Department of Computer Science

Thomas Borchert

Code Profiling

Static Code Analysis

Computer Science
E-level thesis (30hp)

Date:	08-01-23
Supervisor:	Donald F. Ross
Examiner:	Thijs J. Holleboom
Serial Number:	E2008:01

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Code Profiling

Static Code Analysis

Thomas Borchert

© 2008 The author and Karlstad University

This report is submitted in partial fulfilment of the requirements for the Master's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Thomas Borchert

Approved, January 23rd, 2008

Advisor: Donald F. Ross

Examiner: Thijs Jan Holleboom

Abstract

Capturing the quality of software and detecting sections for further scrutiny within are of high interest for industry as well as for education. Project managers request quality reports in order to evaluate the current status and to initiate appropriate improvement actions and teachers feel the need of detecting students which need extra attention and help in certain programming aspects. By means of software measurement software characteristics can be quantified and the produced measures analyzed to gain an understanding about the underlying software quality.

In this study, the technique of code profiling (being the activity of creating a summary of distinctive characteristics of software code) was inspected, formulized and conducted by means of a sample group of 19 industry and 37 student programs. When software projects are analyzed by means of software measurements, a considerable amount of data is produced. The task is to organize the data and draw meaningful information from the measures produced, quickly and without high expenses.

The results of this study indicated that code profiling can be a useful technique for quick program comparisons and continuous quality observations with several application scenarios in both industry and education.

Keywords: code profile, static code analysis, software metrics

Acknowledgements

My first and foremost thanks go to my dissertation supervisor Dr. Donald F. Ross at Karlstad University for the guidance he provided. His suggestions were not only found helpful for the project but also for the life to come. While conducting this research, I was sustained by the love, support, understanding and encouragement of my girlfriend and especially my parents and two siblings who I thank most gratefully. Through out this large and long lasting project, several up and downs were encountered and the people close to me helped me through the rough times and provided me with the support needed. At this point, I take the chance and thank all people directly or indirectly involved in the completion of this research. It has been an interesting and very beneficial journey for me.

Contents

1 Introduction ... 1

1.1 The project ... 2

1.2 Methodology ... 3

1.3 Motivation ... 3

1.4 Organization ... 5

2 Background ... 6

2.1 Terminology ... 6

2.2 Software measurement ... 8

2.2.1 The history and development of software measurement ... 8

2.2.2 Reasons for software measurement ... 11

2.2.3 Problems and issues of software measurement ... 12

2.2.4 Software measurement applied ... 13

2.2.5 Application in education ... 15

2.3 Software metrics ... 17

2.3.1 Static source code metrics ... 17

2.3.2 Dynamic source code metrics ... 17

2.4 Software measures: Tools to produce them ... 17

2.5 Summary ... 18

3 Software measurement: Code analysis ... 19

3.1 Introduction ... 19

3.2 Measurement theory ... 19

3.2.1 Introduction to measurement theory ... 19

3.2.2 Measurement scales for code analysis ... 21

3.2.3 Software measurement error and inaccuracy ... 22

3.3 Software entity code ... 23

3.3.1 Code attributes ... 24

3.4 Measuring code complexity ... 25

3.4.1 Different perspectives on software complexity ... 25

3.4.2 Psychological/cognitive complexity ... 27

3.4.3 Internal code/product complexity ... 27

3.5 Code complexity vs. code distribution ... 29

3.6 Summary ... 31

4 Software metrics: Static code metrics 32
4.1 Introduction 32
4.2 Program size 32
4.2.1 Lines of code 32
4.2.2 Number of constructs 33
4.2.3 Comments and examples 34
4.3 Code distribution 35
4.3.1 Measuring code distribution 35
4.3.2 Code distribution metrics 35
4.3.3 Summary 36
4.4 Halstead code complexity 36
4.4.1 Halstead's software science 36
4.4.2 Halstead's complexity metrics 37
4.4.3 Comments on the Halstead metrics 38
4.5 Control flow complexity 38
4.5.1 Simple control flow metrics 38
4.5.2 Cyclomatic and Essential Complexity 39
4.5.3 Comments on the control flow complexity metrics 41
4.6 Code maintainability 41
4.6.1 Comment Rate 42
4.6.2 Maintainability Index (MI) 42
4.6.3 Maintainability Index summarized 43
4.7 Object oriented metrics 44
4.7.1 Chidamber and Kemerer metrics (CK) 44
4.7.2 The MOOD metrics set 46
4.7.3 Henry and Kafura metrics 49
4.7.4 Other OO metrics 50
4.8 Summary 51
5 Software measurement tools: Static code analysis 52
5.1 Introduction 52
5.2 List of tools 52
5.2.1 CCCC 53
5.2.2 CMT 54
5.2.3 Essential Metrics 54
5.2.4 Logiscope 54
5.2.5 SourceMonitor 55
5.3 Tool comparison 55
5.3.1 Tools in detail: Usage and the metrics supported 55
5.3.2 How the original metrics are implemented 60
5.3.3 How to compare the results 61
5.4 Difficulties experienced while comparing the tools 62
5.5 Summary 63
6 Code analysis: Measuring 64
6.1 Introduction 64
6.2 Metrics of interest 64
6.2.1 Metrics interpretation 64
6.2.2 Metrics combination 66

