Profile Detection Through Source Code Static Analysis

Daniel Ferreira Novais[1], Maria João Varanda Pereira[2], and Pedro Rangel Henriques[3]

1 **Departamento de Informática & Centro Algoritmi, Universidade do Minho, Braga, Portugal**
danielnovais92@gmail.com
2 **Departamento de Informática e Comunicações & Centro Algoritmi, Instituto Politécnico de Bragança, Bragança, Portugal**
mjoao@ipb.pt
3 **Departamento de Informática & Centro Algoritmi, Universidade do Minho, Braga, Portugal**
pedrorangelhenriques@gmail.com

Abstract

The present article reflects the progress of an ongoing master's dissertation on language engineering. The main goal of the work here described, is to infer a programmer's profile through the analysis of his source code. After such analysis the programmer shall be placed on a scale that characterizes him on his language abilities. There are several potential applications for such profiling, namely, the evaluation of a programmer's skills and proficiency on a given language or the continuous evaluation of a student's progress on a programming course. Throughout the course of this project and as a proof of concept, a tool that allows the automatic profiling of a Java programmer is under development. This tool is also introduced in the paper and its preliminary outcomes are discussed.

1998 ACM Subject Classification D.2.8 Metrics, D.3.4 Processors

Keywords and phrases Static analysis, metrics, programmer profiling

Digital Object Identifier 10.4230/OASIcs.SLATE.2016.9

1 Introduction

Proficiency on a programming language can be compared to proficiency on a natural language [7]. 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. Similarly, it may be possible to create a set of metrics and techniques that allow the profiling of programmers based both on proficiency and abilities on a programming language.

In [6], the main inspiration behind this project, Pietrikova 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. It's usually performed with the goal of finding bugs or ensure conformance to coding guidelines.

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

5th Symposium on Languages, Applications and Technologies (SLATE'16).
Editors: Marjan Mernik, José Paulo Leal, and Hugo Gonçalo Oliveira; Article No. 9; pp. 9:1–9:13

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For the present paper, static analysis will be used to extract metrics from source-code related with language usage practices.

Building on the referred paper, 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.

The basic idea is to statically analyse a Java programmer's source code and extract a selection of metrics that can either be compared to *standard solutions* (considered *ideal* by the one willing to obtain the profiles) or, using machine learning techniques, subjected to a classification model in order to be assigned the appropriate profile. The attributes or metrics that will allow us to infer a profile based on sets of previously classified programs can be defined a-priori by hand (using intuition) or can be extracted through data-mining techniques, as Kagdi et al. explored [5]. However this last approach requires the availability of huge collections of programs assigned to each class.

The programmers will be classified generically as for their language proficiency or skill, for example, as novice, advanced or expert. Other relevant details are also expected to be provided, such as the classification of a programmer on his code readability (indentation, use of comments, descriptive identifiers), defensive programming, among others.

Below are some source-code elements that can be analysed to extract the relevant metrics to appraise the code writer's proficiency:

- Statements and Declarations
- Repetitive patterns
- Lines (code lines, empty lines, comment lines)
- Indentation
- Identifiers
- Good practices

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.

To build the system discussed in this paper we intend to develop a metric extractor program, to evaluate the set of parameters that we chose for the profiling process. However this process will be complemented with the use of a tool, called PMD[2], to get information of the use of good Java programming practices. PMD is a source code analyser that finds common programming flaws like unused variables, empty catch-blocks, unnecessary object creation, and so forth. For these reasons it is a tool that may prove to be very useful.

The rest of the paper is organized as follows. In Section 2 we will review the area and present related work, in order to identify techniques and tools commonly used to deal with this problem. Section 3 is devoted to present our proposal for an automatic programmer profiling system based on source code analysis. The analyzer implemented and the set of metrics extracted at the present project stage will be introduced in Section 4. In Section 5 we will discuss the profiling results so far inferred correlating the values provided by the comparison between the programmer's code metrics and the standard solution. The paper is closed at Section 6 with some conclusions and future work.

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

2 Programmer Profiling: approaches and tools

As mentioned before, the main motivation for this project came from the study [6] 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. The authors agree that the tool is promising, but there is still a lot of work that can be done on the subject. To compare programs against models or ideal solutions, by counting language constructs is a common feature between this work and our project. Despite that, in this work, the object profile is optional. The subject profile can be inferred analysing the source code, using as base the language grammar. Considering the language syntax, a set of metrics are extracted from the source code. This can be done to conclude about the complexity of a program or to perform some statistics when analysing a set of programs of one programmer. In our case, we are not concerned with the complexity level of the programs but we analyse the way each programmer solves a concrete problem. So, almost all metrics that we extract only make sense when compared with a standard solution.

