Measuring clone based reengineering opportunities

Magdalena Balazinska[1], Ettore Merlo[1], Michel Dagenais[1], Bruno Lagüe[2] and Kostas Kontogiannis[3]

[1]Department of Electrical and Computer Engineering, École Polytechnique de Montréal,
P.O. Box 6079, Downtown Station, Montreal, Quebec, H3C 3A7, Canada
http://www.casi.polymtl.ca
e-mail: magda@casi.polymtl.ca {ettore.merlo,michel.dagenais}@polymtl.ca

[2]Bell Canada, Quality Engineering and Research Group
1050 Beaver Hall, 2nd floor, Montreal, Quebec, H2Z 1S4, Canada
e-mail: bruno.lague@bell.ca

[3] Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
e-mail: kostas@amorgos.uwaterloo.ca

Abstract

Code duplication, plausibly caused by copying source code and slightly modifying it, is often observed in large systems.

Clone detection and documentation have been investigated by several researchers in the past years. Recently, research focus has shifted towards the investigation of software and process restructuring actions based on clone detection.

This paper presents an original definition of a clone classification scheme useful to assess and measure different system reengineering opportunities. The proposed classification considers each group of cloned methods in terms of the meaning of the differences existing between them.

The algorithm used for automatic classification of clones is presented together with results obtained by classifying cloned methods and measuring reengineering opportunities in six freely available systems whose total size is about 500 KLOC of Java code.

1 Introduction

Source code reuse in object-oriented systems is made possible through different mechanisms such as inheritance, shared libraries, object composition, and so on. Some designs, namely the well-known design patterns [7] particularly facilitate reuse. Nevertheless programmers often need to reuse components which haven't been designed to be reused. This happens mostly when software systems go through the expansion phase and new requirements have to be satisfied periodically [7].

When such a situation arise, ideally, the modules involved should be restructured and the component properly reused. Even better, the whole system could be reorganized, classes could be refactored into general components and their interfaces rationalized. Such a process is known as consolidation and allows a system to become more flexible and easier to expand [7]. Unfortunately, often the process used instead is ”cut-and-paste”, i.e. performing some sort of reuse by manual source code inlining. This other approach produces what we call cloned pieces of code, or clones which will undergo independent successive maintenance.

Previous research has studied both the detection and the use of clones for widely varying purposes including program comprehension, documentation, quality evaluation or system and process restructuring. Several techniques have been investigated in the literature for the detection of clones in software systems. Some techniques are based on a full text view of the source code. Johnson [8] has developed a method for the identification of exact duplications of substrings in source code using fingerprints whereas Baker's tool, ”Dup” [2], reports both identical sections of code and sections that differ only in the systematic substitution of one set of variable names and constants for the other.

Other approaches, such as those pursued by Mayrand et al. [13] and Kontogiannis et al. [10] focus on whole sequences of instructions (BEGIN-END blocks or functions) and allow the detection of similar blocks using metrics. Those metrics relate to aspects of sequences of instructions such as their layout, the expressions inside them, their control flow, the variables used, the variables defined, etc.

In [10], Kontogiannis et al. detect clones using two other pattern matching techniques namely dynamic programming matching which finds the best alignment between two code fragments, and statistical matching between abstract code descriptions patterns and source code.

Yet another clone detection technique relies on the comparison of subtrees from the AST (Abstract syntax tree) of a system. Baxter et al. [3] have investigated this technique.

Several applications of clone detection have also been investigated, Johnson [8] visualizes redundant substrings to ease the task of comprehending large legacy systems. Mayrand et al. [13] as well as Lagüe et al. [11] document the cloning phenomenon for the purpose of evaluating the quality of software systems. Lagüe et al. [11] have also evaluated the benefits in terms of maintenance of the detection of cloned methods. Finally, Baxter and al. [3] restructure systems by replacing clones with macros to reduce the quantity of source code and facilitate maintenance.

The purpose of our research is to investigate the use of clones as a basis for those reengineering actions which are useful to the maintenance of systems. Examples of such reengineering activities include source code transformation, migration from procedural to object-oriented, or simply code restructuring. To achieve this goal, the assessment of reengineering opportunities based on clone information have been investigated. Indeed, before performing any concrete reengineering, the opportunities for such activities have to be determined. Such opportunities have been defined as groups of cloned methods whose characteristics, namely the differences between the copies, can give rise to a concrete reengineering action (parameterization, delegation, moving operations in the class hierarchy, etc.).

To support the approach, a new classification scheme has been developed and is presented in Section 2. It has been implemented in a tool, SMC (Similar Methods Classifier) using algorithms presented in Section 3.

SMC has been applied to six software systems accounting for approximately 500 000 lines of code. Section 4 details the experiment conducted whereas Sections 5 and 6 present and discuss the results. From those results, a definition of a high impact category is developed in Section 6. Such categories are those which are particularly favorable to reengineering based on clone information. The value of the detection of such categories is also discussed.

2 Cloned methods classification

A clone classification scheme has been presented by Mayrand et al. in [13]. It defines an ordinal scale of eight cloning levels based on the degree of similarity between cloned functions. This degree is a function of the names of the clones, their layout, the expressions inside them and their control flow. This classification scheme has been designed for software evaluation purposes.

For reengineering activities based on clone information, other characteristics have to be taken into consideration. For a reengineering action to be performed, a detailed knowledge of the characteristics of clones, namely the differences between copies of a method have to be available. Therefore, we have developed a new classification scheme by taking the meaning of such differences into consideration.