6.3 Measurement stage description 67
6.3.1 Measurement tools used 67
6.3.2 The choice of tool for each metric 67
6.3.3 The set of source programs measured 69
6.3.4 Code measurement preparations 73
6.3.5 The code measurements performed 74
6.4 Expected differences 75
6.4.1 Students vs. industry 75
6.4.2 Open source vs. closed source 76
6.4.3 Expected measurement results 77
6.5 Summary 77
7 Code analysis: Measure interpretation 78
7.1 Introduction 78
7.2 Program size 79
7.2.1 Measurement results 79
7.2.2 Definition revisions and correlations 79
7.2.3 Defining lines of code size groups 80
7.3 Code distribution 82
7.3.1 Measure ranges and measure outliers 82
7.3.2 A closer look and measure combination 84
7.3.3 The big picture 85
7.4 Textual code complexity 87
7.4.1 Problematic aspects 87
7.4.2 Results & normalisations 87
7.4.3 Result interpretation 89
7.4.4 Control flow complexity results 90
7.4.5 Individual differences 91
7.4.6 The importance of measure combination 92
7.5 Maintainability 92
7.5.1 Maintainability measure results 92
7.5.2 Investigating group outliers 93
7.5.3 Interpretation 94
7.6 Object oriented aspects 94
7.6.1 Information flow and object coupling 95
7.6.2 Inheritance and polymorphism 96
7.6.3 Is high good or bad? 96
7.7 Individual comparisons 97
7.7.1 Group differences summarized 97
7.7.2 Individual comparison: BonForum vs. FitNesse 97
7.7.3 Creating an individual code profile 101
7.8 Summary 101
8 Code profiling 102
8.1 Introduction 102
8.2 Reasoning and usage 102
8.2.1 For industry 103
8.2.2 For education 104
8.3 The code profile 104
8.3.1 Defining areas of importance 104

8.3.2 Selecting metrics of importance 105
8.3.3 Defining measure intervals 106
8.3.4 Profile visualisation 107
8.3.5 Profile revision 108

8.4 Profiling the programs 109
8.4.1 BonForum vs. FitNesse 109
8.4.2 Student vs. Student 111
8.4.3 General comparison 112

8.5 Profile drawbacks 112

8.6 Summary 112

9 Conclusion 114

9.1 Review 114

9.2 Project evaluation 114

9.3 Future Work 118

10 References 119

11 Appendix 125

A.1 Software measurement process 125

A.2 Measurement result tables 126
A.2.1 List of software metrics applied 126
A.2.2 Program size & code distribution 127
A.2.3 Textual code complexity and code maintainability 128
A.2.4 Structural code complexity 129
A.2.5 Object oriented aspects 130

A.3 Measurement result diagrams 131
A.3.1 Line diagrams 131
A.3.2 Scatter plots 141

A.4 Program profiles 142
A.4.1 Metric information 142
A.4.2 Kiviat diagrams (competitive measures) 143
A.4.3 Program profiles (full view) 148

A.5 Measurement tools screenshots 176
A.5.1 CCCC 176
A.5.2 CMT (CMT++ / CMTJava) 177
A.5.3 SourceMonitor 178
A.5.4 Essential Metrics 179
A.5.5 Logiscope 180

List of Figures

Figure 1.1: The dissertation project 2
Figure 1.2: Dissertation organization 5
Figure 2.1: Measurement, Metrics and Measures 7
Figure 2.2: Software Measurement domains 14
Figure 2.3: Core entities for the products domain 14
Figure 3.1: Examples of software entities [1] 23
Figure 3.2: Classification of software complexity 26
Figure 3.3: Code distribution example 29
Figure 4.1: Control flow 39
Figure 4.2: Tree reduction [51] 41
Figure 6.1: Tool selection process 67
Figure 6.2: Measurement stage 73
Figure 6.3: Database tables 74
Figure 7.1: Industry NCLOC groups 81
Figure 7.2: Student NCLOC values 81
Figure 7.3: Industry NCLOC per file (avg, max) 83
Figure 7.4: Code distribution charts on file scope (5, 6, 14, 19) 86
Figure 7.5: Halstead Volume results (industry & student) 88
Figure 7.6: Halstead Effort results (industry & student) 89
Figure 7.7: BP and ECC_N for D.CC 92
Figure 7.8: Maintainability Index results 93
Figure 7.9: Comment rate results 93
Figure 7.10: Information flow D.OODM 95
Figure 7.11: File scope code distribution (BonForum vs. FitNesse) 98
Figure 7.12: Method scope code distribution (BonForum vs. FitNesse) 98
Figure 7.13: Textual code complexity (BonForum vs. FitNesse) 99
Figure 7.14: ECC file & method scope (BonForum vs. FitNesse) 99

Figure 7.15: Branch percentage & nesting level (BonForum vs. FitNesse) 100
Figure 7.16: ECC_N and Maintainability Index (BonForum vs. FitNesse) 100
Figure 8.1: Code profiling illustrated 103
Figure 8.2: Examples for data visualization 107
Figure 8.3: Example kiviat diagram 107
Figure 8.4: Profile layout explanation 108
Figure 8.5: Kiviat diagram section example 110
Figure 8.6: kiviat diagrams for course D.CC_L4 111

