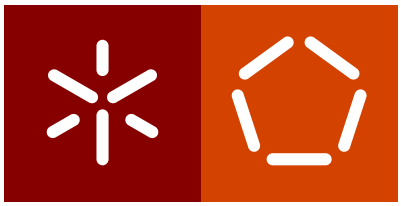
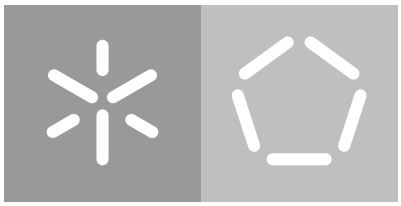


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel José Ferreira Novais

Programmer Profiling through Code Analysis

December 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel José Ferreira Novais

Programmer Profiling through Code Analysis

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Pedro Rangel Henriques
Professor Maria João Varanda

December 2016

AGRADECIMENTOS

Um agradecimento:

- Aos meus orientadores, Pedro Rangel Henriques e Maria João Varanda, pelo acompanhamento contínuo ao longo deste ano, por uma orientação inigualável, e por todas as contribuições que permitiram a realização deste projeto.
- À professora Paula, ao professor Creissac, aos seus alunos e a todos os colegas que dispensaram do seu tempo para contribuir para este projeto.
- Aos meus pais e amigos, por tornarem isto possível.

ABSTRACT

This document serves as a Master's dissertation on a degree in Software Engineering, in the area of Language Engineering.

The main goal of this work is to infer the profile of a programmer, 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 should be developed.

RESUMO

Este documento refere-se a uma dissertação do Mestrado em Engenharia Informática, na área da Engenharia de Linguagens.

O principal objetivo desta dissertação é inferir o perfil de um programador através da análise do seu código fonte. Após a análise, o programador será automaticamente colocado numa escala que o caracteriza quanto às suas capacidades na linguagem.

Existem várias potenciais aplicações para a perfilagem de programadores, por exemplo, avaliar as capacidades e proficiência de um programador numa dada linguagem ou, a avaliação continua de alunos numa disciplina de programação. Como prova de conceito, é esperada a implementação de uma ferramenta que permita perfilar automaticamente programadores.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

1.1 INTRODUCTION

Proficiency on a programming language can be compared to proficiency on a natural language (Poss, 2014). 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.

The motivation to explore this rather recent topic is to be able to classify programmers knowledge on a given language, regardless of matters like time required to solve a problem or actual performance of the program (runtime). This has several real-life applications, for instance, classify students on their language proficiency to help teachers grade them or just generally find out how advance a class is. It could be also useful to discover where a given student is lacking knowledge on, and that way create a more personalized teaching experience. Another area that has potential for this topic is the business fields. Employers could apply these techniques to help select candidates or just generally evaluate their employees' knowledge.

The inspiration behind this dissertation came mainly from the paper *Profile-driven Source Code Exploration* (Pietriková and Chodarev, 2015), which 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 and vulnerabilities, or ensure conformance to coding guidelines. For the present thesis, static analysis will be used to extract metrics from source-code related with language usage practices.

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

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.

There are several approaches that could be taken towards solving this profiling problem:

- Statically analyse a Java programmer's source code and extract a selection of metrics that can either be compared to a *standard solution* (considered *ideal* by the one willing to obtain the profiles) as Pietriková and Chodarev (2015) explored;
- Analyse several solutions to the same problem made by different programmers, and extract the same selection of metrics described above, that can then be compared among themselves and infer who performed better on each metric and globally;
- Use machine learning techniques, subjected to a classification model in order to assign the appropriate profile. In this approach, the attributes or metrics that will allow us to infer a profile based on sets of previously classified programs can be extracted through data-mining techniques, as Kagdi et al. (2007) explored.

The first approach will always require the existence of an *ideal* solution to compare to. This would only allow to look at the profiles through one point of view. There may be more than one correct way to solve a given problem and this approach would only consider one solution as the *correct* one. The third approach requires the availability of huge collections of programs assigned to each class. The second approach, comparison of several solutions among themselves, should be the preferred approach because a *standard* solution is not required and it's much better to compare a set of programmers on a given context (e.g. classroom or job interview).

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. That is, only correct programs producing the desired output should be used for profiling.

To build the system previously discussed 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], that extracts 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.

This topic is challenging and relevant, however it is very recent and requires much more research in order to be possible to produce accurate results.

1.2 OBJECTIVES

This master thesis has the following objectives:

- to study static analysis concepts and how to adapt them to extract important features to characterize a programmer;
- to discuss and define possible programmer's profiles, to set up some metrics adequate to characterize these profiles defining the respective values range;
- to implement a tool based on source code analysis to evaluate the metrics selected in order to infer the profile, testing it exhaustively;
- to explore other approaches to improve the tool effectiveness.

As a final result, it is expected to have a program that can be easily used by software engineers.

1.3 RESEARCH HYPOTHESIS

The research hypothesis is that it is possible to infer the profile of a programmer applying source code analysis techniques.

2 https://pmd.github.io/

1.4 DOCUMENT STRUCTURE

This document is divided in six chapters.

Chapter 1, *Introduction*, gives some motivation on the subject, explains succinctly what is the goal of the thesis and how it may be achieved.

Chapter 2, *Approaches and Tools*, explores the current state of the art of the subject, as well as how previous associated work relates to this thesis.

Chapter 3, *Challenges and our Proposal*, discusses in detail what are the challenges that this dissertation will face and the proposed solutions.

Chapter 4, *Development Decisions and Implementation*, focuses on the implementation aspect, namely what were the decisions made on the beginning of the development phase as well as a detailed explanation on the reasoning behind the full process.

Chapter 5, *Case Studies and Experiments*, describes all the experiments done and tries to justify all results obtained using common knowledge on this subject.

Finally, Chapter 6 gives some conclusions on the work done so far, and lays the foundation of possible future work that could expand this first experiment even further.

2

PROGRAMMER PROFILING: APPROACHES AND TOOLS

The present chapter will focus mainly on studies directly related to the subject of this project. Other researched work on the field is explored in other chapters of this report.

As previously mentioned, the main motivation for this dissertation came from the study of Pietriková and Chodarev (2015). 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, Truong et al. (2004) 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 mentioned above 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. (2004) and Jackson et al. (2005) discuss a tool developed by them, *Gauntlet*, that allows beginner students understanding Java syntax errors made 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* (Hristova et al., 2003) 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. (2014) explain 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 could be very useful to this work in the case of exploring machine learning and data mining techniques on the analysis.

Granger and Rayson (1998) worked on a project that intended to automatically profile essays written in English. The subjects were both native and non-native English speakers (French speakers learning English). Their approach of counting usage of words (articles, determiners, pronouns, propositions, adverbs, nouns and verbs) and then compare the word frequency of non-native to native English speakers is very similar to the intents of this project. The tool they produced concluded that "*the essays produced by French learners display practically none of the features typical of academic writing and most of those typical of speech*".

Shabtai et al. (2010) proposed the use of Machine Learning techniques in the classification of Android applications, trough static analysis of features present in the android application compiled files (APKs). Using the traditional training and testing methodology, they tried several known ML algorithms to discover which worked best for classifying Android apps. Although they intended to detect possible malicious applications, in the end they only managed to differentiate regular applications from games. Nonetheless, the results were very good, achieving a 0.918 accuracy rate using a Boosted Bayesian Networks classifier.

3

PROGRAMMER PROFILING: CHALLENGES AND OUR PROPOSAL

In this chapter, the challenges faced throughout this project will be explained in the form of questions and answers. Here, a detailed explanation regarding our choices can be found, as well as the motivation behind those decisions.

3.1 EARLY DECISIONS

At the early stages of this dissertation some important decisions had to be made. The first step was to choose the programming language to be analysed. Of all the possible languages that could be used to accomplish this project, Java was the most obvious choice. Besides the fact that it's a language the author is quite familiar with, it's one of the most widespread languages both in the academic and professional contexts and there is a lot of available information about what are the best programming practices. It was also decided to only evaluate Java 7 (an leave out Java 8) since, at the moment of this project, it wasn't still very widespread and its construct would add another layer of complexity.

As we have seen in some cases of Chapter 2, Java compiler errors and warnings can be quite hard to properly diagnose. Due to that, syntactic or semantic problems detected by the compiler will not be considered, meaning that this work will not tackle techniques of dynamic analysis. Code with errors may still be analysed, but the errors will not be taken into consideration for the profiling.

Another very important decision is that all programs analysed will be assumed to execute the proposed task, but once again, no dynamic analysis will be done to verify that.

3.2 WHAT ARE PROGRAMMER PROFILES?

Programmer profiling is an attempt to place a programmer on a scale by inferring his profile. As Poss (2014) stated, we can compare proficiency on a programming language with

proficiency on a natural language, and like the *Common European Framework of Reference for Languages: Learning, Teaching, Assessment* (CEFR) has a method[1] 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.

The 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 (2013) presents a classification for programmer categorization: naive, novice, skilled and expert. A similar scale was agreed upon, with what it's believed to be a good starting point for the profile definition:

NOVICE

- Is not familiar with all the language constructs
- Does not show language readability concerns
- Does not follow the *good programming practices* [2]

ADVANCED BEGINNER

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

PROFICIENT

- Is familiar with all the language constructs
- Follows the *good programming practices*
- Shows readability and code-quality concerns

EXPERT

- Masters all the language constructs
- Focuses on producing effective code without readability concerns

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

2 According to McConnell (2004), the best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software.

3 e.g. writes **if (cond==false)** instead of **if (!cond)** as is done by people that have more self-confidence and usually have a less explicit programming style.

The example seen in Listing 3.1 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.

```
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;
}
```

Listing 3.1: "Examples of programs corresponding to different Profile Levels"

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.

3.3 How can we extract this information?

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.

3.4 Which data is relevant for extraction?

On the early stages of this project it was yet not very clear which data would be most useful to the profiling, so on an early development phase a wide range of data was considered to be extracted from the source-code. That data can be seen in the list below and, from here on, will be regarded as metrics.

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

Metrics 1, 3, 4, 6 and 7 show a tendency to be used to infer the skill and language knowledge, and that way know if a solution was that of a programmer with more or less expertise in Java. Metrics 2, 3, 5 and 8 shift more towards concerns with code understandability and readability.

One thing that's clear is that one easy way to spot Novice programmers is to look for the newbie mistakes or bad practices they commit. Instead of implementing features that search for those mistakes, a very useful source code analyser was found, that will be very useful in the detection of those bad practices (this topic will be further explored in Chapter 4).

Finally, the context in which a program is written should be taken into consideration in the profiling process. For example, a teacher that writes code in a safely manner while teaching novices may prioritize code efficiency in personal projects.

3.5 How does that data correlate with programmer 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, clarity and general concern with the understandability of code.

Of the extracted metrics, 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 [4] Violations (e.g. Optimization, Design and Controversial rulesets)

4 More on this in Subsection 4.1.2

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.

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

Table 1.: Proposed correlation

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 1 we can easily infer the programmer's profile in regards to the subject problem.

3.5.1 *A practical example*

The exercise "Read a given number of integers to an array, count how many are even" was proposed to two programmers. An object oriented programming (OOP) teacher and an advanced Java programmer (MsC student). Below we can see both their solutions:

```
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
 */
```

```java
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);
    }

}
```

Listing 3.2: "Teacher Solution"

```java
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);
    }
}
```

Listing 3.3: "Advanced Programmer Solution"

After running both solutions through a first implementation of a metrics extractor created, the results shown in Table 2 were obtained.

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}

Table 2.: PP-Analysis of two solutions

For this example, we will consider the OOP's teacher solution to the problem, a *standard solution*, to help to understand how we can compare two sets metrics extracted from two solutions.

In Table 3 we compare the metrics of the advanced student solution with the ones of the teacher solution solution. In this table, for each analysed metric, the advanced programmer gets 1, 0 or -1 whether his result 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 and how they relate with the groups we have defined, 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 (CFSs) 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.

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**

(+1) - better than the standard solution
(0) - same level as the standard solution
(-1) - worst than the standard solution
(X) - metric no related to this group
PMD results were not considered in this example

Table 3.: Comparing to standard solution

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.

3.6 How to automatically assign a profile to a programmer?

As stated previously, the goal is to calculate scores (Skill and Readability) for the solutions that programmers implement, and then, devise an algorithm that infers the profile of programmers based on those scores.

One of the objectives of the final tool is not to make it dependant on standard (or model) solutions, that are then used to compare with the solutions we want to analyse. The goal is

to make a profile inference that only makes sense in the subset of the programs currently being analysed. That means that, the profiles that are inferred to the programmers could vary if we alter the solutions that are being analysed alongside.

3.7 SYSTEM ARCHITECTURE

Figure 1 shows the block diagram that represents the expected final implementation of the system, as it was envisioned in the early stages of this project. 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 [5]) 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.

Making the profiling an automatic assignment will be the most interesting, challenging and complex part of this project. The goal is not to assign a 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.

5 http://www4.di.uminho.pt/ gepl/PP/

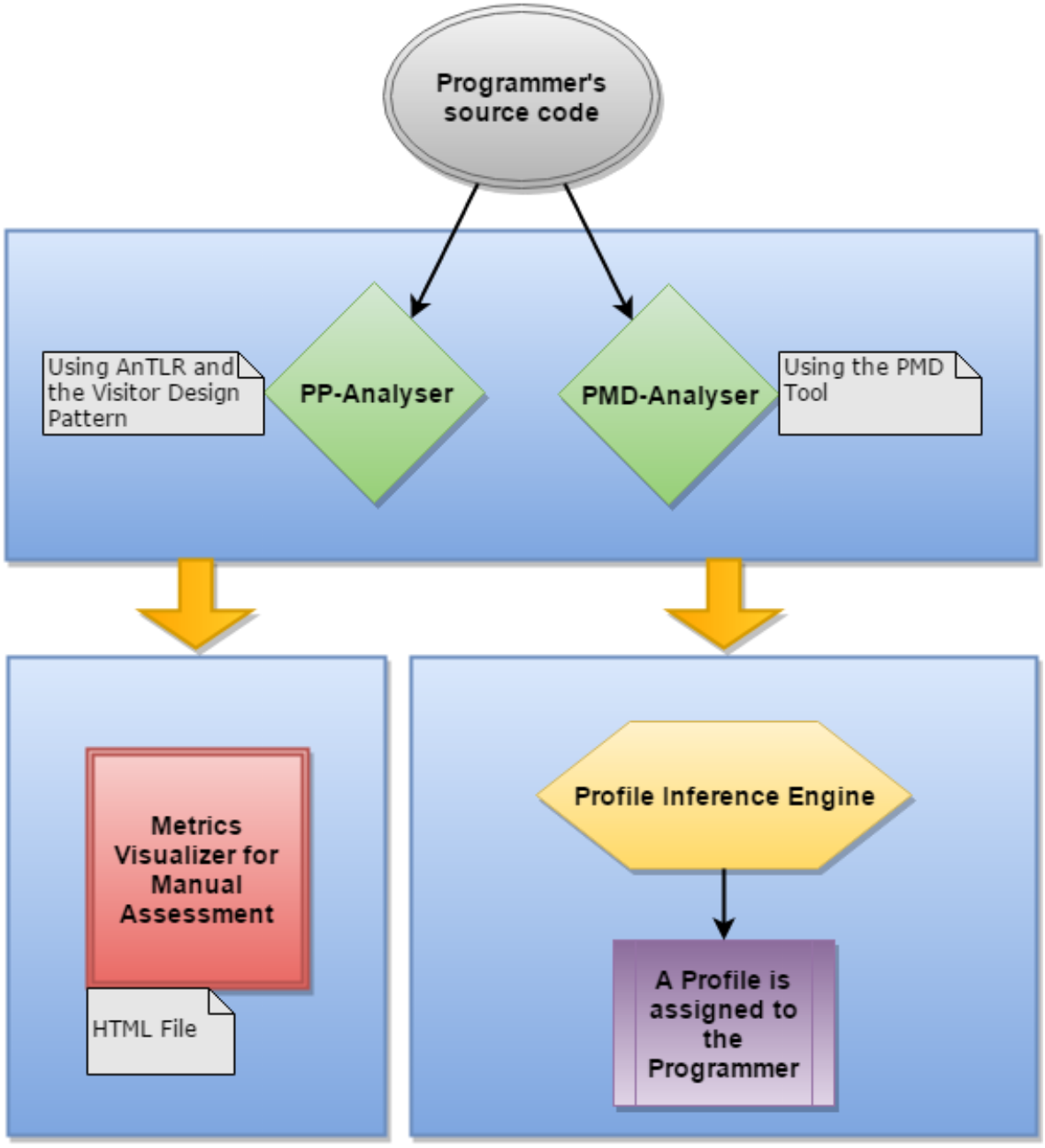


Figure 1.: PP Block Diagram

4

DEVELOPMENT DECISIONS AND IMPLEMENTATION

The following sections will discuss the development phase of the project. At first, the early decisions that had to be made are described, followed by the tools used. Then, the chapter continues with a description of the implementation phase.

