Proceedings of the Federated Conference on Computer Science and Information Systems pp. 701–705
DOI: 10.15439/2017F147
ISSN 2300-5963 ACSIS, Vol. 11

Program analysis for Clustering Programmers' Profile

Daniel José Ferreira Novais
Dpt. Informática, Centro Algoritmi
Universidade do Minho
Braga, Portugal
danielnovais92@gmail.com

Maria João Varanda Pereira
Dpt. Informática e Comunicações, IPB
Centro Algoritmi, Universidade do Minho
Bragança, Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
Dpt. Informática, Centro Algoritmi
Universidade do Minho
Braga, Portugal
pedrorangelhenriques@gmail.com

***Abstract*—Each programmer has his own way of programming but some criteria can be applied when analysing code: there are a set of best practices that can be checked, or "not so common" instructions that are mainly used by experts that can be found. Considering that all programs that are going to be compared are correct, it's possible to infer the experience level of the programmer or the proficiency level of the solution. The approach presented in this paper has as main goal to compare sets of solutions to the same problem and infer the programmers profile. This can be used to evaluate the programmer skills, the proficiency on a given language or evaluate programming students. A tool to automatically profiling Java programmers called PP (*Programmer Profiler*) is presented in this paper as a proof of concept.**

I. Introduction

TWO given solutions that solve the same problem can be very different. The style of programming, the proficiency on the programming language, the conciseness of the solution, the use of comments and so on, allow to compare programmers through static analysis of their code. It is possible to measure the proficiency on a programming language in the same way that we measure the proficiency on a natural language [Pos14]. Using, for example, the *Common European Framework of Reference for Languages: Learning, Teaching, Assessment* (CEFR) method[1] it is possible to classify individuals based on their proficiency on a given foreign language. Statically analysing code, it should be possible to extract a set of metrics and using a set of best practices to infer the proficiency and style of programming. The main idea is to evaluate the programmers' profiles, comparing code, without the need to construct a standard solution to perform that comparison. When facing a class of students or when evaluating a group of candidates to a programmer position at a company, we only need to compare them to each other to find the best one or to create a rank. Of course we can include a best solution in the group in order to perform an absolute evaluation, especially needed in non-academic environments. The attributes or metrics that will allow to infer a profile can be defined a-priori by hand (using intuition) or can be extracted through data-mining techniques as can be seen in [KCM07]. However this last approach requires the availability of huge collections of programs assigned to each class.

Pietrikova in [PC15] also explores techniques aiming the evaluation of Java programmers' abilities through the static analysis of their source code. Static code analysis may be defined as the act of analysing source-code without actually executing it, as opposed to dynamic code analysis which is done on executing programs. The latter is usually performed with the goal of finding bugs or ensure conformance to coding guidelines. In our approach the goal is to further explore the discussed techniques and introduce new ones to improve that evaluation, with the ultimate goal of creating a tool that automatically profiles a programmer only using static analyse of code. Notice that in our work we do not cope at all with automatic code assessment or program verification; we only focus on the programmers' ability to master a programming language.

Concerning the knowledge about a language or the capability to write 'naive/expressive' sentences on that language, a possible set of profiles would be: novice, advanced and expert. Moreover, other relevant information is expected to be extracted, such as the classification of a programmer on his code readability (indentation, use of comments, descriptive identifiers), hid defensive programming style, among others.

There are some source-code elements that can be analysed to extract the relevant metrics to appraise the code writer's proficiency such as: number of statements and declarations, existence of some repetitive patterns, number of lines (code lines, empty lines, comment lines), use of indentation, quality of the identifiers, use of not so common instructions and other characteristics considered as good practices. In this work code with errors will not be taken into consideration for the profiling. This is, only correct programs producing the desired output will be used for profiling.

In order to build the PP tool to automatically extract metrics from programs and to profile the owners of those programs, language processing techniques will be used. This process will be complemented with the use of a tool, called PMD[2], to get information on the use of good Java programming practices. PMD is also a source code analyser that finds common programming flaws like unused variables, empty catch-blocks,

[1] http://www.coe.int/t/dg4/linguistic/cadre1_en.asp

[2] https://pmd.github.io/

unnecessary object creation, and so forth. For these reasons this tool proved to be very useful.