List of tables

Table 3.1: Scale types ... 22

Table 4.1: Program size example ... 35

Table 5.1: Static code analysis tools list ... 53

Table 5.2: Tools' metric support ... 59

Table 6.1: Metrics interpretation ... 65

Table 6.2: Tools and metrics selection ... 68

Table 7.1: List of programs measured ... 78

Table 7.2: Different program sizes ... 79

Table 7.3: Correlations with the total_LOC measures ... 80

Table 7.4: Correlations[7], NCLOC, #STAT, HN ... 80

Table 7.5: Code distribution measure ranges (file scope) ... 82

Table 7.6: Code distribution measure ranges (method scope) ... 83

Table 7.7: Program size (ID 5,6,14,19) ... 84

Table 7.8: Code distribution sample group (file scope) ... 84

Table 7.9: Code distribution sample group (method scope) ... 85

Table 7.10: ECC range measures (industry & student) ... 90

Table 7.11: BP, NBD, ECC_N range measures (industry & student) ... 90

Table 7.12: Control flow complexity results (5, 6, 40, 45) ... 91

Table 7.13: MI measure comparison (6, 14) ... 94

Table 7.14: Information flow and object coupling measure ranges ... 95

Table 7.15: File scope code distribution (BonForum vs. FitNesse) ... 98

Table 7.16: Method scope code distribution (BonForum vs. FitNesse) ... 98

Table 8.1: List of comparative metrics ... 108

1 Introduction

In industry, with increasing demands for shorter turnaround in constructing and programming computer systems, it is important to quickly identify the parts of a system which may need further scrutiny in order to improve the system. One technique to achieve this is code profiling i.e. using a selection of metrics to build a description, or profile, of the program. This in turn requires tools to produce software measures, each of which is a value for a given metric, for example the number of lines of code, Cyclomatic Complexity and Maintainability Index. These measures may then be combined to produce a profile of the program. In this project the goal is to produce a static profile for a given program i.e. measures derived from the program text. This profiling technique may also be applied to student programs in order to detect those students who may need extra help in laboratory assignments. Finally, the profiles for both student and industry programs may be compared in order to ascertain whether the teaching of programming in the university environment is actually preparing the student for industrial scale programming. This project has been undertaken in two parts: (i) an investigation into available tools for software measurement and (ii) a study of how such measures may be combined to produce a profile for a single program.

1.1 The project

In this dissertation, the feasibility of using software measurement for both industry and computer science education is examined. The preparations as well as the measurement process required are explained and the use of code profile generation in order to compare student programs is discussed. Figure 1.1 gives an overview of the project.

