Scaling up a Programmers' Profile Tool

Martinho Aragão[1]
Algoritmi R.C./Dep. of Informatics, Univ. of Minho, Portugal
martinhoaragao@gmail.com

Maria João Varanda Pereira
Algoritmi R. C./ CeDRI/ DIC/ Polytechnic Inst. of Bragança, Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
Algoritmi R.C./Dep. of Informatics, Univ. of Minho, Portugal
pedrorangelhenriques@gmail.com

Abstract

The style of programming, the proficiency on the programming language, the conciseness of the solution, the use of comments and so on, allow comparison of programmers through static analysis of their code. The Programmer Profiler Tool, which has been commonly named PP Tool, is an open source profiling tool for Java language where the programmer's ability can be classified in one out of five possible profiles and the distinction among them falls upon the levels of both skill and readability. Taking a set of correct solutions the comparison between solutions for the same problems is fundamental to evaluate proficiency on the analysed criteria. As such, there was a need to tune the tool in order to handle, simultaneously, with a bigger amount of programs and with a wider scope of solutions. By scaling up PP Tool it will be possible to apply it in a far wider scope of situations as it will be able to cope with programmers from different geographies, with or without formal education, between 1 and 20 years of experience amongst other factors. For that, a set of features were implemented and tested and are described in this paper.

2012 ACM Subject Classification General and reference → General literature; General and reference

Keywords and phrases Programmers Profiling, Code Analysis, Programming Skills, Code Readability

Digital Object Identifier 10.4230/OASIcs.SLATe.2019.

Acknowledgements 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.

1 Introduction

The PP Tool [6] is based on program analysis and can be applied in educational and professional contexts to compare the proficiency of a set of solutions. The main idea is to profile different programmers by using their solutions to the same problem in terms of bad-practices, ability to master a programming language and code readability (indentation, use of comments, descriptive identifiers). In this work only correct programs producing the desired output were used and the efficiency of the solution is not analysed. A programmer's ability can be classified as one of a set possible profiles and the distinction among them falls upon the levels of both skill and readability that are evaluated based on code metrics. By aiming at proficiency on these criteria one can achieve a more experienced profile.

The basic idea is to statically analyse Java source code and extract a selection of metrics. Some metrics can be directly extracted from source-code and provide a lot of information to understand the programmer proficiency like number of files, classes, methods and statements; number of lines code and comments, and their ratios; usage of control flow statements (if,

[1] Corresponding author

Symposium on Languages, Applications and Technologies (SLATe 2019).
Editors: Hugo Oliveira and Ricardo Rodrigues; Article No. ; pp. :1–:8
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

while, for, etc); variable declarations and datatypes used; usage of advanced Java operators (bitshift, bitwise, etc); usage of repetitive patterns; usage of indentation and identifiers of good quality.

Moreover, it is possible to detect automatically bad-practices using the PMD tool [2]. PMD is a free source code analyser that finds common programming flaws like unused variables or code, empty catch-blocks, unnecessary object creation, poor identifier names, non-optimised code, inappropriate code size and so on.

Based on the metrics described above and the number of violations detected by PMD Tool, values are given to the parameters skill and readability. Skill is defined as the language knowledge and the ability to apply that knowledge in an efficient manner and to measure that the most important metrics are: number of statements; use of control flow statements (if, while, for, etc) and advanced Java operators; number and datatypes used. Readability is defined as the aesthetics and general concerns related with code legibility, so other metrics are taken into account: number of methods, classes and files; total number and ratio of code, comments and empty lines.