The paper will follow with Section II where related work will be reviewed in order to identify techniques and tools commonly used to deal with this problem. Section III is devoted to present our proposal for an automatic programmer profiling system based on source code analysis. The analyser implemented and the set of metrics extracted are presented in Section IV. In Section V we will discuss the correlation between metric values and profiles. A complete case study will be described in Section VI in order to show all the PP functionalities. The paper is closed at Section VII with conclusions and future work.

II. Related work

As it was said, the main motivation for the work described in this paper came from the study [PC15] of Pietriková and Chodarev. These authors propose a method for profiling programmers through the static analysis of their source code. They classify knowledge profiles in two types: subject and object profile. The subject profile represents the capacity that a programmer has to solve some programming task, and it's related with his general knowledge on a given language. The object profile refers to the actual knowledge necessary to handle those tasks. It can be viewed as a target or a model to follow. The profile is generated by counting language constructs and then comparing the numbers to the ones of previously developed optimal solutions for the given tasks. Through that comparison it's possible to find gaps in language knowledge.

In [TRB04], Truong et al. suggest a different approach. Their goal is the development of a tool, to be used throughout a Java course, that helps students learning the language. Their tool provides two types of analysis: software engineering metrics analysis and structural similarity analysis. The former checks the students programs for common poor programming practices and logic errors. The latter provides a tool for comparing students' solutions to simple problems with model solutions (usually created by the course teacher).

Flowers et al. [FCJ+04] and Jackson et al. [JCC05] present a tool, *Gauntlet*, that allows beginner students understanding Java syntax errors committed while taking their Java courses. This tool identifies the most common errors and displays them to students in a friendlier way than the Java compiler. *Expresso tool* [HMRM03] is also a reference on Java syntax, semantic and logic error identification. Both tools have been proven to be very useful to novice Java learners but they focus mainly on error handling.

Hanam et al. explain [HTHL14] how static analysis tools (e.g. *FindBugs*) can output a lot of false positives (called unactionable alerts) and they discuss ways to, using machine learning techniques, reduce the amount of those false positive so a programmer can concentrate more on the real bugs (called actionable alerts). We are not considering the use of machine learning and data mining techniques in our approach. Our idea is to use a set of pre−defined criteria to evaluate programs and infer profiles.

III. Profile detection: our proposed solution

Programmer profiling is an attempt to place a programmer on a scale by inferring his profile. The first step towards achieving this profiling is to define what will be the profiles. A classification that could encapsulate a broad range of programming knowledge was developed.

The **Novice** is someone who's not familiar with all the language constructs, does not show language readability concerns and does not follow the *good programming practices*. The **Advanced Beginner** starts to shows variety in the use of instructions and data-structures. He also begins to show readability concerns by writing programs in a safely manner. The **Proficient** is a programmer who is familiar with all the language constructs, follows the *good programming practices* and shows readability and code-quality concerns. Finally, the **Expert** is someone that masters all the language constructs and focuses on producing effective code, sacrificing on readability.

The example seen in Listing 1 could be a bit exaggerated but may help shed some light on what is meant by the previous scale. Each one of the following methods has the same objective: calculating the sum of the values of an integer array, in Java. Each method has features of what may be expected from each profile previously defined. It's hard to represent all 4 classifications on such a small example, so the Advanced Beginner profile was left out.

Listing 1. "Examples of programs corresponding to different Profile Levels"

```
int novice (int[] list) {
    int a=list.length;
    int b;int c= 0;
    for (b=0;b<a;b++) {
      c=c+list[b];}
    return c;
}

//Sums all the elements of an array
int proficient (int[] list) {
    int len = list.length;
    int i, sum = 0;
    for (i = 0; i < len; i++) {
        sum += list[i];
    }
    return sum;
}

int expert (int[] list) {
    int s = 0;
    for (int i : list) s += i;
    return s;
}
```

The Novice has little or no concern with code readability. He will also show lack of knowledge of language features. In the example we can see that by the way he spaces his code, writes several statements in one line or gives no meaning in variable naming. He also shows lack of advanced knowledge on assignment operators (he could have used the *add and assignment* operator, +=).

The Expert, much like the Novice, shows no concern for language readability, but unlike the latter, he has more language knowledge. That means that the Expert has a different kind of bad readability. The code can be well organized but the programming style is usually more compact and not so explicit.

As an example of language knowledge, the Expert uses the *extended for loop*, making his method smaller in lines of code.