4.1 EARLY DECISIONS

To path to solve the *Programmer Profiler* problem was still quite unexplored. Taken that, the implementation component had to be present from the beginning so we knew we were travelling down the right path.

Firstly, it was decided that an appropriate name for the tool, proposed in this thesis, would be *Programmer Profiler* (PP). Another one of the first steps was to analyse simple Java source-files, so it was decided that the appropriate tool for that would be AnTLRv4, mainly because of the familiarity of the authors with the tool and the fact that there are already Java grammars freely available.

4.1.1 *AnTLR*

As taken from the website[1]:

> 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.

Nowadays, AnTLR is one of the most used Compiler Generators in the world in both professional and academic environments. By using only a Java grammar, AnTLR will generate a parser that will recognize any syntactically correct Java source file. This allows us to work with the information that a source file contains in any desired way. Additionally,

1 http://www.antlr.org/

AnTLR is also used to extract, with more or less ease, any kind of needed metric from the Java files.

On the other hand, 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. The book, *The definitive ANTLR 4 reference* (Parr, 2013) appears to be a very useful resource for the development of this tool.

4.1.2 *PMD*

PMD[2] is a source code analyser. It finds common programming flaws like unused variables, empty catch-blocks, unnecessary object creation, and so forth.

PMD was selected among other similar tools mainly due to its large support in many aspects of the Java language (bad practices, code smells, etc). It's an open-source tool and it has a good API to work with.

PMD executes a thorough analysis over source-code (since it supports several languages) and reports back with the violations found. Violations are predefined rules that are triggered when source is analysed using this tool. The rules are grouped in rulesets of related flaws.

PMD looks for dozens of poor programming practices, nonconformity to conventions and security guidelines.

Listing 4.1.2 depicts some of the main PMD 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

2 https://pmd.github.io/

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

Detailed information on all rules our tool detects can be found in Appendix B.

4.1.3 *Metrics*

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. Currently, for each solution to a exercise, the metrics being extracted by the *ProgramerProfiler Tool* are the following:

Code Size Metrics

These metrics are related with code size. We believe code size is mainly related with readability concerns. A greater number of methods, lines of comment, and empty lines shows that the programmer has a concern with the understandability of code. Although not being so obvious, we also believe that a greater number of lines of code also shows that the programmer wanted to make the code as comprehensible as possible.

Another metric worth mentioning is the *Percentage of Lines of Code*, where we consider that a lower percentage will imply higher readability. In terms of percentage, all code is divided into lines of code, comment, and empty lines. So a lower percentage of lines of code will mean a higher percentage of comments and empty lines, which benefices readability.

Of all these metrics, the only one we believe differs from all the other is the *Number of Statements*. By statement we mean the smallest standalone element of the Java language, that expresses some action to be carried out. We believe that the number of statements is closely related to the skill of a programmer, meaning that if a programmer can implement a equivalent algorithm as another, using fewer statements, than we can assume, heuristically, that programmer has higher language knowledge and, therefore, higher skill.

Below is our proposal of the metrics and how we believe these metrics may affect the groups we previously defined, skill and readability.

NUMBER OF CLASSES

- **Affects:** Readability and Skill
- **How:** More classes implies +R (more Readability) and +S (more Skill)
- **Priority:** 2
- **Description:** Total number of classes

NUMBER OF METHODS

- **Affects:** Readability and Skill
- **How:** More methods implies +R and +S
- **Priority:** 2
- **Description:** Total number of methods

NUMBER OF STATEMENTS

- **Affects:** Skill
- **How:** Less statements implies +S
- **Priority:** 8
- **Description:** Total number of statements

NUMBER OF LINES OF CODE (LOC)

- **Affects:** Readability
- **How:** More lines of code implies +R
- **Priority:** 5
- **Description:** Total number of lines of code
- **Also Known as:** LOC

PERCENTAGE OF LINES OF CODE

- **Affects:** Readability
- **How:** Lower percentage of lines of code implies +R
- **Priority:** 5
- **Description:** Percentage of LOC in total number of lines
- **Also Known as:** %LOC

NUMBER OF LINES OF COMMENT

- **Affects:** Readability
- **How:** More lines of comment implies +R
- **Priority:** 3
- **Description:** Total number of lines of comment
- **Also Known as:** LOCom

PERCENTAGE OF LINES OF COMMENT

- **Affects:** Readability
- **How:** Higher percentage of lines of comment implies +R
- **Priority:** 3
- **Description:** Percentage of LOCom in total number of lines
- **Also Known as:** %LOCom

NUMBER OF EMPTY LINES

- **Affects:** Readability
- **How:** More empty implies +R
- **Priority:** 3
- **Description:** Total number of empty lines
- **Also Known as:** Blank lines

PERCENTAGE OF EMPTY LINES

- **Affects:** Readability
- **How:** Higher percentage of empty lines implies +R
- **Priority:** 3
- **Description:** Percentage of empty lines in total number of lines

Control Flow Statements Metrics

Control flow statements (CFS) are the heart of the algorithms. These are statements whose execution results in a choice being made as to which of two or more paths should be followed. the control flow statements we consider are: if-statements (if-then and if-then-else), while-statements (while and do-while), for-statements (for and enhanced for), switch-statements and inline-if-statements (also know as the ternary operator).

Usually, programmers either opt for using more CFSs and make their algorithm more explicit and easier to understand or for using less CFS making it less explicit and harder to comprehend. One thing that we can all agree is that creating an alternate, much more implicit, algorithm is something that not every programmer can do. Reducing the size and execution time of the program by cutting on the use of controls is a task only for those more skillful and expert. As an example we can imagine a programmer which does all tasks inside a loop statement and another that has to use two or more to accomplish the same thing.

Almost all CFSs are interchangeable. For-loops can be replaced with while-loops, switch-statements can be replaced with nested if-statements and so on. Variety on the use of CFSs will show that the programmer knows the language and what controls are better for each situation.

Finally, there are some CFS that we consider not so common and, therefore, show language knowledge. Those are the do-while-loop, the enhanced-for-loop and the inline-if-condition. Using these will increase the *skill points*.

TOTAL NUMBER OF CONTROL FLOW STATEMENTS

- **Affects:** Skill and Readability
- **How:** Less CFSs implies +S and more CFSs implies +R
- **Priority:** 5
- **Description:** Total number of Control Flow Statements
- **Also Known as:** CFS

VARIETY OF CONTROL FLOW STATEMENTS

- **Affects:** Skill
- **How:** More variety of CFS implies +S
- **Priority:** 4
- **Description:** Variety of CFS used

TOTAL NUMBER OF NOT SO COMMON CFSS

- **Affects:** Skill
- **How:** More *Not So Common CFS* implies +S
- **Priority:** 6
- **Description:** Total number of *Not So Common CFS*
- **Also Known as:** NSCCFS

Not So Common Operators Metrics

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

```
SHIFT ('<<', '>>' and '>>>')
BIT_AND ('&'), BIT_OR ('|')
CARET ('^')
ONE_ADD_SUB ('++' and '--'), ADD_ASSIGN ('+='), SUB_ASSIGN ('-=')
MULT_ASSIGN ('*='), DIV_ASSIGN ('/='), AND_ASSIGN ('&=')
OR_ASSIGN ('|='), XOR_ASSIGN ('^='), RSHIFT_ASSIGN ('>>=')
URSHIFT_ASSIGN ('>>>='), LSHIFT_ASSIGN ('<<='), MOD_ASSIGN ('%=').
```

We consider the usage of these operators as an indication of skill.

VARIETY OF *NOT SO COMMON OPERATORS*

- **Affects:** Skill
- **How:** More variety of NSCO implies +S
- **Priority:** 5
- **Description:** Variety of *Not So Common Operators* used
- **Also Known as:** NSCO

Variable Declaration Metrics

Similarly to the case of the Control Flow Statements, while using variables a programmer can reuse non-needed variables to reduce the size of the code and memory used by the system. But just like in the example above, this will cause the code to become harder to understand (this requires the variables to be given more generic names, for example).

Also, like in the case of the *Not So Common Operators*, there are a lot of types available for the programmer to use, and sometimes there are some more appropriate then others. For instance, an expert programmer should know when it's best to use *float* or *double*.

TOTAL NUMBER OF DECLARATIONS

- **Affects:** Skill and Readability
- **How:** Lower number of Declarations implies +S and -R
- **Priority:** 5 for the first and 3 for the second
- **Description:** Total number of variables declared

TOTAL NUMBER OF TYPES

- **Affects:** Skill
- **How:** More variety of types implies +S
- **Priority:** 4
- **Description:** Variety of types used

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. So far, the only metric being analysed is the *Readability Related Relevant Expressions* on which the main output methods are being considered.

System.out and *System.err* methods can be very helpful in understanding code (mainly when accompanied by some kind of message or debug information). For that reason they are being considered as readability-increasing. In the future other methods or constructs could be added to this section.

TOTAL NUMBER OF READABILITY RELEVANT EXPRESSIONS

- **Affects:** Readability
- **How:** Higher number of Relevant Expressions implies +R
- **Priority:**3
- **Description:** Total number of readability-related Relevant Expressions
- **Also Known as:** RE

PMD Violations Metrics

Lastly we have the PMD Violations Metrics. These metrics are very important because they allow us to detect problems in code that otherwise would be very hard to catch, as seen in subsection 4.1.2, PMD is devided into rulesets, and each ruleset into rules. When these rules are detected in source-code they are referred as violations. Each rule also as assigned a priority (or weight).

Our approach was to assign each PMD rule to one of our previously defined Groups (Skill and Readability) whether that rule as an influence on that given group. Some rules can affect both Readability and Skill. Than we measure the number of detected violations to punish the programmers in that Group to which that rule belongs to.

The rules, descriptions, priorities and groups can be consulted in Appendix B.

4.1.4 *Enhancing the Profiles*

As we've seen in Chapter 3, the idea behind the inference of profile is to calculate a Skill and Readability scores, and combining those values, assign a profile to the programmer.

As time progressed, our idea of the profiles shifted a bit from the original idea that we saw in Table 1. We decided that the Experts should be the ones with maximum focus on Skill, the Proficients on Readability and the Advanced Beginners should be divided also 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 R (AB-R)**: Low Skill and Average Readability
- **Advanced Beginner S (AB-S)**: Average Skill and Low Readability
- **Advanced Beginner + (AB +)**: Average Skill and Average Readability
- **Proficient (P)**: Low-to-Average Skill and High Readability
- **Expert (E)**: High Skill and Low-to-Average 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.

Figure 2 visually shows how we envision the profiles.

4.2 IMPLEMENTATION

The following sections discuss the implementation process of PP and the reasoning behind the decisions made.

4.2.1 *Setting up*

The first step of the implementation was setting up AnTLRv4. The *Integrated Development Environment* chosen was IntelliJ IDEA[3], a very powerful and modern Java IDE. IntelliJ IDEA

3 https://www.jetbrains.com/idea/

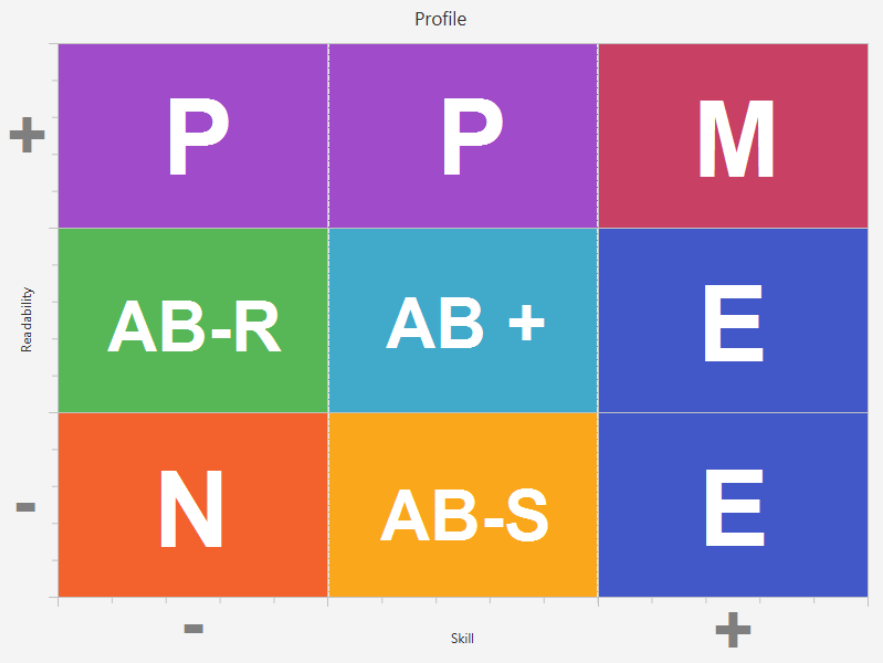


Figure 2.: Correspondence between scores and profiles

has available an AnTLRv4 plug-in, so setting it up was straightforward. The plugin[4] offers all features present in ANTLRWorks[5], a grammar development environment, with the advantage of already being integrated in the IDE used to create the PP tool. It is also maintained by the developers of AnTLR themselves.

Then, a Java grammar for AnTLRv4 was needed. Among the several available, the one chosen[6] was provided by the creator of AnTLR himself.

4.2.2 *Attribute Grammar vs Visitor Pattern*

When we first began testing static code analysis, the most immediate solution was to add attributes to the productions of the grammar, with the purpose of counting the occurrences of the language constructs.

Soon enough it was obvious that, despite being a possible solution, this approach would not be viable. That's due to the fact that with the amount of metrics needed, the resulting grammar would become unreadable and very hard to maintain.

4 https://github.com/antlr/intellij-plugin-v4
5 http://tunnelvisionlabs.com/products/demo/antlrworks
6 https://github.com/antlr/grammars-v4/tree/master/java

Given that, the solution was to implement a visitor design pattern. In short, a visitor class is defined, which implements a method for each production of the grammar. When the Abstract Syntax Tree (AST) is being traversed by the visitor, each node calls its respective method. That method can then be overridden to extract any kind of information. For example, the method *visitIfStatement*:

```
@Override
public Object visitIfStatement(JavaParser.IfStatementContext ctx) {
    incr(JConstruct.IF);
    return super.visitIfStatement(ctx);
}
```

is called whenever an if-statement is found. The **incr** method increments the if-statement counter.

AnTLR generates automatically the BaseVisitor, which can then be implemented in our classes.

4.2.3 *Fixing up the Grammar*

Almost all metrics needed to be extracted are of numeric kind, meaning, the number of occurrences of given language constructs. That means that the goal of the metrics is, for example, to know how many if-statements a solution implements.

The way the original Java grammar was implemented, had many of these Java constructs defined as terminals:

```
statement
    :   block
    |   ASSERT expression (':' expression)? ';'
    |   'if' parExpression statement ('else' statement)?
    |   'for' '(' forControl ')' statement
    |   'while' parExpression statement
    |   'do' statement 'while' parExpression ';'
    |   'try' block (catchClause+ finallyBlock? | finallyBlock)
    |   'try' resourceSpecification block catchClause* finallyBlock?
    |   'switch' parExpression '{' switchBlockStatementGroup* switchLabel* '}'
    |   'synchronized' parExpression block
    |   'return' expression? ';'
    |   'throw' expression ';'
    |   'break' Identifier? ';'
    |   'continue' Identifier? ';'
    |   ';'
```

```
    |   statementExpression ';'
    |   Identifier ':' statement
    ;
```

The problem with this, is that we can only implement a visitor for the *statement* production rule. In this way, when that visitor is triggered by a construct, such as, a *if*, *while* or *switch* statement, we couldn't know exactly which one triggered it. To know that, we would have to manually compare the beginning of the expressions to the constructs we were looking for. That would make the visitors very big and ugly methods.