The paper is organised as follows. In Section 2 the work done by others on profiling is reviewed and compared to ours. In Section 3 a brief introduction to the main components and techniques of the original PP Tool is presented. Also in this section the original profile classes are characterised, and a refinement of that initial classification is discussed; at last, the metrics used to measure programmers' level of skill and readability, necessary to determine the profile class, are listed. After describing the problems encountered when PP Tool was applied to a big collection of programs gathered from a new source, in Section 4 we enumerate the various and important decisions taken to scale up the tool and cope properly with this kind of program sets. Then Section 5 will contain a detailed discussion on the results attained with the new version of PP Tool to enhance the gains. Section 6 concludes the paper with a summary of the work reported and a mention to the generation of detailed feedback on programmers improvement as a future research direction.

2 Related Work

Before deciding on pursuing improvements to a tool which uses a source code analysis, other alternatives of profiling were explored.

Perhaps the most used way is actually through their experience. Often one of the first steps for companies when recruiting is in the form of a *curriculum vitae*. However, this has been known to be flawed, hence requiring other methods.

One technique which has been growing in popularity employs the use of *gamification*. Particularly one can use the example of code challenge websites where programmers are ranked based on the number and difficulty of the challenges that they have solved. Scoring systems feed leaderboards and these approaches are also evidenced on [2]. However, this feeds on very particular knowledge as it completely disregards efficiency, how long it took to solve the exercise and code legibility as the only information it provides is how many challenges have been solved. It also only capable of profiling users after several exercises, while difficult exercises can take hours to be solved.

In recent years, the surge of software communities has accumulated countless data of their users. *GitHub* tracks number of commits and their information as well as pull requests and even project popularity. *StackOverflow* also tracks number of answers divided by topics

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

and with a voting system on both the answers and the questions. In [4] the CPDScorer is introduced which aggregates the information of the platforms mentioned previously to claim very high precision. However, it once again requires a lot of information and is dependant on popularity.

Pietrikova [7] also explores techniques aiming the evaluation of Java programmers' abilities 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 is the model to follow and refers to the actual knowledge necessary to handle those tasks. This work is also based on metrics whose values are compared with an optimal solution. In PP Tool [5] there is no need to define an optimal solution because it is based on the relative position between a set of solutions.

There are other tools more concern with learning programming. The tool presented In [8], provides two types of analysis: software engineering metrics analysis to look for poor programming practices and logic errors in student programs and structural similarity analysis for comparing students' solutions to a model solution. Flowers et al. present a tool, Gauntlet [1], that allows beginner students understanding their Java syntax errors. It is based on a set of the most common errors for these kind of students and it uses a very friendly and helpful way of displaying those errors. Also concerned with error handling, Expresso tool [3] is a reference on Java syntax, semantic and logic error identification.

3 PP Tool at a glance

PP Tool, whose architecture implementation and tests were described in detail in [6], uses language processing techniques for static analysis and automatically extracts metrics from programs aiming to profile their writers. As was said, this process will be complemented with the use of PMD Tool, to get information on the use of good Java programming practices.

The PP Tool has two key moments for analysis, one for scoring and finally one for profiling. First, on the PP Analysis, metrics are extracted from the source code and stored on specially created class. On the second one, the PMD Analyser is used to identify common programming flaws which are also called violations. During scoring, both of the previously obtained information is transformed to impact in either skill or readability. Finally, all the solutions are provided profiles based on the comparison between their scores.

3.1 Code analysis

For each set of metrics a class with the purpose of extracting those metrics was created.

These metrics can be customised on an auxiliary file such as whether they have a positive or negative effect to skill or readability, or even the weight of the impact.

The PMD Analyser has a set of rules which can also be customised. Currently the quickstart set is used which provides a general list of rules which are valid for most situations. However if the PP Tool is to be applied on a controlled environment then it is recommended to set its own list of rules.

Each rule has a priority associated with the penalty to be inflicted. When running the analyser, rule violations are registered with information regarding the line where they occurred. Violations are then summed up based on number of occurrences and the priority to inflict a penalty.

