Software Evolution
Series 2

Rasha Daoud - 11607394
Ighmelene Marlin - 10296050

December 18, 2017

1 Introduction

This document presents our solution for detecting certain types of code cloning in open-source Java project. Furthermore, detected clones are presented in different ways. Our tool visualizes clones per each java file, and clone classes of multiple clone fragments.

The main motivation behind building the tool is to improve code quality, and avoid redundant code. We believe that being aware of clones helps the system maintainer to estimate the changeability costs and to better analyze the system on hand (Table 1).

The document explains the basic algorithm of the tool, design decisions, clone visualization, and testing properties to ensure the correctness of the tool.

The following clone types are detected:

- *Type1:* identical code segments, except for whitespaces and comments
- *Type2:* Syntactical copy, ignoring variable names & types, and function identifiers.

properties / characteristics	volume	complexity per unit	duplication	unit size	unit testing
analysability	x		x	x	x
changeability		x	x		
stability					x
testability		x		x	x

Table 1: SIG maintainability scores

2 Design rationale

In this section we will give a brief description of the design algorithm. And how the tool satisfies the requirements of code maintainers following the suggested literature.

2.1 Clone detection algorithm

Our clone detection tool finds clones of type 1 and 2. The algorithm is similar, we use AST comparison for detection.

However, for clone detection of type 2, when visiting AST, we normalize nodes following type 2 definition. We replace each type by wildcard(), and we rename all methods to same string name.

Type1 & type 2

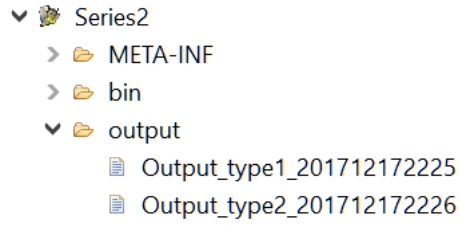
```
1   Function cloneDetection(project)
2   {
3         // Extract information
4          Extract AST from project
5
6         // Extract blocks
7          Visit (AST)
8          {
9                For every subtree get code segment & exact location (begin & end)
10               Fill out a storage with information
11               If (type2)
12               {
13                   Normalize node before storing data
14               }
15
16              "|unknown:///|" is ignored
17         }
18
19        // Link code segments
20        The storage map is processed to have all possible sequential code blocks and locations
21        Blocks that are smaller than 6 loc are ignored, following SIG
22
23        // Store duplicated items
24        For (every item in the storage)
25        {
26              If (size(storage[item]) >=2) // cloned segment
27              {
28                  Store storage[item] in cloneClasses map structure
29              }
30        }
31
32        // Post processing, get rid of subsumptions
33         For (every key in cloneClasses map)
34         {
35             For (every key2 in cloneClasses map)
36             {
37                  //Check string content is not the same
38                  If (content is the same & keys are different)
39                  {
40                        //Compare locations (overlap, contained, none)
41                        If (strictly contained)
42                        {
43                              Drop subsumed clone class from the map
44                        }
45                  }
46             }
47        }
48  }
```

```
node convert(node subtree) {
return visit(subtree){
        case \simpleName(_) => \simpleName("simple")
        case \variable(_,ext) => \variable("var",ext)
        case \variable(_,ext,i) => \variable("var",ext,i)
        case \method(t,_,ps,es,impl) => \method(t,"unit",ps,es,impl)
        case \method(t,_,ps,es) => \method(t,"unit",ps,es)
        case \parameter(t, _,ext) => \parameter(t,"parameter",ext)
        case Modifier _ => \private()
        case Type _ => wildcard() // will normalize all types
        }
}
```

2.2 Output

Clone classes, biggest clone and biggestclone class, and other statistics such as duplication percentage are written to an output text file under /output/ folder in Rascal project.

