AUTOMATIC DETECTING AND REMOVING CLONES IN JAVA SOURCE CODE

Tomáš Bublík, Miroslav Virius
Fakulty of Nuclear Science and Physical Engineering, Czech Technical University in Prague
tomas.bublik@gmail.com

ABSTRACT:
This paper deals with the detection of the clones in the Java language source code. Especially the „non-ideal“ clones are discussed. These are pieces of code that are not exactly the same. This paper focuses on the most important part in the case of the non-ideal clones: extracting and defining a new method. Some improvements are suggested.

KEY WORDS:
clone detection, abstract syntax tree, tree algorithm, Java source code

1. INTRODUCTION

Nowadays, it is more or less usual to use already written pieces of code. The main reason is to save the programmer’s time, of course. In the programming, the initial step is to check whether the intended algorithm or technology already exists, and if it is the case, try to use it in the developed code. For larger projects, which are developed in team, and which have thousands of code lines, clones occur. Unintended clones cause many troubles in the further code development and maintenance. Each error or functionality change must be corrected in every clone occurrence. In the case of large projects, the clone detection could be a serious problem. It may be the cause of one the worst error type: the compiler finds no error, but the program doesn‘t works properly. According to [9], the clone occurrence in an average project is between 7-23%. Moreover, it is usual that 70 % of the programming is only the maintenance of the existing code.

Many studies deal with the detection and the removal of the clones. This is a relatively difficult discipline which improves continuously. The possibilities of the clone detection grow hand in hand with the computational power. According to [6], the whole process can be composed of the following steps:

1) Preparation and transformation of the original code to another structure
2) Clones detection in this new structure
3) Connection between the clones and source code
4) Transformation of the clones into newly created structures

In this article, we describe these steps from various points of view and mention some improving procedures. However, many proposals of the transformations exist. There are 4 essential groups of the approaches to this issue: textual, lexical, syntactic and semantic one.

2. TEXT ORIENTED APPROACH

This approach uses simple algorithms for code comparison. The implementation of these algorithms is simple and it is independent on the programming language used. In the first stage, the code is normalized. The text normalization consists of the elimination of gaps, blank lines, comments and others unnecessary stuff. In next stage, the lines or blocks of various lengths are compared. This is $O(n^2)$ complexity algorithm. One of the possible improvements is the hashing of the lines or code blocks into buckets.

The text oriented approach has small ability of clone detection, because the code duplicities must perfectly match. The text algorithms fail on very small differences in the clones, e.g. if the variable names differ. Therefore, an improved version exists based on parameterized search. Parameterized search uses regular expressions to treat tiny code differences.
On the other hand, text approaches are very fast and suitable as a part of more advanced processing.

3. LEXICAL APPROACH

This approach is based on the transformation of the source code into a sequence of lexical tokens. It operates like the compilers; the sequences and the subsequences of tokens are scanned and in the case of the match, the corresponding original code is returned as a clone. In comparison with the textual approach, the lexical ones are more robust, but dependent on the programming language used. However, it is necessary to know the grammar of the language. They are resistant to spaces etc., but they also can‘t find semantic similarities. Lexical approaches are usually used together with the text search.

3.1 Tree structure analysis

Tree analysis belongs to the category of the syntactical approaches. These approaches are now the most quickly developing ones. They have wide range of settings, adjustments and use. Principle of this approach is the source code transformation to the syntax tree and processing it by already known algorithms. The source code can be expressed as a graph in many ways: As a flow graph, as a dependence graph, as a class graph, etc. Probably the most common form is the Abstract Syntax Tree (AST). AST is the code representation that is used by the compilers. In AST, each meaningful element is modeled as a node. As in previous cases, the comments and the spacing are omitted.

