Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

APA
Plagiarism Dectection: a tool and a comparison

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Contributions by Peter Rademaker

Department of Information and Computing Sciences, Universiteit Utrecht

June 7, 2012

1. Introduction

Overview

What is plagiarism?

> *het in een scriptie of ander werkstuk gegevens of tekstgedeelten van anderen overnemen zonder bronvermelding. (Docenthandleiding Dept. Informatica)*

which translates to

> *to copy information or textual passages written by others into a paper or other artifact without proper citation.*

Why do we need tools?

- Detecting plagiarism in computer programs is hard to do by hand:
  - discoveries tend to be accidental, based on remarkable similarities
  - only between assignments handed in in the same year
  - fewer discoveries if the group of students becomes very large
    - assignments are checked by various people
- Support is essential when students number in the hundreds, and the same assignment is given repeatedly

2. What can Marble do?

Marble

- Lends support in discovering plagiarism in (mainly Java/C#) programs
  - listing pairs of files, sorted on amount of similarity
  - results in an executable script that shows these files with their similarities
  - also compares against a collection of assignments of previous years
  - is relatively fast (20,000 in 6 minutes and 20 seconds)
  - and was little work to program
- Marble is tailored to Java/C#, but variants made and applied to PHP, Perl and XSLT

Some numbers on plagiarism cases

- Since 2006:
  - IMP 2006: five new cases
  - IMP 2007, mandelbrot: 5 cases of plagiarism, 4 of selfplagiarim
  - IMP 2007, reversi: 8 new cases of plagiarism, 2 of selfplagiarism
  - IMP 2008, reversi: 11 cases of plagiarism, 2 cases of collaboration, 4 of selfplagiarism
  - IMP 2009, mandelbrot: 1 case of plagiarism
  - IMP 2009, reversi: 5 cases of plagiarism
  - IMP 2010: no cases.
  - DS 2011: 10 and DS 2012: currently at 3
- Note: selfplagiarism is allowed, but still useful to know.
- Patterns: Computer Science vs. Information Science

Characteristics of Marble

- Compares all newly handed in assignments to
  - each other
  - to all formerly handed in assignments
  - by comparing them source file to source file
- Comparison is insensitive to
  - names of variables/identifiers
  - string, character or numerical constants
  - indentation
  - position or contents of comments
  - package structure (to some extent)
  - order of definition of methods, inner classes and attributes
  - how classes are distributed over source files
- Some identifiers remain untouched.
- No deletion of template code.

3. How does Marble work?

How is Marble organized

- Two phases
  - the normalisation phase
    - Transforms source code into a special form suited for literal comparison
  - the detection phase
    - actually performs the comparisons and ranks the results
- Some assumptions are made about how assignments are organized:

  `halloworld/0405period1/jur/assignment2/`
- For submission directories, `assignment2`, we make no assumptions how they are organized.

Normalisation

Normalisation removes unessential detail from source files.

In particular, details that are easy to change without changing the behaviour of the program.

Either by tool, or by hand!

Normalisation in Marble

- Consider each Java source file in turn
  - Anywhere inside the `assignment2` directory
- Split them up into a separate file for each class
- Normalise the names
  `assignment2/src/Hello World.java` becomes
  `assignment2/src!Hello@World.java`
- For each of these files, residing at top level, we normalise the Java source code.

Code normalisation in detail

- Remove comments and literal strings and characters
- Map identifiers to `X`, except
  - keywords (`while`), special constants (`true`), special methods (`wait`) and special types (`String`)
- We keep these special identifiers to avoid false positives
- Decimal and octal numbers $\Rightarrow N$
- Hexadecimal numbers $\Rightarrow H$
- Essentially, we map the tokens in the program to special uppercase letters.
- Retain symbols like assignments, braces, arithmetic symbols.
- Each token on a separate line (or almost)