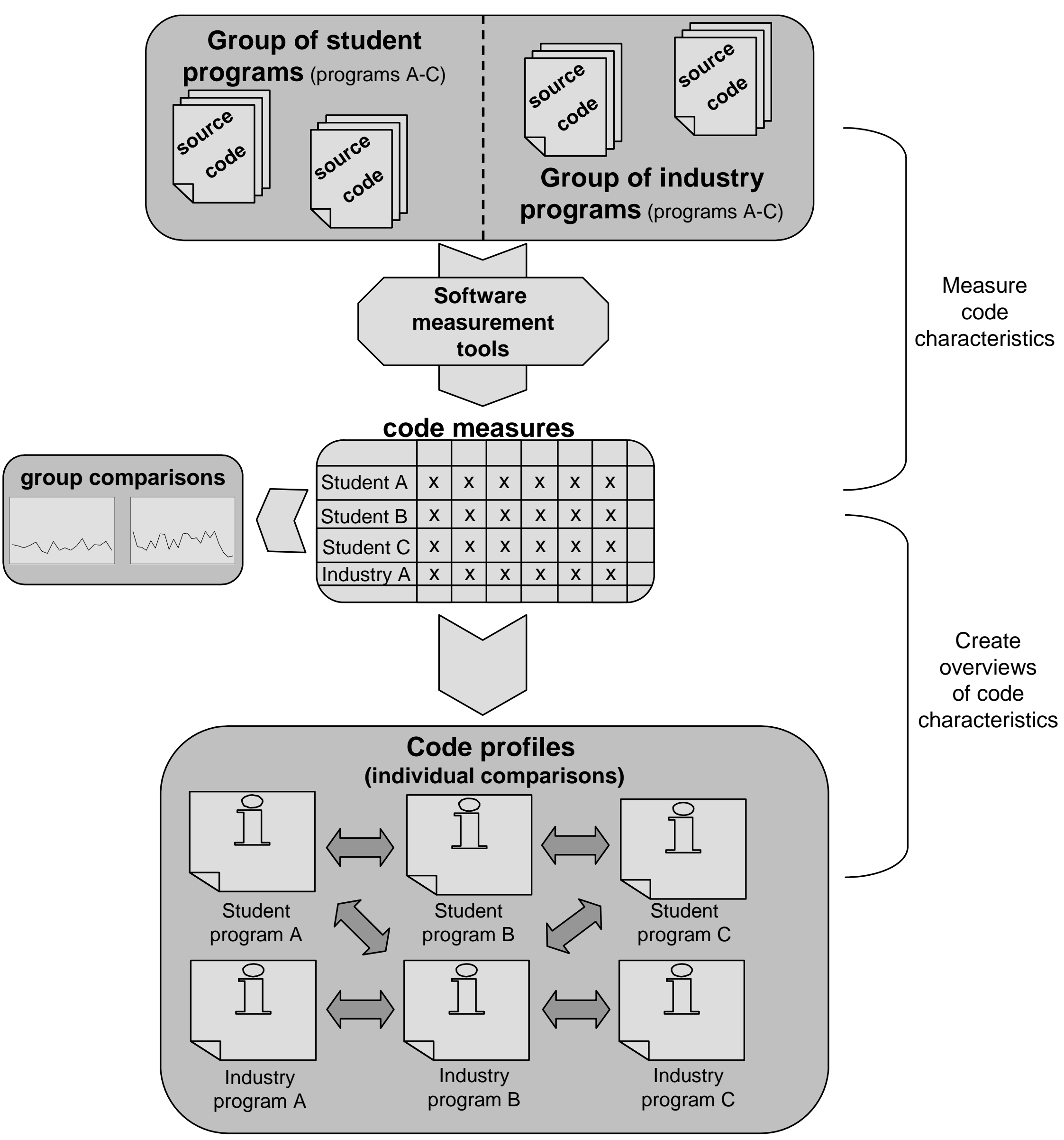


Figure 1.1: The dissertation project

1.2 Methodology

In this study a quantitative research methodology was followed. The first step was to undertake a literature study in the area of software measurement followed by a survey of software metrics for static code analysis. The next step was a survey of the available tools for producing software measures and the suitability of these tools for this project. The third step was to select industry and student programs as a test sample and derive a number of software measures for each program. Finally, a study was undertaken on how these measures could be combined to create a profile for each program.

1.3 Motivation

The motivation for this dissertation can be summarized by the following key questions:

(1) Is it useful to perform static software measures on programs for use in education and industry?
(2) How does student source code differ from industry source code and are students prepared for industrial scale programming?
(3) Is student code profiling feasible as a means of quick program comparison?

The first question suggests a deeper view into and understanding of the discipline of software measurement. When performed, additional questions might arise such as:

- *Which software metrics are available and useful for the analysis of source code?*
- *Which software measurement tools are available and how do they differ?*
- *Are the measures produced by different tools comparable?*

In order to answer the second question, not only an understanding of software measurement is required, but also knowledge about the general differences between the two areas (student vs. industry programming). Here subsidiary questions similar to the following might be asked:

- *What code characteristics can be compared?*
- *When may code be defined as complex, and what does complex in this sense mean?*

The third and last key question raises another set of subsidiary questions:

- *What is a code profile?*
- *What is needed for a code profile?*
- *Can code profiling be used to improve the students' awareness of programming?*

These three main questions and their subsidiary questions will be considered and answered in the course of this dissertation project. The following sections present the way in which the reader will be directed through the dissertation and the questions presented.

1.4 Organization

This section gives an overview on the organization of the dissertation. The dissertation can be viewed as consisting of two parts. Part 1 describes the generation of the measures and part 2 the analysis of these measures. The first part of the dissertation is divided as follows (see Figure 1.2):

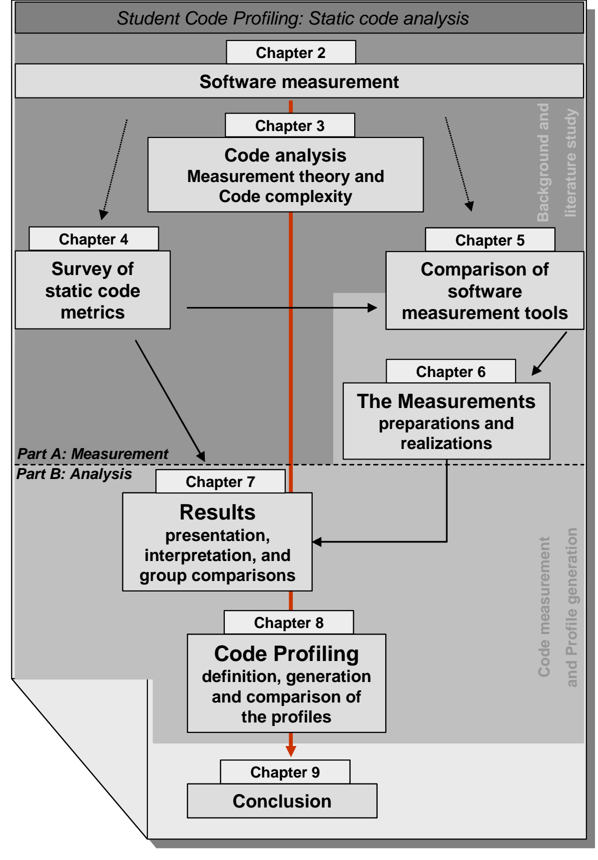


Figure 1.2: Dissertation organization

Chapter 2 presents an introduction and background to software measurement as well as the terminology used in this dissertation. Chapter 3 serves as a more detailed background of code analysis within the area of software measurement; special focus is placed on software complexity.

Chapter 4 presents and discusses the survey of software metrics for static code analysis.

Chapter 5 gives a survey of software measurement tools available. In addition, a selection of tools is compared in terms of usability and applicability for this study. Chapter 6 deals the student and industry code measurements themselves and the preparations for them. Next to the introduction of the program sample selection a discussion about the student vs. industry issue is presented.