The solution was to separate the desired constructs into different production rules. This means changing the original grammar a bit:

```
statement
    :   statementBlock
    |   ASSERT expression (':' expression)? ';'
    |   ifStatement
    |   forStatement
    |   whileStatement
    |   doWhileStatement
    |   'try' block (catchClause+ finallyBlock? | finallyBlock)
    |   'try' resourceSpecification block catchClause* finallyBlock?
    |   switchStatement
    |   'synchronized' parExpression block
    |   returnStatement
    |   'throw' expression ';'
    |   breakStatement
    |   continueStatement
    |   ';'
    |   statementExpression ';'
    |   Identifier ':' statement
    ;

statementBlock
    : block
    ;

ifStatement
    : 'if' parExpression statement ('else' statement)?
    ;

forStatement
    : 'for' '(' forControl ')' statement
    ;
```

With these changes, we could define a visitor for each metric being extracted, as we saw above.

4.2.4 *Early Metrics Extracted*

As said in 3.3, the first metrics to be extracted were:

- Class hierarchy
- Class and method names and sizes
- Variable names and types
- Number of classes, methods, statements
- Control Flow Statements (if, while, for, etc)

Other metrics that look for poor practices usually adopted by beginner programmers (e.g. several returns in one method or leaving empty catch-blocks) were also implemented. However, after discovering PMD (which does that and much more), those implementations were discarded.

4.2.5 *Preliminary PMD Integration*

The integration of PMD was quite straightforward. First we imported the JAR file with the PMD solution. Then, to run the tool we just have to pass arguments: a folder to examine, the rulesets to analyse and the desired format to output the result. Below is presented an example of a violation detected by PMD in XML format:

```
<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>
```

In this example of a violation we can see that it was detected a fault of type *CollapsibleIfStatements*, which belongs to the *Basic* ruleset. As the description explains this violation means: *These nested if statements could be combined*. We can also see in which file, line and column the violation was detected.

All the information that the PMD provides has a lot of potential aiming at classifying programmers by analysing their code. But in order to achieve that we must give some meaning to the PMD data. As previously mentioned PMD violations refer to bad practices and possible code-smells, so we can generalise those violations to mistakes done by programmers (although some of them may be nothing more than conventions that have really no impact in quality of code).

The goal is to penalize the programmers by the amount and type of PMD violations found in their code. To do that a CSV file was created that has almost all possible PMD violations and, for each, a priority (or weight) and a group (remember Skill and Readability) to which the violation is more related too. Below we can see some violations as example:

```
"Code Size","ExcessiveMethodLength","When methods are excessively long this
    usually indicates that the method is doing more than itsname/signature might
    suggest. They also become challenging for others to digest since excessive
    scrolling causes readers to lose focus.Try to reduce the method length by
    creating helper methods and removing any copy/pasted code.","3","R"

"Braces","ForLoopsMustUseBraces","Avoid using 'for' statements without using
    curly braces. If the code formatting or indentation is lost then it becomes
    difficult to separate the code being controlled from the rest.","3","R"

"Design","SimplifyBooleanReturns","Avoid unnecessary if-then-else statements when
     returning a boolean. The result of the conditional test can be returned
    instead.","3","S"

"Empty Code","EmptyIfStmt","Empty If Statement finds instances where a condition
    is checked but nothing is done about it.","3","B"

"Basic","ExtendsObject","No need to explicitly extend Object.","4","S"
```

The last two values of each entry are the priority (1-5) and the group (S, R and B, for Skill, Readability and Both). The priority was assigned by the developers of PMD themselves. The group to which they are more relevant was assigned by the authors of this work. Later this information will be used to help calculating a score to the solution of a given problem.

4.2.6 *Solutions Input*

In order to properly analyse the files that contain the various solutions provided by different programmers to the same problem, they must be previously carefully prepared. For each problem to be analysed a folder must be created. The name of that folder will be the name

of the problem. That folder will be organized in subfolders, each containing the solution to that given problem. The name of each subfolder will be the name of the person that solved it. Each folder must contain the file (or files) needed to solve the exercise. The name of those files is not relevant.

A folder with the base solution must be created and given separately as input, in order to differentiate it from all the others.

The first step of the PP process will be to cycle through all the folders of the problem and retrieve the paths of the Java files, and link them to the programmer that created that solution. After that we are ready to start the PP analysis.

4.2.7 *PP Analyser*

The PP Analyser is the class responsible for extracting all metrics from the Java source code.

The process starts by receiving the paths of Java files. Then using the Lexer and Parser, generated from the Java grammar using AnTLR, the Parse Trees (also known as Concrete Syntax Trees) will be generated for each file, according to the following code snippet:

```
public ParseTree generateParseTree (String filePath) throws IOException {
    ANTLRInputStream input;
    input = new ANTLRInputStream(new FileInputStream(filePath));
    JavaLexer lexer = new JavaLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    JavaParser parser = new JavaParser(tokens);
    ParseTree tree = parser.compilationUnit();
    return tree;
}
```

With all the Parse Trees generated we can begin extracting the metrics. As it was previously mentioned, this project will implement the visitor design pattern to traverse the trees in order to extract the metrics. For each set of related metrics a class with the purpose of extracting those metrics was created. Each class implements the JavaBaseVisior, generated with AnTLR from the grammar, that will allow us to use the visitor methods.

A visitor is a method generated, that is triggered when a given production of the grammar is visited. For example, in the grammar we have a production:

```
localVariableDeclaration
    :   variableModifier* type variableDeclarators
    ;
```

Implementing the BaseVisitor we will be able to override the method:

```
@Override
public Object visitLocalVariableDeclaration(JavaParser.
    LocalVariableDeclarationContext ctx) {
    int count = ctx.getChild(1).getChildCount();
    String type = ctx.getChild(0).getText();
    for (int i = 0; i < count; i++) {
        ParseTree c = ctx.getChild(1).getChild(i);
        insertVar(c, true, type);
    }
    return super.visitLocalVariableDeclaration(ctx);
}
```

This method will be called whenever that production is visited in the Parse Tree. In this case, information about local variable declaration is retrieved (name and type).

This is done for almost all metrics, and the following classes were created:

- CFSExtractor: for Control-flow statements related metrics
- VariableExtractor: for Variable declaration related metrics
- CounterExtractor: for counting classes, methods and statements
- NSCOExtractor: For uncommon and advanced java constructs
- REExtractor: For other relevant metrics

The only exception are the metrics related to code and comments lines, that use a separated tool for its extraction.

Almost all the metrics are of numeric kind (most of them are counters). After the extraction, all metrics are stored in a specially created class that holds all the metrics for a given solution. Adequate data-structures were chosen so that the data could be easily manipulated.

4.2.8 *PMD Analyser*

The PMD Analyser is the class responsible for extracting all the violations detected in the source code by the PMD tool.

For each problem to be analysed, we run the PMD tool with a set of previously chosen rulesets; as output we chose the CSV file format. The CSV file produced for each program analysed will be placed in the corresponding directory, according to the structure explained above.

The first approach was to extract this data using XPath queries on the XML files, but due to the way the information was structured on the files, that proved to be rather difficult (would require very complex queries). An approach similar to the early metrics extraction was adopted, meaning that the PMD data was structured in Java classes.

Instead, the CSV output files are analysed with a CSVReader tool and their contents are stored on a defined PMDRule class. The final goal is to have a list of all violations detected on a file, separated by problem and by type.

4.2.9 *Projects Comparison*

In the ProjectsComparison class we assemble all data collected so far (both PP and PMD Analysis). Firstly an auxiliary file containing all existing PMD violations and all information related is loaded. That way we can link all violations detected with important data, such as, the priority, group or a description of the violation.

The second feature of this class is the generation of an HTML file with all information collected by the analysis. That page is composed of tables where we can see all the results extracted both from PP and PMD. The main goal of that page is to enable a manual assessment of the results. This feature proved to be very useful in a first project stage while we were trying ways to automate the profile calculation.

4.2.10 *Score Calculator*

The ScoreCalculator class is the heart of the Programmer Profiler Tool. Its job is to analyse all information that was previously extracted using PP and PMD Analysis, and calculate a numeric score value for each programmer's solution.

As described in Chapter 3, each metric has an effect on one or both groups (Skill and Readability), and that effect can either be positive or negative. For example, the *NumberOfStatements* metric has the effect "*lower is better*" on the Skill group. If that metric value is low when comparing to other solutions that effect will be positive, otherwise there will be a negative effect. In other words, a positive effect will lead to a greater increment of the Skill group numeric score, while a negative will cause the opposite.

The first step of the ScoreCalculator is to load the PP Analyser metric rules. For that, a JSON file was created that contains all the information about how the metrics influence the score of a solution. Each JSON object contains:

- The name of the method that extracts the referred metric

- Whether it's a high(+) or low(-) value that affects the group
- The group being affected (and if it's positively or negatively)
- The priority (or weight) of that metric

```
[
  {
    "methodname": "getNumberOfMethods",
    "_this"     : "+",
    "implies" : "+R",
    "priority": 2
  },
  {
    "methodname": "getNumberOfMethods",
    "_this"     : "+",
    "implies" : "+S",
    "priority": 2
  },
  {
    "methodname": "getNumberOfStatements",
    "_this"     : "-",
    "implies" : "+S",
    "priority": 8
  }
]
```

Listing 4.1: "Metrics rules excerpt"

Listing 4.1 shows a small excerpt of the referred JSON file. As seen in Section 4.1.3, the way we should interpret this should be:

- For the metric *NumberOfMethods*, a high value will imply an increase in the Readability score with a weight of 2.
- For the metric *NumberOfMethods*, a high value will imply an increase in the Skill score with a weight of 2.
- For the metric *NumberOfStatements*, a low value will imply an increase in the Skill score with a weight of 8.

After all this metric data is loaded to the system, the actual score calculation process can begin. The following paragraphs try to demonstrate the algorithm behind this calculation for a set of solutions of a given programming challenge.

First we start by setting everyone's base skill and readability scores to zero. Then, for each metric, we calculate which solution has *the best result*. *The best result* is the lowest (or highest) value obtained by comparing all the results of a given metric. The lowest or the highest value is chosen depending on the **_this** field of the metrics file that we saw in Listing 4.1. For instance, for the *NumberOfMethods* we calculate the highest result obtained by all solutions, as opposed to the *NumberOfStatements*, where we pick the lowest.

After that, and still for each metric, by comparing all other solutions to *the best one*, a ratio is calculated for each solution. *The best solution* will always get a ratio of 1. All other solutions will have lower values depending on how distant those values are from *the best one*. Usually that value falls in the range [0-1] but it can be below zero if that metric is very deviated from *the best one*.

As explained before, each metric also has a priority (or relevance) field, which gives us an idea of how important that metric is for inferring the profile. The low relevance metrics have a priority of around 1 to 3 while the high profile ones have a priority of 5 to 8.

Recall that each solution starts with a score of zero in both Skill and Readability. For each metric processed, that value is updated with the value obtained using the formula $\boldsymbol{I} * \boldsymbol{P} * \boldsymbol{R}$, where:

- $\boldsymbol{I}$ is the impact of the metric (1 for positive and -1 for negative). This can be found in the *implies* field of the metrics JSON file
- $\boldsymbol{P}$ is the priority (or weight) of the metric. Between 1 and 8.
- $\boldsymbol{R}$ is the ratio for the result calculated in that metric. Usually between 0 and 1, but could be less than 0 if the metrics' results are to deviated from the average values.

As we can perceive through the formula, if the impact is positive, *the best results* for the given metric will yield a greater increment of the group score that metric affects. A not so good result, will cause a smaller increase. The reverse applies to a negative impact. *Best results* cause low decrease of the score, while *worst results* cause a greater decrease. In the rare situation where a metric has a ratio below zero that means that even if the impact is positive it will result in a decrease of that group score.

Tables 4, 5 and 6 depict a small fictitious example using the metrics from Listing 4.1. Table 4 starts by calculating the ratio for each metric, by comparison among all solutions. Table 5 applies the formula $\boldsymbol{I} * \boldsymbol{P} * \boldsymbol{R}$ that we saw before. The result of this formula will be used to update the group scores as we can see in Table 6.

	# Statements	# Methods	NoS Ratio	NoM +R Ratio	NoM +S Ratio
Ex1	5	1	1	0.5	0.5
Ex2	8	1	0.625	0.5	0.5
Ex3	10	2	0.5	1	1
Ex4	7	1	0.714...	0.5	0.5
Ex5	8	1	0.625	0.5	0.5

Table 4.: Ratios obtained for a small example

	NoS	NoM +R	NoM +S
Ex1	1 * 8 * 1 = 8	1 * 2 * 0.5 = 1	1 * 2 * 0.5 = 1
Ex2	1 * 8 * 0.625 = 5	1 * 2 * 0.5 = 1	1 * 2 * 0.5 = 1
Ex3	1 * 8 * 0.5 = 4	1 * 2 * 1 = 2	1 * 2 * 0.5 =1
Ex4	1 * 8 * 0.714 = 5.7...	1 * 2 * 0.5 = 1	1 * 2 * 0.5 = 1
Ex5	1 * 8 * 0.625 = 5	1 * 2 * 0.5 = 1	1 * 2 * 0.5 = 1

Table 5.: Metric score calculated for a small example

	Skill	Readability
Ex1	+9	+1
Ex2	+6	+1
Ex3	+5	+2
Ex4	+6.7...	+1
Ex5	+6	+1

Table 6.: Resulted increase in groups for a small example

After calculating Skill and Readability scores for the PP metrics it's time to update the scores with the PMD results. The process is quite similar. Currently we are only considering the number of PMD violations detected (and not their weight). So, for each PMD violation detected, a unit (1) is deducted from the corresponding group score (remember that each violation is connected to a group, or both).

When this process ends, each solution has two value scores assigned to them, one for each group.

4.2.11 *Profile Inferrer*

The ProfileInferrer class is responsible for the mapping of the results obtained in both groups (Readability and Skill) to the profiles.

As we've seen in Figure 2, the idea is to have the pair (skill,readability) plotted onto a graph and then divide the graph into proportionally sized areas. To each area, there will be an assigned profile. The lowest and highest values obtained in both skill and readability groups will define the outer boundaries of the plot and then, the plot is divided in a grid 3x3, where each cell has the same area. Each cell as a profile associated, and all solution that fall on a given area will be assigned that profile.

4.2.12 *Log Generator*

Log information was very helpful throughout the development of all the algorithms for the profile inference. The information provided by logging allowed an easier debugging when developing all the algorithms.

Every time the PP is run, a log file is created with all the numbers and calculations done in order to infer the profile. The LogGenerator class is responsible for the creation of such file. While all the classes of the PP are being executed a *StringBuffer* is storing a lot of relevant information that is, in the end, written to a file.

4.2.13 *Results Plotter*

Lastly, the ResultsPlotter class uses the JavaFx [7] library to create a graph with all the scores obtained and the profile inference in action.

This plot shows the results achieved in the groups, by every solution, and also which profile was assigned. This was very helpful in the development phase, and will certainly be

7 http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html

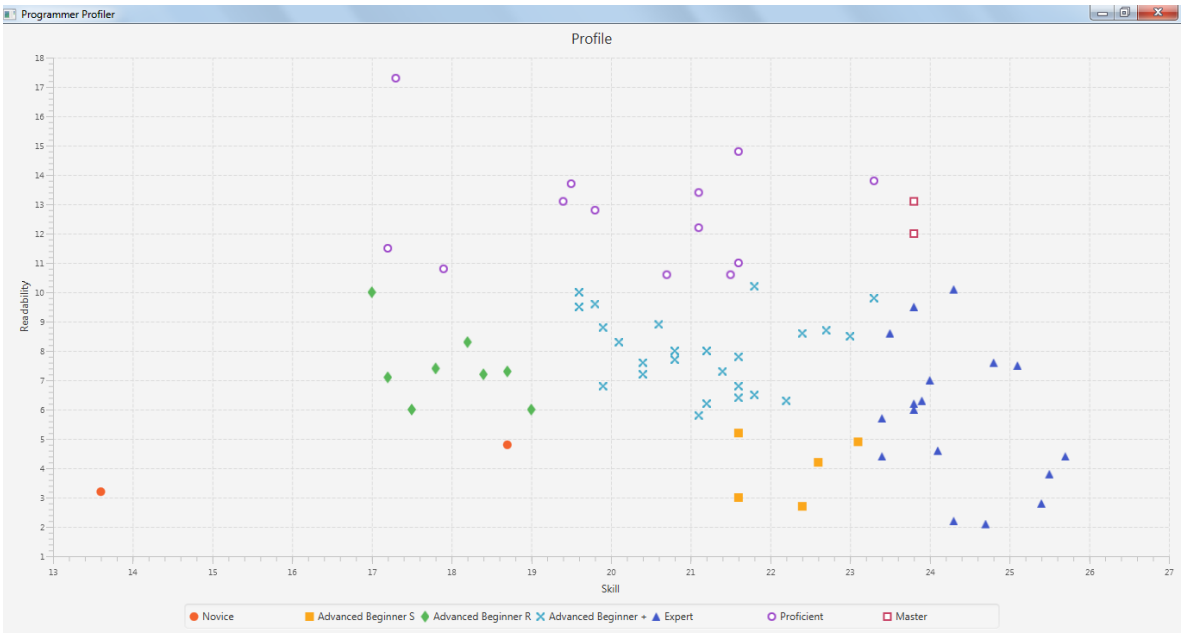


Figure 3.: PP Plot for Numbers Challenge

in the production phase, because it allows us to visually understand how the programmers performed in both skill and readability categories. We can see if there are any solutions very deviated from the usual, and also to see if a solution got very close or far from another profile.