The classification scheme has been determined after manually examining some 800 cloned methods, extracted from six freely available Java software systems which are described in Section 4. The clones used for the elaboration of the classification were extracted from the systems using Patenaude et al.'s [17] approach which extracts and groups similar methods using metrics.

During the observation phase, the differences existing between clones have been listed. It has been noted that many clones are strictly identical or contain very superficial differences, i.e. differences that affect neither the output produced by the method nor its behavior. Therefore, categories "Identical" and "Superficial changes" have been defined for such clones.

For the other differences, three categories clearly appeared: differences affecting only one lexical token at a time, differences affecting sequences of tokens and differences affecting attributes of methods (public, static, synchronized, list of thrown exceptions, etc.). The first group of differences have been further subdivided using the meaning of the single token differences (type of a variable, name of a parameter, etc.). This has lead to the following definitions of categories 3 through 9:

- Called methods: The single-token differences existing between the clones correspond to method calls. Stated differently, when clones differ only in some method calls, they belong to this category.
- Global variables: The single-token differences correspond to non-local variables or constants.
- Return type: The single-token difference corresponds to the return type.
- Parameters types: The single-token differences correspond to parameter types.
- Local variables: The single-token differences correspond to the types of local variables.

Table 1. Cloned methods classification

Category number	Type of clones
1	Identical
2	Superficial changes
3	Called methods
4	Global variables
5	Return type
6	Parameters types
7	Local variables
8	Constants
9	Type usage
10	Interface changes
11	Implementation changes
12	Interface and implementation changes
13	One long difference
14	Two long differences
15	Several long differences
16	One long difference, interface and implementation
17	Two long differences, interface and implementation
18	Several long differences, interface and implementation

- Constants: The single-token differences correspond to constants hard-coded in the methods.
- Type usage: The single-token differences correspond to types explicitly manipulated in expressions such as "instanceof" or "typecast".

Some clones differ in several of the previously defined single token entities. For those clones, categories 10 through 12 have been defined:

- Interface changes: The single-token differences correspond to called methods and/or global variables and/or parameters types and/or return type.
- Implementation changes : The single-token differences correspond to types of local variables and/or constants used and/or types explicitly manipulated.
- Interface and implementation changes: The single-token differences correspond to any difference used in the definition of the previous categories.

For the differences affecting sequences of tokens, it has been noted that most cloned methods either contain only one or two of such differences or are completely different. Therefore categories 13, 14 and 15 have been defined as follows:

- One long difference: Only one entity (an expression, a statement or other) is affected by a long difference.
- Two long differences: Exactly two entities are affected by long differences.
- Several long differences: Three or more entities are affected by long differences.

Some entities are more important in size than others, though. For two completely different methods, one could say that they differ in one entity, the whole method, but this wouldn't be of much help for reengineering. Therefore, we have defined a threshold for the maximum percent of differences in a method. We have used an arbitrary threshold of 30% but this threshold can be further refined.

It has also been noted that many cloned methods differed in both single token and sequences of tokens differences. Hence, categories 16, 17 and 18 have been defined as the unions of the definitions of the three previous categories (13, 14 and 15 respectively) with category 12 ("Interface and implementation changes").

Finally the differences affecting the list of thrown exceptions and the attributes of methods (public, static, etc.) have been kept as parameters for the reengineering phases because they don't affect directly the choice of appropriate reengineering actions.

The categories defined are summarized in Table 1. All similar methods according to the metrics, that didn't correspond to any of the previous definitions are currently not categorized and kept for further research. Nevertheless, the analysis of some distributions of cloned methods (c.f. Section 4) showed that the classification covers a wide spectrum of cloning cases.

Moreover, the results presented in Figure 7 show that when several systems are considered, there exists clones of every category defined by the scheme except category 7 ("Local variables") as will be discussed later. This result validates the choice of the categories.

3 Comparison and classification algorithms

This section presents the algorithms that allow to classify similar methods automatically. We first describe the source code representation used to get information on the meaning of the result of comparisons. We then discuss, the choice of the comparison algorithm. We finally present the comparison algorithm followed by the classification algorithm.

3.1 Source Code Representation

The first step towards analyzing a software system is to represent the code in a higher level of abstraction. A number of program representation schemes have been proposed in

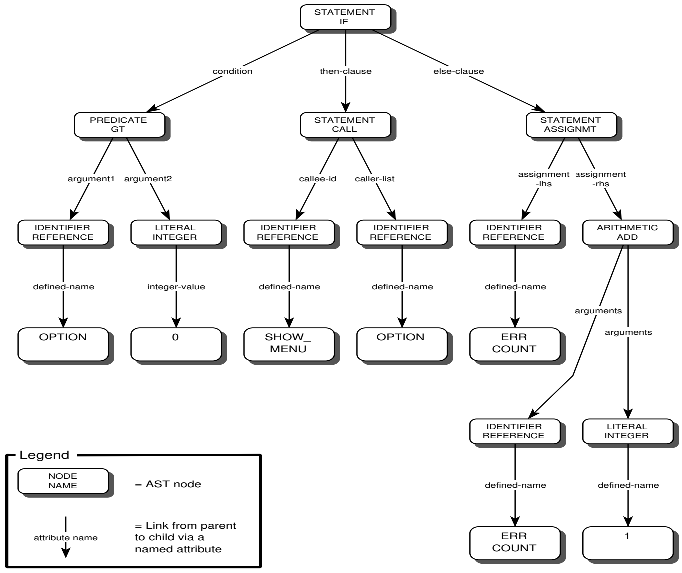

Figure 1. A sample AST.

the relevant literature. These include frames [16], annotated data and control flow graphs [21], Abstract Syntax Trees [15], logic formulas on program dependencies [5] and, relation tuples based on a language domain model [14].