In another paper [9], 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). Despite having several limitations, teachers have been giving this tool a positive feedback. As stated before, this thesis will be taking a similar approach to this software engineering metrics. However, the tool above mentioned was only used on an academic context while the purpose of this project is to develop a tool that can also be applied in another contexts.

Flowers et al. [1] and Jackson et al. [4] discuss a tool developed by them, *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* [3] 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 since they focus mainly on error handling, they will not be very useful for this project.

Hanam et al. explain [2] 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). This study may prove to be very useful to this work since there is an intention of exploring similar machine learning and data mining techniques on the analysis.

3 Programmer Profiling: Our proposal

3.1 Programmer Profiles

Programmer profiling is an attempt to place a programmer on a scale by inferring his profile. As Poss stated [7], we can compare proficiency on a programming language with proficiency on a natural language, and like the CEFR has a method of classifying individuals based on their proficiency on a given foreign language, it is believed that the same can be done for a programming language.

CEFR defines foreign language proficiency at six levels: A1, A2, B1, B2, C1 and C2 (A1 meaning the least proficient and C2 the most proficient). A similar method for classifying programmers was considered at first, but due to the fact that the levels were not very descriptive, a more self-described scale was preferred.

Sutcliffe presents [8] a classification for programmer categorization: naive, novice, skilled and expert. A similar scale was agreed upon, with what it is believed to be a good starting point for the profiling:

Novice
- Is not familiar with all the language constructs
- Does not show language readability concerns
- Does not follow *good programming practices*

Advanced Beginner
- Shows variety in the use of language constructs and data-structures
- Begins to show readability concerns
- Writes programs in a safely[3] manner

Proficient
- Is familiar with a great variety of language constructs
- Follows *good programming practices*
- Shows readability and code-quality concerns

Expert
- Masters a great variety of language constructs
- Focuses on producing efficient code (minimizing resources or lines of code) without readability concerns

The following (a little bit exaggerated) example (see Listing 1) may help to shed some light on what is meant by the previous scale. Each method has the same goal: to calculate the sum of the values of an integer array, and has features of what may be expected for each class. It's hard to represent all 4 classifications on such a small example, so the Advanced Beginner was left out.

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

[3] e.g. writes `if (cond==0)` instead of `if (!cond)` as is done by people that have more self-confidence and usually have a not so obvious way of programming.