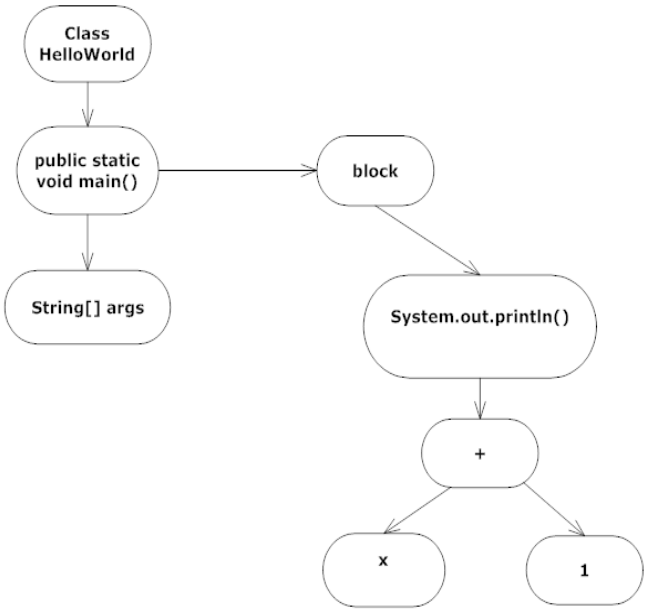


Fig. 1 Example of the abstract syntax tree

AST in the Figure 1. represents following code snippet:

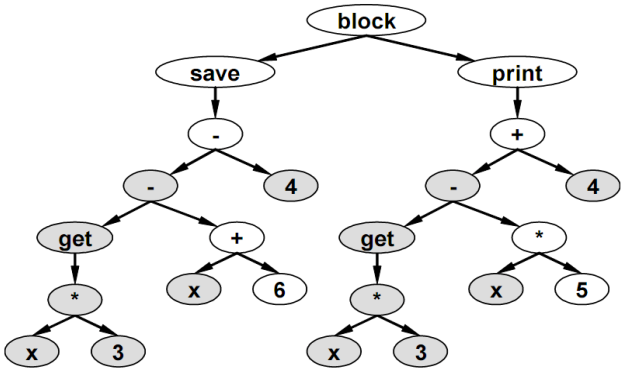
```
public class HelloWorld {
      public static void main(String[] args) {
      System.out.println(x+1);
      }
}
```

The rules for the AST are:

- Each node represents an element in the source code,
- there are exactly specified node types which depend on the used language,

- each node type has exactly specified properties, sub nodes etc.

A pioneer in this area is I. D. Baxter who described in [5] an enhancement of clone search procedure. Baxter showed that the algorithm for clones searching has $O(n^3)$ complexity, where n is number of sub trees in AST of source code. In [5], he described enhancement using heuristics which assigns a hash code to each sub tree, and accordingly classifies it into a bucket. Next, only the sub trees in the same bucket are compared. Baxter shows that this improved algorithm has $O(n^2)$ complexity in the worst case. This algorithm served as the basis for the following research in field of clone detection. N. Juillerat [1] presented detection method based on sub trees, too, which aren't quite the same, but they have the same discontinuous parts. Consider this graph:



He performs clone detection using a post-order processing of the tree. The result is a list of tokens. By using simple search commands on this list, he receives subsequences of clones. Example of tokens from the previous figure:

[x, 3, *, get], [x], [-], [4]

All these algorithms using graph techniques which are known for many years. After successful clone detection, it is necessary to find the connection between the newly represented code and original one. Next, the original code is replaced by extracted methods. In this stage, it is impossible to use the well-known algorithms like in clone searching stage. All the restrictions of the current programming language must be considered. For example, in Java, it is impossible to extract the method with more than one return value from a code with multiple outgoing data flows.
For example:

```
varA = x + 1;
varB = y + x;
….
```

The variables *varA* and *varB* are the output, but it is not easy to extract this code and to insert it into a new method, because Java is unable to return two values. Further, it is impossible to extract code snippet, which contains multiple commands transferring out the control flow (i.e. in which the statements like break, continue and return are used). For example:

```
Iterator something = …;
while (i.hasNext()) {
      MyClass myClass = (MyClass) i.next();
```

```
      if (myClass.getVarA.equals(„end“)) {
            continue;
      }
      if (myClass.getVarA.equals(„end2“)) {
            continue;
      } else {
      return;
      }
}
```

Juillerat [1] described a restriction list, which must be considered during method extraction process. He proposed how to bypass some of these restrictions. For multiple outgoing data flows, it is suitable to choose the longest subsequence of clone tokens, which has only one outgoing data flow. However, it is necessary to make data flows analysis before the method extraction itself. This process is described by [11]. It consists of the function separation which indicates interrupted flow and cancels the rest of the commands. Unfortunately, these solutions suits only for the C language. It is not applicable in Java due to the one return value restriction.

Semantic approach

The last type of clone detection approach is the semantic analysis. A source code is transferred into program dependency graph (PDG). The nodes of this graph represent the expressions and the statements, while the edges represent the control and data dependencies. The clone searching problem is then turned into the isomorphic sub graph searching problem. The practices described can be found in [12]. This kind of the clone searching is usually used in the plagiarism detectors.