Each profile has a color assigned, and mouse-clicking a solution will open the folder containing the source files for that implementation.

4.2.14 *General Profile Inferer*

For each set of solutions for a given exercise being evaluated, a JSON file is generated with the results obtained on the profiling. For each solution, the JSON file contains the name of the programmer and the Skill/Readability scores obtained by them.

```
[
    {
        "name": "Student1",
        "skill": 22.8,
        "readability": 12.8
    },
    {
        "name": "Student2",
        "skill": 22.9,
```

```
        "readability": 19.2
    },
    ...
]
```

Listing 4.2: "Excerpt of results JSON File"

This file contains the necessary information to generate the results plot, as seen in subsection 4.2.13.

In a scenario where a teacher is evaluating his students using our PP Tool by giving them several problems to solve, for each exercise that the tool analyses a file like this will be generated. After evaluating several exercises made by the same students, the goal is to generate a more precise profile inference (by combining the results obtained on each exercise).

By normalising the results obtained by the students and averaging the scores of all exercises each student solved we are finally able to more precisely profile a programmer, and accomplish the hypothesis of this thesis.

Figure 4 shows the final system architecture implemented. As you can see, it's divided in two modules. The first one is responsible for calculating the scores and inferring the programmer's profile for several solutions to the same problem. Of course, inferring a profile based on a single exercise may not be very accurate. The goal of the second module is to, using the scores calculated by the first module for several exercises, infer a much more reliable profile for the programmers that solve those exercises.

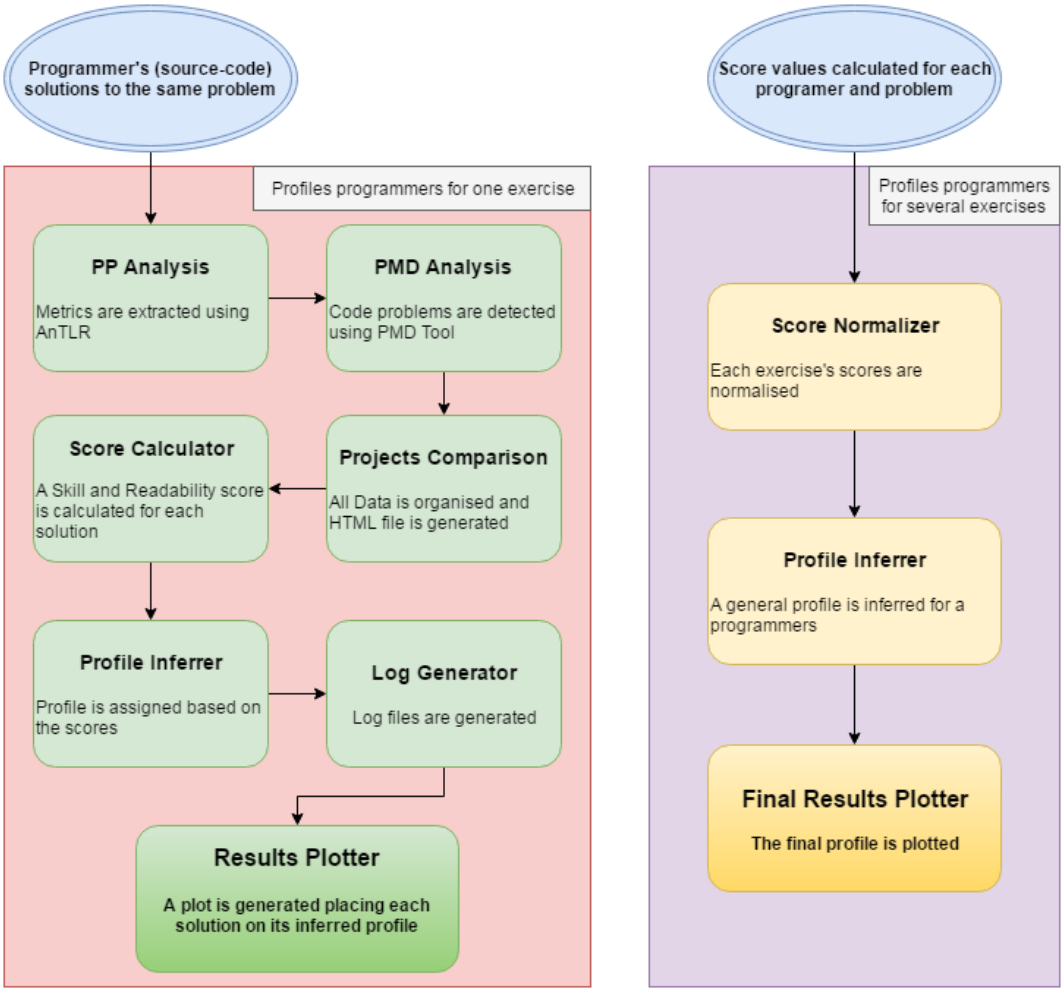


Figure 4.: Final System Architecure

5

CASE STUDIES AND EXPERIMENTS

5.1 EXPERIMENT SETUP

5.1.1 *Development Phase Setup*

Ever since the early stages of this project, development and experiment have been walking hand in hand. When the first draft of the tool was up and running a small amount of exercises were used to begin tests and reach early conclusions.

Professors of object-oriented courses (that used the Java language in their classes) supplied some basic to medium difficulty exercises. We then requested several students to solve those exercises. The students were mainly Master's students of Computer Science (CS) that already had completed the course and had a good amount of experience in Java. A few were still novices in the language. For each exercise we also had a solution made by the course professor.

Listing 5.1 shows the four exercises proposed for students to solve.

```
P1) Write a Java program that reads a non-defined amount of positive integers (
    number 0 will terminate the input phase). Compute and print the amount of even
     numbers, odd numbers, and the average (real number) of the even numbers.
For example, given:
1
2
12
7
15
0
The output should be:
2
3
7.0
```

```
P2) Write a Java program that, given two ages (integers, M and N) reads N ages
    and outputs all ages greater than M, and the average (real number) of all ages
    .
For example, given:
20 //-> M
5  //-> N
15
20
21
40
5
The output should be:
21
40
20.2

A1) Write a Java program that, given and integer N greater than 0, reads to a
    vector (unidimensional array) N integers and in the end prints the amount of
    even numbers in that array.
For example, given:
10 //-> N
1
2
12
7
5
20
3
5
8
10
The output should be:
5

S1) Write a Java program that, given two string, counts the occurrences of a
    string in another.
For example, given:
"BABABABA"
"BAB"
The output should be:
3
```

Listing 5.1: List of exercises given to students

On this first phase of the experiment, there were a total of 10 participants, but not all of them solved all four exercises. Here is a list of all participants, and their previous background in Java (we shall refer to them by their initials):

- **P**: OOP Professor
- **D, G, V, Z, B, E**: Master's students in CS with solid Java knowledge
- **A, S**: Under-graduated CS students still taking OOP course

And below is a list of the participants and the exercises they solved:

- **Exercise P1**: P, A, D, G, V, Z, S
- **Exercise P2**: P, A, D, B, E, J, Z
- **Exercise A1**: P, A, E, G, S, Z
- **Exercise S1**: P, A, D, V

5.1.2 *Post-Development Setup*

Later on, as the development was on its final phase, a larger and more complete experiment was conducted. After reaching out to a OOP teacher at Minho University, we agreed that he would propose the previously mentioned exercises to his students.

This experiment was conducted at the end of the course and the goal was to infer the profile of most of the students that took that OOP course.

We had 70 student submitting their solutions to the proposed exercises, and adding the 10 that we already had, made a total of about 80 solutions for each one of the four exercises.

5.2 RESULTS

In this section we will reenact the first experiment conducted. We will take a look at some solutions of the problems stated before and make a manual assessment, explaining how our tool will interpret these solutions in order to infer a profile.

5.2.1 *Development Phase Results*

This smaller experiment had the main goal of working on a manual assessment of the profiles, in order to shape the PP tool that was being developed alongside.

The careful inspection of the solutions made by the (mainly advanced) students and the teacher, helped us to understand the familiarity these programmers already had with the language and the paradigm, and also identify different styles of programming.

Case Study 1

Taking, for instance, two solutions of the P1 exercise seen in Listings 5.2 and 5.3.

```java
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 input = new Scanner(System.in);
            int num = input.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 Average: " + average);
    }
}
```

Listing 5.2: "Solution to P1 made by S"

```java
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;
```

```
12              } else odds++;
13              value = in.nextInt();
14          }
15          System.out.println(evens + "\n" + odds);
16          System.out.println(evens >0?evensSum/evens:evensSum);
17      }
18  }
```

Listing 5.3: "Solution to P1 made by Z"

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 inputted and the variables are updated. 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)*. The bitwise *AND* operation, used with the *&* character, takes two binary representations and performs the logical *AND* operation on each pair of the corresponding bits, by multiplying them. Using this method we only have to check if the last bit of a binary representation of an integer is 1 or 0 (1 means the number is odd and 0 means the number is even). 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.

Wrapping up the analysis, we can 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. As you can see in Figure 5, using only these small examples, *S* was classified as *Advanced Beginner* with a readability focus, obtianing a (S,R) score of (20.9, 15.9). *Z* was classified as *Expert* with a score of (32.2, 10.7). This complies to their programming background, which was stated previously.

Table 7 contains all metrics that were extracted from this problem solutions. The focus of attention should be the columns of the two programmers we are analysing (*S* and *Z*) but the others should not be disregard, since the results are obtained by comparing all solutions among each other.

In this table, if we pay enough attention (refer to Subsection 4.1.3), we are able to see all the characteristics and differences that were mentioned above.

	A	D	G	**S**	V	**Z**	P
Files	1	1	1	**1**	1	**1**	1
Classes	1	1	1	**1**	1	**1**	1
Methods	1	1	1	**1**	2	**1**	1
Statements	7	7	9	**9**	6	**6**	10
LOC	17	19	17	**19**	19	**16**	24
%LOC	73,9	67,9	54,8	**65,5**	65,5	**84,2**	70,6
LOCom	0	3	1	**0**	0	**1**	1
%LOCom	0	10,7	3,2	**0**	0	**5,3**	2,9
Empty Lines	2	5	10	**6**	5	**0**	5
%Empty	8,7	17,9	32,3	**20,7**	17,2	**0**	14,7
Total Lines	23	28	31	**29**	29	**19**	34
CFSs	{IF=2, FOR=1}	{DOWHILE=1, IF=2}	{IF=1, WHILE=1}	**{IF=3, WHILE=1}**	{IF=1, WHILE=1}	**{IIF=1, IF=1, WHILE=1}**	{DOWHILE=2, IF=1, WHILE=1}
NSC CFSs	0	1	0	**0**	0	**1**	2
CFS Variety	2	2	2	**2**	2	**3**	3
CFS Total	3	3	2	**4**	2	**3**	4
NSCOs	{ADD_ASSIGN=1, ONE_ADD_SUB=3}	{BIT_AND=1, ADD_ASSIGN=1, ONE_ADD_SUB=2}	{ADD_ASSIGN=1, ONE_ADD_SUB=2}	**{ADD_ASSIGN=1, ONE_ADD_SUB=2}**	{BIT_AND=1, ADD_ASSIGN=1, ONE_ADD_SUB=2}	**{BIT_AND=1, ADD_ASSIGN=1, ONE_ADD_SUB=2}**	{}
NSCOs Variety	2	3	2	**2**	3	**3**	0
Var Decl	{float=1, int=4}	{Scanner=1, int=4}	{Scanner=1, float=1, int=4}	**{Scanner=1, double=1, int=4}**	{Scanner=1, int=4}	**{Scanner=1, double=1, int=3}**	{Scanner=1, float=1, int=4}
Total Decls	5	5	6	**6**	5	**5**	6
Decl Variety	2	2	3	**3**	2	**3**	3
Other	{SYSERR=1, SYSOUT=3}	{SYSOUT=2}	{SYSOUT=1}	**{SYSOUT=4}**	{SYSOUT=3}	**{SYSOUT=3}**	{SYSOUT=5}
Skill PMD	3	4	4	**6**	4	**4**	4
Read PMD	6	4	3	**4**	2	**4**	3

Table 7.: P1 Metrics extracted

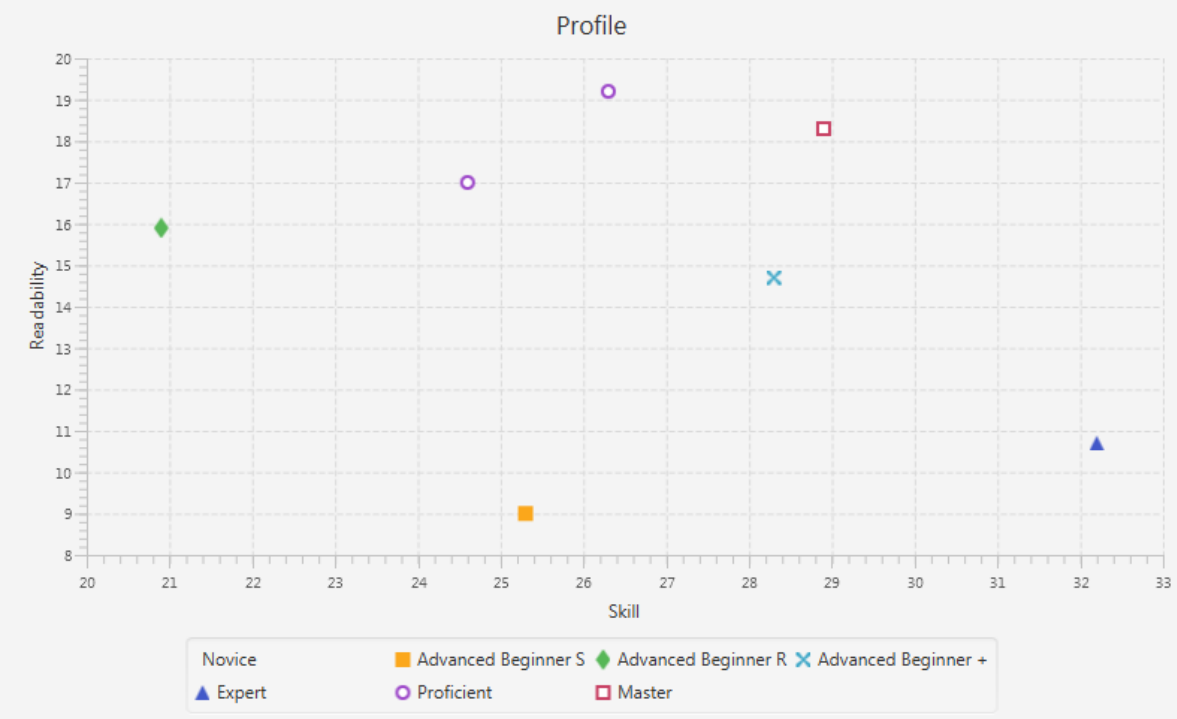


Figure 5.: Profile inference made for Exercise P1

Case Study 2

In Listings 5.4, 5.5 and 5.6 we can take a look at another set of solutions, for a different problem. In this case it was problem P2 (read N ages and print all above M, and also their average) and we will analyse three solutions.

```
1  import static java.lang.System.out;
2  import static java.lang.System.err;
3  import java.util.Scanner;
4
5  public class P2_A {
6      public static void main(String args[]){
7          int m = Integer.parseInt(args[0]),n=Integer.parseInt(args[1]),i,aux;
8          float average=0.0f;
9          String in; String[] nums;
10         Scanner input = new Scanner(System.in);
11         if (m <= 0 || n <=0) {
12             System.err.println("Both numbers have to be positive.");
13             return;
14         }
15         System.out.println("Insert the ages separated by spaces.");
16         in = input.nextLine();
17         nums=in.split(" ");
```

```
18          if (nums.length!=n) {
19              System.err.println("You must insert "+n+" ages.");
20              return;
21          }
22          System.out.printf("Ages above %d:\n",m);
23          for(i=0;i<n;i++) {
24              aux=Integer.parseInt(nums[i]);
25              if (aux>m) System.out.println(aux);
26              average+=aux;
27          }
28          System.out.println("Average: "+average/n);
29      }
30  }
```

Listing 5.4: "Solution to P2 made by A"

```
1  import java.util.Scanner;
2  public class P2_P {
3
4          public static void main(String[] args) {
5          Scanner read = new Scanner(System.in);
6          int n, m, age;
7          int total=0;
8          float average;
9
10         do {
11             System.out.println("Insert the number for comparing");
12             m = read.nextInt();
13         } while (m <=0);
14
15         do {
16             System.out.println("Insert the number of ages");
17             n = read.nextInt();
18         } while (n <= 0);
19
20         //Read the age values
21         for (int i = 0; i < n; i++) {
22             age = read.nextInt();
23
24             if (age > m) {
25                 System.out.println(age);
26             }
27
28             total=total+age;
29         }
30
31         //The division result must be casted to float!
```

```
32          average=(float)total/n;
33          System.out.println(average);
34      }
35  }
```

Listing 5.5: "Solution to P2 made by P"

```
1  import java.util.Scanner;
2
3  public class P2_Z {
4
5      public static void main(String[] args) {
6          Scanner in = new Scanner(System.in);
7          int threshold = in.nextInt(), i, ages = i = in.nextInt(), age, sum = 0;
8          while(i > 0){
9              sum += age = in.nextInt();
10             if(age > threshold)System.out.println(age);
11             i--;
12         }
13         //Double or float, according to the desired decimal digits precision
14         System.out.println(ages > 0 ? (double) sum/ages : 0.0);
15     }
16 }
```

Listing 5.6: "Solution to P2 made by Z"

Once again we can start by looking at the sizes of the three solutions. *Z's* solution is clearly the shortest one, which starts to prove his tendency to write concise and effective code. *A* and *P* on the other hand, wrote more extensive programs. But a closer look will reveal that, apart from the number of lines, their two solutions do not relate much.