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;
}
```

of bad readability. The code can be well organized but the programming style is usually more compact and it's harder to understand. As an example of language knowledge the expert uses the *extended for loop*, making his code a lot smaller.

Finally, the proficient will show the skill and knowledge of an 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.

3.2 System architecture

As mentioned previously, this project will be complemented with a tool, developed in Java, that intends to put into action the theory behind the project.

Figure 1 shows the block diagram that represents the expected final implementation of the system. The tool will be named *Programmer Profiler* (PP).

The programmer's Java source code is loaded as PP input. Then, the code goes through two static analysis processes: the analyser implemented (PP-Analyser) using AnTLR with the goal of extracting a set of metrics and the PMD-Analyser, an analyser that resorts to the PMD Tool to find a set of predefined metrics regarding poor coding practices.

Both analysis' outcomes will feed two other modules: A *Metrics Visualizer* (a generator of HTML pages [4]) which will allow us to make a manual assessment of the source code to infer the programmer's profile; and a *Profile Inference Engine* whose goal is to make the profile assignment an automatic process.

[4] `http://www4.di.uminho.pt/~gepl/PP/`

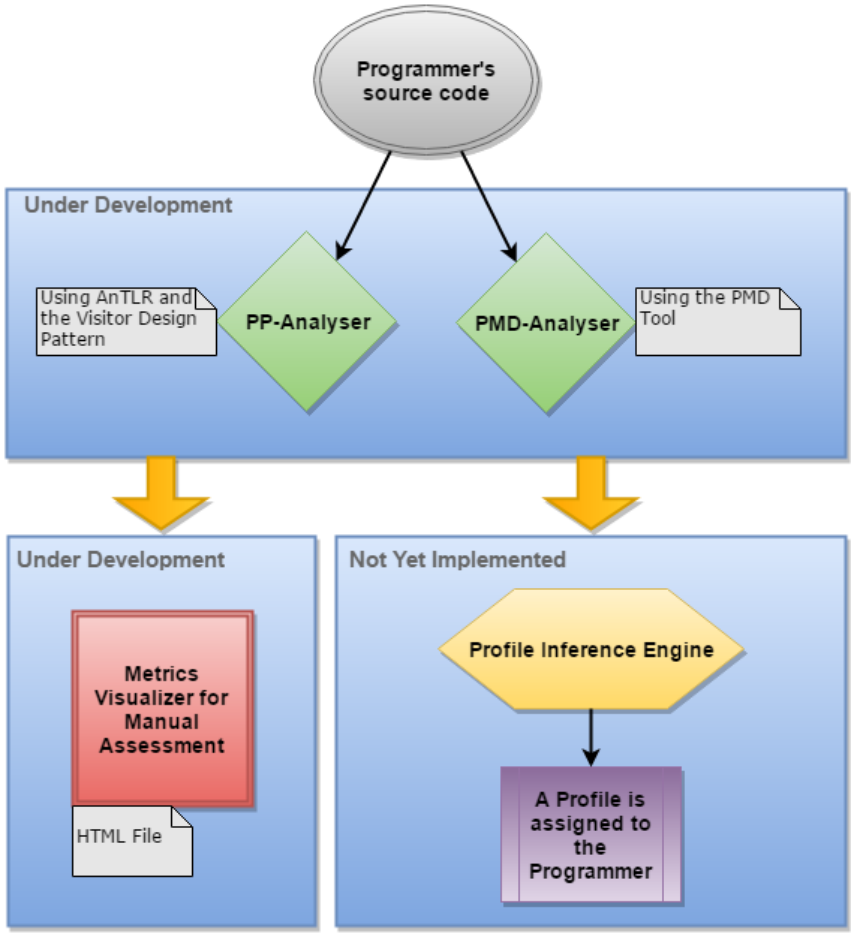

Figure 1 PP Block Diagram.

Making the profiling an automatic assignment will be the most interesting, challenging and complex part of this project. The goal is not to assign an absolute value that characterizes a programmer's proficiency on the Java language, but instead to give a general classification in regards to a resolution of a given problem or task.

3.3 Tools being used

To implement PP some tools were very helpful throughout the development process. Below we describe the two most relevant: AnTLR and PMD.

3.3.1 AnTLR

As taken from the website[5]:

> ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text (...). From a grammar, ANTLR generates a parser that can build and walk parse trees.

[5] `http://www.antlr.org/`

Using only a Java grammar, AnTLR will generate a parser that will read any syntactically correct Java source file. This allows us to easily manipulate the information that a source file contains. This means that AnTLR allows us to extract, with more or less ease, any kind of metric needed from the Java files.

As mentioned on the previous chapter, instead of implementing features that search for poor Java practices and novice programming mistakes, PMD, a very useful source code analyser, was selected.

3.3.2 PMD

PMD[6] is a source code analyser. It finds common programming flaws like unused variables, empty catch-blocks, unnecessary object creation, and so forth.

PMD executes a thorough analysis over source-code (it supports several languages) and reports back the possible programming flaws in the form of violations. PMD looks for dozens of poor programming practices, nonconformity to conventions and security guidelines, being a promising asset to this project.

Below are some of the main PMD Rulesets[7] that may be the most useful to our goals:

Unused Code
The Unused Code Ruleset contains a collection of rules that find unused code

Optimization
These rules deal with different optimizations that generally apply to performance best practices

Basic
The Basic Ruleset contains a collection of good practices which everyone should follow

Design
The Design Ruleset contains a collection of rules that find questionable designs

Code Size
The Code Size Ruleset contains a collection of rules that find code size related problems

Naming
The Naming Ruleset contains a collection of rules about names - too long, too short, and so forth

Braces
The Braces Ruleset contains a collection of braces rules

4 Metrics extraction: source code analysis

As mentioned before the *Programmer Profiler* (PP) tool will consist of two performed analysis. An implemented PP-Analyser, which resorts to metrics extraction, with the goal of comparing with model solutions and a PMD-Analyser, which uses the PMD tool to detect poor programming practices in an absolute manner (not resorting to comparison).

[6] `https://pmd.github.io/`
[7] `https://pmd.github.io/pmd-5.4.1/pmd-java/rules/index.html`

Listing 2 Violation detected by PMD.