Possible improvements

A disadvantage of the text oriented approaches is that they fail in ostensibly simple cases. For example:

```
a = x + 1 ; a = 1 + x
```

Two different trees will arise from this code snippet and no duplicity will be found. A solution could be to implement some text search or test interchange of the elements in the statement. Unfortunately, this could bring the exponential complexity and the algorithm would be useless.

This problem can be avoided by sorting applied already during the tree assembling process. But not all the content can be sorted. We can define simple rules for sorting of the variables in assignment statements. For example, it is suitable for the operations, for which the commutative law is applicable (operations „+“, „-“, „*“). Individual elements can be sorted alphabetically, then by the number and then special order of symbols could be defined.

This algorithm can proceed recursively from the inside of the smallest operation, which is compliant for interchange of the operands according to the commutative law. We will demonstrate sorting on example:

```
one = 1 + (1 + x) + (y + 1) + (a * b)
two = (b * a) + (x + 1) + (1 + y) + 1
```

First, the inside parts of the parentheses are sorted. These are the shortest operations capable to interchange of operands.

```
one = 1 + (1 + x) + (1 + y) + (a * b)
two = (a * b) + (1 + x) + (1 + y) + 1
```

Second, the higher lever elements are sorted. Parenthesis counts as one unit and their contents are amendable to the same rules.

```
one = 1 + (1 + x) + (1 + y) + (a * b)
two = 1 + (1 + x) + (1 + y) + (a * b)
```

It is obvious, that these statements are the same and it is easy to extract a new method. Nevertheless, not everything can be sorted by this way. The sorting without any loss of the meaning is possible in the case of methods and attributes in a class, too. It is not so easy in the case of the blocks of statements. The article [11] indicates some rules which it allows. However, there are some cases, for which is it almost impossible. For example:

```
logger.info(„Going to instantiate new variable“);
String hello = „Hello wordl!“;
```

Because we cannot assume the existence of predetermined rules for the logging and its relationship to the code, we are unable to determine whether these two statements belong together. It is impossible to change the order of the parameters of the method, too:

```
someMethod(int param1, String param2)
someMethod(String param2, int param1)
```

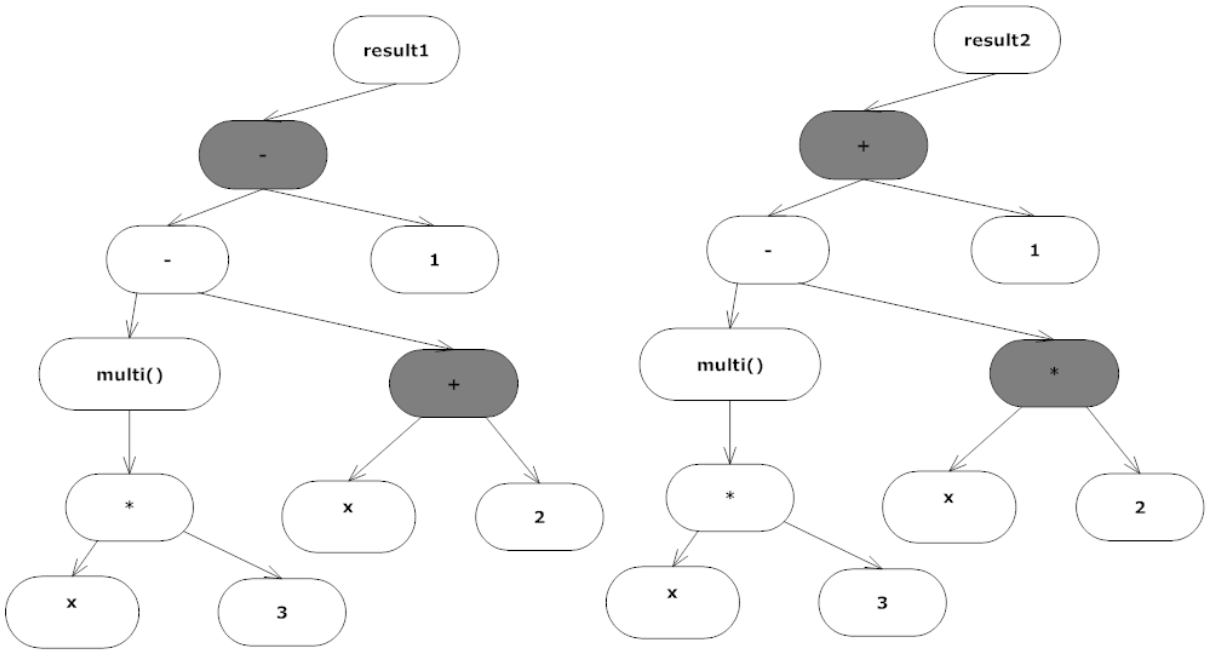
Despite of the same name and the same name of parameters, these methods may be completely different. Certainly, many other examples can be found.
Defining various levels can provide more options to the improvement. One kind of level could set a number of occurrences required to be clone detection meaningful. If we find this statement:

```
a = 1 + x
```