Different output file is written for each type of clones.




```
Output from analyzing clone classes of type2 for small project:

Project volume after clean up= 23589.
Project contains 80 clone classes and the biggest clone is 134 lOC.
The biggest clone class has 6 clones.

1)
block([declarationStatement(variables(wildcard(),[variable("var",0,methodCall(false,simpleName("s
|project://smallsql0.21_src/src/smallsql/junit/TestOperatoren.java|(5185,192,<96,11>,<101,9>)
|project://smallsql0.21_src/src/smallsql/junit/TestFunctions.java|(22560,192,<375,11>,<380,9>)
|project://smallsql0.21_src/src/smallsql/junit/TestMoneyRounding.java|(1823,192,<54,11>,<59,9>)
|project://smallsql0.21_src/src/smallsql/junit/TestDataTypes.java|(3089,192,<69,11>,<74,9>)



2)
block([expressionStatement(assignment(arrayAccess(simpleName("simple"),simpleName("simple")),"=",
|project://smallsql0.21_src/src/smallsql/database/SSPreparedStatement.java|(8336,237,<247,29>,<25
|project://smallsql0.21_src/src/smallsql/database/SSStatement.java|(6988,324,<269,35>,<276,13>)



3)
block([declarationStatement(variables(wildcard(),[variable("var",0,newObject(wildcard(),[]))])),e
|project://smallsql0.21_src/src/smallsql/junit/TestOperatoren.java|(11816,180,<267,75>,<273,5>)
|project://smallsql0.21_src/src/smallsql/junit/TestJoins.java|(10508,180,<204,75>,<210,5>)
```

Figure 1 – Output files location & example

2.3 Maintainer requirements

As stated in multiple literatures, software system maintainer might have different types of requirements in a clone detection tool. We built our tool to fulfil some of these requirements.

Facilitating navigation: our clone classes visualization is interactive. The user can click on one of the file names, the action will open the java file and highlight the cloned fragment. The user can click on all clones in one clone class and access all cloned fragments in java code.

On the other hand, in the view of clones per file, lines of clones are reported, so the maintainer can easily see where to manage these cloned lines.

Improving comprehension: we believe that our tool fulfils this requirement. We report multiple statistics and we point out the biggest clone. Furthermore, our visualization provides sufficient overview for clones.

Understanding programming patterns: Our tool reports and visualize clone fragments per file, which can help to reveal the copy/paste pattern used in certain components of the software system. Developers who are working on such components can be guided to reduce duplication in the future.

Reducing disorientation: Again, the output and visualization produced by the tool guide the maintainer to where the duplicated code is. Therefore, we believe that this requirement is fulfilled by out tool.

3 Visualization

We used RASCAL *Vis library* to visualize clone fragments per file, and clone classes.

We also designed a simple main page, with a drop-down-menu that includes one-source projects of interest.

The user can either run type1 or type2. Different views open on clicking on one of the types buttons. The user can go back to the main page from all other screens at any time.

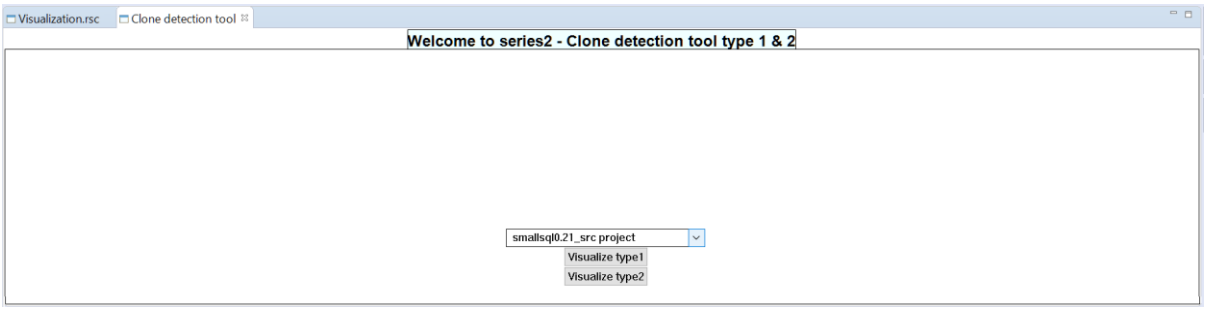


Figure 2 – Main page

Clones per file: we got inspirited by the paper of Koschke. We've used a box to represent a java file. And we used outlines with some markers & info to highlight clones in the box.