There is a clear lack of readability in *A's* code. That is mainly due to the lack of empty lines in key zones of the program. *P's* implementation, with 68% of lines of code looks much cleaner in contrast to the 80% of *A*.

In regards to the use of control-flow statements, *Z* was once again the most economical, using only an if-condition, an iff-condition and a while-loop. *P* used three loops and a if-condition and *A* required 1 loop and 3 if-conditions.

This will make *Z* benefit once again from skill points.

The most remarkable differences among these three projects came, in fact, from the PMD analysis. In regards to skill-related violations, *P* and *Z* had the same number of violations (4). *A* on the other had 6, so it will impact the skill result.

As for readability-related violations, *A* had 2 more violations than *P*, which will have a big impact on the readability-level also. Those extra violations were:

- LocalVariableCouldBeFinal (**Skill**)
- OnlyOneReturn (**Skill** and **Readability**)
- PrematureDeclaration (**Skill**)
- IfStmtsMustUseBraces (**Readability**)

All these factors combined, made the following profile inference for these three solutions: *A* was classified as Novice, *Z* was classified as Expert, once again, and *P*, the OOP teacher was classified as Proficient. Like before, these classifications agree with what is known about the programmers and also to the manual assessment made.

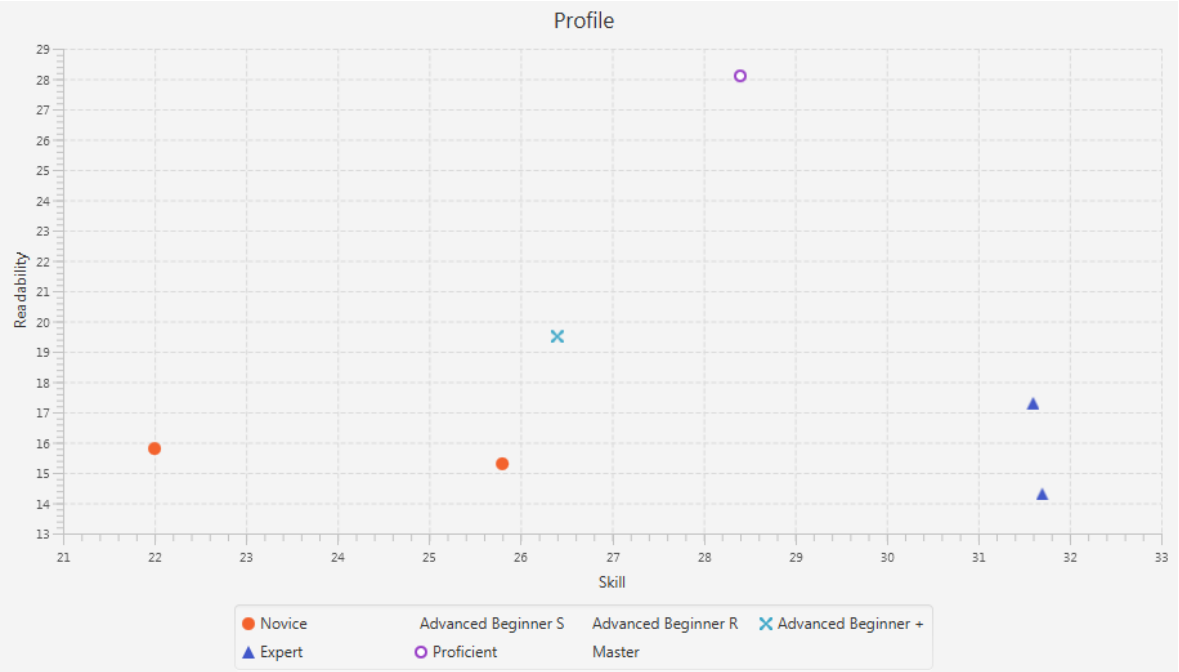


Figure 6.: Profile inference made for Exercise P2

Figure 6 shows an overall compare of several solutions to exercise P2. We can see *A* in the bottom-left corner with a (22, 15.8) score, *Z* in the bottom-right corner with a (31.6, 17.3) score, and finally *P* in the top-middle area with a (28.4, 28.1) score.

5.2.2 *Post-Development Results*

The second experiment conducted, consisted on the analysis of several dozens of solutions to the problems aforementioned, made by students ending the course of *Object-Oriented Programming* at Minho University. From here on, these students shall be referred as Std1 to Std69.

To that set of solutions we also added the ones we already had and used, throughout the development of the PP Tool.

Despite wanting to make the experiment as untampered as possible, some solutions had to be modified or even discarded. The main reason for changing files was that some students made all four exercises on the same file, and our tool requires each solution to be on each own file. The main reasons to discard some solutions was due to the fact that some students didn't understand the problem and their solution was not correct. Programs that used Java 8 constructs were also discarded because it's not supported by our tool.

After setting up all the solutions in the correct folder structure, they were all analysed using the PP Tool. The solutions were analysed and the results obtained for each exercise can be seen in the plots of Appendix A. The plot in Figure 7 shows the final profile inference for the four exercises combined.

Table 8 depicts all the profile inferences that were automatically calculated by the PP Tool. The first column identifies the programmer that solved the exercises. The following four columns represent the profile that was inferred to each programmer, by exercise (Listing 5.1). The *final* column displays the final profile inference by combining all results obtained from the exercises, as seen in Subsection 4.2.14.

	P1	**P2**	**A1**	**S1**	***Final***
P	P	E	P	P	***P***
D	E	AB +	-	AB +	***AB +***
G	AB +	-	AB +	-	**-**
V	E	-	E	AB-R	***E***
Z	E	E	E	-	***E***
A	AB-S	AB-R	AB-R	AB-R	***N***
S	AB-R	-	AB +	-	**-**
M	AB-R	-	E	AB +	***AB +***
F	E	AB +	AB +	AB-R	***AB-S***
J	-	E	-	-	**-**

Std1	-	-	-	P	**-**
Std2	AB-R	AB-S	-	AB-R	***N***
Std3	P	P	P	P	***P***
Std4	P	-	P	P	***P***
Std5	AB +	AB-S	AB-S	AB +	***AB-S***
Std6	AB +	AB-R	AB +	AB +	***AB-R***
Std7	-	M	P	-	**-**
Std8	AB +	AB-S	AB +	N	***N***
Std9	P	P	P	P	***P***
Std10	AB-R	AB +	AB +	E	***AB +***
Std11	E	AB-S	-	AB-S	***AB-S***
Std12	E	AB-S	AB-S	AB-S	***AB-S***
Std13	AB +	P	AB +	AB +	***AB +***
Std14	P	AB +	M	AB +	***P***
Std15	E	AB +	AB +	AB +	***AB +***
Std16	P	P	P	E	***P***
Std17	AB +	AB-R	AB-R	AB +	***AB-R***
Std18	AB-S	AB-S	AB-S	AB-S	***AB-S***
Std19	AB +	AB +	AB +	P	***AB +***
Std20	AB +	E	AB +	E	***E***
Std21	AB +	AB +	AB +	AB +	***AB +***
Std22	AB-R	E	AB-S	AB-S	***AB-S***
Std23	AB +	AB +	AB +	E	***E***
Std24	AB +	P	P	P	***P***
Std25	AB +	AB-S	AB-S	AB +	***AB +***
Std26	P	AB +	P	P	***P***
Std27	AB +	AB-S	AB-S	AB +	***AB +***
Std28	AB +	AB +	AB-S	E	***E***
Std29	E	AB-R	AB-S	AB-S	***AB-S***
Std30	P	P	P	P	***P***
Std31	AB +	E	AB +	E	***E***
Std32	E	AB +	AB +	AB +	***AB +***
Std33	AB +	P	P	P	***P***
Std34	P	P	P	AB +	***P***
Std35	-	-	AB +	AB +	**-**
Std36	P	P	P	P	***P***
Std37	AB +	-	-	N	**-**

Std38	AB +	P	AB +	AB +	***AB +***
Std39	AB +	-	P	AB +	***AB +***
Std40	AB +	E	AB +	AB-S	***AB +***
Std41	AB-R	AB-S	E	AB-S	***AB-S***
Std42	P	P	P	P	***P***
Std43	AB +	AB +	AB +	E	***AB +***
Std44	E	AB +	AB +	AB +	***AB +***
Std45	AB +	AB +	AB +	AB +	***AB +***
Std46	P	AB +	AB +	P	***P***
Std47	AB-S	AB-S	AB +	AB-S	***AB-S***
Std48	AB-S	E	AB-S	AB-S	***AB-S***
Std49	AB +	AB +	P	P	***AB +***
Std50	P	AB +	AB +	AB +	***AB +***
Std51	AB-S	E	AB-S	E	***E***
Std52	P	E	P	AB +	***P***
Std53	AB-R	AB-S	AB-S	AB-R	***N***
Std54	P	AB +	AB +	AB-S	***AB +***
Std55	P	AB +	AB +	AB +	***AB +***
Std56	AB-S	P	AB +	AB +	***AB +***
Std57	AB-S	AB-R	-	AB +	***AB-R***
Std58	-	P	P	P	***P***
Std59	N	AB-S	E	AB-S	***AB-S***
Std60	AB-S	AB-S	AB-S	E	***AB-S***
Std61	P	E	P	E	***AB +***
Std62	E	AB-S	AB-S	E	***E***
Std63	AB-S	-	AB-S	AB-S	***AB-S***
Std64	AB-S	AB-S	AB-S	AB +	***AB-S***
Std65	P	M	P	P	***P***
Std66	AB-S	P	AB +	AB +	***AB-R***
Std67	AB +	P	P	P	***P***
Std68	AB +	AB-R	AB-R	AB-R	***AB-R***
Std69	N	AB-S	AB-S	AB-S	***AB-S***

Table 8.: Final Results of Profile Inference using PP tool

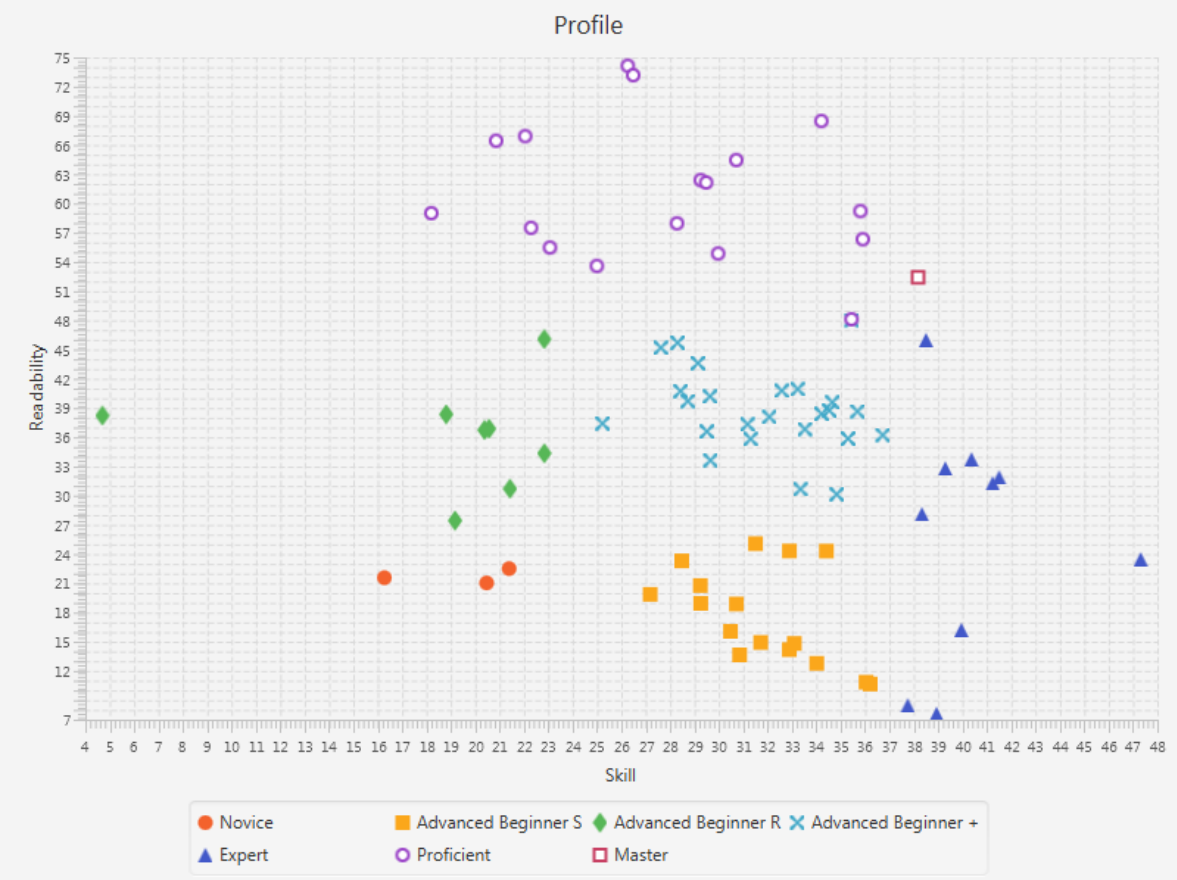


Figure 7.: Profile inference made with all four exercises combined

Looking at the final results on Table 8, in the majority of cases there seems to be a correlation between the profiles inferred in most of the exercises and the final profile inferred for all exercises combined. The most recurrent profile in the exercises seems to be the one inferred for that programmer.

The profiles calculated for the exercises (columns P1, P2, A1, and S1 of Table 8) were not directly used to infer the final profile for the given programmer. These profiles just act as "syntactic sugar" to increase the perception of how the programmer performed on those exercises. Only the raw numbers calculated in 4.2.10 were used to infer the final profile.

This explains the odd case of student *A*, who got profiled as Advanced Beginner in all individual exercises, but ended with Novice as the final profile. This means that, probably, although being classified with AB in all four exercises, the actual Skill and Readability values obtained were pretty low (probably near the boundaries of the Novice profile) which caused this profile to be inferred for student *A*. A manual assessment of the results obtained on each problem proves that assumption.

Another interesting result to analyse is the case of student *F*. The exercises profiles were quite diverse with E, AB+, AB+ and AB-R as the exercises profiles and the final profile being AB-S. If we look again at Figure 2, we can see that the profile AB+ occupies an average position on the plot (right in the middle). Profiles E and AB-R could cancel out, but that wouldn't explain the final AB-S profile inferred for *F*. The answer is that the Expert profile was very low in Readability, pulling what could be a final AB+ profile to an AB-S. A similar case to this one is student *Std66*.

