Zjednodušení zdrojového kódu pomocí grafové struktury

Ing. Tomáš Bublík

1. Introduction

Nowadays, there is lot of programming languages. These languages differ in syntax, usage, and processing. Keep in mind the rules of all of them is not an easy task. For better functionality understanding, it is possible to use some other language expressions then the text ones. The talk is about some graph-based expressions. Probably the most used one is Abstract Syntax Tree (AST) [1]. Because this expression is independent of used language, it is possible to use it in many cases. By default, an AST is used by compilers. a code in the form of a tree is more readable for those users who do not want to deal with a specific syntax of the source code, or for those, for who if more difficult to learn the language due to the higher age. Regardless the age, both this groups of users want to understand the language and work with the code. Thanks to an AST, it is, for example, possible to perform the code refactoring and optimizing without the knowledge of syntax. Further, it is possible to transfer some older applications to the newer version independently of the programming language. It is not necessary to learn any older languages; just a tool for creating trees is needed.

2. Abstract Syntax Tree

A 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 an Abstract Syntax Tree (AST). An AST belongs to the graph category of source code expression. Principle of this approach is the source code transformation to a syntax tree. An AST is the code representation that is used by the compilers. In an AST, each meaningful element is modeled as a node. Therefore, the comments and the spacing are omitted.

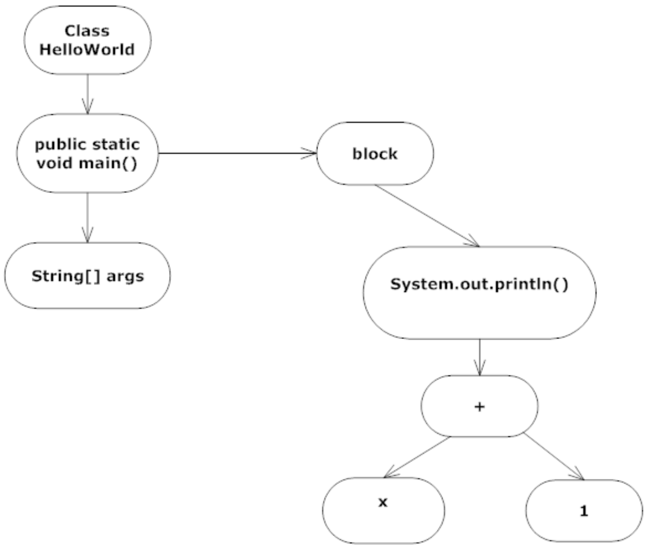

Fig. 1. Example of abstract syntax tree

The AST in the Figure 1 represents following code snippet:

Example 1.

```
public class HelloWorld {

        public static void main(String[] args) {

        System.out.println(x+1);

        }

}
```

The rules for an AST are:

- each node represents an element in the source code,
- there are exactly specified node types dependent on the used language,
- each node type has exactly specified properties, sub nodes etc.

An AST is not exactly specified standard; it differs a little in each usages. It is generated by gradual class passing. For the Java language is even available an API for work with a tree made by a compiler. Sideways, abstract syntax trees are a common tool for compilers to avoid tangling syntax and semantics. During the compile process, these trees are a result of the syntax analysis step, and serves as an input for the semantic analysis step.

An important thing is that an AST in not similar to any of the UML diagrams. An AST is a completely different graph and should not be interchanged, for example, with a class diagram. Despite of its divergence, from both the graphs it is possible to assemble the source code. But the class diagram is more abstract form intended for modeling purposes.

3. Clones detection in Java source code

While programming large application with many co-developers, it is usual that a source code becomes messy. The code is disordered and difficult to read. Many bad "smells" occurs. One of the unintended things is a cloned source code. In various parts of code are the same snippets. It could be the same methods, the same classes, or even the same small parts of code. These snippets are called clones or duplicities, and the code is considered as cloned. In this state, it is time to refactoring. In bad code, there exist many types of week parts or refactoring candidates. One of the types, described next, is a cloned source code.

