the shortest signature in the cluster; obviously their difference must lie within the threshold. This simple check decreases drastically the execution time. Additional gains are provided by using the CCT algorithm as a filter as already mentioned in Section 4. At the end clusters containing only one signature are discarded.

It should be noticed that our heuristics produces a collection of clusters satisfying the definition but this solution is not necessarily unique. Different processing orders may produce different results, but in practice these differences do not seem to be significant.

Cluster construction consumes most of the execution time. Both algorithms used to determine similarities have execution times at least proportional to the product of the lengths of the two signatures. In the worst case (when all signatures have the same length), cluster construction might require pairwise similarity comparison of all signatures, and thus result in N^2 similarity computations where N is the total number of signatures under analysis. In practice, the distribution of signature lengths is very random and a very small fraction (in general less than 1% in our tests) of the similarity calculations is carried out.

6 Experimental results

Our experimental results are still preliminary. We applied the beta version of our tool to several systems and the results are encouraging. As one of the examples we processed a publicly available package `apache-tomcat-6.0.18-src` [1] released by the Apache Software Foundation.

In this experiment we concentrated on class similarity analysis (see Section 2) which seems to be the most interesting. In what follows we present and comment some of quantitative processing results. The tests were executed on a standard PC with an Intel(R) Core(TM) Duo CPU, 2.66GHz, 2.8 GB RAM, running Linux kernel version 2.6.28 and SUN Java 1.6.0_07.

As already mentioned our system offers many different options and parameters which influence the collection of similarity clusters which are determined. As a matter of fact the determination of the right combination of these options is part of our ongoing study. For the purpose of this example we decided to explore several choices in order to compare the results:

- minimum signature lengths s: 10, 20, 50, 100, 200, 500, 1000 and 2000
- minimum tile lengths m: 3 and 6
- similarity thresholds t: 0.8, 0.9 and 1.0
- similarity formula: $S = \min(q/p_1, q/p_2)$
- detailed treatment of expressions
- inclusion of constructors
- algorithms: GST and LCS

It should be noticed that the similarity definition adopted in this experiment is the strongest one. Our system generated from this example 1146 class signatures (including the inner classes) with the following lengths distribution (the longest signature has 5343 characters):

- less than 10: 178
- 10 or more: 968
- 20 or more: 852

- 50 or more: 650
- 100 or more: 493
- 200 or more: 342
- 500 or more: 159
- 1000 or more: 60
- 2000 or more: 18

The table shown on the next page summarizes some of the quantitative results (no clusters were found for $s = 2000$):

For each algorithm used, the first figure denotes the number of clusters identified. The figure between parentheses is the number of clusters included by this algorithm but not included in the other algoritm (this allows for the comparison of the two algorithms). The next number below is the size of the largest cluster found and finally the last number is the approximate execution time in seconds. We do not show the numbers but almost in all cases the number of string comparison applications eliminated by our heuristics is above 99%.

We can draw several conclusions from our results. As expected the GST algorithm identifies a larger number of clusters than LCS, but the difference is rather small and tends to disappear as the value of the threshold t is increased. The few cases in which the LCS algorithm seems to have produced clusters that were not found by GST are somewhat misleading: in every case such a cluster found by LCS is covered by a larger cluster found by GST.

Another conclusion is that using $m = 3$ or $m = 6$ does not affect very significantly the results, but the number of clusters determined for a higher value of $m = 6$ is somewhat smaller which is also to be expected.

With regard to execution times the LCS algoritm is significantly more efficient. The difference is in general of an order of magnitude. It should be noticed that the signature generation time was always about 5 seconds.

We also examined some of the clusters that were found. For instance two of the five clusters found for $s = 50$ and $t = 1.0$ are (they are the same for all choices of parameters):

- Cluster 1: signatures length 90
  classes:
  `org.apache.tomcat.util.digester.SetNextRule.java`
  `org.apache.tomcat.util.digester.SetTopRule.java`
- Cluster 2: signatures length 180
  classes:
  `org.apache.catalina.tribes.util.StringManager.java`
  `org.apache.catalina.util.StringManager.java`

Both classes in the first cluster are identical except for the word and identifier pairs systematic exchanges: `top` and `next`, and `child` and `parent`. Clearly one of them was produced by copying the other one (including comments) and minor editing. In the case of the second cluster, both classes are identical except obviously for their `package` specification.