Also regarding student *A*, notice that while in 5.2.1 *A* was profiled as Novice, in the general experiment for exercise P2 the profile inferred was Advanced Beginner R. That's due to the fact that in the first experiment there were only 6 solutions, and most of them were of experienced Java programmers. Later, in the second experiment, most of the solutions were of students of the same level of *A*, so in lack of better words, the standards were lowered, making the scores increase.

Here's the distribution of profiles assigned to the OOP students (Std1 - Std69), after completing the course.

- Novice: 3 (4.6%)
- Advanced Beginner (Readability): 5 (7.7%)
- Advanced Beginner (Skill): 14 (21.5%)
- Advanced Beginner (Both): 20 (30.8%)
- Proficient: 17 (26.2%)
- Expert: 6 (9.2%)
- Master: 0 (0%)

Looking at Figure 7 one thing easy to notice is that, unlike the inferences made for the exercises, we can see here clusters forming around the profile zones. This shows that programmers (mostly students in this case) tend to fall towards a profile when comparing several of their solutions.

Most of students were placed in the Average-to-High profiles, which is expected since these exercises were solved at the end of the year. The OOP professor was mainly proficient, which was also expected of him. The more advanced Java programmers were classified as Advanced Beginner + and Expert.

As a final step of this experiment, there was the intention of collecting the grades obtained on the OOP course by the students whose code we analysed. Comparing the grade

obtained by each student to the inferred profiles would be very useful to validate the results that the students obtained on the PP Tool. Unfortunately, that was not possible to do.

Nevertheless, we can compare the profiles obtained with the grades attributed in the course in a general way.

For this we collected the grades of the students that were taking this year's *OOP* course for the first time and have passed it (grades between 10 and 20).

- 10-12: 19 (20%)
- 13-15: 50 (52.6%)
- 16-18: 26 (27.4%)
- 19-20: 0 (0%)

The results obtained seem to be consistent with the grades attributed to the students. Most of the students fell on the average grades (13-15) and on the average profiles (Advanced Beginner). In the list bellow we aggregate the profiles to better illustrate this.

- N: 3 (4.6%)
- AB: 39 (60%)
- P/E: 23 (35.4%)
- M: 0 (0%)

Also, the manual assessments done on, not only students but also experienced Java programmers, seem to agree to the results obtained in the experiment.

6

CONCLUSION

In the beginning of this dissertation, it was presented a proposal to develop a system (called Programmer Profiler Tool) that allowed the profiling of a programmer through the static analysis of his source code. The hypothesis is that such profile inference is possible.

Currently, there is a working implementation that can be used to infer the profile of a programmer, thus proving the research hypothesis.

6.1 WORKING PLAN

To accomplish this master project, an iterative methodology based on literature revision, solution proposal, implementation, testing and experimentation was followed. The working plan followed to carry out this approach, was composed by the following steps:

1. Bibliographic search;
2. Reading and synthesis of the bibliography selected;
3. Definition of a set of profiles and selection of metrics that characterize them; identification of ways to extract data and evaluate those metrics, and a set of deduction rules to build the profile;
4. Development of a tool, based on AnTLR, that implements the metrics evaluation and the profiling; Integration of the PMD tool to improve such analysis;
5. Testing of the developed tool
6. Experimentation of the tool with real life examples to prove its effectiveness on the programmer profiling problem;

All proposed steps were successfully completed.

The PP Tool is open-source and is freely available on a GitHub repository[1]. It was developed in Java, with the IntelliJ IDEA IDE and the following support software:

- AnTLR v4.5.2
- PMD Tool v5.4.1
- AnTLR v4 grammar plugin for IntelliJ IDEA v1.8.1
- PMD plugin for IntelliJ IDEA v1.8
- Apache Commons IO v2.4
- Apache Commons Lang v3.4
- Google GSON v2.6.2
- OpenCSV v3.7

6.2 OUTCOMES

This dissertation's research hypothesis was whether it 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.

It uses static analysis to extract a set of metrics that we think are indicators of the coding style and capabilities of a programmer. It also uses an external tool, PMD, that detects bad coding practices and code smells, on source code.

Each one of the possible metrics and bad practices are already 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.

1 https://github.com/danielnovais92/ProgrammerProfiler/

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 then mapped to a set of previously defined programmer profiles, and thus the profile is inferred for each one of programmers.

The results are 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 expectations based on the subjects background and known capabilities as seen on Chapter 5, which leads us to state that the PP Tool can correctly infer the profile of Java programmers, whether the analysed subjects are beginner or more advanced Java programmers.

As part of this work there was also written a paper (Novais et al., 2016), that was published on the 2016 edition of SLATE [2] (*Symposium on Languages, Applications and Technologies*). The article covered Chapters 1, 2 and 3 of this dissertation. The rest of the work developed will be published on an upcoming paper.

6.3 Final Remarks

There are some issues that need to be addressed about the profiling and the PP Tool itself.

The first and most important is that, like any other profiling tool, the PP Tool is susceptible to error, and no result should be taken as absolute. The tool, although functional, is still in a very beta phase, and is very prone to exploits that could interfere the profile results. An example of this is that, although the tool works under the assumption that the exercises are correctly solved (from an input/output standpoint), a programmer with knowledge of the mechanisms used by the PP tool could devise a solution to an exercise that may trick the tool and infer him a higher profile.

Another limitation of the tool, is scalability of the exercises analysed. The PP tool works well with small exercises (that can be solved in a few methods), but if we try to scale this to bigger and more robust projects, the tool will most likely fail. That's because with larger projects, the frequency of certain language constructs or usage of programming styles, stops making sense due to software design choices being more relevant. For example, larger

2 http://slate-conf.org/2016/home

Java programs will likely use the OOP paradigm, which largely increases the methods declaration with, for instance, the getters and setters.

Another issue worth mentioning, is that the profiles do not intend to reflect the full extensive capabilities of a programmer. Although the profiles we infer are also the adjectives we use to classify programmers, it's important to establish that, with the small number of exercises analysed by the tool, the final profiles are a reflection of the set of capabilities that a programmer must have to solve the exercises analysed. Other knowledge that the programmer may have will, obviously, not be considered.

Despite that, the profiles inferred strongly correlate to what was expected of the programmers (at least to the ones whose background was familiar) which leads us to believe that the tool tends to be correct. After all, in most fields, the subject evaluation is usually made targeting only a subset of the knowledge involved.

For example, a skillful programmer could opt to make a more extensive solution because he knows that in the end, the compiler will optimize is code to have the same efficiency of a less intuitive but more effective version.

Finally, the context in which a program is written should be also taken into consideration. For instance, a teacher that writes code in a safely and extensive manner while teaching students, may prioritize code efficiency in personal projects.

6.4 FUTURE WORK

Although the general idea was not new, much of the work developed during this dissertation (the theory, the algorithms and the tool) was a new approach to the programmer profiling problem. Because of that, it's understandable that many ideas were not possible to explore due to time restrictions.

Those ideas, that could help improve the work done so far are left here as possible future work for someone that has interest in taking the PP Tool to the next level.

6.4.1 *Theory*

In this section, the work that probably could be improved is the scale used to infer the profile of the programmer based on the skill and readability values.

Inference Plot Areas

A closer look at Figure 2 will reveal some potential flaws. The most obvious is that there are areas in the plot that are very close to four different profiles, and looking only at profile

inference could give a biased idea of the profile that was inferred for a given programmer. The most obvious cases of this are the area where the profiles AB+, P, E and M meet up. A mere Skill or Readability point could dictate the difference between an Advanced Beginner or Master profile inference. The similar happens between the N, AB-R, AB-S and AB+ profiles.

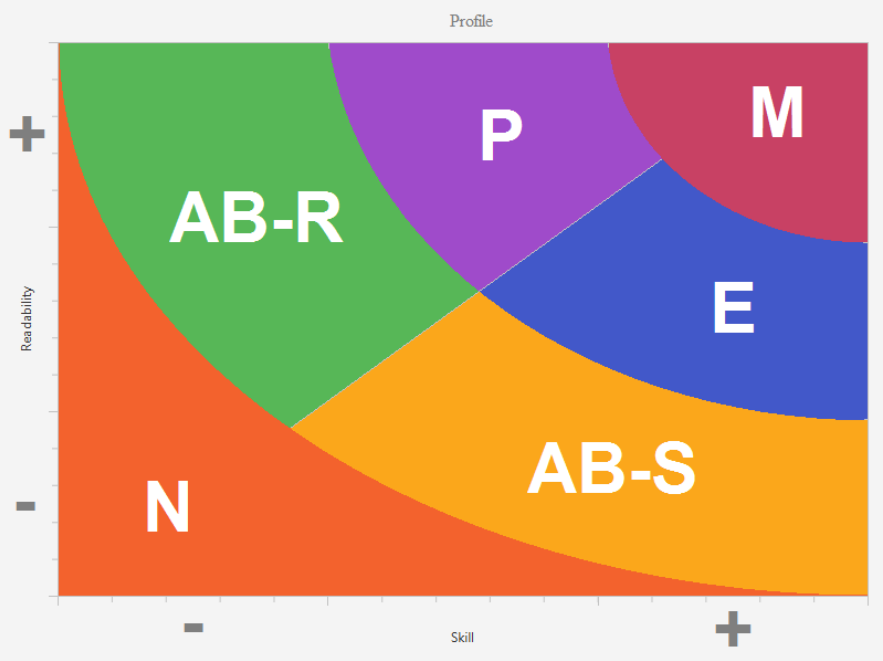


Figure 8.: Possible future implementation of correlation between scores and profiles

A proposed solution for this problem is shown in Figure 8. Here we reduce the zones where several areas meet up, which could lead to less biased profiles. Another advantage of using a plot like this is that it's overall more just. For example, in the current version of the implementation (seen in Figure 2), programmers with very low Skill and very high Readability are classified as Proficient. That's not very fair and does not agree at all with out initial profile definition (Section 3.2) where we say that Proficient still show a lot of language knowledge (although focusing more on code readability). This new solution would make it harder to reach the top profiles (Proficient, Expert and Master) and would also remove the profile Advanced Beginner +, which is not a very useful after all.

On a implementation point of view, this solution would require a more complex mapping from the scores to the profile that the one in the current implementation, due to the curves the profiles have on the plot.

6.4.2 *Algorithms*

Regarding the algorithms used for the profile inference, the biggest issue with them is that they are a little too heuristic. By that we mean that they resort to simple mathematical techniques like averaging, inverting and normalising values in order to make the calculations that will later be used to infer the profiles.

A suggestion for future work is to investigate existing techniques for profiling that could probably make better calculations with the metrics used by our tool.

6.4.3 *Tool*

Regarding the tool itself, it's where we find more room for improvements.

Language Support

The first point to address is the lack of Java8 support. One of its main new features are the lambda expressions. These expressions facilitate the programming in the functional paradigm, and make code a lot cleaner and easier to understand. Adding support to Java8 would help on the differentiation of Novice and Advanced programmers. Besides being quite a new feature of Java it's also something that, at least for now, is usually not taught in beginner Java courses.

Skill Metrics

Besides the lambda expressions there are other programming features that influence the Skill scores and currently are not being identified by the PP Tool. The main one is defensive programming.

Defensing programming is intended to create robust software by design. That usually means programming for every scenario that an application could possible undergo. In the context of the small examples we analysed in this dissertation, we can find defensive programming in the validation of the inputs passed by the user, usually in if or while conditions. Doing these validations will cause a reduction on skill-related metrics because it will increase the global number of control flow statements (and statements in general). So, in short, the programmer will be penalized for using something that should increase his score. As possible future work, there could be an investigation on techniques that help identify the use of defensive programming to properly reward the programmers.

Readability Metrics

The Readability group is clearly the most problematic.The metrics that relate to this group can sometimes be misleading. Indentation is probably the most important aspect of readable code, but the PP Tool does not take that into consideration. Adding that support would make the tool much better at classifying code in terms of Readability. Another feature that could be added is the analysis of variable, method and class naming. For example, using full words as variable names is much more meaningful than just using letters.

Weighted Exercises

Lastly, a feature that could also help the PP Tool is attributing weights to exercises. That would mean that some exercises could be more important (or harder do solve) than others and make an improved final score when combining the scores the programmers got in each exercise to infer the final profile.

6.5 END NOTE

This last section concludes the dissertation. We hope this was an interesting and pleasant reading, and that this dissertation helps continuing the research on the programmer profiling subject, which is still taking its first steps in a exponential growing world of programmers.

BIBLIOGRAPHY

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.

Sylviane Granger and Paul Rayson. Automatic profiling of learner texts. *Learner English on computer*, pages 119–131, 1998.

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.

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.

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.

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.

Steve McConnell. *Code complete*. Pearson Education, 2004.

Daniel Novais, Maria João Pereira, and Pedro Rangel Henriques. Profile detection through source code static analysis. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*, volume 51, pages 1–13. OASICS, 2016.

Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

Emília Pietriková and Sergej Chodarev. Profile-driven source code exploration. In *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, pages 929–934. IEEE, 2015.

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.

Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010.

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

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.

A

FINAL PROFILE INFERENCES

The following figures contain the final results obtained by the developed PP Tool. Four exercises were distributed to several dozens of programmers (most of them were students learning OOP) the their solutions were analysed by the tool producing this results. For more information, see Chapter 5.