Finally, the Proficient will display skills and knowledge, much like the Expert programmer, while keeping concern with code readability and appearance. The code will feature advanced language constructs while maintaining readability. His code will be clear and organized, variable naming has meaning and code will have comments for better understanding.

Since the goal is to classify programmers automatically, that classification can only be carried through the analysis of the programmers' source code. Since the interest is in language usage, in various aspects, static code analysis was the selected technique to perform the extraction of the data to be analysed.

The two main aspects of code that were of interest to this project are the language knowledge and the readability of code. To classify the abilities of a programmer regarding his knowledge about a language and the way he uses it, we considered two profiling perspectives, or group of characteristics: language **Skill** and language **Readability**.

- **Skill** is defined as the language knowledge acquired and the ability to apply that knowledge in an efficient manner.
- **Readability** is defined as the aesthetics, clarity and general concern with the understandability of the code written.

We believe that these two groups contain enough information to obtain a profile of a programmer, regarding his ability to write proper language sentences to solve problems. Then, for each group, and according to the score obtained by the programmer, Table I gives a general idea of how programmers can be profiled. Notice that (+) means a positive score, while (-) means a negative one.

TABLE I
PROPOSED CORRELATION

Profile	Skill	Readability
Novice	-	-
Advanced Beginner	-	+
Expert	+	-
Proficient	+	+

What constitutes a lower and a higher score for each group must be defined. For every programmer, the goal is to compare each metric value among all solutions to identify those who performed better or worse on that metric, and then, assemble a mathematical formula which allows to combine the metrics' results into a grade for each of the two groups. Taking those two grades and resorting to Table I we can easily infer the programmer's profile in regards to the subject problem.

IV. SOURCE CODE ANALYSIS: METRICS EXTRACTED

After some testing and experimenting, we've created a set of metrics that we consider appropriate for programmer profiling. The range of metrics extracted is quite large, and it's obvious that not all metrics should have the same weight towards inferring the profile of programmers. Considering that, each metric has an associated priority (or weight) that directly relates to the impact that metric will have towards inferring the profiles. Table II formally specify the following rules that we are extracting for each solution to a given exercise. For instance, the first rule, should be read as: More classes imply more Skill points and more Readability points.

TABLE II
METRICS EXTRACTED AND RULES WITH THEIR PRIORITIES

Metric	Rule	Priority
Number of Classes	+ =>+R +S	2
Number of Methods	+ =>+R +S	2
Number of Statements	- =>+S	8
Number of LOC	+ =>+R	5
Percentage of LOC	- =>+R	5
Number of LOCom	+ =>+R	3
Percentage of LOCom	+ =>+R	3
Number of Empty Lines	+ =>+R	3
Percentage of Empty Lines	+ =>+R	3
# Control Flow Statements	- =>+S & + =>+R	5
Variety of Control Flow Statements	+ =>+S	4
# Not So Common CFSs	+ =>+S	6
Variety of *Not So Common Operators*	+ =>+S	5
# Declarations	- =>+S -R	5
# of Types	+ =>+S	4
# Readability Relevant Expressions	+ =>+R	3

Code Size Metrics

- These metrics are related with code size. We believe code size is mainly related with readability concerns.

Control Flow Statements Metrics

- Control flow statements (CFS) are the heart of the algorithms. Knowing how to properly use them says a lot about programming knowledge.

Not So Common Operators Metrics

- Java is a vast language with numerous operators. Some of them are very specific and most programmers don't know about them. When correctly applied these can reduce the code size and even improve the program's performance.

Variable Declaration Metrics

- Similarly to the case of the Control Flow Statements, the usage of Declarations could be an indication of a programmer's capabilities.

Other Relevant Expressions Metrics

- This metric was created to hold other important language features that fore some reason or another didn't fit in the other descriptions.

PMD Violations Metrics

- The PMD Violations Metrics are very important because they allow us to detect problems in code that otherwise would be very hard to catch. PMD rules have their own priorities.

V. RELATING METRICS WITH PROFILES

As time progressed, our idea of the profiles shifted a bit from the original idea that we saw in Table I. We decided that the Experts should be the ones with maximum focus on Skill,

the Proficients on Readability and the Advanced Beginners should more precisely divided. A new profile was also created. The final version of the profiles is the following:

- **Novice (N)**: Low Skill and Low Readability
- **Advanced Beginner (AB)**: Low-to-Average (LtA) Skill and Readability
- **Proficient (P)**: LtA Skill and High Readability
- **Expert (E)**: High Skill and LtA Readability
- **Master (M)**: High Skill and High Readability

Keep in mind that the definition of the groups (Readability and Skill) is not the common meaning of the word. Saying that an Expert has low Readability means only that he scored a low value on our axis of Readability (based on the metrics we've seen in the previous section) when comparing to other solutions to the same problem.