```
<violation beginline="274" endline="276" begincolumn="33"
endcolumn="33" rule="CollapsibleIfStatements" ruleset="Basic"
package="(...)" class="IMC" method="MT"
externalInfoUrl="https://pmd.github.io/pmd-5.4.0/pmd-java/rules/
java/basic.html#CollapsibleIfStatements" priority="3">
These nested if statements could be combined</violation>
```

4.1 PP-Analyser

The PP-Analyser, which extracts metrics that allow comparing possible solutions with optimal ones was implemented using AnTLRv4 and so far a variety of metrics is being extracted:

1. Class hierarchy
2. Class and method names and sizes
3. Variable names and types
4. Number of files, classes, methods, statements
5. Number of lines code and comment
6. Control Flow Statements (if, while, for, etc)
7. Advanced Java Operators (Bitshift, Bitwise, etc)
8. Other relevant Java Constructs

Metrics 1, 3, 4, 6 and 7 can be used to compare a given solution of a problem to an optimal solution, and that way know if a solution was that of a programmer with more or less expertise in Java. Metrics 2, 3 and 5 can be use to find concerns with code understanding and readability.

4.2 PMD-Analyser

As mentioned before, PMD is a source-code analyser that looks for poor practices usually adopted by beginner programmers (e.g. several returns in one method or leaving empty catch-blocks).

To illustrate PMD behaviour, listing 2 below shows an example of a violation detected and reported by the tool.

Each violation found contains a good amount of information about the violation itself and where it was found. A large-sized project, with a lot of rulesets being examined, could return hundreds of violations which may prove very helpful in the profiling of programmers.

5 Correlating metrics with profiles

Correlating metrics with profiles has proved to be a challenging task. After much consideration, we came up with a proposal, presented below, that we think to be as accurate as possible.

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 and the ability to apply that knowledge in a efficient manner.
- *Readability* is defined as the aesthetics, readability and general concern with understandability of code.

Table 1 Proposed correlation.

Profile	*Skill*	*Readability*
Novice	−	−
Advanced Beginner	−	+
Expert	+	−
Proficient	+	+

Of the metrics extracted, some show a tendency towards classifying Skill while others towards classifying Readability. Here's a breakdown of where each metric may fall:

Skill

- Number of statements
- Control flow statements (If, While, For, etc)
- Advanced Java Operators
- Number and datatypes used
- Some PMD Violations (e.g. Optimization, Design and Controversial rulesets)

Readability

- Number of methods, classes and files
- Total number and ratio of code, comments and empty lines
- Some PMD Violations (e.g. Basic, Code Size and Braces rulesets)

These two groups contain enough information to obtain a profile of a programmer, regarding a given task. Then, for each group, and according to the score obtained by the programmer, Table 1 gives a general idea of how programmers can be profiled. (+) means a positive score, while (−) means a negative one.

What constitutes a lower and a higher score on each group must be defined. For every programmer, the goal is to compare each metric's value to the *standard solution*, which is by default considered a proficient solution (high skill and readability), and then, assemble a mathematical formula which allows to combine the metrics' results into a grade for each one of the two groups. With those two grades and resorting to Table 1 we can easily infer the programmer's profile in regards to the subject problem.

The exercise: "Read a given number of integers to an array, count how many are even", as proposed to two programmers. A Java and OOP teacher and an advanced Java programmer (master student). Listings 3 and 4 show both solutions.

After running both solution through the PP-Analyser we get the results shown in Table 2.

In Table 3 we compare the metrics of the obtained solution with the ones of the standard solution. In this table, for each metric analysed, the programmer gets 1, 0 or -1 whether his value on that metric is better, the same level or worst when compared to the standard solution, respectively. In this case the programmer got +3 points in skill -6 in readability when comparing to a proficient solution, making him an expert according to Table 1. As a general rule of thumb, for the readability group, more is better. Of course the score was obtained in a very naive way. As mentioned previously a mathematical formula which takes into consideration the importance of the metrics is expected to be developed to make this classification as precise a possible.

Another problem that is yet to be tackled is how to automatically compare some complex metrics like control flow statements and variable declarations, but we already know that it will be important to classify *CFS* and the datatypes as common or not so common in order to evaluate the programming language knowledge level.

Listing 3 Teacher Solution.