We have chosen as a program representation scheme, the program's annotated abstract syntax tree (AST). We believe that this scheme is most suitable because:

- it does not require any overhead to be computed as it is a direct product of the parsing process and,
- it can be easily analyzed to compute several data and control flow program properties

The tree is created during the parsing phase, and is annotated in a post-processing phase where linking information is added. An example AST is illustrated in Figure 1

3.2 Code Segmentation, Preprocessing

One of the important issues in Code Cloning detection is the segmentation of the source code. This issue relates to which parts of the system are selected for matching. One technique is to use a sliding window, and try to match all possible combinations of source code text which may be of a certain length (i.e. the length of the sliding window). Another way is to use some type of preprocessing so that potential clones can be identified first.

In Buss et. al [4] a metrics based approach has been proposed. The metrics based matching technique is based on the assumption that if two code fragments are clones, then they share a number of structural and data flow characteristics that can be effectively classified by these metrics. It has been shown that the metrics-based approach provides a fast approximation of the code cloning recognition problem. Experimental results [9] have indicated that we can effectively retrieve 60% of the code cloning instances sought, and maintain a *precision* of approximately 41.0% at the final results. The drawback on the speed and ease of use in the metrics based method is that at higher *recall* levels noise can be introduced and low *precision* values be obtained. At a *recall* level of 70.0% the *precision* can drop to 19.2%. Moreover, the metrics based clone recognition does not offer any explanation or classification whatsoever on the nature or the category of the cloning. In this context, we can use the metrics based approach for a fast selection of potential clones, at a pre-processing stage in order to limit the search space when using the more accurate but more computationally expensive method of dynamic programming.

3.3 Comparison algorithm

The comparison algorithm used is based on Kontogiannis et al.'s Dynamic Pattern Matching algorithm [10]. The main difference resides in the granularity of the comparison of the code fragments. Kontogiannis et al. use whole source code statements as comparison units. Statements are first abstracted into feature sets. The matching is then realized on the feature vectors corresponding to blocks of statements. The features used include metrics values and specific data- and control-flow properties. The approach described in this paper uses the tokens of the programming language as comparison units. Only in a later phase, sequences of tokens are constructed thus achieving a finer granularity of analysis.

The dynamic matching is performed on vectors corresponding to the sequences of tokens forming the code fragments. Let $t1_i$ represents the i-th token of the first code fragment and $t2_j$ represents the j-th token of the second code fragment, with i and j no greater than the total amounts of tokens of each code fragment. The vectors of tokens corresponding to the code fragments can be defined as $v1 =< t1_1, t1_2, ..., t1_n >$ and $v2 =< t2_1, t2_2, ..., t2_m >$ where n and m are the lengths of the code fragments. We also have $v1[i] = t1_i$ and $v2[j] = t2_j$.

From the vectors, a grid is constructed which holds the partial results of the dynamic matching. The dynamic algorithm takes as input the grid and both vectors v1 and v2. It returns the distance between v1 and v2 as well as the updated grid containing the details of the match.

```
function match(c: Grid; v1,v2: Sequence) => (cost: Integer)
  for ( i ← 1 to size(v1) )
    for ( j ← 1 to size(v2))
      tempCost ← computeCost(v1[i],v2[j])
```

$$c[i][j].\mathrm{cost} \leftarrow \min \begin{cases} c[i-1][j].cost + 1, \\ c[i][j-1].cost + 1, \\ c[i-1][j-1].cost + tempCost \end{cases}$$

$$c[i,j].\mathrm{previous} \leftarrow \begin{cases} c[i-1][j], \\ c[i][j-1], \\ c[i-1][j-1] \quad \text{depending on} \\ \qquad\qquad\qquad \text{the minimal cost} \end{cases}$$

```
  return c[size(v1)][size(v2)]
```

Figure 2. Core method of the matching algorithm.

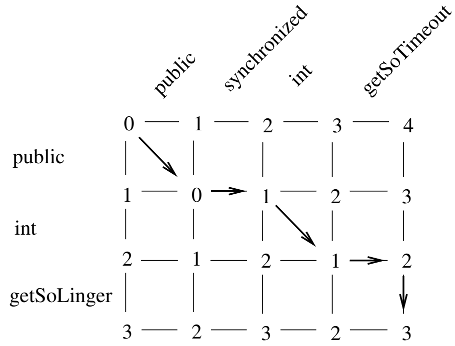

Figure 3. Grid after matching the code fragments v1 = <public,int,getSoLinger> and v2 = <public,synchronized,int,getSoTimeout> from the methods presented in Figure 4. The arrows show the path of the optimal match between the code fragments.

The distance between two vectors of tokens is defined as the minimal amount of tokens that have to be inserted or deleted to transform one vector into the other.

The core of the algorithm which is defined in function *match* is presented in Figure 2.