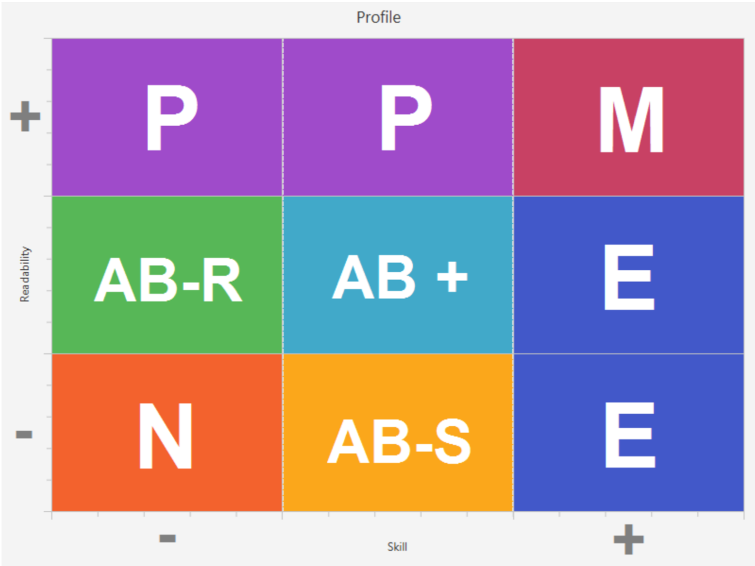

Figure 1 Profiling Distribution

3.2 Profiling

There are 4 main profiles. The novice profile (N) identifies a programmer that is not yet familiar with all the language constructs and usually does not show language readability or good programming practices concerns. The advanced beginner (AB) programmer shows variety in the use of language constructs and data-structures, starts showing some readability concerns but still writes programs in a safely manner. The proficient programmer is familiar with a great variety of language constructs, usually follows good programming practices, has readability and code-quality concerns. The expert programmer masters a great variety of language constructs and is focuses on producing efficient code usually without readability concerns.

As time progressed, the profiles shifted a bit from the original idea. The Experts should be the ones with maximum focus on Skill, the Proficients on Readability, the Advanced Beginners were divided in three subsets and a new profile called Master was created to be associated to a high level of skill and readability.

So the profiles used in this work are 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.

Profiling is the last step of the tool. A grid is created with the lowest and highest values of skill and readability in mind, and all results are distributed in the grid. The grid is divided in 9 blocks of equal size as can be seen at Figure 1.

4 Scaling Up

When testing the scalability of the tool by using a big amount of programs, it lead to a great variety of results that are semantically different from the ones got from the analysis of a small amount of programs. One of the problems was the lack of distinction between solutions. Although each metric has different impact it was common to find very different solutions that had practically the same readability and skill results. It was concluded that several metrics should be better calculated taking into account, for instance, the priority and the number of occurrences.

Two important decisions were made:

- Refine some criteria, rules and values (to cope with a bigger variety of solutions):
  - use only the percentage of blank lines and comment lines instead of also their absolute values;
  - introduction of the notion of criterion weight;
  - increase variety of violations;
  - the profile is always based on solution comparison but "isolated" solutions (very very good or very very bad) must have lower impact on the results;
  - assign weight and number of occurrences to each violation in order to tune the influence of it in skill and readability;
  - change violations impact to be proportional to readability and skill score to remove negative values due to "isolated" solutions;
  - adjust the number and the impact of each metric in order to balance both skill and readability results.
- Improve PMD performance (to cope with a bigger amount of solutions):
  - introduce a new caching option that speed up the tool;
  - turn easier the system maintenance associating the impact attribute to each group of violation rules and not to each violation individually;
  - the violations belonging to the same type are grouped and it is much more easier to associate each group to the factors skill and readability;

All of these changes lead to a more robust system that could handle the new multitude of scenarios. The scoring system changed considerably, as metrics became the only source of positive score, and violations the only source of negative ones. PMD violations now can provoke up to 50% penalty in a given score (if the solution is the one with the most severe penalties) and metrics no longer reduce score.