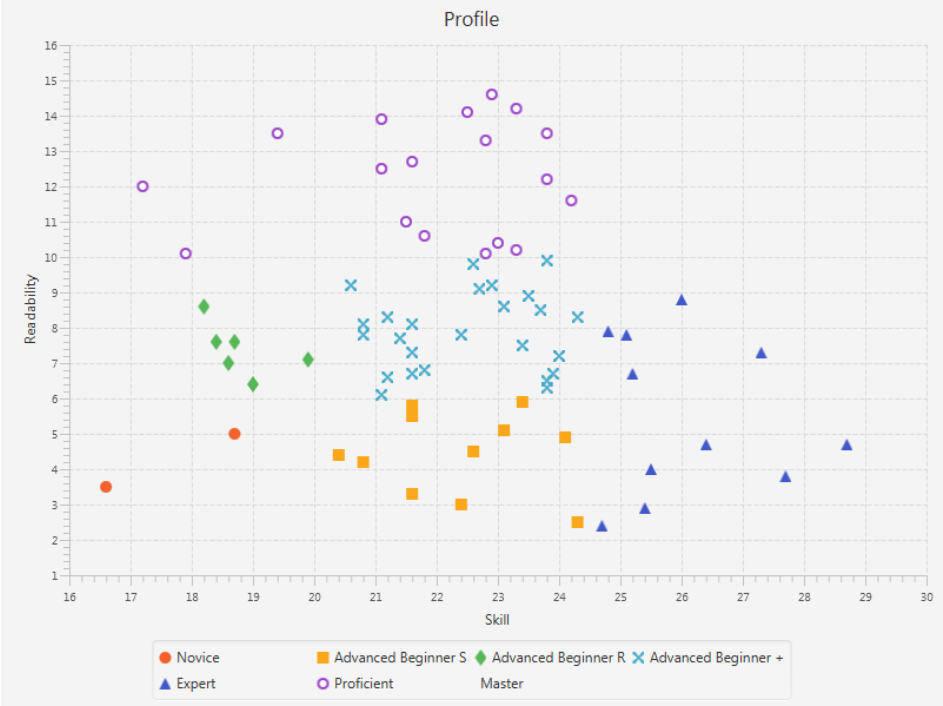


Figure 9.: Profile inference made for Exercise P1

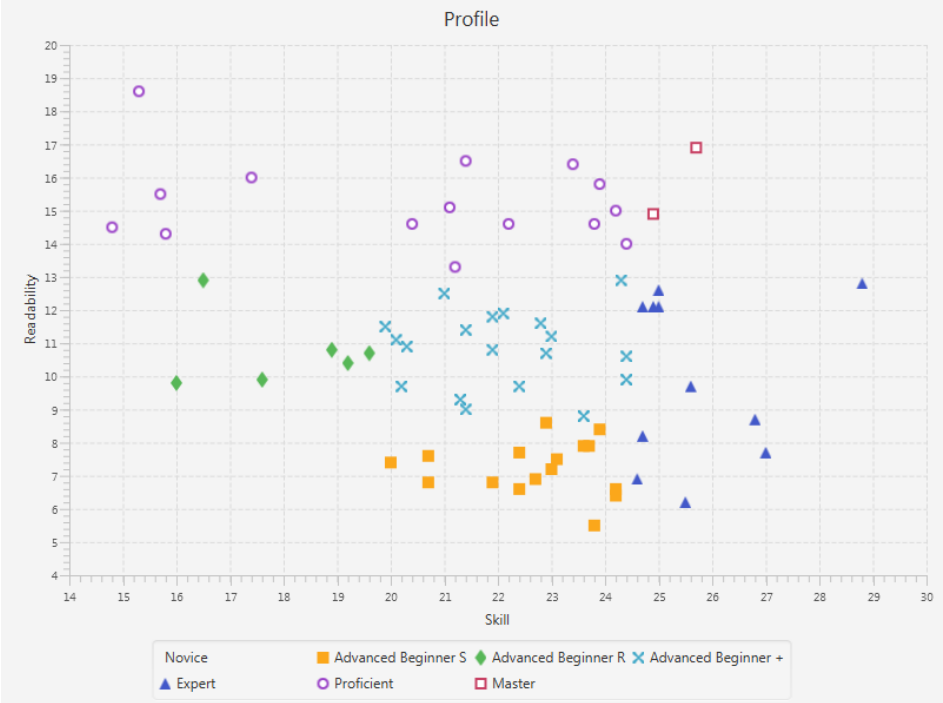


Figure 10.: Profile inference made for Exercise P2

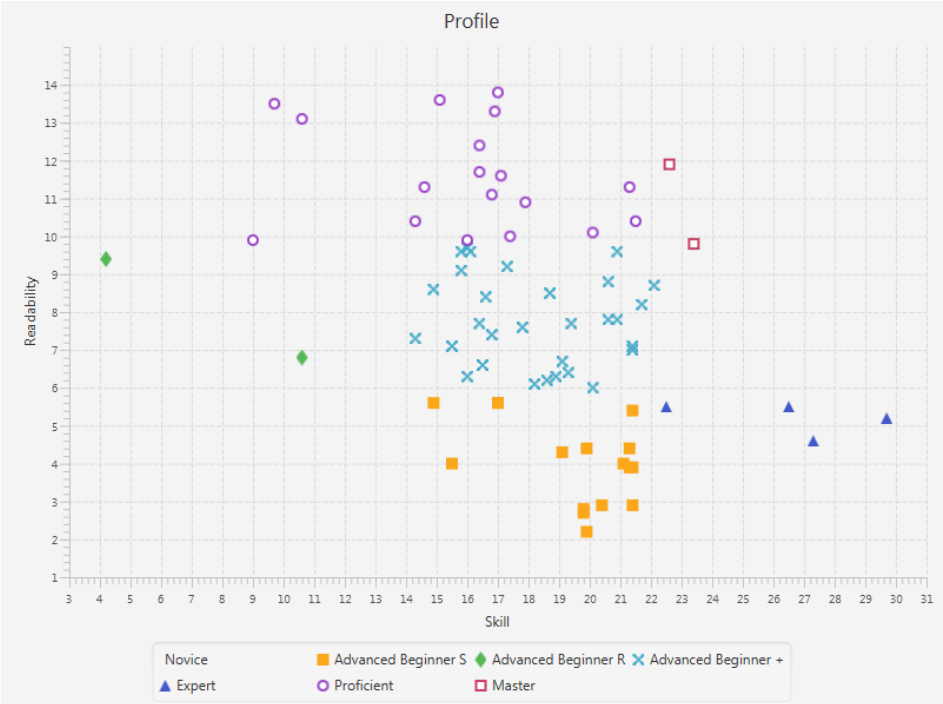


Figure 11.: Profile inference made for Exercise A1

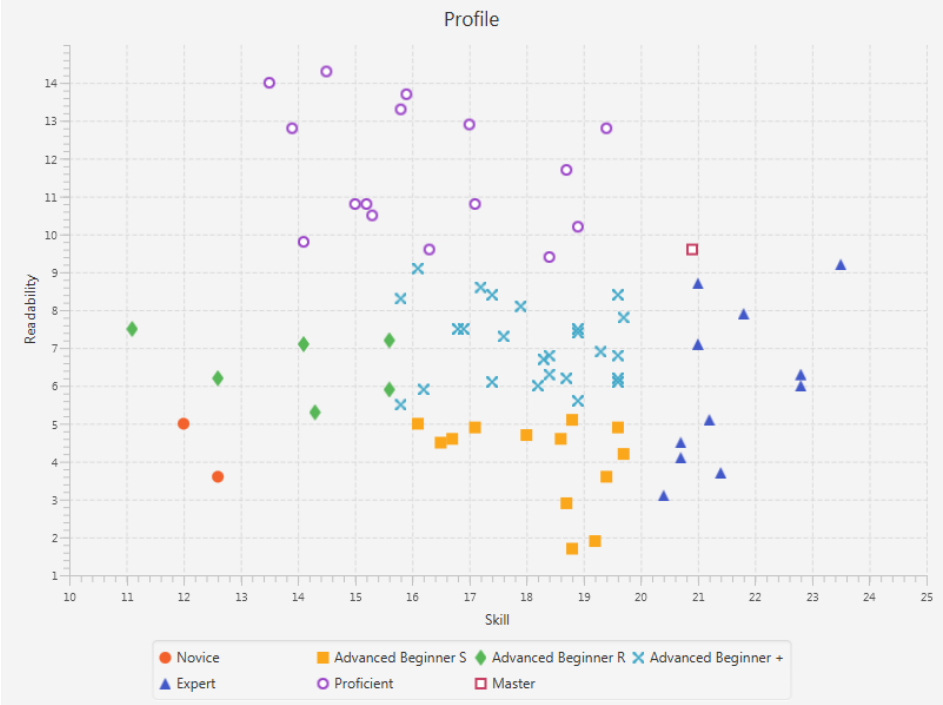


Figure 12.: Profile inference made for Exercise S1

B

PMD RULES

The following tables depict all the Java PMD rules that are relevant for the PP Tool. This is integral information obtained from the PMD website[1]. This is directly related to Subsection 4.1.2.

Rule	Description	Priority	Group
Jumbled Incrementer	Avoid jumbled loop incrementers - its usually a mistake, and is confusing even if intentional.	3	B
ForLoop Should Be WhileLoop	Some for loops can be simplified to while loops, this makes them more concise.	3	R
Override Both Equals AndHash code	Override both public boolean Object.equals(Object other), and public int Object.hashCode(), or override neither. Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your superclass.	3	S
Double Checked Locking	Partially created objects can be returned by the Double Checked Locking pattern when used in Java.An optimizing JRE may assign a reference to the baz variable before it creates the object thereference is intended to point to.	1	S
Return From Finally Block	Avoid returning from a finally block, this can discard exceptions.	3	S

Table 9.: Basic PMD Ruleset - I

1 https://pmd.github.io/pmd-5.5.1/pmd-java/rules/index.html

Unconditional IfStatement	Do not use 'if' statements whose conditionals are always true or always false.	3	S
Boolean Instantiation	Avoid instantiating Boolean objects; you can reference Boolean.TRUE, Boolean.FALSE, or call Boolean.valueOf() instead.	2	S
Collapsible IfStatements	Sometimes two consecutive 'if' statements can be consolidated by separating their conditions with a boolean short-circuit operator.	3	B
Class Cast Exception With ToArray	When deriving an array of a specific class from your Collection, one should provide an array ofthe same class as the parameter of the toArray() method. Doing otherwise you will will resultin a ClassCastException.	3	S
Avoid Decimal Literals In BigDecimal Constructor	One might assume that the result of 'new BigDecimal(0.1)" is exactly equal to 0.1, but it is actuallyequal to .1000000000000000055511 1231257827021181583404541015 62	3	S
Misplaced Null Check	The null check here is misplaced. If the variable is null a NullPointerException will be thrown.Either the check is useless (the variable will never be 'null') or it is incorrect.	3	S
Avoid Thread Group	Avoid using java.lang.ThreadGroup; although it is intended to be used in a threaded environmentit contains methods that are not thread-safe.	3	S
Broken Null Check	The null check is broken since it will throw a NullPointerException itself.It is likely that you used || instead of && or vice versa.	2	S
BigInteger Instantiation	Don't create instances of already existing BigInteger (BigInteger.ZERO, BigInteger.ONE) andfor Java 1.5 onwards, BigInteger.TEN and BigDecimal (BigDecimal.ZERO, BigDecimal.ONE, BigDecimal.TEN)	3	S
Avoid Using Octal Values	Integer literals should not start with zero since this denotes that the rest of literal will beinterpreted as an octal value.	3	S
Avoid Using Hard Coded IP	Application with hard-coded IP addresses can become impossible to deploy in some cases.Externalizing IP adresses is preferable.	3	S

Table 10.: Basic PMD Ruleset - II

Check Result Set	Always check the return values of navigation methods (next, previous, first, last) of a ResultSet.If the value return is 'false', it should be handled properly.	3	S
Avoid Multiple Unary Operators	The use of multiple unary operators may be problematic, and/or confusing.Ensure that the intended usage is not a bug, or consider simplifying the expression.	2	B
Extends Object	No need to explicitly extend Object.	4	S
Check Skip Result	The skip() method may skip a smaller number of bytes than requested. Check the returned value to find out if it was the case or not.	3	S
Avoid Branching Statement As Last In Loop	Using a branching statement as the last part of a loop may be a bug, and/or is confusing.Ensure that the usage is not a bug, or consider using another approach.	2	B
Dont Call Thread Run	Explicitly calling Thread.run() method will execute in the caller's thread of control. Instead, call Thread.start() for the intended behavior.	4	S
Dont Use Float Type For Loop Indices	Don't use floating point for loop indices. If you must use floating point, use doubleunless you're certain that float provides enough precision and you have a compellingperformance need (space or time).	3	S
Simplified Ternary	Look for ternary operators with the form condition ? literalBoolean : fooor condition ? foo : literalBoolean.These expressions can be simplified respectively tocondition || foo when the literalBoolean is true!condition && foo when the literalBoolean is falseor!condition || foo when the literalBoolean is truecondition && foo when the literalBoolean is false	3	S

Table 11.: Basic PMD Ruleset - III

Rule	Description	Priority	Group
IfStmts Must Use Braces	Avoid using if statements without using braces to surround the code block. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.	3	R
WhileLoops Must Use Braces	Avoid using while statements without using braces to surround the code block. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.	3	R
IfElse Stmts Must Use Braces	Avoid using if..else statements without using surrounding braces. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.	3	R
ForLoops Must Use Braces	Avoid using 'for' statements without using curly braces. If the code formatting or indentation is lost then it becomes difficult to separate the code being controlled from the rest.	3	R

Table 12.: Braces PMD Ruleset

Rule	Description	Priority	Group
NPath Complexity	The NPath complexity of a method is the number of acyclic execution paths through that method.A threshold of 200 is generally considered the point where measures should be taken to reduce complexity and increase readability.	3	S
Excessive Method Length	When methods are excessively long this usually indicates that the method is doing more than its name/signature might suggest. They also become challenging for others to digest since excessive scrolling causes readers to lose focus.Try to reduce the method length by creating helper methods and removing any copy-/pasted code.	3	R
Excessive Parameter List	Methods with numerous parameters are a challenge to maintain, especially if most of them share the same datatype. These situations usually denote the need for new objects to wrap the numerous parameters.	3	R
Excessive Class Length	Excessive class file lengths are usually indications that the class may be burdened with excessive responsibilities that could be provided by external classes or functions. In breaking these methods apart the code becomes more manageable and ripe for reuse.	3	R
Cyclomatic Complexity	Complexity directly affects maintenance costs is determined by the number of decision points in a method plus one for the method entry. The decision points include 'if', 'while', 'for', and 'case labels' calls. Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denotehigh complexity, and 11+ is very high complexity.	3	B

Table 13.: Code Size PMD Ruleset - I

Std Cyclomatic Complexity	Complexity directly affects maintenance costs is determined by the number of decision points in a method plus one for the method entry. The decision points include 'if', 'while', 'for', and 'case labels' calls. Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denotehigh complexity, and 11+ is very high complexity.	3	B
Modified Cyclomatic Complexity	Complexity directly affects maintenance costs is determined by the number of decision points in a method plus one for the method entry. The decision points include 'if', 'while', 'for', and 'case labels' calls. Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denotehigh complexity, and 11+ is very high complexity.	3	B
Excessive Public Count	Classes with large numbers of public methods and attributes require disproportionate testing effortssince combinational side effects grow rapidly and increase risk.	3	R
Too Many Fields	Classes that have too many fields can become unwieldy and could be redesigned to have fewer fields,possibly through grouping related fields in new objects. For example, a class with individual city/state/zip fields could park them within a single Address field.	3	R
Ncss Method Count	This rule uses the NCSS algorithm to determine the number of linesof code for a given method. NCSS ignores comments, and counts actual statements.	3	R
Ncss Type Count	This rule uses the NCSS algorithm to determine the number of linesof code for a given type. NCSS ignores comments, and counts actual statements.	3	R
Ncss Constructor Count	This rule uses the NCSS algorithm to determine the number of lines of code for a given constructor. NCSS ignores comments, and counts actual statements.	3	R
Too Many Methods	A class with too many methods is probably a good suspect for refactoring, in order to reduce its complexity and find a way tohave more fine grained objects.	3	B

Table 14.: Code Size PMD Ruleset - II

Rule	Description	Priority	Group
Comment Required	Denotes whether comments are required (or unwanted) for specific language elements.	3	R
Comment Size	Determines whether the dimensions of non-header comments found are within the specified limits.	3	R
Comment Content	A rule for the politically correct. We don't want to offend anyone.	3	R
Comment Default Access Modifier	To avoid mistakes if we want that a Method, Field or Nested class have a default access modifier we must add a comment at the beginning of the Method, Field or Nested class. By default the comment must be /* default */, if you want another, you have to provide a regex.	1	R

Table 15.: Comments PMD Ruleset

Rule	Description	Priority	Group
Unnecessary Constructor	This rule detects when a constructor is not necessary; i.e., when there is only one constructor,its public, has an empty body, and takes no arguments.	3	S
Null Assignment	Assigning a 'null' to a variable (outside of its declaration) is usually bad form. Sometimes, this typeof assignment is an indication that the programmer doesn't completely understand what is going on in the code. NOTE: This sort of assignment may used in some cases to dereference objects and encourage garbage collection.	3	S
Only One Return	A method should have only one exit point, and that should be the last statement in the method.	3	B
Assignment In Operand	Avoid assignments in operands; this can make code more complicated and harder to read.	3	R
At Least One Constructor	Each class should declare at least one constructor.	3	S
Dont Import Sun	Avoid importing anything from the 'sun.*' packages. These packages are not portable and are likely to change.	4	S
Suspicious Octal Escape	A suspicious octal escape sequence was found inside a String literal.	3	S
Call Super In Constructor	It is a good practice to call super() in a constructor. If super() is not called but another constructor (such as an overloaded constructor) is called, this rule will not report it.	3	S
Unnecessary Parentheses	Sometimes expressions are wrapped in unnecessary parentheses, making them look like function calls.	3	R
Default Package	Use explicit scoping instead of accidental usage of default package private level.The rule allows methods and fields annotated with Guava's @VisibleForTesting.	3	S

Table 16.: Controversial PMD Ruleset - I

Dataflow Anomaly Analysis	The dataflow analysis tracks local definitions, undefinitions and references to variables on different paths on the data flow.From those informations there can be found various problems.1. UR - Anomaly: There is a reference to a variable that was not defined before. This is a bug and leads to an error.2. DU - Anomaly: A recently defined variable is undefined. These anomalies may appear in normal source text.3. DD - Anomaly: A recently defined variable is redefined. This is ominous but don't have to be a bug.	5	S
Avoid Final Local Variable	Avoid using final local variables, turn them into fields.	3	S
Avoid Using Short Type	Java uses the 'short' type to reduce memory usage, not to optimize calculation. In fact, the JVM does not have any arithmetic capabilities for the short type: the JVM must convert the short into an int, do the proper calculation and convert the int back to a short. Thus any storage gains found through use of the 'short' type may be offset by adverse impacts on performance.	1	S
Avoid Using Volatile	Use of the keyword 'volatile' is generally used to fine tune a Java application, and therefore, requiresa good expertise of the Java Memory Model. Moreover, its range of action is somewhat misknown. Therefore,the volatile keyword should not be used for maintenance purpose and portability.	2	S
Avoid Using Native Code	Unnecessary reliance on Java Native Interface (JNI) calls directly reduces application portability and increases the maintenance burden.	2	S
Avoid Accessibility Alteration	Methods such as getDeclaredConstructors(), getDeclaredConstructor(Class[]) and setAccessible(),as the interface PrivilegedAction, allows for the runtime alteration of variable, class, ormethod visibility, even if they are private. This violates the principle of encapsulation.	3	S