Function *match* iterates over all the elements of the grid and computes the distance for consecutive sequences s1 and s2 using previously computed distances between shorter sequences as well as the cost of matching the current tokens $< tail(s1) >$ and $< tail(s2) >$. This latter cost is determined by *computeCost*.

Function *computeCost* compares two tokens t1 and t2 by testing for equality of types and values. Two nodes match perfectly if they belong to the same type, except if they're literals or identifiers. Then they must also have the same value. For example, the token of type "if" can only be equal to another token of type "if" whereas an identifier can only be equal to another identifier whose string value represents the same sequence of characters.

Function *computeCost* returns 0 if the tokens are equal and can be matched. Otherwise, it returns 2 (the equivalent of the cost of removing one token and then adding the other instead).

Figure 3 presents an example of grid after matching two code fragments. Those fragments have been extracted from real cloned methods presented in Figure 4. The numbers placed at each grid[i][j] are the distances between the sequence <v1[1],...,v1[i]> and <v2[1],...,v2[j]> The optimal match can be represented as a path in the grid: p = <<0,0>, <1,1>, <1,2>, <2,3>, <2,4>, <3,4>> The optimal match determined by the algorithm is thus represented as a path in the grid where each token is associated with an action: insertion (horizontal arrow), deletion (vertical arrow) or match (diagonal arrow).

The comparison isn't completely over at this point, though. Indeed, in order for the classification to proceed, the match has to be defined in terms of differences in the code fragments. The differences can be represented as pairs of sequences of consecutive tokens taken from each of the code fragment and associated with an action (insertion, deletion or replacement). The set of all the actions allow to transform one code fragment into the other.

Such sequences can nevertheless easily be extracted from the grid by walking the optimal match path from the last element (grid[size(V1)][size(V2)]) to the first element (grid[0][0]). Consecutively inserted, deleted or matched tokens are grouped into sequences and the corresponding action is registered. Later, consecutive actions of insertion/deletions are changed for replacement actions yielding the final result.

For the example of Figure 3 such a transformation returns the optimal match as: <{<public>, <public>, perfect match}, {<ϵ>, <synchronized>, insertion}, {<int>, <int>, perfect match}, {<getSoLinger>, <getSoTimeOut>, replacement}>. Figure 5 gives the result for the complete methods.

The whole comparison algorithm, as presented has a complexity of $\theta(n * m)$, where n and m are the respective sizes (numbers of tokens) of the two code fragments. Usually, for similar methods $n \approx m$ and the complexity can be approximated by $\theta(n^2)$. This complexity can easily be reduced by using a "beam search" with an appropriate threshold, i.e. discarding all the matching alternatives that demand the deletion and/or insertion of an amount

```
———— Copy 1 ——————
public int getSoLinger() throws SocketException {
   Object o = impl.getOption( SocketOptions.SO_LINGER);
   if( o instanceof Integer) {
      return(( Integer) o). intValue();
   }
   else return - 1;
}
———— Copy 2 ——————
public synchronized int getSoTimeout()
                              throws SocketException {
   Object o = impl.getOption( SocketOptions.SO_TIMEOUT);
   if( o instanceof Integer) {
      return(( Integer) o). intValue();
   }
   else return 0;
}
```

Figure 4. Example of cloned Java methods. Example taken from JDK 1.1.5. File Socket.java, class Socket.

<public> in the original method and <public> in the cloned method	Perfect match
ϵ and <synchronized>	Insertion
<int> and <int>	Perfect match
<getSoLinger> and <getSoTimeout>	Replacement
< ( ) throws ... SocketOptions .> and < ( ) throws ... SocketOptions .>	Perfect match
<SO_LINGER> and <SO_TIMEOUT>	Replacement
<) ; ... else return> and <) ; ... else return>	Perfect match
<- 1> and <0>	Replacement
<; }> and <; }>	Perfect match

Figure 5. Optimal match of the code fragments of Figure 4.

of tokens greater than the threshold. This optimization is possible because already similar fragments of code are compared and the optimal match lies necessarily not far from the diagonal of the grid.

3.4 Classification algorithm

Once the optimal match between two code fragments has been found, the classification can proceed. During this second phase, each previously found difference is examined and its type is determined using the information provided by the AST. The set of all the types of differences determines the category of the cloning relation. The following algorithm is used:

```
function classify(s: Sequence) => (type: TypeCategory)
  setTypes = ∅
  forall pairs p of sequences in s
     if ( action(p) ≠ perfect match)
        setTypes ⋃ typeDifference(p)
  return determineCategory(setTypes)
```

This algorithm takes as input the pairs of sequences with their actions as a sequence of 3-tuples. It returns the type of the category of clones.

Function *typeDifference* takes a pair of sequences of tokens and determines the type of the programming language entity (variable, type, expression, etc.) corresponding to the sequences, using informations from the ASTs. The type of this entity directly determines the type of the difference.

Function *determineCategory* examines all the types of the differences and determines the category using the category definitions presented in Section 2.

If we consider the methods presented in Figure 4, with their optimal match presented in Figure 5 we see that the two methods have four differences. The first one is the insertion of the attribute synchronized to the second methods. This difference affects the attributes of the method and, as explained previously, is not used during the classification. The second difference is the modification of the name of the methods, its type is thus MethodName. The next difference, is a substitution of a non-local constant. The type of this difference is GlobalVariable. Finally, the last difference is a substitution of constants. It's type is thus Constant. The set of types of differences is then: MethodName, GlobalVariable and Constant. Therefore, the corresponding category is "Interface and implementation changes".