Part two of this dissertation is presented in Chapters 7, and 8. In Chapter 7 the code measurement results are presented, discussed and analyzed. Furthermore, group differences are pointed out and individual comparisons made. The profile generation and comparison is discussed and analyzed in Chapter 8. Here the profile defined is illustrated by comparing two programs, for which the results were to a great extent discussed in the chapter before. Chapter 9 evaluates and reviews the methodology used and the problems experienced, as well as the outcome produced.

In the Appendix Measurement results and profiles generated can be found.

2 Background

A more comprehensive understanding of the area of software measurement is required, before specific static code analysis can be discussed in detail. In this chapter a description of software measurement is presented and general information on the topic and the related tasks is given.

2.1 Terminology

The first set of terms consists of *measurement, metrics*, and *measures*. Before understanding these terms in the context of software measurement, some definitions are presented:

- "**Measurement** is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules." [1]
  It should be noted at this point that an entity is any object, either real or abstract, whose characteristics can be described, and captured as attributes [2].
- "**Metrics** are a system of parameters or ways of quantitative and periodic assessment of a process that is to be measured, along with the procedures to carry out such measurement and the procedures for the interpretation of the assessment in the light of previous or comparable assessments" [3].
- A **Measure** is
  - "a basis for comparison; a reference point against which other things can be evaluated; they set the measure for all subsequent work" [4]
  - "how much there is of something that you can measure" [4]

These terms are quite clear in literature. The terms software measurement and software metrics, however, are often used interchangeably [5]. In this paper the understanding of software measurement as being the process of quantifying selected characteristics of software entities with the goal of gathering meaningful information about these entities, is followed. How the selected characteristics are quantified is determined by the software metrics used in the measurement process. Applying a software metric results in a measure for the characteristic which subsequently must be interpreted. The measure produced is a value that can be used as the basis for comparison. Information can be retrieved through the interpretation of the software measures. The first set of terms stands in this dissertation for the

code measurements and represents a combination of software measurement, metrics and measures (see Figure 2.1).

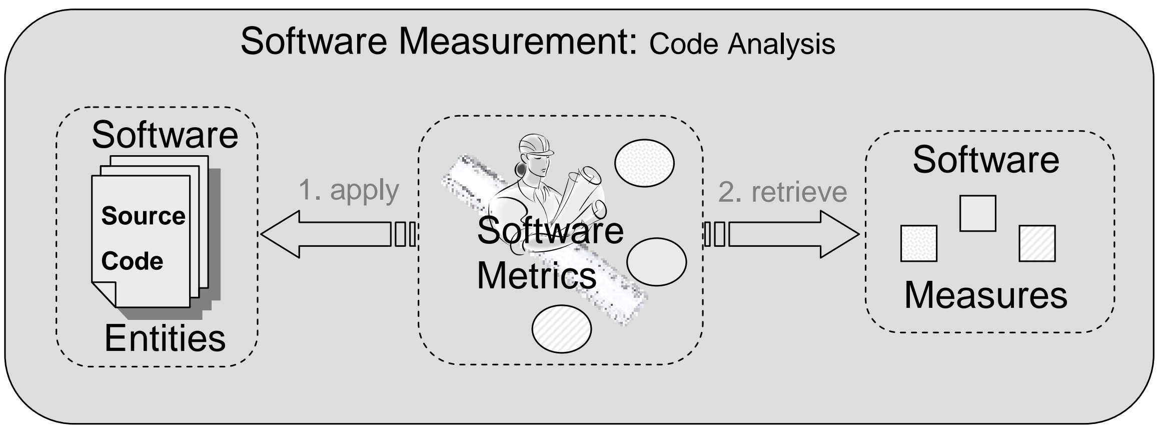


Figure 2.1: Measurement, Metrics and Measures

The second set of terms consists of *information* and *profile*. The measures produced and the metrics applied in the first part of this dissertation require to be understood in order to gain information about the measures produced. What does the term information actually mean? A general definition says that:

- *"**Information** is the result of processing, manipulating and organizing data in a way that adds to the knowledge of the person receiving it."* [6]

Within the context of software measurement, information is knowledge about software entities together with the understanding of associated software attributes and their characteristics. In other words information is retrieved when the measure for the attribute is interpreted. In this dissertation, the focus is on gaining an understanding of code characteristics. As indicated, the information gained is analyzed to profile the programming ability and style of students.

A **profile** is *"a formal summary or analysis of data, often in the form of a graph or table, representing distinctive features or characteristics"* [7].

The information about selected code characteristics will be summarized and stored in a code profile for a particular program.

2.2 Software measurement

2.2.1 The history and development of software measurement

The practice of measurements in the area of software engineering has a history of almost 25 years [8]. The idea of creating and defining software metrics started in the 1960s and the 1970s and was mainly developed in the 1980s and 1990s.

The following information about the history and development of software measurement were mainly gathered from *A Frame work of Software Measurement* [8] and *History of Software Measurement* [9].

1950 – 1970

Probably the first step in the area of software measurement was done in 1955 by John Backus. He proposed the software metric LOC (Lines of Code) and used the metric to analyze the size of a FORTRAN compiler [8]. Counting the lines of a program is a fairly easy process and therefore made LOC a widely used metric. However, the problem of LOC lies in possible variations in counting LOCs [10] for example by including or excluding comments. However, LOC became the basis for several other software metrics as presented in this section as well as in Chapter 4. Later in the 1960s, the first cost prediction methods in the area of software engineering were introduced; these were namely the Delphi and Nelson's SDC method [8]. Probably the first paper in the area of software measurement that dealt with software complexity was *"Quantitative Measurement Program Quality"* [11] in 1968.