Table 17.: Controversial PMD Ruleset - II

Do Not Call Garbage Collection Explicitly	Calls to System.gc(), Runtime.getRuntime().gc(), and System.runFinalization() are not advised. Code should have thesame behavior whether the garbage collection is disabled using the option -Xdisableexplicitgc or not.Moreover, 'modern' jvms do a very good job handling garbage collections. If memory usage issues unrelated to memory leaks develop within an application, it should be dealt with JVM options rather than within the code itself.	2	S
One Declaration Per Line	Java allows the use of several variables declaration of the same type on one line. However, itcan lead to quite messy code. This rule looks for several declarations on the same line.	3	R
Avoid Prefixing Method Parameters	Prefixing parameters by 'in' or 'out' pollutes the name of the parameters and reduces code readability.To indicate whether or not a parameter will be modify in a method, its better to document methodbehavior with Javadoc.	4	R
Avoid Literals In IfCondition	Avoid using hard-coded literals in conditional statements. By declaring them as static variablesor private members with descriptive names maintainability is enhanced. By default, the literals '-1' and '0' are ignored.More exceptions can be defined with the property 'ignoreMagicNumbers'.	3	B
Use Object For Clearer API	When you write a public method, you should be thinking in terms of an API. If your method is public, it means other class will use it, therefore, you want (or need) to offer a comprehensive and evolutive API. If you pass a lot of information as a simple series of Strings, you may think of using an Object to represent all those information.	3	B
Use Concurrent HashMap	Since Java5 brought a new implementation of the Map designed for multi-threaded access, you can perform efficient map reads without blocking other threads.	3	S

Table 18.: Controversial PMD Ruleset - III

Rule	Description	Priority	Group
Use Utility Class	For classes that only have static methods, consider making them utility classes. Note that this doesn't apply to abstract classes, since their subclasses maywell include non-static methods. Also, if you want this class to be a utility class,remember to add a private constructor to prevent instantiation.(Note, that this use was known before PMD 5.1.0 as UseSingleton)	3	S
Simplify Boolean Returns	Avoid unnecessary if-then-else statements when returning a boolean. The result ofthe conditional test can be returned instead.	3	S
Simplify Boolean Expressions	Avoid unnecessary comparisons in boolean expressions, they serve no purpose and impacts readability.	3	B
SwitchStmts Should Have Default	All switch statements should include a default option to catch any unspecified values.	3	S
Avoid Deeply Nested IfStmts	Avoid creating deeply nested if-then statements since they are harder to read and error-prone to maintain.	3	B
Avoid Reassigning Parameters	Reassigning values to incoming parameters is not recommended. Use temporary local variables instead.	2	B
Switch Density	A high ratio of statements to labels in a switch statement implies that the switch statement is overloaded. Consider moving the statements into new methods or creating subclasses based on the switch variable.	3	R
Constructor Calls Overridable Method	Calling overridable methods during construction poses a risk of invoking methods on an incompletely constructed object and can be difficult to debug.It may leave the sub-class unable to construct its superclass or forced to replicate the construction process completely within itself, losing the ability to call super().	1	S

Table 19.: Design PMD Ruleset - I

Accessor Class Generation	Instantiation by way of private constructors from outside of the constructor's class often causes the generation of an accessor. A factory method, or non-privatization of the constructor can eliminate this situation. The generated class file is actually an interface. It gives the accessing class the ability to invoke a new hidden package scope constructor that takes the interface as a supplementary parameter. This turns a private constructor effectively into one with package scope, and is challenging to discern.	3	S
Final Field Could Be Static	If a final field is assigned to a compile-time constant, it could be made static, thus saving overhead in each object at runtime.	3	S
Close Resource	Ensure that resources (like Connection, Statement, and ResultSet objects) are always closed after use.	3	S
Non Static Initializer	A non-static initializer block will be called any time a constructor is invoked (just prior to invoking the constructor). While this is a valid language construct, it is rarely used and is confusing.	3	B
Default Label Not Last In SwitchStmt	By convention, the default label should be the last label in a switch statement.	3	B
Non Case Label In Switch Statement	A non-case label (e.g. a named break/-continue label) was present in a switch statement.This legal, but confusing. It is easy to mix up the case labels and the non-case labels.	3	R
Optimizable ToArray Call	Calls to a collection's toArray() method should specify target arrays sized to match the size of thecollection. Initial arrays that are too small are discarded in favour of new ones that have to be createdthat are the proper size.	3	S
Bad Comparison	Avoid equality comparisons with Double.NaN. Due to the implicit lack of representationprecision when comparing floating point numbers these are likely to cause logic errors.	3	S
Equals Null	Tests for null should not use the equals() method. The '==' operator should be used instead.	1	S

Table 20.: Design PMD Ruleset - II

Confusing Ternary	Avoid negation within an 'if' expression with an 'else' clause. For example, rephrase: if (x != y) diff(); else same();as: if (x == y) same(); else diff();Most 'if (x != y)" cases without an 'else' are often return cases, so consistent use of this rule makes the code easier to read. Also, this resolves trivial ordering problems, suchas 'does the error case go first?" or 'does the common case go first?'.	3	R
Instantiation To Get Class	Avoid instantiating an object just to call getClass() on it; use the .class public member instead.	4	S
Idempotent Operations	Avoid idempotent operations - they have no effect.	3	B
Simple Date Format Needs Locale	Be sure to specify a Locale when creating SimpleDateFormat instances to ensure that locale-appropriateformatting is used.	3	S
Immutable Field	Identifies private fields whose values never change once they are initialized either in the declaration of the field or by a constructor. This helps in converting existing classes to becoming immutable ones.	3	B
Use Locale With Case Conversions	When doing String.toLowerCase()/toUpperCase() conversions, use Locales to avoids problems with languages thathave unusual conventions, i.e. Turkish.	3	S
Avoid Protected Field In Final Class	Do not use protected fields in final classes since they cannot be subclassed.Clarify your intent by using private or package access modifiers instead.	3	S
Assignment To Non Final Static	Identifies a possible unsafe usage of a static field.	3	S
Missing Static Method In Non Instantiatable Class	A class that has private constructors and does not have any static methods or fields cannot be used.	3	S
Avoid Synchronized At Method Level	Method-level synchronization can cause problems when new code is added to the method. Block-level synchronization helps to ensure that only the code that needs synchronization gets it.	3	S
Missing Break In Switch	Switch statements without break or return statements for each case option-may indicate problematic behaviour. Empty cases are ignored as these indicate an intentional fall-through.	3	S

Table 21.: Design PMD Ruleset - III

Use Notify All Instead Of Notify	Thread.notify() awakens a thread monitoring the object. If more than one thread is monitoring, then onlyone is chosen. The thread chosen is arbitrary; thus its usually safer to call notifyAll() instead.	3	S
Avoid Instanceof Checks In Catch-Clause	Each caught exception type should be handled in its own catch clause.	3	B
Abstract Class Without Abstract Method	The abstract class does not contain any abstract methods. An abstract class suggests an incomplete implementation, which is to be completed by subclasses implementing theabstract methods. If the class is intended to be used as a base class only (not to be instantiateddirectly) a protected constructor can be provided prevent direct instantiation.	3	B
Simplify Conditional	No need to check for null before an instanceof; the instanceof keyword returns false when given a null argument.	3	S
Compare Objects With Equals	Use equals() to compare object references; avoid comparing them with ==.	3	S
Position Literals First In Comparisons	Position literals first in comparisons, if the second argument is null then NullPointerExceptions can be avoided, they will just return false.	3	S
Position Litera ls First In Case Insensitive Comparisons	Position literals first in comparisons, if the second argument is null then NullPointerExceptions can be avoided, they will just return false.	3	S
Unnecessary Local Before Return	Avoid the creation of unnecessary local variables	3	S
Non Thread Safe Singleton	Non-thread safe singletons can result in bad state changes. Eliminate static singletons if possible by instantiating the object directly. Staticsingletons are usually not needed as only a single instance exists anyway.Other possible fixes are to synchronize the entire method or to use an initialize-on-demand holder class (do not use the double-check idiom).	3	S
Single Method Singleton	Some classes contain overloaded getInstance. The problem with overloaded getInstance methodsis that the instance created using the overloaded method is not cached and so, for each call and new objects will be created for every invocation.	2	S

Table 22.: Design PMD Ruleset - IV

Singleton Class Returning New Instance	Some classes contain overloaded getInstance. The problem with overloaded getInstance methodsis that the instance created using the overloaded method is not cached and so, for each call and new objects will be created for every invocation.	2	S
Uncommented Empty Method Body	Uncommented Empty Method Body finds instances where a method body does not containstatements, but there is no comment. By explicitly commenting empty method bodiesit is easier to distinguish between intentional (commented) and unintentionalempty methods.	3	R
Uncommented Empty Constructor	Uncommented Empty Constructor finds instances where a constructor does notcontain statements, but there is no comment. By explicitly commenting empty constructors it is easier to distinguish between intentional (commented)and unintentional empty constructors.	3	R
Avoid Constants Interface	An interface should be used only to characterize the external behaviour of an implementing class using an interface as a container of constants is a poor usage pattern and not recommended.	3	S
Unsynchronized Static Date Formatter	SimpleDateFormat instances are not synchronized. Sun recommends using separate format instances for each thread. If multiple threads must access a static formatter, the formatter must be synchronized either on method or block level.	3	S
Preserve Stack Trace	Throwing a new exception from a catch block without passing the original exception into the new exception will cause the original stack trace to be lost making it difficult to debug effectively.	3	S
Use Collection Is Empty	The isEmpty() method on java.util.Collection is provided to determine if a collection has any elements.Comparing the value of size() to 0 does not convey intent as well as the isEmpty() method.	3	S
Class With Only Private Constructors Should Be Final	A class with only private constructors should be final, unless the private constructor is invoked by a inner class.	1	S

Table 23.: Design PMD Ruleset - V

Empty Method In Abstract Class Should Be Abstract	Empty methods in an abstract class should be tagged as abstract. This helps to remove their inappropriate usage by developers who should be implementing their own versions in the concrete subclasses.	1	S
Singular Field	Fields whose scopes are limited to just single methods do not rely on the containing object to provide them to other methods. They may be better implemented as local variables within those methods.	3	S
Return Empty Array Rather Than Null	For any method that returns an array, it is a better to return an empty array rather than a null reference. This removes the need for null checking all results and avoids inadvertentNullPointerExceptions.	1	S
Abstract Class Without Any Method	If an abstract class does not provides any methods, it may be acting as a simple data container that is not meant to be instantiated. In this case, it is probably better to use a private or protected constructor in order to prevent instantiation than make the class misleadingly abstract.	1	S
Too Few Branches For A SwitchStatement	Switch statements are indented to be used to support complex branching behaviour. Using a switch for only a few cases is ill-advised, since switches are not as easy to understand as if-then statements. In these cases use theif-then statement to increase code readability.	3	R
Logic Inversion	Use opposite operator instead of negating the whole expression with a logic complement operator.	3	B
Use Varargs	Java 5 introduced the varargs parameter declaration for methods and constructors. This syntactic sugar provides flexibility for users of these methods and constructors, allowing them to avoid having to deal with the creation of an array.	4	S
Field Declarations Should Be At Start Of Class	Fields should be declared at the top of the class, before any method declarations, constructors, initializers or inner classes.	3	B

Table 24.: Design PMD Ruleset - VI

God Class	The God Class rule detects the God Class design flaw using metrics. God classes do too many things, are very big and overly complex. They should be split apart to be more object-oriented.	3	B
Avoid Protected Method In Final Class Not Extending	Do not use protected methods in most final classes since they cannot be sub-classed. This should only be allowed in final classes that extend other classes with protected methods (whose visibility cannot be reduced). Clarify your intent by using private or package access modifiers instead.	3	S

Table 25.: Design PMD Ruleset - VII

Rule	Description	Priority	Group
Empty Catch Block	Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.	3	B
Empty IfStmt	Empty If Statement finds instances where a condition is checked but nothing is done about it.	3	B
Empty WhileStmt	Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it is a while loop that does a lot in the exit expression, rewrite it to make it clearer.	3	B
Empty Try Block	Avoid empty try blocks - what's the point?	3	B
Empty Finally Block	Empty finally blocks serve no purpose and should be removed.	3	B
Empty Switch Statements	Empty switch statements serve no purpose and should be removed.	3	B
Empty Synchronized Block	Empty synchronized blocks serve no purpose and should be removed.	3	B
Empty Statement Not In Loop	An empty statement (or a semicolon by itself) that is not used as the sole body of a 'for' or 'while' loop is probably a bug. It could also be a double semicolon, which has no purpose and should be removed.	3	B
Empty Initializer	Empty initializers serve no purpose and should be removed.	3	B
Empty Statement Block	Empty block statements serve no purpose and should be removed.	3	B
Empty Static Initializer	An empty static initializer serve no purpose and should be removed.	3	B

Table 26.: Empty PMD Ruleset

Rule	Description	Priority	Group
Local Variable Could Be Final	A local variable assigned only once can be declared final	3	S
Method Argument Could Be Final	A method argument that is never re-assigned within the method can be declared final	3	S
Avoid Instantiating Objects In Loops	New objects created within loops should be checked to see if they can created outside them and reused	3	S
Use ArrayList Instead Of Vector	ArrayList is a much better Collection implementation than Vector if thread-safe operation is not required	3	S
Simplify StartsWith	Since it passes in a literal of length 1, calls to (string).startsWith can be rewritten using (string).charAt(0) at the expense of some readability	3	R
Use StringBuffer For String Appends	The use of the '+=' operator for appending strings causes the JVM to create and use an internal StringBuffer.If a non-trivial number of these concatenations are being used then the explicit use of a StringBuilder or threadsafe StringBuffer is recommended to avoid this.	3	S
Use Arrays As List	The java.util.Arrays class has a 'asList' method that should be used when you want to create a new List froman array of objects. It is faster than executing a loop to copy all the elements of the array one by one.	3	S
Avoid ArrayLoops	Instead of manually copying data between two arrays, use the efficient System.arraycopy method instead.	3	S
Unnecessary Wrapper Object Creation	Most wrapper classes provide static conversion methods that avoid the need to create intermediate objectsjust to create the primitive forms. Using these avoids the cost of creating objects that also need to be garbage-collected later.	3	S

Table 27.: Optimization PMD Ruleset - I

Add Empty String	The conversion of literals to strings by concatenating them with empty strings is inefficient.It is much better to use one of the type-specific toString() methods instead.	3	S
Redundant Field Initializer	Java will initialize fields with known default values so any explicit initialization of those same defaultsis redundant and results in a larger class file (approximately three additional bytecode instructions per field).	3	S
Premature Declaration	Checks for variables that are defined before they might be used. A reference is deemed to be premature if it is created right before a block of code that doesn't use it that also has the ability to return or throw an exception.	3	S

Table 28.: Optimization PMD Ruleset - II

Rule	Description	Priority	Group
Unnecessary Conversion Temporary	Avoid the use temporary objects when converting primitives to Strings. Use the static conversion method son the wrapper classes instead.	3	S
Unnecessary Return	Avoid the use of unnecessary return statements.	3	B
Unnecessary Final Modifier	When a class has the final modifier, all the methods are automatically final and do not need to be tagged as such.	3	S
Useless Overriding Method	The overriding method merely calls the same method defined in a superclass.	3	B
Useless Operation On Immutable	An operation on an Immutable object (String, BigDecimal or BigInteger) won't change the object itself since the result of the operation is a new object. Therefore, ignoring the operation result is an error.	3	S
Unused Null Check In Equals	After checking an object reference for null, you should invoke equals() on that object rather than passing it to another object equals() method.	3	S
Useless Parentheses	Useless parentheses should be removed.	4	R
Useless Qualified This	Look for qualified this usages in the same class.	3	S

Table 29.: Unnecessary PMD Ruleset

Rule	Description	Priority	Group
Unused Private Field	Detects when a private field is declared and/or assigned a value, but not used	3	S
Unused Local Variable	Detects when a local variable is declared and/or assigned, but not used	3	S
Unused Private Method	Unused Private Method detects when a private method is declared but is unused	3	S
Unused Formal Parameter	Avoid passing parameters to methods or constructors without actually referencing them in the method body	3	S
Unused Modifier	Fields in interfaces are automatically public static final, and methods are public abstract.Classes or interfaces nested in an interface are automatically public and static (all nested interfaces are automatically static).For historical reasons, modifiers which are implied by the context are accepted by the compiler, but are superfluous	3	S

Table 30.: Unused Code PMD Ruleset