Furthermore, boxes of files were stacked in one view horizontally. And statistics are shown on same screen.

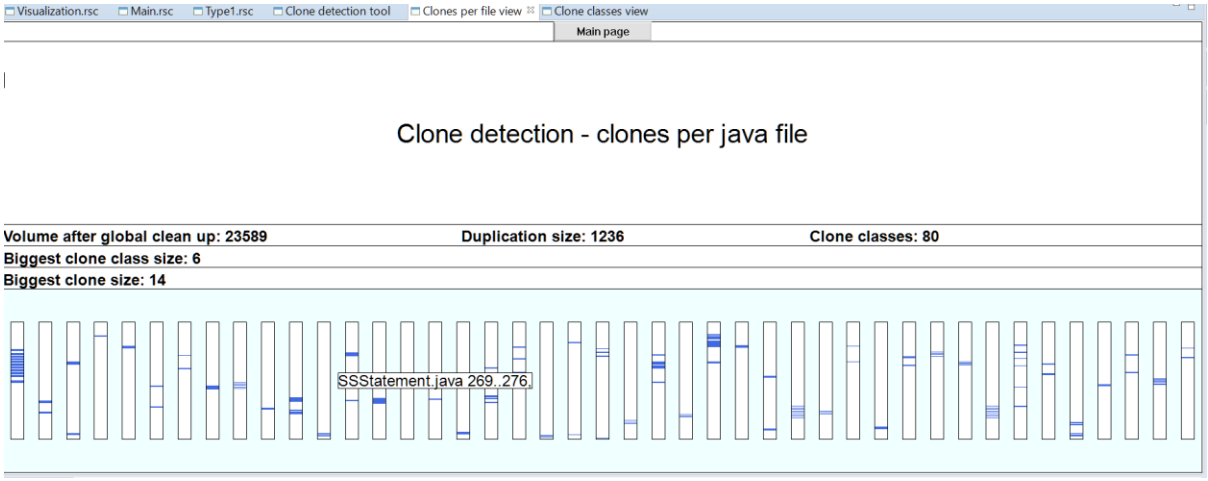


Figure 3 – Clones per file – static height

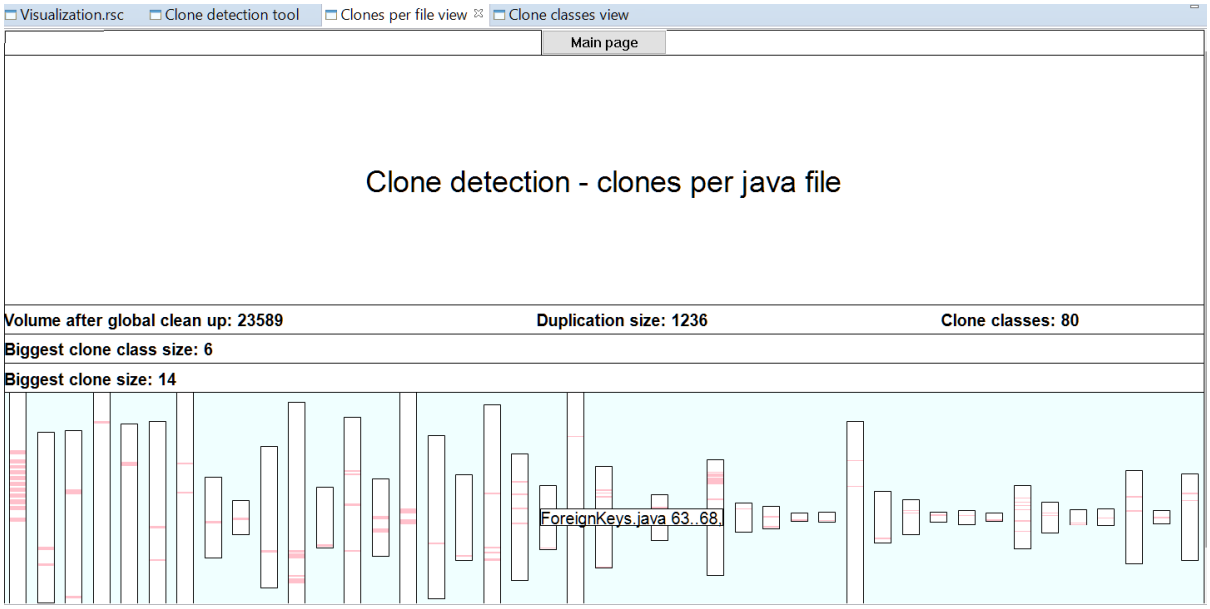


Figure 4 – Clones per file – relative height/ file size