VI. Case Study

Taking, for instance, two students solutions for the following Java exercise: *Write a Java program that reads positive integers (number 0 will terminate the input). Compute and print the amount of even numbers, odd numbers, and the average (real number) of the even numbers.*

Listing 2. "Solution to P1 made by S"

```
import static java.lang.System.out;
import java.util.Scanner;

public class P1_S {

  public static void main(String[] args) {
    int nEven = 0, nOdd = 0, sum = 0;

      while(true){
        out.println("Insert a number:");
        Scanner ipt = new Scanner(System.in);
        int num = ipt.nextInt();

        if(num == 0) break;
        if(num%2 == 0) {
          nEven++;
          soma += num;
        }
        else nOdd++;
      }

      double average = 0;
      if (nEven != 0) average = sum / nEven;

      out.println("Even: " + nEven);
      out.println("Odd: " + nOdd);
      out.println("Even Avrg: " + average);
  }
}
```

Listing 3. "Solution to P1 made by Z"

```
import java.util.Scanner;
public class P1_Z {
 public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  int value = in.nextInt(), evens = 0,
  odds = 0;
  double evensSum = 0;
  /*I'm assuming the input is viable,
  i.e. all input numbers are
  positive integers*/
  while(value != 0){
   if((value & 1) == 0){
    evens++;
    evensSum += value;
   } else odds++;
    value = in.nextInt();
   }
   System.out.println(evens + "\n" + odds);
   System.out.println(evens >0?
   evensSum/evens:evensSum);
  }
}
```

Looking at the structures of both solutions, we can see they are both divided in the same way. Inside the main method, the first lines are used for variable declaration and initializations. Then we have the main cycle, where numbers are read and the variables are assigned. Finally, in the last lines we have our results output.

One thing we can easily observe is the size of both solutions, in regards to the number of lines. The first solution has 61% more lines of code than the second one. A closer inspection shows us that *S* had the concern of leaving empty lines between some code instructions, while *Z* didn't leave a single one. This is one of the most clear signs of concern for readability. Empty lines and indentation are probably most important things when creating readable code. Although it was not possible to implement the verification of correct usage of indentation (tabs or spaces) the usage of empty lines was, and it will weight for the readability grade.

Regarding the use of variables, *S* declares a total of 4 ints, 2 Scanners and 1 double while *Z* only needs 3 ints, 1 Scanner and 1 double.

The number of required variables reflects the capacity that the programmer has in reusing variables. Therefore, less number of needed variable declaration reflects a higher skill in the language. Of course that has the fallback of generally making the code less understandable (the same variable has different purposes), so there is a loss in readability as well.

That takes us to the main loop. *S* makes the mistake of reinitializing a Scanner and a int in every cycle iteration, that is a violation that is detected by the PMD tool. *Z* on the other hand reuses his variables.

Another bad practice detected by PMD on *S's* solution is the use of a *while(true)* cycle. This is generally regarded as an avoidable practice, because it then forces the programmer to explicitly end the cycle using, as is seen in this case, a *break* condition. *Z* avoids this by simply reading the numbers in the cycle's test and checking if the number is equal to zero. As explained in the previous chapter, detected PMD violations are "punished" in the skill or readability grades. Each violation is related to one of the groups. In this case, both violations punish the skill group.

The parity check was also made differently in the two solutions. While *S* compares with the traditional (and easier to understand) way of *if(n % 2 == 0)*, *Z* used the more advanced approach of *if((n & 1) == 0)*. This is much more efficient than using the % operator, especially for large numbers. The bitwise and bitshift operators allow programmers to perform bit-level operations and have a very high potential to those who know to use them. These operators are considered advanced, so their usage will increase the skill level of a programmer when detected by PP.