twice, it does not make sense to write a new method for it. This could be helpful already by classification of sub trees into the bucket. If the bucket contains only small number of sub trees, we can afford to skip it.
Another kind of level would be set a minimal required length for a similar sequence of tokens or its rareness. For example, if there are few scattered subsequences of maximal length of 2, the transformation and extraction of the new method will not pay off.
But there are also cases (quite common), when trees differ a little and extraction could be easily done by parameterization.

```
int result1 = (multi(x * 3) - (x + 2) - 1);
int result2 = (multi(x * 3) - (x * 2) + 1);
```

The trees differ in two nodes (operators „+“, „-„ and „*“). By comparison of sequences created from post-order processing of the trees, we receive 3 identical subsequences. Instead of approach suggested by Juillerat [1], another approach is possible. We can create a new enum type, which has values equivalent to the operators mentioned before.



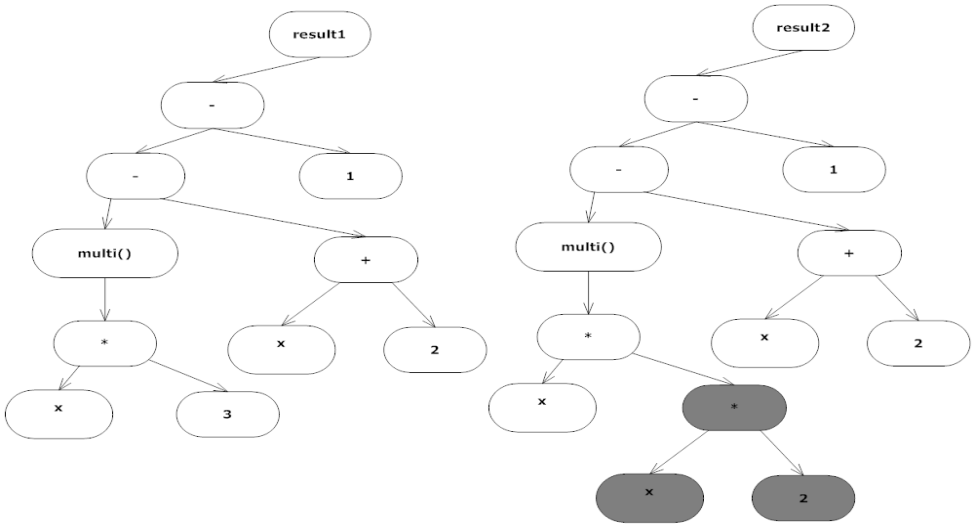

```
public static enum OperationType {
        ADD,
        MULTIPLY,
        SUBSTRACT;
}
```

Furthermore, the extracted method has a parameter x, but also parameters to determine the required type of operation.

```
private static int refactor(int x, OperationType firstOp, OperationType
secondOp) {
        int temp = 0;
        int leftOperator = x;
        int righOperator = 2;
        if (firstOp.equals(OperationType.ADD)) {
            temp = leftOperator + righOperator;
        }
        if (firstOp.equals(OperationType.MULTIPLY)) {
            temp = leftOperator * righOperator;
        }
        leftOperator = multi(x * 3) - temp;
        righOperator = 1

        if (secondOp.equals(OperationType.SUBSTRACT)) {
            temp = leftOperator - righOperator;
        }
        if (secondOp.equals(OperationType.ADD)) {
            temp = leftOperator + righOperator;
        }
        return temp;
    }
```

Moreover, the already known algorithms for the equation solving can be applied during the method extraction. A method with the parameter could also replace these code snippets which are the same except for a few statements that are easy to remove. We can identify them by extra branches in the tree.