The cloned source code is not only the code 100% equal to the original, but also the similar one. In this case, we talk about the "non-ideal" clones. The duplicated code is slightly modified, but does the same operations. If there are changes only in the method names or variables, the clone could be revealed comparing a decompiled code. It is known that while compiling a code, the information about names is lost. By decompiling, the comments a text formatting are also lost. Nevertheless, here will be described the clone detection technique using not by decompiling discoverable clones for the detection. For example, in this paper is the code with changed type of a variable. a source code with other changes produces the different byte code. These changes are difficult to reveal by classic methods, therefore the new method detecting this kind of the plagiarism will be introduced. This method uses an AST.

3.1. Preparing and normalizing the code

Before an AST creation, the code has to be processed a little. This preparation makes the further processing more effective. One of the adjustment sorts the code elements. This can be done already during the tree assembling process. But not all the content can be sorted. We can define the 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 compliant for interchange of the operands according to the commutative law. We will demonstrate sorting on example:

Example 2.

varA = 1 + (1 + x) + (y + 1) + (u * v)

varB = (v * u) + (x + 1) + (1 + y) + 1

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

Example 3.

varA = 1 + (1 + x) + (1 + y) + (u * v)

varB = (u * v) + (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.

Example 4.

varA = 1 + (1 + x) + (1 + y) + (u * v)

varB = 1 + (1 + x) + (1 + y) + (u * v)

It is obvious, that these statements are the same and an AST will be the same too.

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 allowing it.

3.2 Code processing and tree shape

Next, from the prepared code, an AST is assembled and compared. Methods for comparing graph are described in [4] and more developed in [1]. Also methods for the clone detection are described here [6]. But in this paper, there will be some differences. The "non-ideal" clones are searched, and the tree shapes are compared. It is expected that the code is modified, thus the tree shape is very important for the decision if the code is duplicated.

The plagiarism proving is made as follows: During the sub-trees comparison, the percentage of conformity to an AST in the current class is also compared. For example, if there are all the trees similar to each other of more than 70%, the class could be classified as similar. By the post-order walk through an AST, tokens sequence will be created. Further, it is possible to compare these tokens by already known methods; the percentage of the identical ones will determine the similarity degree. For example, two slightly different classes with the proper AST are shown. The only difference is the text variable initialization, but both the classes have the same output.

Example 5.

```
public class SomeClass {

public static void main(String [] args) {

System.out.println(„Hello world!“);

 }

}
```

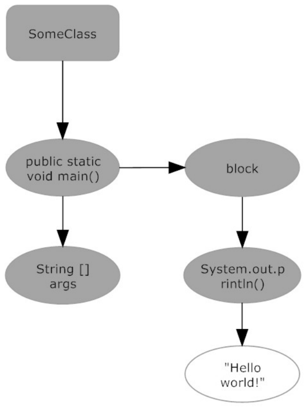

Fig. 2. Example of AST

Example 6.

```
public class SomeClass {

 public static void main(String [] args) {

 String toPrintValue = "Hello world!";

 System.out.println(toPrintValue);

 }

}
```