5 Testing the tool

In order to ensure the Programmer Profiler Tool was ready to be used in a more generic environment, we needed to test it with a far more diverse input of exercises. As such, instead of requesting more exercises from a classroom we looked into platforms which provided hundreds of challenges and solutions. In that search, online programming exercise platforms came up as an ideal solution. These type of platforms have several years worth of exercise solutions from all experience levels and with users across the globe. Other services are often either tailored for specific use cases such *Stack Overflow* with just code bits or there is great difficulty in comparing solutions for profiling which is the context of whole Open Source projects like found in *Github*.

By request CodeChef, a not-for-profit educational initiative, supplied the solutions.

In order to test the results of the changes, an exercise of medium difficulty has been chosen. Specifically we will be looking at the following solutions: solution A, solution B, solution C.

The Figure 2 represents the distribution at that stage of all 300 solutions. It's clearly visible that almost all solutions are profiled as "Experts". With the average skill being higher than the average readability, which seems consistent with the programming challenges environment expectations. However, the distribution is also very tight with several points

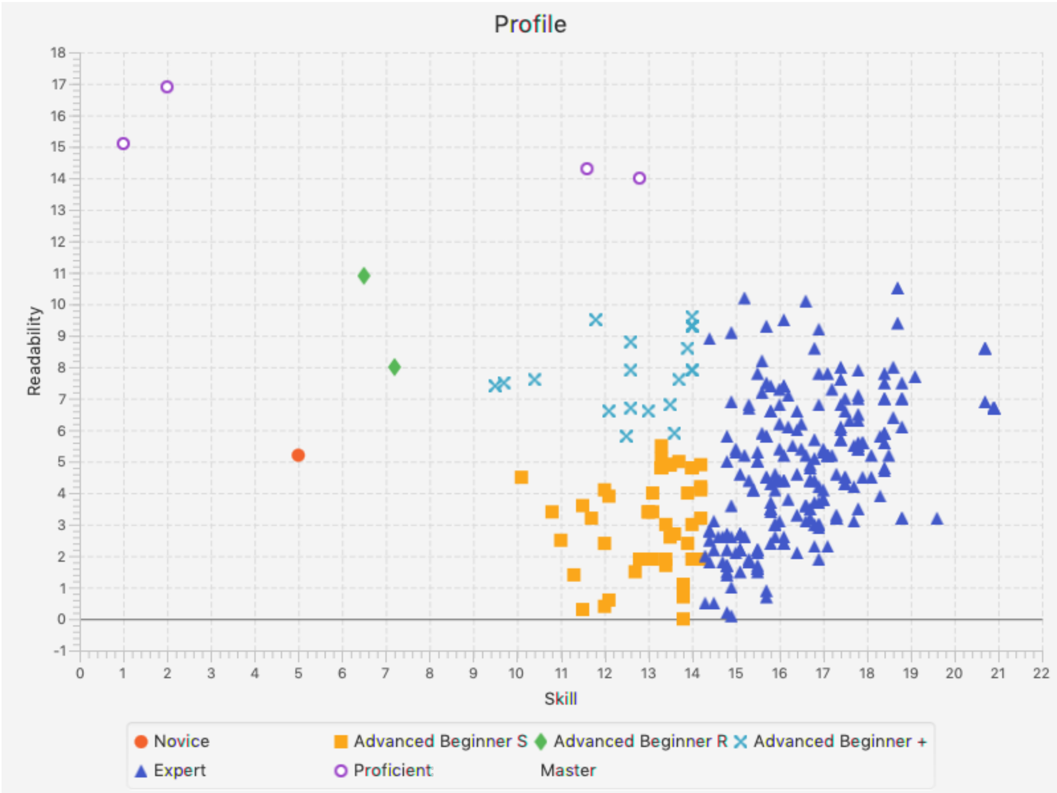

Figure 2 Distribution of solutions without scaling changes

practically on top of one another. Solution A was profiled as Proficient while B and C were as Experts.