1970 – 1980

In the 1970s and 1980s the area of software measurement was further developed and several papers and books were published as well as new software metrics introduced; especially metrics trying to quantify the complexity of source code. The complexity aspect of software is presented in Chapter 3.

One of the first metrics trying to quantify programmer productivity was proposed by R.W. Wolverton in 1974 and was based on the LOC metric. Wolverton proposed object instructions per man-month as a productivity measure and suggested threshold intervals for the metric. In the same decade several complexity metrics were introduced. In 1976, the metric Cyclomatic Complexity [12] was introduced by Thomas McCabe [8]. This metric tries to quantify the complexity of a program by measuring the number of linearly independent paths through a

program. The Cyclomatic Complexity metric is still widely in use and is discussed further in Chapter 4.

One year later, Maurice Halstead [13] introduced a set of metrics targeting a program's complexity; especially the computational complexity [14]. Halstead based the metrics on operators and operands count within the source code of a program as explained in his book *"Elements of Software Science"* [13]. Several companies, including IBM and General Motors, have used this metric set in their software measurement processes. Today the Halstead metrics are still widely in use, including Halstead Length, Volume, Difficulty and Effort which are the most common. Further complexity metrics were introduced by Hecht in 1977 and by McClure in 1978 [9].

The first book which focussed entirely on software measurement was *"Software Metrics"* by Tom Gilb, was published in 1976 [9]. Also in the 1970s, the terms *software physics* and *software science* were introduced in order to construct computer programs systematically. The former was proposed by Kolence in 1975 and the latter was proposed by Halstead in 1977 in the previously mentioned book "*Elements of* Software Science" [13].

Then in the late 1970s (1979), Alan Albrecht introduced his idea of function points at IBM [8]. The Function-Point metric captured the size of a system based on the functionality provided to the user in order to quantify the application development productivity.

1980 – 1990

Also in the 1980s several complexity metrics were introduced, including Ruston's flowchart metric [8], which describes a program's flowchart with the help of the flowchart's elements and underlying structure. In 1981 Harrison et al [8] introduced metrics to determine the nesting level of flow graphs. One year later Troy et al defined a set of metrics, which tried to quantify modularity, size, complexity, cohesion and coupling of a program [8].

In addition to the Function Point method from 1979, another widely spread cost estimation metric with the COCOMO (Constructive Cost Model) [15] was introduced two years later in 1981. The COCOMO, based on the Metric LOC, estimates the number of man-months it will take to finish a software product. A further cost estimation metric is the Bang metric was proposed by DeMarco six years later [9].

In 1984, NASA was one of the first institutions to begin with software measurement. In the same year the metric GQM (Goal Question Metric) was developed by Victor Basili and the NASA Software Engineering Laboratory. GQM is a approach to establish a goal-driven

measurement system for software development applicable to applied to all life-cycle products, processes, and resources [16] [1] .

In 1987, Grady and Castwell established the first company wide metric program at Hewlett Packard and published their findings in [17].

Since then, not much attention had been given to object oriented programming in the area of software measurement. Yet in 1988, Rocacher [9] undertook early investigations regarding the measurement of object oriented designs, while establishing software metrics for the object oriented programming language Smalltalk. One year later, Kenneth Morris followed by discussing software metrics for an object oriented system in his publication *"Metrics for Object-Oriented Software Development Environments"* [9]. Bigger breakthroughs in object oriented measurement were made in the 1990s.

Since 1990

As object-oriented programming became a more widely accepted programming methodology during the 90s [18], the focus also changed within software measurement toward object oriented programs. Thus, several books and papers on object oriented software measurement were published as well as many object oriented metrics were introduced [19]. Furthermore, methods to quantify external attributes such as maintainability, readability and quality were introduced. The following is a brief presentation of events related to the field of software measurement since 1990.

In 1992, the first IEEE-Standard for quality metrics, namely „ IEEE Std 1061-1992 IEEE Standard for a Software Quality Metrics Methodology", was established [9]. The standard depicts methods for establishing quality specifications as well as the identification, analysis and validation of quality metrics.

In 1994, the first book about object oriented software metrics, written by Lorenz et al., was published [9]. In the same year, Chidamber and Kemerer [20] introduced a set of object oriented metric set, widely known as CK metrics. The CK metrics are widely used as metrics to quantify object oriented design approaches. The MOOD metric set, introduced by Abreu et al. [21], followed in 1995. Chapter 4 gives a more detailed description of the CK as well as the MOOD set and other object oriented metrics.

More books and papers on software measurement and best practices in software measurement appeared after the mid 1990s. However, fewer new software metrics were created. Examples of books which appeared in the later 1990s are "Object-oriented metrics: measures of complexity" by Henderson-Sellers (1996) [18], "A Framework of Software

Measurement" by Zuse (1997) [8] and especially "Software Metrics: A Rigorous Approach" by Fenton (second edition 1997) [1]. Despite new approaches in the 1980s and 1990s, many of the metrics applied in industry were established in the 1970s [22].

2.2.2 Reasons for software measurement

There are several reasons for using software measurement in practice. The key benefits of software measurement can be classified as follows [16]:

- Understanding
- Evaluation ⎤ Control [1]
- Prediction ⎦
- Improvement

According to Fenton [1], the categories evaluation and prediction can be combined to control and control is essential to any important activity.