Clone classes: we've used similar approach with boxes and onMouseDown function, to make the view of clone classes interactive. Each big box has small boxes of clones. The user can click on any of the file names (clones), that will open the java file and highlight the clone.

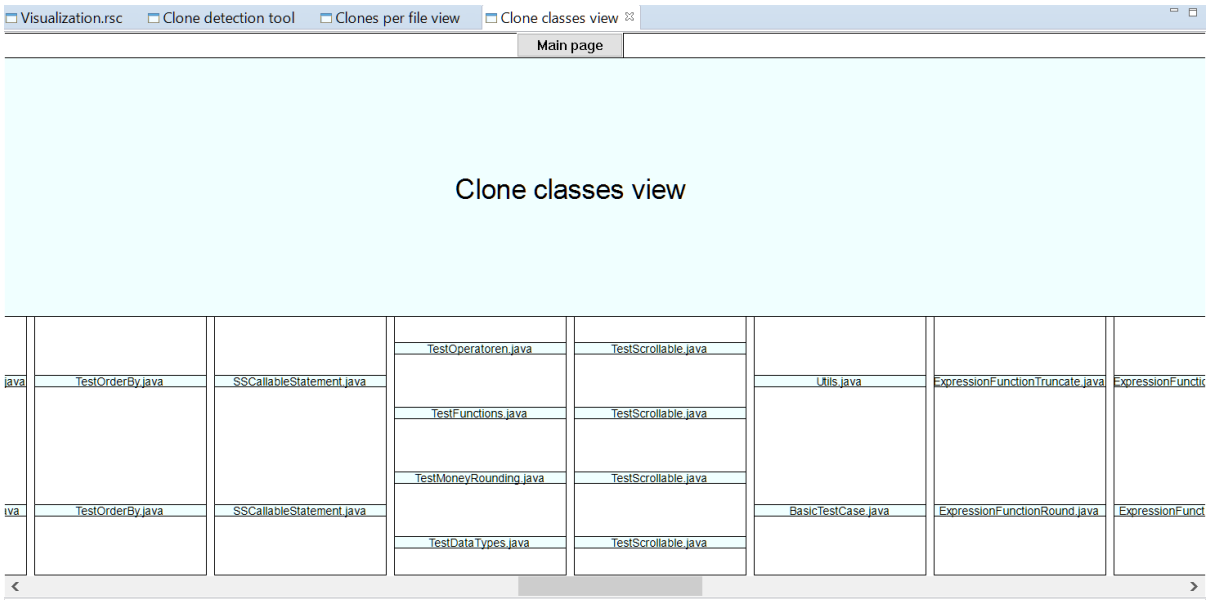


Figure 5 – Clone classes (interactive)