An example

The class

```
class Bliep extends Zwiep {
   String glob (int z) {
     int cnt = x;
     cnt = cnt*2;
   }
}
```

becomes

```
CLASS X EXTENDS X {
   STRING X ( INT X ){
     INT X = X;
     X = X * N;
   }
```

Actually...


```
CLASS
X
EXTENDS
X
{
STRING
X
(
INT
X
){
INT
X
=
X
;
X
=
X
*
```

Two variants

- In one variant (.nf) we are now done.
- In another variant (.nfs) we "sort" the methods, attributes and inner classes:
  - annotate each brace, { and }, with its nesting depth.
  - extract inner classes, methods and attributes based on positions of paired {1 and }1 and semi-colons ;
  - group methods, attributes and inner classes together
  - sort within each group on the length of normalised code, then alphabetically
- If students actually moved methods around, using the .nfs version for comparison gives much better results

The detection phase

- Consider all files with extension (.nf or .nfs)
- Compare them using the standard `diff` utility

```
differentlines = diff file1 file2 || wc -l
len1 = wc -l file1
len2 = wc -l file2
measure =
  100 - 100 * differentlines / (len1 + len2)
```

- measure is 100 if very similar, 0 when very dissimilar.

The generated output

- Each score above a given threshold generates one line

```
echo 100 59 59 85  S && vimdiff \
  ../org/jur/origineel/QSortObserver.java \
  ../hist/testset/versie9/QuickSortObserver.java
```

- 100 is the score attained for the Sorted version, 59 and 59 are the respective "file sizes"
- 85 is then the score for unsorted
- File is sorted in descending order (score than size)
- File can be run as a script (under Linux/Unix)
- Typically, a lecturer goes through these until he/she
  - discovers that the last five cases show similarities, but within limits
  - or gets fed up

4. A small experiment

Experiment set-up

- Two student assistants, Arjen Swart and Arie Middelkoop,
- were handed somebody else's assignment for an exercise they also made themselves
- Their task: change the program as much as possible to avoid detection, but
- the program should behave in the same way and should be human readable.

Arjen Swart's nine versions

Version	modification	nf score	nfs score
1	comment and layout changes	100	100
2	interchanged method declarations	96	100
3	attribute declarations exchanged	96	100
4	calls to GUI methods exchanged	87	99
5	imports changed	87	99
6	GUI text and colours changed	86	99
7	identifier names changed	86	99
8	rewrote some expressions	86	98
9	get/setmethods inlined	86	98

- Scores are the highest ones for a significantly large class file
- For non-plagiarism: highest scores obtained are 51 for nf, 52 for nfs
- nfs score is sometimes worse!

5. How does Marble do it?

Main characteristics

- Marble should be short, easy to implement and maintain
  - 440 lines of code, 220 line of comment
- Significantly flexible to change
  - no parsing, only work on lexical level
- Programming language is Perl
  - ugly and obscure
  - regular expressions supported directly in the language
  - meant for report generation, text manipulation
  - why: familiarity and regexps

On extensability

- Moving to Java 1.5:
  - add two new keywords to a specific array in the program (`enum` and `assert`)
  - verify that generics do not interfere too much
- Moving to C#:
  - after reading up on C# syntax and finding a token description
  - a few hours of work to deal with nested namespaces
  - namespace declarations are first deleted, and then proceed as usual

Regular expressions

- The main tool for normalisation
- First map the whole program to lower case
- Capture tokens, replace them by an upper case character or remove them
- Uppercase parts never matched: transformed text never touched again

Regular expression examples

- To replace all octal and decimals numerals by `N`:

```
$prog =~ s/\d+/N/gs;
```

- To remove all comments and literals in the program:

```
$prog =~
    s/(\/\*(.|\n)*?\*\/) # multi-line comments
     |(\/\/.*?\n)        # end-of-line comments
     |(".*?")            # string literals
     |('.*?')            # character literals
     / /g;               # replace all with space
```

- Why *? and not *?

Annotation of braces


```
sub annotateAccolades ($) {
  my $src = $_[0];
  my $depth = 0;
  my $dest = "";
  while ($src =~ /(.*?)(\{|\})(.*)/s) {
    ($upto,$match,$src) = ($1, $2, $3);
    if ($match eq "{") {
        $dest .= "$upto$match$depth";
        $depth++;
    }
    else {
        $depth--;
        $dest .= "$upto$match$depth";
    }
  }
  return $dest;
}
```