> "We must control our projects, not just run them" [1]

However, only ongoing software projects require control. A completed project will not change in state and therefore does not need to be controlled, but may be compared to other projects. Since the final source code (as entity code) itself and not its creation processes is the central point in this dissertation, the main focus lies on the aspects of *understanding* and *improvement* of software.

Before knowing how to improve an attribute (where to go), we first need to understand the meaning of the attribute and its current characteristics (where we are).

> "If you don't know where you are a map won't help" [23]

In the context of measuring source code, understanding is defined here as to gain meaningful information about code attributes (as illustrated in Figure 2.2). With the understanding of the code characteristics within their current states, areas of improvement [16] can be identified and threshold intervals which should be met can be defined. Threshold intervals define a value range in which the measures should occur.

2.2.3 Problems and issues of software measurement

The major problems and issues in the field of software measurement can be pointed out as follows:

- **Interpretation of measures**

  Each measure produced on its own does not give us more than a single value. It requires education and experience to make software measures useful [2]. Furthermore, information regarding the context in which the measures were produced is needed [8]. For this project, the measure of cyclomatic complexity of a student program might have a value x. However this measure has to be interpreted in the context of the program size, since a longer program is more likely to be more complex, before information about the complexity can be retrieved. This point is further discussed in Chapter 3.

- **Scale types**

  Care has to be taken about the *scale types* involved. According to measurement theory [24], every scale type has a different set of applicable statistics. E.g. the set of statistics for the interval scale does not include multiplication. Therefore saying today's temperature is twice as hot as yesterday's is not a valid statement for Fahrenheit and Celsius, which are based on the interval scale [1] (see Chapter 3). This means in the context of software measurement that comparing and analyzing the measures produced has to be done with caution. More information about measurement scale types are presented in Section 3.2.2.

- **Lack of specific standards**

  According to Munson [2], the biggest problem in the field of software measurement, for each measured object, is the lack of standards.

  > "NIST[1] is not motivated to establish standards in the field
  > of software measurement."

  Therefore, measures retrieved for the same software entity are not comparable unless additional information about the way they were quantified is available and the methods are identified as equivalent. This means for this project, that the tools have to be compared by analyzing which of the software measures are produced in the same manner.

- **Issues of people measures**

With software measurement the productivity of programmers can be measured. However, judging a programmer's performance by the use of a metric (such as LOC/hour) can be seen as unethical [2]. Furthermore, the outcome can be influenced by the programmer measured. For example, the programmer writes unnecessary LOC to increase the measure for his productivity. Since not the students but their code is of interest, this does not represent a problem in this dissertation project.

- **Reliable and valid data**

Gathering measures is not sufficient in order to obtain reliable measurement results, since the way the data are collected and analyzed has a great influence on the outcome. As M.J. Moroney (1950) said:

> *"Data should be collected with a clear purpose in mind. Not only a clear purpose but a clear idea as to the precise way in which they will be analyzed so as to yield the desired information."*

In this dissertation, the purpose of data collection is to gain information about the size, structure, complexity and quality of the available source code.

2.2.4 Software measurement applied

Section 2.2.1 presented the history and development of Software Measurement. In that section, the reasons for Software Measurement and the related problems were illustrated. This section focuses on the application of software measurement; the domains, their entities, and the measurement process itself.

Software measurement domains (and their entities)

Software entities can be classified into product entities, process entities and resource entities [1]. Since software measurement tries to quantify software entities, the areas of application can be categorized correspondingly as displayed in Figure 2.2. Further examples for software entities can be seen in Figure 2.2.

---

[1] National Institute of Standards and Technology

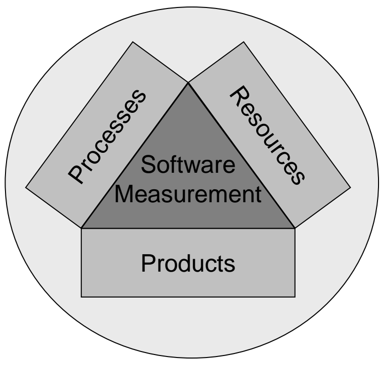


Figure 2.2: Software Measurement domains

The *Processes* domain holds software entities that represent steps in the software engineering process. Quantifying and predicting attributes within this domain (such as time, effort and cost) is especially of interest for managers and team leaders [1, 8]. Boehm's COCOMO method [15] and Albrecht's function point model, which are introduced in the history and development section of this paper, accomplish this job.

The *Resources* domain contains software entities, which are used or required by software entities within the Processes domain (such as personnel, software and hardware) [1]. In this respect, personnel are required to complete a development process. Attributes of interest for this entity are e.g. the number of software engineers involved, their skills and their performance. Munson [2] classifies this as an individual area called people measurement domain.

The *Products* domain holds software entities that result from a process activity (thus an entity within the processes domain). Figure 2.3 illustrates the core software entities within this domain, as Fenton [1] sees them.

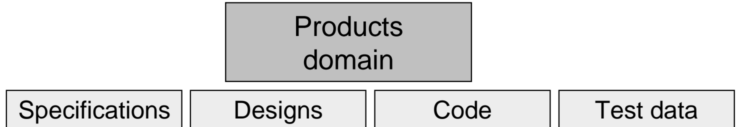


Figure 2.3: Core entities for the products domain

In this dissertation project, software measurement is applied in the products domain and focus lies on the entity code and its attributes. Software measurement of the entity code is discussed in more detail in Chapter 3.