4 Experimental set-up

The SMC (Similar methods classifier) tool implements the classification algorithm. It has been developed in Java, using jdk 1.1.7. For the ASTs, SMC uses a java parser generated with javacc version 8 (first pre-release).

We have applied SMC to six freely available Java software systems of some 500 000 lines of code.

- JDK [18], a development kit from Sun Microsystems with 145 000 lines of code.
- SableCC [6] a parser generator from McGill University with 32 000 lines of code.
- ANTLR [12] a parser generator from MageLang Institute, with some 25 000 lines of code.

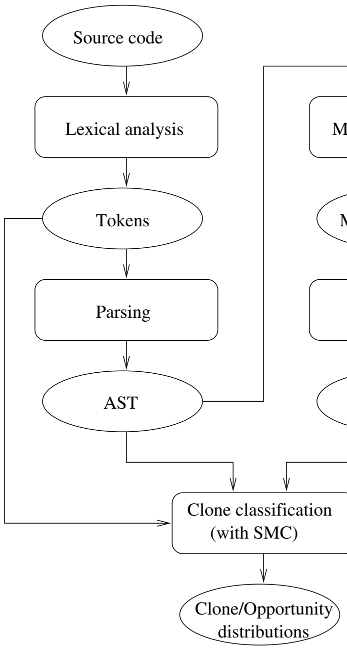

Figure 6. Process used for the experiment.

- SWING [19] a user interface toolkit from Sun Microsystems accounting for 215 000 lines of code.
- KFC [22] a user interface toolkit by K. Yasumatsu of 57 000 lines of code.
- HTTPCLIENT [20] a web browser made by R.Tschalaer with 21 000 lines of code.

The process used for the experiment is depicted in Figure 6. For each system, we have first applied Patenaude et al.'s approach [17] to find clusters of similar methods using metrics. We have used the output of this tool as input to SMC. For each cluster of similar methods, SMC determined groups of cloned methods and the category of their cloning relation.

The experiment was conducted on a Pentium Pro 180MHz with 64MB RAM running Linux version 2.0.27.

5 Results

The goal of the experiment has been to measure, in several systems, reengineering opportunities based on clone information. Opportunities have been defined as groups of cloned methods to which a concrete reengineering action can be applied.

Therefore, several values have been computed during the experiment. The quantities of groups of methods that belong to each category of clones have been determined. For each of those groups, we have counted the number of methods involved and their size in lines of code. We have also added the measures obtained for each group in order to get the total numbers for each category of clones.

Some methods were involved in more that one clone relation. We have computed the different measures counting methods in every category separately. The results for the different categories of clones should thus be considered separately. Those results can be found in Table 2.

5.1 Overall results

The detailed results obtained (c.f. Table 2) show the amounts of opportunities in the different categories of clones defined (column "Groups") as well as the amounts of methods and code corresponding to clones of each category.

Figures 7 through 13 show the distributions of opportunities in categories, for the systems analyzed. Figure 7 shows the distribution when the totals of the opportunities are considered. A novel result appears from those graphs. Indeed, when the overall results are considered category 1 ("Identical") contains significantly more opportunities than any other category. This is also true for two of the systems considered separately (ANTLR and SWING). In the other systems, except JDK, category 1 appears in the first two categories. Such a result is very interesting since identical clones will obviously be the easiest to manipulate.

The next three categories, in the histogram of Figure 7, are categories 16 ("One long difference, interface and implementation"), 12 ("Interface and implementation changes") and 3 ("Called methods"). This result is strongly influenced by results from SWING whose size is significantly greater (200 KLOCs) than the size of the other systems (300 KLOCS altogether). If we consider instead the first five categories in each system and determine the categories that come up most often in those first five, we can show that categories 16 and 3 remain most important but instead of category 12, category 4 seems to play a leading role. Two conclusions can be drawn from this result:

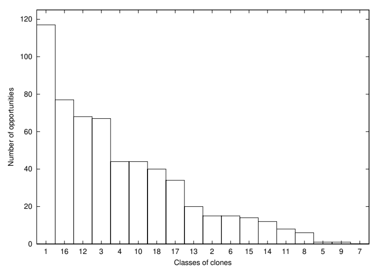

Figure 7. Overall distribution of opportunities.

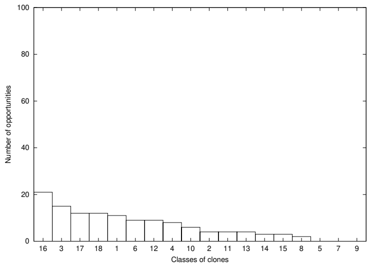

Figure 8. Distribution of opportunities in JDK.

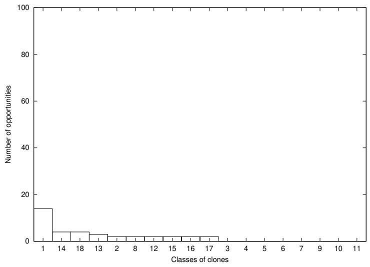

Figure 9. Distribution of opportunities in ANTLR.

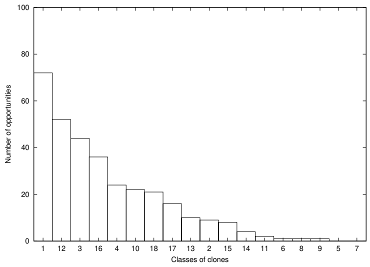

Figure 10. Distribution of opportunities in SWING.