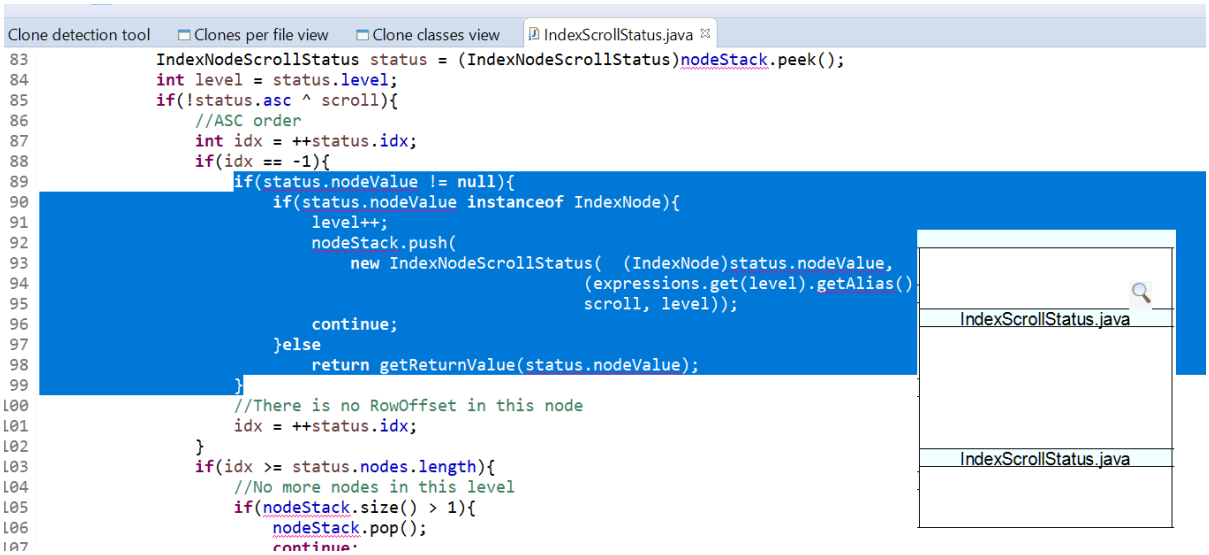


Figure 6 – Java file on click on one file name in a clone class

Clone classes (hasse diagram): We also visualize hasse diagram to visualize clones within a clone class. Each node is interactive as well. The user can click any box to open the cloned fragment in java file.

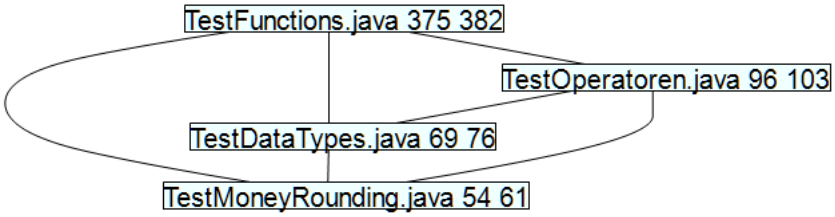


Figure 6 – hasse diagram – clone class (interactive)

Extra visualization: Next to that, we also render biggest clone and biggest clone class in the same interactive way.

4 Testing

RASCAL project includes a unit-test of multiple testable properties, to check the correctness of the algorithm. Three major properties were implemented:

- *Clone classes correctness:* a clone class should not be duplicated in the resulting map. If two keys don't match, the content should not match
- *Count clone correctness:* Clone volume which is found by parsing the resulting map, and counting lines based on the stored location (begin & end lines), should match the sum of node sizes found in the AST and reported as clones. This is after cleaning up the subsumptions.
- *Clone size:* Each clone class should have clones with same size (LOC), and if commands are extracted, strings should match.

On top of that, the tool is used on pre-prepared java projects with known clones of type 1 & 2. The statistical results were as expected.

5 Limitation:

The tool causes eclipse to hang for some time when running visualization on the big project.

References

[1] Paul Klint. Towards visual software analytics.

[2] M.-A.D Storey, F.D Fracchia, and H.A Müller. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. Journalof Systems and Software, 44(3):171 – 185, 1999.

[3] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Sci. Comput. Program., 74(7):470–495, May 2009.

[4] Clone Detection Using Abstract Syntax Trees Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, Lorraine Bier.