On the Figure 3 the graph shows the final distribution after all the changes explained on the previous subsections. Now most of the profiles are considered Advanced Beginner S. There is still a larger influence on the skill score, but the distribution is slightly more spread about. Solution A was profiled as Expert, solution B as Advanced Beginner + while C as Proficient.

To summarise the results of some of these changes Table 1 can be viewed. Only some of the key metrics have been listed. Solution A had been profiled as "Proficient", this is a profile leaning towards more readability than skill, however it has: The least number of skill penalties; The smallest number of statements; Far less total lines, almost a 1 to 10 factor compared to solution B; Just 2 methods and 1 class; Quite a few readability penalties and no comment lines.

By looking at these factors it's obvious the solution A leans towards skill instead of readability. In fact, we can make a direct contrast to solution C, in fact they swapped profiles. Solution C leans towards readability while keeping a good skill score, some of the factors for comparison with solution A: One skill penalty; Three more classes; Four time more the number of lines of code and of statements; 2.7 percent of lines of comment; Just 2 methods and 1 class; Quite a few readability penalties.

Finally, solution B clearly is too long compared to the others, with the most penalties and no good points in its favour. However, it doesn't necessarily lean more towards either skill or readability, hence the profile given is "Advanced Beginner +".

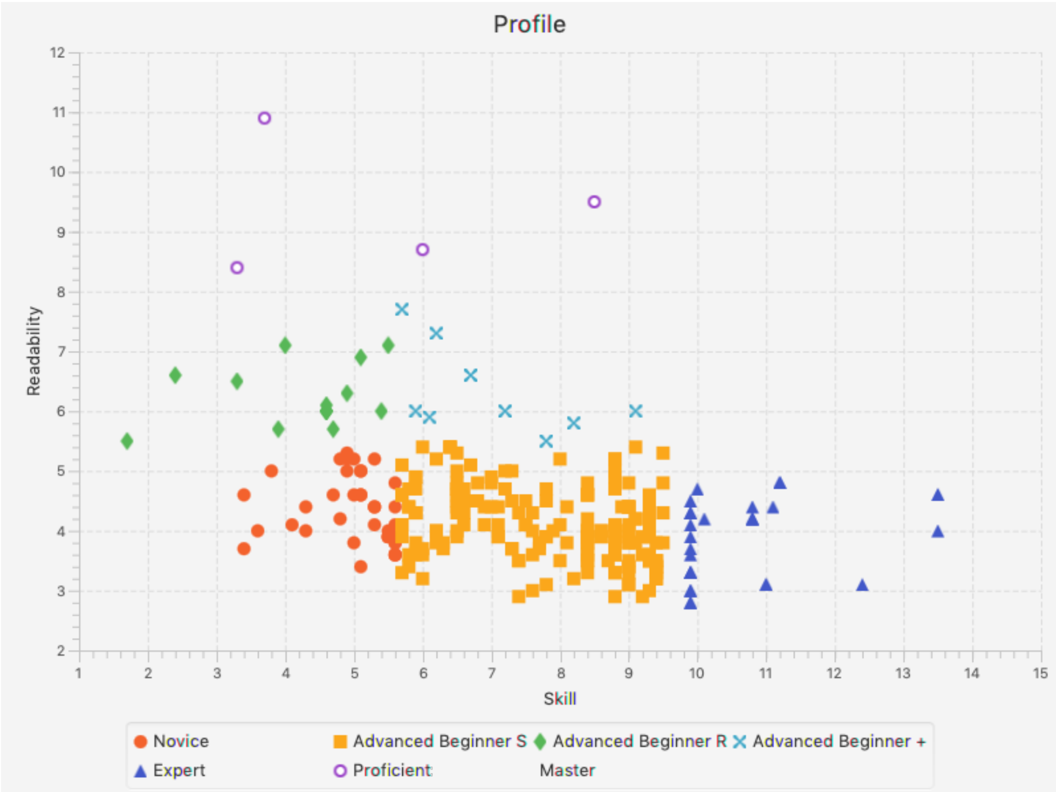

Figure 3 Final distribution of solutions

Finally, and just from a programmer's direct point of view, there are some things that are easily noticeable and also serve as a validation of the adjustments made.