Finally, we can see that in the first solution, *S* has to declare one last variable, use another if-condition, and call one final

println method just to compute and output the average of the even numbers. *Z* on the other hand does everything in a single line, using the ternary operator (also know as inline if). All these extra statements used by *S* will have a negative effect on *S's* Skill (or a positive effect on *Z's*). After all, *Z* did the same in less statements. The usage of the ternary if condition is considered an advanced operator that also benefits Skill.

After running these two solutions (together with five other) through the PP tool, all data detailed in IV will be extracted. Then, those individual metric values are normalised across all solutions. Finally we apply the weight to those normalised values and achieve a final score by adding the individual metrics results. That final score is composed by two numbers: one for Skill another for Readability.

Wrapping up the analysis, we see that *Z* shows greater language knowledge and skill, but not much concern for readability. *S* is less skillful and programs in a more novice way. Figure 1 shows the final scores obtained by all seven solutions that were analysed for this particular case study. In these small examples, *S* was classified as *Adv. Beginner* (leftmost on the plot) with a readability focus, obtaining a (S,R) score of (20.9, 15.9), and *Z* was classified as *Expert* (rightmost on the plot) with a score of (32.2, 10.7). This complies to their programming background, which was stated previously.

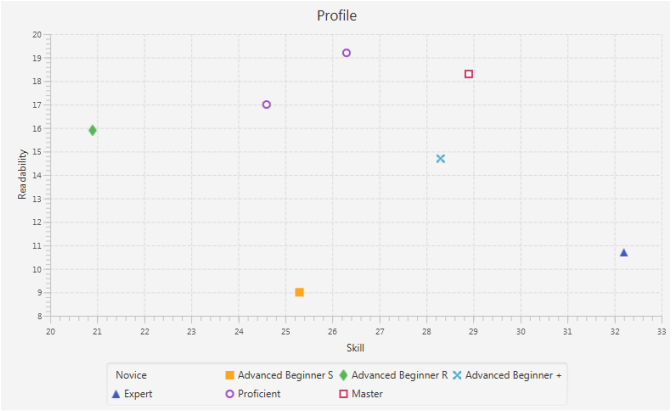

Fig. 1. Profile inference made for Exercise P1

VII. Conclusion

The research hypothesis that led the project here reported was *whether was possible to infer the profile of a programmer through the analysis of his source code*. We proved that research hypothesis by means of demonstration.

The developed tool, Programmer Profiler Tool, takes as input a set of correct solutions to a given programming problem, written in Java, by different programmers.

Each one of the metrics extracted and bad practices identified are linked to one of two groups, Skill and Readability, and can have a positive or negative effect on the two groups. The Skill group is related to language knowledge and ability of creating effective code. The Readability group relates more to understandability of code, and coding style related practices.

By comparing all results among each other, and applying previously defined rules of how the metrics and defects affect the groups, a numeric score is calculated for each group and for each programmer. Each one of these rules, applies the results of an extracted metric (or PMD violation) to either increase or decrease the score of the two groups (S and R), thus reaching a final value for each group.

By applying the described method to several exercises, a set of scores is calculated for each programmer, and by combining those scores a final score is calculated, for each group, that portraits how the programmer performed in comparison to the solutions of other programmers.

The final scores are mapped to a set of previously defined programmer profiles, and thus the profile is inferred for each one of programmers. The results can then displayed in a plot, to better interpret how each programmer performed on the different exercises as well as on the global scope.

All the profiles inferred on the tests performed agreed to the teacher's manual evaluation done in Java course of the University of Minho. Which leads us to state that PP Tool can correctly infer the profile of Java programmers. Although this it would be interesting in the future to include more information about each student namely learning ability and soft skills.

More data and information regarding this project can be found at http://www4.di.uminho.pt/~gepl/PP/ .

Acknowledgment

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

References

[FCJ+04] Thomas Flowers, Curtis Carver, James Jackson, et al. Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H–10. IEEE, 2004.

[HMRM03] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[HTHL14] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 152–161. ACM, 2014.

[JCC05] James Jackson, Michael Cobb, and Curtis Carver. Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*, pages T4C–T4C. IEEE, 2005.

[KCM07] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[PC15] Emília Pietriková and Sergej Chodarev. Profile-driven source code exploration. *Computer Science and Information Systems (FedCSIS), pp. 929-934, IEEE.*, 2015.

[Pos14] Raphael 'kena' Poss. How good are you at programming?—a CEFR-like approach to measure programming proficiency. July 2014.

[TRB04] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 317–325. Australian Computer Society, Inc., 2004.