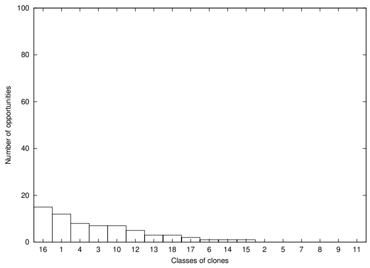

Figure 11. Distribution of opportunities in KFC.

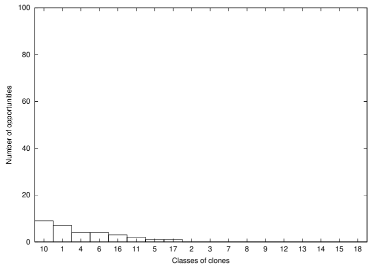

Figure 12. Distribution of opportunities in SABLECC.

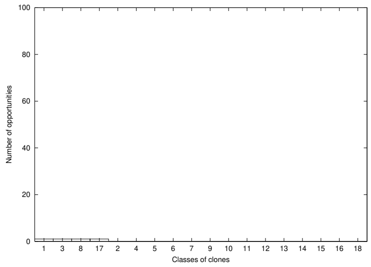

Figure 13. Distribution of opportunities in HTTPCLIENT.

- The overall results do not reflect the distributions of opportunities in the individual systems separately.
- Besides category 1, opportunities tend to belong to categories 16 (”One long difference, interface and implementation”), 3 (”Called methods”) and 4 (”Global variable”);

Therefore, reengineering categories 1, 16, 3 and 4 will have a large impact on all systems.

If we consider the size of the source code involved in clones in the different categories and thus in the different reengineering opportunities, similar categories appear in the lead (1, 16, 12 and 3). This result is interesting because it shows that categories containing most opportunities also cover the most of the source code.

Finally, it can be noted from the results of Table 2 that category 7 (”Local variable types”) is empty in all the systems. Hence, in the systems investigated, local variables types are never modified without side effects in the bodies of the methods. This category might therefore be removed from the scheme.

5.2 Results dependent on systems

The previous results hold independent of the system analyzed. Nevertheless, many characteristics of the measures of reengineering opportunities depend on systems. Indeed, histograms from Figures 8 through 13 show that distributions of opportunities vary with systems. Some systems such as SABLECC have all their clones in a few categories whereas other systems like JDK have their clones more evenly distributed between categories. This new result is extremely important in that it implies that design reengineering decisions based on clone information necessitate the detailed analysis of clone distribution in systems.

5.3 Impact of different opportunities

Another original result is that all opportunities are not equivalent. Indeed, opportunities vary in the amount of methods and amount of code they involve. Those two latter measures aren't relative either. The following three examples prove this point:

- If the results of Table 2 are considered, category 8 from JDK contains 2 opportunities that involve 54 lines of code whereas category 2 contains 4 opportunities that involve only 42 lines of code. Thus opportunities vary in the amount of source code they involve.
- Category 8 from ANTRL contains 2 opportunities corresponding to 4 methods whereas category 12 with also 2 opportunities affects as much as 29 methods. Hence, opportunities contain different amounts of methods.
- Finally, even though category 1 from SWING contains as much as 3332 lines of code, it contains only 245 methods. This is less than category 12 which contains only 2729 lines of code but as much as 269 methods. Therefore, amounts of methods contained in categories are independent of the corresponding size in lines of code.

6 Discussion

6.1 Clone based reengineering

In the previous section, we have measured reengineering opportunities and determined their characteristics. In this section, we'll go further by analyzing which categories of opportunities present potential greater impacts for reengineering and how such categories can be detected. Those categories correspond to what we call "High impact categories".

High impact categories are defined as categories whose reengineering is relatively easy to automate and/or has important assets in terms of source code or methods involved in every single reengineering action.

The identification of such high impact categories will allow to put priorities on categories of clones and guide reengineering decisions.

6.2 High impact categories

This section presents a suite of characteristics making some categories potentially represent a high impact from a reengineering perspective. The order in which they appear is arbitrary.

6.2.1 Categories easy to manipulate and containing a fair amount of code

Categories easy to manipulate are considered as "high impact" from a reengineering perspective because the automation of their transformations should be quite easy to implement. If such categories don't contain much of the source code though, their reengineering is of little interest. Manipulations of any categories are worth pursuing only if those categories contain a fair amount of code.

We believe that categories 1 ("Identical") through 9 ("Type usage") will be easy to manipulate because their definitions are based on very specific, single token differences. Categories 10 ("Interface changes"), 11 ("Implementation changes") and 12 ("Interface and implementation changes") are a composition of the previous categories and shouldn't be much more difficult to reengineer. Therefore they might also be considered during the selection of high impact categories.

Such high impact categories can easily be selected. When inspection of results yields that the amounts of source code per category, for one or some of the chosen categories, exceed a threshold of interest, high impact categories have been found.

The analysis of SABLECC shows that there are 1640 lines of code involved in clones of category 6 ("Parameter types") and 954 lines of code are in methods from category 1 ("Identical"). This represents respectively 61 and 35.5 percent of all the code involved in cloning. Definitively, categories 1 and 6 are of high impact in this system.