Retaining untouchables


```
sub untouchabilize ($) {
  my $line = $_[0];
  $line =~ tr/A-Z/a-z/; # Line to lower case
  foreach $untouchable (@untouchables) {
    $UNTOUCH = "\U$untouchable";
    $line =~ s/(\W|\s)$untouchable(\W|\s)/$1$UNTOUCH$2/gs;
  }

  foreach $class (@preserveclasses) {
    $CLASS = "\U$class";
    $line =~ s/(\W|\s)$class((\s+)[a-z\$_])/$1$CLASS$2/gs;
  }
  return $line;
}
```

A class (use) is an identifier that is followed directly by another

What is untouchable?


```
@keywordsunsorted =
  ("abstract", ... "enum", "true", ..., "null");
@additionals =
  ("system.out.println", "tostring", "wait", "notify");
@untouchables =
   sort bylength (@keywordsunsorted, @additionals);
```

Order can be essential

- Remove escaped quotes and backslashes
- Remove comments and literals
- Remove superfluous whitespace
- Preserve untouchables
- Replace octal and decimal numbers
- Replace hexadecimal numbers
- Separate $X{+}N$ into $X\ +\ N$.
- Replace remaining identifiers with X

Summary

- Marble has been an ongoing side-track for many years
- Discovered quite a few actual plagiarism cases
  - More than documented by the Exam Committee
- Characteristics of the system are
  - little code, lots of documentation
  - Uses token-abstraction to normalize code
  - by means of Perl's regular expression.
  - For the rest, the code is mainly administration.
  - command-line scripts with a script as output
  - Running that script gives the most suspect cases first.
  - Using the right editor, quickly shows the lecturer whether it's plagiarism or not.

6. A comparison of plagiarism detection tools

Where to next?

- Slides on our comparative study of plagiarism detection tools.
- Published/presented at CSERC 2011, Heerlen.
  - 1st International Computer Science Education Research Conference

7. Holmes

What is Holmes?

- Marble for (all of) Haskell
- But with ...
  - detemplating
  - multiple heuristics
  - dead code removal
- Based on `haskell-src-exts`
- Joint work with Brian Vermeer and Gerben Verburg

How does it work

- Similar to Marble:
  - `holmes-prepare`: computes token stream, but this time also other information
  - `holmes-compare`: compares submissions/files pairwise (mostly submission at this time)

Detemplating

- Code marked as template is removed (various pragma's exist)
- Allows us to deal with lecture provided code
- Per module, and/or per function

Dead code removal

- Lecturer provides "starting points".
- Only code reachable from starting point will be retained.
- Specification examples `Main.*` and `*.main`

```
useful = .....
spurious = ....
cleverlyHidden = ...

main =
  let
    f = (spurious, useful)
    g = cleverlyHidden
    h = useful
  in const h g
```

Implemented heuristics

- Implementation by Brian Vermeer had more than a dozen
- In the final version only five are used for comparison
  - fingerprinting
    - Taken from MOSS, information theory, generic
  - tokenstream
    - As in Marble
  - indegree signature of top-level functions (compared in three different ways)

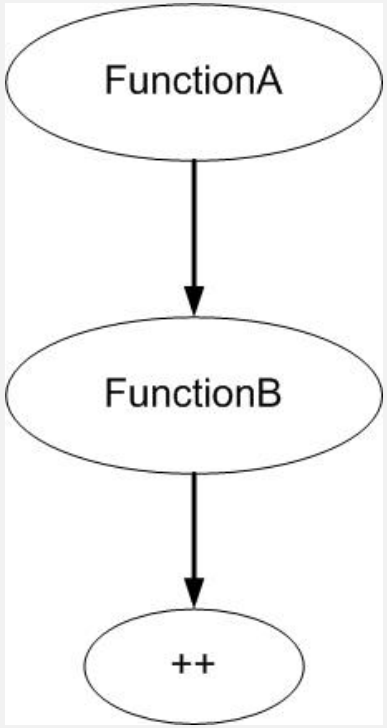
Token stream (slight return)


```
module Token where
f y = 3
main :: Int -> IO()
main x = do
           putStrLn (f (x + x))
           return ()

X X = I X X = do X ( X ( X 0 X ) ) X ()
```