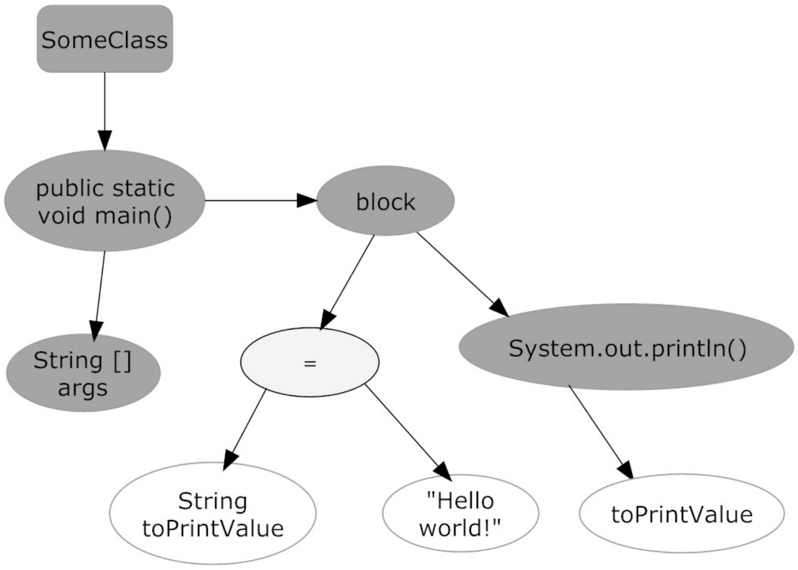

Fig. 3. Example of AST

As can be seen, the graphs differ just slightly, and after assembling and comparing the tokens sequence, it would be obvious that the graphs are 88% similar. The same parts of the graphs are distinguished by a grey color. Despite of its ostensible similarity, it is not easy to discover this “non-ideal” clone by a computer. Also the code refactoring is not an easy task. Many approaches have been introduced. Here will be mentioned a method using an AST. The desired final state is that a new method or variable will be extracted. The originally two different snippets will be the same now, and one of them can be deleted. Another approach is to use another type of graph. Also this method will be mentioned.

4. Code refactoring with AST

With and AST can be a source code refactored and optimized. The refactoring is a code change that does not affects external behavior of an application, but makes the system well-arranged. If the development gets into the stage where the code is messy and hard sustainable, it is necessary to use refactoring. On the other hand, the system becomes prone to errors and maintenance requirements are getting higher. Refactoring deals the publication [7].

The code transfer into a graph is very useful, because it is possible to perform (with respecting the rules) graph operations. This is much easier approach then performing these operations on the pure source code. The original source code is full of a superfluous, for functionality uninteresting, material. With an AST, all the comments, spaces, empty lines, or other syntax material can be omitted. The big advance is that the graph operations are very well known and described. The algorithms for the tree comparing, sorting, and walking through a tree can be useful. On the contrary, these calculations are very demanding to the computer hardware resources. But there exists a whole line of tools using the trees, and capable to perform fully automated refactoring operations. Many of those operations are already implemented in every IDE and using the AST operations. The typical example of refactoring can be a variable extraction. As an example, the trees and the code snippets from the above text are used. The variable extraction is the easiest way to solve this case.

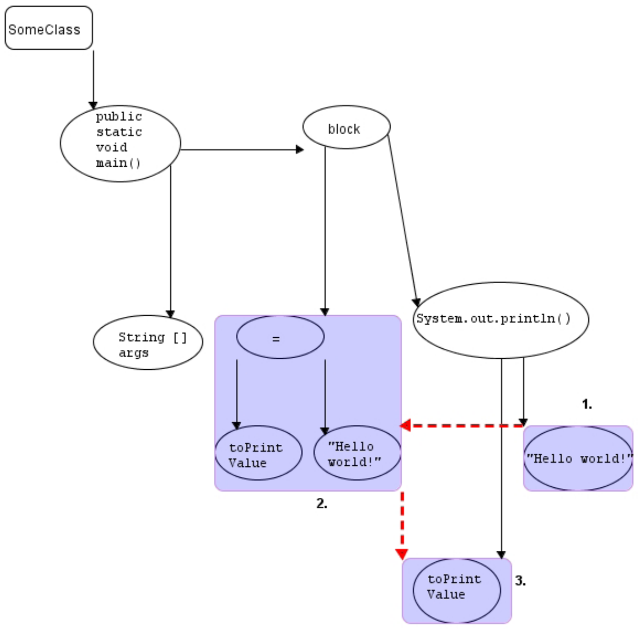

Fig. 4. Refactoring with AST

The new variable extraction takes place in few steps. First, it is obvious from the graph that the text value "Hello world!" moves to the new variable. Next, the new instance of the same type variable with the name "toPrintValue" is made. Finally, the new variable is used as a parameter to the "println()" method. The result is that the first case is converted to second one. The new method extracting is also a very frequently used method. When a two parts (or more) of a code are considered a clone, the desired target is to extract a new method with common operations of this clones. The greater number of elements will be extracted, the better it is the refactoring. However, the set of common elements is large, it is also important that is common for a great number of the considered snippets. Then the refactoring can be considered successful.

5. Program dependence graph (PDG)

Another approach is to use a PDG. This type of graph is even more abstract then an AST. It reflects data and control flow between the statements, expressions and operators. The graph nodes represent these elements while the edges define the data flow and its direction. a PDG can be drawn like the nodes representing the code lines while the edges making connections. Each edge has a label defining the data flow. a PDG express also the line dependence. If some statement uses the previously defined variable, then is dependent on the line defining it.