There can be, for instance, a block of several statements and another similar block with few extra statements. The newly created method will have the parameter (among the others), which will determine whether to exclude different statements of not. But again, it is necessary to follow the rules for the method extraction [11].
The other option for the code with multiple outgoing flows is to create a new object, which will hold the values for these data flows. Consider, for instance, the code snippet with 2 outgoing values (one and two):

```
int one = 1 + x;
int two = 1 + y;
System.out.println(one);
System.out.println(two);
```

The newly created class will be:

```
public static class NewClass {
    private int one;
    private int two;
    public int getOne() {
        return one;
    }
    public void setOne(int one) {
        this.one = one;
    }
        public int getTwo() {
        return two;
    }
  public void setTwo(int two) {
        this.two = two;
    }
```

```
    }
```

And the extracted method with the new object as an outgoing flow:

```
private static NewClass newMethod(int x, int y) {
        NewClass newObject = new NewClass();
        newObject.setOne(newOperation(x));
        newObject.setTwo(newOperation(y));
        return newObject;
    }

private static int newOperation(int input) {
        return 1 + input;

    }
```

The original code snippet will be replaced by this:

```
NewClass newObject = newMethod(x,y);
```

All the next occurrences of the use of these variables will be replaced by the *newObject.get** call. For example:

```
   System.out.println(newObject.getOne());
   System.out.println(newObject.getTwo());
```

Using this type of code modification, it is important that all the input variables are be of the basic types. Otherwise, the newly created method cannot change the values of the input variables. It could lead to a complication, if their values are used in the next code.

Conclusion

According to the methods described above, it is obvious that the tree approach is the most promising one. That is because the major part of the algorithms is well known. Moreover, the access to the AST used by the compiler is available since Java version 1.6. Another advantage is the tree independence on the programming language. It can be created very powerful and effective tool for removing clones by defining additional constraints.

The disadvantage of the tree approach is the difficult implementation and the memory and CPU requirements. These algorithms will probably fail in the case of a very large project. Currently, I'm involved in research of implementation enhanced capabilities of algorithms for removing so-called „non-ideal" clones. It shows up, that using is very easy for simple cases, but for more complex ones is the situation very difficult. The ability to recognize the cases in which the refactoring is dangerous seems to be the key feature.

REFERENCES

1. Juillerat, N., Hirsbrunner, B. *An Algorithm for Detecting and Removing Clones in Java Code*, 2006. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.3829

[cit. April 2, 2011]

2. Juillerat, N. *Models and Algorithms for Refactoring Statements,* PhD thesis. Fribourg: University of Fribourg, Switzerland, 2009

3. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, N. *A Metamodel for Language-Independent Refactoring*. In: Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00), IEEE Computer Society Press, 2000. p. 157–167

4. Nouza, O. *Automatizované transformace objektově orientovaných modelů*. PhD Thesis. Praha: ČVUT 2010

5. Baxter, I.D., Yahin, A., Moura, L. Sant' Anna. M. Bier, L. *Clone Detection Using Abstract Syntax Trees.* In: ICSM '98 Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Washington, DC, USA 1998

6. Roy C.K., Cordy, J.R. *Scenario-Based Comparison of Clone Detection Techniques*, In:, 2008. ICPC 2008. In: The 16th IEEE International Conference on Program Comprehension, June 2008

7. Ducasse, S., Rieger, M. Demeyer, S. *A Language Independent Approach for Detecting Duplicated Code*, ICSM '99 Proceedings of the IEEE International Conference on Software, IEEE Computer Society Washington, DC, USA 1999

8. Van Rysselberghe, F., Demeyer S., Antwerpen, B., Antwerpen B. *Evaluating Clone Detection Techniques*, 2003. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.4.1398 [cit. April 2, 2011]

9. Baker, B.S. *On Finding Duplication and Near-Duplication in Large Software Systems*. IEEE Computer Society Press, July 1995

10. Juillerat, N, Hirsbrunner, B. *FOOD: An Intermediate Model for Automated Refactoring*, In: The 5th International Conference on Software Methodologies, Tools and Techniques, SoMeT 06, Québec, Canada, 25 - 27 October 2006. pp. 452 - 461

11. Komondoor R., Horwitz, S. *Tool Demonstration: Finding Duplicated Code Using Program Dependences.* In: Proceedings of the European Symposium on Programming (ESOP'01), Vol. LNCS 2028, 2001, 383386

12. Komondoor R., Horwitz, S. *Using slicing to identify duplication in source Code.* In: Proceedings of the 8th International Symposium on Static Analysis, Paris, France, July 16-18, 2001