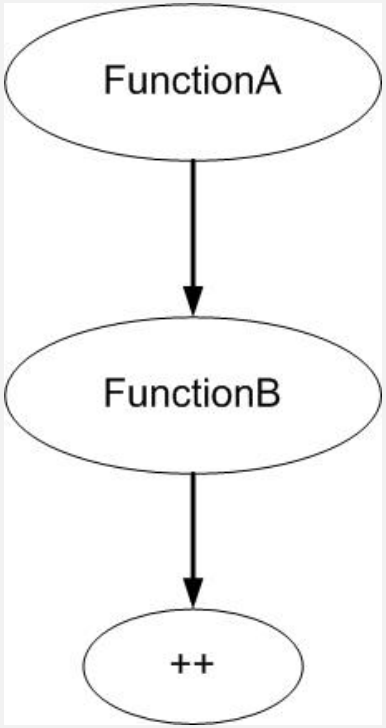
Sorting the functions: by arity, number of tokens, alphabetically
Haskell's Diff library used for comparing

Control flow graph


```
functionA :: String
functionA = let
               f =  functionB "foo"
            in
               f "bar"

functionB :: String -> String -> String
functionB str1 str2 = str1 ++ str2
```



In-degree signature



For each vertex, compute the number of incoming edges; sort the list of degrees: 0, 1, 1.

`FunctionA` has in-degree 0, `FunctionB` has in-degree 1.

Sample output


```
fingerprints; tokens; ind1; ind2; ind3; sub VS sub;
015; 067; 076; 048; 079; 2007/xx-yy VS 2007/zz1;
019; 067; 052; 034; 069; 2007/xx-yy VS 2001-hugs/zz2;
026; 064; 068; 068; 079; 2007/xx-yy VS 2004/zz3;
```

Import into Excel for easy manipulation.

Sensitivity Analysis again

Single refactorings	
Name	*Description*
nc	changed identifier names
tc	translated comments from Dutch to English
rl	changed the order of the function declarations
rw	simple transformations like `where` to `let - in`
trc	declared a trace function similar to the Debug module and let all functions call trace
cp	move single used functions to local scope
un	declared a unit test function that calls all functions declared in the module

Sensitivity results

original VS ...	tks	in1	in2	in3	fps
nc	100	100	100	100	68
bogus	3	12	4	12	0
trc	85	92	46	92	68
tc	100	100	100	100	100
rl	100	100	100	100	91
rw	87	85	86	94	78
compact	86	94	58	94	99
unit	91	61	60	80	86
nc_rw	87	85	86	94	53
nc_rw_tc	87	85	86	94	53
nc_rw_tc_cp	77	83	62	89	53
nc_rw_tc_cp_trc	74	84	67	92	42
nc_rw_tc_cp_trc_un	68	58	58	81	37
nc_rw_tc_cp_trc_un_rl	68	58	58	81	36

Applying Holmes to real life data

- On all submissions for FP from Submit
- Organised per assignment/incarnation/submission
- No detemplating, `*.*`, only submission level

Some statistics

total submissions	2122 (out of 2250)
different assignments	18
total incarnations	36
max repeats for an assignment	7
total students	1042
max assignment for any student	11

Corpus

Assignment name	incarnations	submissions
fp-afschrift	1	65
fp-afschriftgui-ghc	1	62
fp-agenda	1	78
fp-beeldverwerking-ghc	1	59
fp-creditcardvalidation	1	93
fp-fpcal	1	68
fp-fql	6	420
fp-getallen	1	95
fp-html	1	68
fp-kalender	1	6
fp-mastermind	2	156
fp-propositielogica	7	380
fp-river	1	70
fp-rocks	1	70
fp-soccer	2	52
fp-spreadsheet	1	5
fp-turtlegraphics	4	163
fp-wiki	1	74
fp-wisselkoers	1	163
fp-wxcal	1	52

Results

- 63 cases of clear cut plagiarism, 3 cases of fraud
- 12 additional cases that were less clear cut
- 27 cases of plagiarism through previous incarnation
- Only seven cases had a lot of identical code
  - Refactoring/rework have been performed otherwise
- Tokenstream and fingerprinting each have something to contribute
- Indegree signatures often also high accidentally

Summary

- Results are very promising, even without file-to-file submissions
- With better suited assignments, Holmes is likely to do better
  - room to roam
  - one starting point
  - template annotations if necessary
- Holmes will be used next year during FP