Solution C is clearly the most readable, it has good descriptions, spacing, more classes and methods. Solution A, was able to solve the exercise in simply 25 lines of code, and one of the smallest number of statements. On the other hand Solution B is very long, it is more complex than necessary compared to other alternatives, it seems more the work of a beginner.

To conclude, the comparison between the images shows that with this new version of PP Tool the results are more distributed across the chart.

6 Conclusion

We can anticipate several situations where it is necessary to carry out programmer's profiling: programming contests, contracting of new programmers, evaluation of programming students, analysis of source code quality for some purpose and so on. As we presented in this paper, it's possible to extract important information from the static analysis of source code in order to obtain values for parameters like skill and readability and following that approach, PP Tool infers the programmer's profile. This profile varies from novice to master passing through advanced beginner, proficient and expert. PP Tool was tested in a different more demanding environment and it did not scale up conveniently. So we extended it with some new features to obtain a finer and more efficient metrics evaluation method (also weights were tuned) in order to cope with a bigger diversity of solutions for more complex problems. Some tests

	Solution A	Solution B	Solution C
Skill PMD Penalty	0	1	1
Readability PMD Penalty	7	14	8
# Classes	1	2	3
# Methods	2	18	6
# Statements	4	60	17
Lines of Code	13	99	52
Percentage of Comment	0	2.3%	2.7%
Total Lines	26	214	73
# Declarations	4	16	10
Profile - Before	Proficient	Expert	Expert
Profile - After	Expert	Advanced Beginner +	Proficient

Table 1 Comparison 3 solutions before and after the PP Tool scaling adjustments

were made, as discussed in this paper, showing that the accuracy of the new version of our programmer's profiling tool was actually improved. The direction for future research will include the generation of detailed feedback on programmers performance based on the bad practices detected. The idea is to open the possibility to use PP Tool not only for profiling but as a recommendation tool that will contribute to improve the quality of programmer's code specially for students that are learning their first programming language.

References

1. T Flowers, Curtis Carver, and J Jackson. Empowering students and building confidence in novice programmers through gauntlet. pages T3H/10 – T3H/13 Vol. 1, 11 2004.
2. Markus Fuchs and Christian Wolff. Improving programming education through gameful, formative feedback. *2016 IEEE Global Engineering Education Conference (EDUCON)*, pages 860–867, 2016.
3. 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.
4. Weizhi Huang, Wenkai Mo, Beijun Shen, Yu Yang, and Ning Li. Automatically modeling developer programming ability and interest across software communities. *International Journal of Software Engineering and Knowledge Engineering*, 26(09n10):1493–1510, 2016. URL: `http://www.worldscientific.com/doi/abs/10.1142/S0218194016400143`, `arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0218194016400143`, `doi:10.1142/S0218194016400143`.
5. Daniel Novais, Maria Joao Varanda Pereira, and Pedro Rangel Henriques. Program analysis for Clustering Programmers' Profile. In Ganzha, M and Maciaszek, L and Paprzycki, M, editor, *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 701–705. PTI; IEEE, 2017. FedCSIS, Prague, Czech Republic, Sep 03-06, 2017. `doi:{10.15439/2017F147}`.
6. Daniel José Ferreira Novais. Programmer profiling through code analysis. Master's thesis, December 2016.
7. Emília Pietriková and Sergej Chodarev. Profile-driven source code exploration. *Computer Science and Information Systems (FedCSIS), pp. 929-934, IEEE.*, 2015.
8. 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.