```
package ex1_arrays;

import java.util.Scanner;

/**
 * Escreva um algoritmo que leia e mostre um vetor de n elementos
 * inteiros e mostre quantos valores pares existem no vector.
 *
 * @author Paula
 */
public class Ex1_Arrays {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        int cont = 0, N;

        N = in.nextInt();

        int vec[] = new int[N];

        for (int i = 0; i < N; i++) {
            vec[i] = in.nextInt();
        }

        for (int i = 0; i < N; i++) {
            if (vec[i] % 2 == 0) {
                cont = cont + 1;
            }
        }

        System.out.println(cont);
    }

}
```

The goal for the *Programmer Profiler*, and especially the *Profile Inference Engine* is to be able to automatically make that classification and that way infer the profile of the programmer.

The (alpha version) PP-Analyser has already been applied on source-code developed by programmers on different levels of Java proficiency to start acquiring the values (metrics) that characterise the profiles. The code analysed was of moderate diversity, ranging in size and programming background (teachers, students and professional programmers).

6 Conclusion

Along this article, it was presented a proposal to develop a system (called Programmer Profiler) that allows to profile a programmer through the analysis of his source code. The hypothesis is that such profile inference is possible.

Listing 4 Advanced Programmer Solution.

```
import java.util.Scanner;
public class Even {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] numbers = new int[n];
        int result = 0;
        for (int i = 0; i < n; i++) {
            int input = in.nextInt();
            numbers[i] = input;
            result += (input & 1) == 0 ? 1 : 0;
        }

        System.out.println(result);
    }
}
```

Table 2 PP-Analysis of two solutions.

Metric	*Teacher*	*Expert*
Total Number Of Files	1	1
Number Of Classes	1	1
Number Of Methods	1	1
Number Of Statements	6	3
Lines of Code	17 (48,6%)	12 (75%)
Lines of Comment	3 (8.3%)	0
Empty Lines	10 (28.6%)	1 (6.3%)
Total Number Of Lines	35	16
Control Flow Statements	{FOR=2, IF=1}	{FOR=1, IIF=1}
Not So Common CFSs	0	1
Variety of CFSs	2	2
Number of CFSs	3	2
Not So Common Operators	{}	{BIT_AND=1}
Number of NSCOs	0	1
Variable Declarations	{Scanner=1, int[]=1, int=4}	{Scanner=1, int[]=1, int=4}
Number Of Declarations	6	6
Number Of Types	3	3
Relevant Expressions	{SYSOUT=1}	{SYSOUT=1}

Until now, the main contributions of this work consist in: defining a set of possible profiles and their main characteristics; constructing the architecture of the system and the used tools; and performing experiments that allowed us to manually profile a programmer.

Currently, there is a working implementation that can be used to visualize [8] extracted metrics, both by the implemented PP-Analyser and the PMD Tool. That generated data is also being properly stored.

[8] http://www4.di.uminho.pt/~gepl/PP/

Table 3 Comparing to standard solution.

Metric Name	*Skill*	*Readability*
Number Of Files	X	0
Number Of Classes	X	0
Number Of Methods	X	0
Number Of Statements	+1	X
Number Of Lines of Code	X	−1
% Code	X	−1
Number Of Lines of Comment	X	−1
% Comment	X	−1
Number Of Empty Lines	X	−1
% Empty	X	0
Total Number Of Lines	X	−1
Control Flow Statements	+1	X
Variable Declaration	0	X
Total Number Of Declarations	0	X
Total Number Of Types	0	X
Advanced Operators	+1	X
PMD	N/A	N/A
Total	*+3*	*−6*

a) (+1) – better than the standard solution
b) (0) – same level as the standard solution
c) (−1) – worst than the standard solution
d) (X) – metric no related to this group
e) PMD results were not considered in this example

Some manual assessments are already being made with the objective of finding patterns and correlations that will make the PP a fully automatic tool.

We intend to go on conducting more and more experimental case studies to extract as much data as possible to refine the conclusions so far attained to improve our inference process aiming at finding a set of rules to automatically profile programmers.

References

1 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.

2 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.

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 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.

5 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.

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

7 Raphael 'kena' Poss. How good are you at programming? – a CEFR-like approach to measure programming proficiency, July 2014. URL: `http://science.raphael.poss.name/programming-levels.html`.

8 Alistair Sutcliffe. *Human-computer interface design*. Springer, 2013.

9 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.