In KFC, a smaller system, categories 1 ("Identical") and 3 ("Called methods") account for only a bit more than 200 lines of code each. But this represents as much as 11 and 8 percent of the source code involved in cloning, in this system. Hence, in KFC, those categories can be considered having a high potential impact.

6.2.2 Categories containing an important amount of code

Some categories can be difficult to reengineer, but because of the quantity of source code they contain, it will be worthwhile to pursue their reengineering.

The measure of the quantity of code in each category of clones is a straightforward indicator of such categories.

Let's consider the classification of clones in SWING. In this system, category 1 ("Identical") contains over 3 K-LOCs whereas categories 12 ("Interface and implementation changes") and 16 ("One long difference, interface and implementation") have almost 3 KLOCs. Those two latter categories will probably be more difficult to reengineer than category 9 ("Type usage") and any lower indexed category, but because of the quantity of source code they cover, the results of their manipulation will be more noticeable than the result of the manipulation of any other category (but category 1 which also contains 3 KLOCs). Therefore, we consider them as having a potential high reengineering impact in this system.

6.2.3 Categories containing an important amount of methods

It has been shown, in the previous section, that the amount of methods per category is partially independent of the amount of code. For certain reengineering activities, the amount of methods that can be reached through some similar actions is important. An example of such activities would be clone removal. From such perspective, different categories might be considered "high impact" than those simply containing a lot of code.

If we consider SWING again, we can see that categories 1, 3, 4, 10, 12 and 16 contain over 100 methods and can be considered as having a high potential impact. If we push the threshold up to 250 methods then only categories 12 and 16 remain.

6.2.4 Categories containing highly clustered groups of methods

Large clusters imply many copies of one method. Therefore they represent "sensitive" parts of a system and should be more eagerly considered for reengineering. Moreover, each reengineering action when applied to larger clusters will have a greater impact on the system.

To make such categories appear from the results of a classification, the ratio of the number of methods per group must be computed for each category. Categories having those ratios exceed a certain threshold will be considered having a potential high impact.

In the SABLECC system, there are two very large clusters of methods. One is in category 6 and the other in category 10. The average amounts of methods per group in those categories are 41 and 26. Those categories have thus a potential high reengineering impact in this system.

The analysis of ANTLR shows that categories 12 and 16 each contain 29 methods in only two groups. These categories have definitively a high potential impact related to highly clustered groups of methods.

6.2.5 Categories containing large methods

When large methods are manipulated, each reengineering action, will have a greater impact in terms of the size of source code involved. But this is not the only reason for considering large methods for reengineering. Indeed, those methods are also more interesting because their size increases their probability of containing bugs. Thus, merging the common parts of the code of longer methods should benefit more the reduction of the probability of bugs.

The computation of the ratio of the size of the source code with the amount of methods for every category of clones will allow the identification of such categories.

In JDK for example, even though category 15 contains only 135 lines of code, it's a high impact category because the average size of the methods of this category is 15 lines.

In HTTPCLIENT, categories 1, 3 and 17 contain methods whose average size is above 20 lines of codes. Those categories can hence be considered having a high potential reengineering impact in this system.

As the results of the previous discussion show, the set of high impact categories vary with systems. Their identification should thus help guide reengineering decisions uniquely for each system.

The categories with high potential impacts also depend on the kind of impact considered during the analysis and the thresholds selected. Therefore, reengineering decisions will vary with the goal sought.

7 Conclusions

This paper has presented a definition of a reengineering opportunity based on clone information. To make the study of such opportunities possible in large software systems, a classification scheme has been defined along with algorithms that automatically compare and categorize cloned methods along the scheme.

Reengineering opportunities have been studied in six systems. The results of the study have shown that some categories contain most opportunities independently of the system analyzed. Those categories have been identified. Nevertheless, the distributions of opportunities in categories vary with systems which implies that a careful analysis of opportunities has to be performed whenever a system is to be reengineered.

The study performed has also shown that reengineering opportunities based on clone information vary in size and amount of code affected. Thus, the sole amount of opportunities isn't enough to guide reengineering decisions. Other aspects have to be considered. Those aspects are defined in what is called "high impact categories", i.e. categories whose reengineering presents a potentially high impact on the system in terms of source code affected, amount or size of methods affected, size of clusters involved or simply degree of difficulty of implementation and automation.

The identification of such categories could be used to guide reengineering decisions for systems especially that the set of high impact categories varies with systems and kinds of impact sought.

8 Future work

The next step in the investigation of reengineering based on clone information would be to investigate different concrete possible transformation: system redesign, clone removal, clone restructuring, and so on. The benefits and pitfalls of such transformations would have to be carefully studied.

Depending on the transformations chosen, concrete reengineering actions could be investigated for each category of clones (or opportunities) defined in the scheme along with algorithms for their automation.

9 Acknowledgements

This research project has been funded by both the Natural Sciences and Engineering Research Council of Canada (NSERC) and Bell Canada.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-wesley, 1988.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*. IEEE Computer Society Press, July 1995.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntaxt trees. In *Proceedings of the International Conference on Software Maintenance 1998*, pages 368–377. IEEE Computer Society Press, 1998.

[4] Buss and al. Investigating reverse enginering technologies for the cas program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.

[5] G. Canfora, A. Cimitile, and A. DeLucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance 1994*, pages 424–433. IEEE Computer Society Press, 1994.