	$t = 0.8$				$t = 0.9$				$t = 1.0$			
	$m = 3$		$m = 6$		$m = 3$		$m = 6$		$m = 3$		$m = 6$	
	GST	*LCS*	*GST*	*LCS*	*GST*	*LCS*	*GST*	*LCS*	*GST*	*LCS*	*GST*	*LCS*
$s = 10$	76	69	62	61	51	50	42	42	25	25	25	25
	(9)	(2)	(2)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	6	6	7	7	6	6	6	6	5	5	5	5
	77.9	7.8	60.3	8.4	3.5	0.9	5.3	1.0	1.7	0.8	1.7	0.4
$s = 20$	60	53	51	50	40	39	33	33	18	18	18	18
	(8)	(1)	(2)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	4	4	4	4	4	4	4	4	4	4	4	4
	76.5	7.9	62.4	9.6	3.0	0.7	4.5	0.8	1.5	0.3	0.7	0.7
$s = 50$	32	27	26	25	19	18	17	17	5	5	5	5
	(6)	(1)	(2)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	3	3	3	3	3	3	3	3	2	2	2	2
	70.2	7.3	54.7	7.8	4.5	0.9	4.8	0.7	1.2	0.2	1.1	0.3
$s = 100$	22	18	19	18	13	12	11	11	1	1	1	1
	(5)	(1)	(2)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	3	3	3	3	3	3	3	3	2	2	2	2
	68.6	7.2	53.7	7.1	4.1	0.7	4.5	0.6	0.2	0.2	0.4	0.2
$s = 200$	15	14	14	14	10	9	9	9	0	0	0	0
	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	3	3	3	3	3	3	3	3	0	0	0	0
	71.4	7.6	57.4	7.1	4.0	0.6	4.6	0.6	0.8	0.1	0.8	0.0
$s = 500$	9	8	8	8	4	3	3	3	0	0	0	0
	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	3	3	3	2	2	2	2	2	0	0	0	0
	63.0	6.2	49.0	6.2	4.0	0.5	4.6	0.6	0.4	0.0	0.2	0.0
$s = 1000$	4	3	3	3	2	1	1	1	0	0	0	0
	(1)	(0)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
	3	3	3	2	2	2	2	2	0	0	0	0
	41.0	4.1	33.8	4.4	3.6	0.6	5.1	0.6	0.0	0.0	0.1	0.0

As expected $t = 1.0$ imposes a very extreme similarity criterion, and we get some more interesting results for lower values. If we look at the combination $s = 50$ and $t = 0.9$, one of the clusters covers the Cluster 1 above with the additional class `org.apache.tomcat.util.digester.SetRootRule` of slightly shorter length of 89. Its similarity to the two other classes is $S = 0.98$ and again it is identical to the two other classes except for word and identifier exchanges and a minor difference in one expression (access to a parent object). As a matter of fact this last difference would not appear if we chose the option of less detailed treatment of expressions.

Most clusters include only two classes. For low values of s larger clusters appear more often; for $s = 10$ and $t = 0.8$ we have the cluster (signatures lengths in parentheses):

`org.apache.el.parser.AstDiv` (14)
`org.apache.el.parser.AstMinus` (14)
`org.apache.el.parser.AstMod` (14)
`org.apache.el.parser.AstMult` (14)
`org.apache.el.parser.AstPlus` (14)
`org.apache.el.parser.AstEqual` (15)
`org.apache.el.parser.AstNotEqual` (16)

Another interesting case is the cluster found also for $t = 0.8$:

`org.apache.coyote.ajp.AjpProtocol$AjpConnectionHandler` (173)
`org.apache.coyote.ajp.AjpAprProtocol$AjpConnectionHandler` (178)
`org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler` (195)

All three are inner classes (as denoted by their qualified names with \$). Again their visual inspection shows that they were produced as copies and then went through some adaptations.

7 Conclusions

Our preliminary results show that the system can be a valuable tool in detecting cloning program parts. Depending on the size of the system analyzed and the choice of parameters the execution times can be significant (order of minutes). However we envision that such a tool would be applied to a system not more than once every few days during its development so that such times are not prohibitive.

An important conclusion is the obvious need to display the results in a graphical way so that eventual candidates for refactoring can be easily identified starting from the clusters determined by the system. The exhibition model used by the abovementioned JPLAG system provides a nice example of such a facility. It should be noticed that both algorithms GST and LCS provide the set of tiles necessary for such display of results.

Acknowledgments

The authors would like to thank José Vahl, José Ventura and Kleber Bacili from Sensedia, for their support in developing this project.

References

[1] `http://tomcat.apache.org`. Visited on May 12,2009.

[2] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC-FSE'05*, 2005.

[3] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *ESEC-FSE'07*, 2007.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[5] Júlio Vilmar Gesser. Java 1.5 Parser and Abstract Syntax Tree. `http://code.google.com/p/javaparser`.

[6] Java Compiler Compiler. `https://javacc.dev.java.net`.

[7] JPlag. `https://www.ipd.uni-karlsruhe.de/jplag/`.

[8] Alan Bustos Kleiman and Tomasz Kowaltowski. Qualitative analysis and comparison of plagiarism-detection systems in student programs. Technical Report IC-09-08, Institute of Computing, University of Campinas, March 2009. `http://www.ic.unicamp.br/~reltech/2009/09-08.pdf`.

[9] Tomasz Kowaltowski. Longest common subsequence with minimum length subsegments, 2008. Private communication.

[10] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, 8(11), 2002.

[11] Michael J. Wise. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. Unpublished, 1993.