Like an AST, a PDG is abstract, language independent representation of a source code. Also a PDG is used by the compilers, and it is possible to use it for the program refactoring and optimizing. From a PDG, the code can be generated into the various programming languages without the knowledge of the original source syntax.

Example 7.

```
int a = someFunction();

int b = someFunction();

while (a<b) {

        a++;

}

return a + b;
```

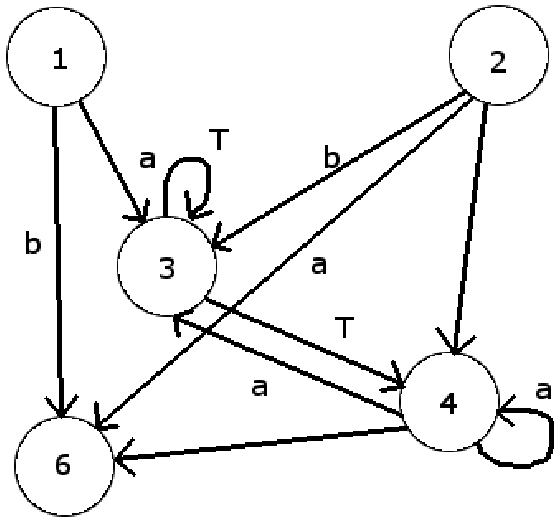

Fig. 5. Example of PDG

The advantage of a PDG over an AST is not only the language independence, but the ability to elevate to syntax. Two code snippets with the same elements and the same data flow can be transformed into a new, highly optimized, form. Moreover, this form can be different from the original code. With a PDG, total different parts of a code can be discovered as the clones.

6. Conclusion

Probably the biggest advance of the tree code expression is that it is independent on the language specific syntax rules. Next, the problems generated by the classical text representations are eliminated. It is not necessary to deal with the spacing, space, and other redundant material. The methods for the source code work using the tree expression are very interesting alternative to the ordinary ones, however, its implementation is very demanding on the computer performance. The algorithm for the tree comparing has O(n3) complexity [4], and the consequent processing is also very consuming. But these methods offer the wild range of the opportunities and optimizations. Therefore, the problems can be partially eliminated by the proper adapting to actual uses.

The tree expression of source code can also serve as a teaching tool. Despite of the used programming language, students can see how the programming operations work on a graph, how difficult is to repair broken code, and how the programs like PMD, FindBugs etc. works. While their future programming, they will perhaps consider the written code, and they will more think about what they write.

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

[2] 2. Jeanne Ferrante , Karl J. Ottenstein , Joe D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems (TOPLAS), v.9 n.3, p.319-349, July 1987 [doi>10.1145/24039.24041]

[3] 3. George K. Baah, Andy Podgurski, Mary Jean Harrold, The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis, IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 528-545, July/Aug. 2010, doi:10.1109/TSE.2009.87

[4] 4. 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

[5] 5. 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

[6] 6. Bublík T., Virius M., Automatic detecting and removing clones in Java source code, Tworba Software 2011, Ostrava 2011

[7] 7. Fowler M., Beck K., Refactoring: improving the design of existing code, Addison-Wesley Professional, 2001

[8] 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] 9. Juillerat, N. Models and Algorithms for Refactoring Statements, PhD thesis. Fribourg: University of Fribourg, Switzerland, 2009

Ing. Tomáš Bublík, VŠMIE, tomas.bublik@gmail.com

Simplification of source code with graph structure expression

Ing. Tomáš Bublík

Abstract. This paper deals with abstraction of a source code. It is possible to express source code like a graph. This abstraction is suitable for easier understanding the source codes with more complex syntax. Two structures will be introduced on Java source code: an abstract syntax tree and a program dependence tree. Next, its properties and advantages will be introduced. Although its possibilities are wide, some of them will be mentioned. Moreover, the area with successful graph usages will be described. Clone detection, refactoring, or plagiarism could be as an example. The examples in this paper are introduced in one of the most popular language – Java. For most of readers will be probably easy to understand it.

Keywords: Java, abstract syntax tree, program dependence graph, clones detection, source code, refactoring

JEL classification: C61