[6] E. Gagnon, Sable Research Group, School of Computer Science, McGill University. Sablecc 2.5: Object-oriented comiler framework. http://www.sable.mcgill.ca/sablecc/.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-wesley, 1997.

[8] J. H. Johnson. Identifying redundancy in source code using fingerprints. *CASCON'93*, pages 171–183, October 1993.

[9] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 44–54, 1997.

[10] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:77–108, March 1996.

[11] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance 1997*, pages 314–321. IEEE Computer Society Press, 1997.

[12] MageLang Institute. Antlr 2.2.3.: Predicated-ll(k) parser generator. http://www.antlr.org.

[13] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance 1996*, pages 244–253. IEEE Computer Society Press, 1996.

[14] H. Müller. Understanding software systems using reverse engineering technology perspectives from the Rigi Project. *CASCON'93*, pages 217–226, October 1993.

[15] P. Newcomb and P. Scott. Requirements for advanced year 2000 maintenance tools. *IEEE Computer*, pages 52–57, March 1997.

[16] J. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, 1994.

[17] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th. International Workshop on Program Comprehension. IWPC'99*. IEEE Computer Society Press, 1999.

[18] Sun Microsystems Inc. Jdk 1.1.5.: Java development kit.

[19] Sun Microsystems Inc. Swing component set 1.0.2. http://www.javasoft.com/products/jfc/tsc/.

[20] R. Tschalaer. Httpclient 0.3.:http client library. http://www.innovation.ch/java/HTTPClient.

[21] L. Wills. Automated program recognition by graph parsing. MIT Technical Report 1358, MIT, AI Laboratory, 1993.

[22] K. Yasumatsu. Kfc 1.0 beta: Kazuki yasumatsu's foundation classes. http://ring.aist.go.jp/openlab/kyasu/.

Table 2. Results of the classification of cloned methods in the six systems. The first column is the number of the category. The second column indicates the amount of source code in terms of lines of code covered by the category. Column three is the translation of the amount as a percent of the total amount of code involved in cloning. Columns 3 and 4 represent respectively the amount of methods and groups of methods belonging to each category of clones. This latter measure is also the amount of reengineering opportunities in the category. Some methods have cloning relations in more than one category. In this table results for each category have been computed independently.

	JDK				SABLECC				ANTLR			
Category	LOCs	Percent clones	Methods	Groups	LOCs	Percent clones	Methods	Groups	LOCs	Percent clones	Methods	Groups
1	432	5.9	42	11	954	35.5	94	7	593	24.2	51	14
2	42	0.6	8	4	0	0.0	0	0	40	1.6	4	2
3	464	6.4	36	15	0	0.0	0	0	0	0.0	0	0
4	198	2.7	26	8	476	17.7	49	4	0	0.0	0	0
5	0	0.0	0	0	12	0.4	2	1	0	0.0	0	0
6	229	3.2	24	9	1640	61.0	164	4	0	0.0	0	0
7	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
8	54	0.7	4	2	0	0.0	0	0	44	1.8	4	2
9	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
10	162	2.2	21	6	2016	75.0	215	9	0	0.0	0	0
11	117	1.6	13	4	70	2.6	4	2	0	0.0	0	0
12	354	4.9	35	9	0	0.0	0	0	375	15.3	29	2
13	134	1.8	10	4	0	0.0	0	0	69	2.8	11	3
14	56	0.8	6	3	0	0.0	0	0	198	8.1	8	4
15	135	1.9	9	3	0	0.0	0	0	157	6.4	4	2
16	742	10.2	58	21	52	1.9	7	3	363	14.8	29	2
17	233	3.2	28	12	14	0.0	2	1	173	7.1	7	2
18	334	4.6	30	12	0	0.0	0	0	201	8.2	9	4
Total	N/A	N/A	N/A	120	N/A	N/A	N/A	31	N/A	N/A	N/A	37

	SWING				KFC				HTTP.			
Category	LOCs	Percent clones	Methods	Groups	LOCs	Percent clones	Methods	Groups	LOCs	Percent clones	Methods	Groups
1	3332	14.5	245	72	294	10.7	29	12	42	17.0	2	1
2	274	1.2	24	9	0	0.0	0	0	0	0.0	0	0
3	1793	7.8	217	44	228	8.3	16	7	63	25.5	3	1
4	1420	6.2	192	24	135	4.9	20	8	0	0.0	0	0
5	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
6	12	0.05	2	1	117	4.3	9	1	0	0.0	0	0
7	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
8	28	0.1	2	1	0	0.0	0	0	21	8.5	3	1
9	28	0.1	2	1	0	0.0	0	0	0	0.0	0	0
10	1115	4.8	133	22	140	5.1	16	7	0	0.0	0	0
11	88	0.4	4	2	0	0.0	0	0	0	0.0	0	0
12	2729	11.9	269	52	111	4.0	13	5	0	0.0	0	0
13	314	1.4	21	10	40	1.5	6	3	0	0.0	0	0
14	68	0.3	8	4	16	0.6	2	1	0	0.0	0	0
15	278	1.2	22	8	144	5.2	16	1	0	0.0	0	0
16	2684	11.7	363	36	450	16.4	51	15	0	0.0	0	0
17	669	2.9	75	16	40	1.5	4	2	50	20.2	2	1
18	712	3.1	53	21	201	7.3	21	3	0	0.0	0	0
Total	N/A	N/A	N/A	323	N/A	N/A	N/A	65	N/A	N/A	N/A	4