The measurement process

The term software measurement process can be defined as follows:

"That portion of the software process that provides for the identification, definition, collection, and analysis of measures that are used to understand, evaluate, predict, or control software processes or products." [25]

According to McAndrews [25], the software measurement process can be split into the four activities: planning, collecting data, analyzing data and evolving the process. During the planning a purpose and a plan for measuring has to be defined. Furthermore the software entities and attributes of interest have to be identified and suitable software metrics selected or new ones defined. During the data collecting phase the selected metrics are applied manually or by the use of software measurement tools [26]. Software measurement tools are discussed later in Section 2.4 and in Chapter 5. The measures/data produced are then analyzed with regard to the measurement goals defined.

In this dissertation the measurement process is split into two stages, namely a measurement stage and an analysis stage. The measurement stage includes the planning, preparations for code measurements and the code measurements themselves. In the analysis stage, the measures produced are analyzed and interpreted for later use in code profiling.

2.2.5 Application in education

Despite its age and its application in industry, the discipline of software measurement is not often used for the education sector [27]. Some of the related studies about the application of software measurement in education were found as follows:

- *Using Verilog LOGISCOPE to analyze student programs* [28]

This paper (published in 1998) describes the application of a software measurement tool to measure student programs. The results of analyzing programs are given to show the diversity of results. In addition a discussion on how the software measurement tool can be used to help students improve their programming and help instructors evaluate student programs better is presented.

- *A case study of the static analysis of the quality of novice student programs* [27]

In this paper (published in 1999) builds up on the paper listed above. The authors used a software measurement tool and sample student programs to affirm that static code analysis

is a viable methodology for assessing student work. Further work is considered by the author to help to confirm the study's results and their practical application.

- *Static analysis of students' Java programs* [29]

This paper (published in 2004) introduces framework for a static analysis. According to the author this framework can be used to give beginning students practice in writing better programs and to help in the program grading process. The framework suggests both software metrics and relative comparison to evaluate the quality of students' programs.

- *Student portfolios and software quality metrics in computer science education* [30]

This paper (published in 2006) discusses the usage of student program collections (portfolios) to conduct long term studies of student performance, quantitative curriculum and course assessment, and prevention of forms of cheating. The authors suggest the use of quality software metrics to analyze the portfolios, however only requirements for those are outlined.

2.3 Software metrics

As indicated in Section 2.1 software metrics are instruments applied within the software measurement process to quantify attributes of software entities. Each software metric holds information about its targeted attribute(s) as well as how the metric can be applied to quantify the attribute(s). Thus, software metrics are specific defined instruments for targeting an attribute of a software entity and explains the application of the metric within a software measurement process. For example, LOC is a metric that targets the source code size (length) attribute. As will be explained below in Chapter 4, LOC quantifies the size by counting the lines of code. In this dissertation project, the focus is on the source code and source code attributes. Source code metrics can be further classified as follows:

2.3.1 Static source code metrics

Static code metrics attempt to quantify source code attributes from the program text [2, 31]. Such metrics may quantify control flow graphs, data flows and possible data ranges. Static source code metrics are further reviewed in Chapter 4.

2.3.2 Dynamic source code metrics

Dynamic source code metrics quantify characteristics of a running system and how it interacts with other processes. Compared with the usage of static source code metrics, the measurement of a running system is rather resource consuming due to the massive data overhead. However, dynamic source code metrics represent a great help in error detection, security aspects and program interaction [2].

2.4 Software measures: Tools to produce them

There is a large number software measurement tools available [32], most of which are commercial. Some of the available tools focus on a specific area of software measurement, such as software process or resource measurement. In this dissertation, the focus lies on tools suitable for static code analysis. A more detailed description of software measurement tools is given in Chapter 5.

2.5 Summary

In conclusion, software measurement is the process of applying software metrics with the goal of quantifying specified attributes of software entities to gain meaningful information about the entity itself. The main advantages of software measurement are control, understanding and improvement. The disadvantages are the lack of standards and the experience required in order to gain meaningful information. Software measurement tools exist to simplify and automate the application of software metrics within the software measurement process.

3 Software measurement: Code analysis

3.1 Introduction

In order to measure software, the basic theory about measurement theory has to be known and understood. Section 3.2 presents a short introduction and aspects of interest. Since the analysis of source code is the major aspect in this dissertation project, a deeper look at the entity code and its attributes is presented in Section 3.3. One frequently discussed and difficult quantifiable attribute is complexity [5]. The measurement of code complexity and the term software complexity itself are discussed in Section 3.4.

3.2 Measurement theory

Measurement theory is a branch of applied mathematics and defines a theoretical basis for measurement. According to Zuse [8], the major advantages of measurement theory are hypotheses about reality, empirical investigations of measures produced, and the understanding of the real world through an interpretation of numerical properties.

3.2.1 Introduction to measurement theory

As with other measurement disciplines, software measurement also requires to be based on measurement theory [33]. For the area of software measurement, measurement theory holds the following [8]: (1) clear definitions of terminology, (2) strong basis for software metrics, (3) criteria for experimentation, (4) conditions for validating software metrics and (5) criteria for measurement scales. Henderson-Sellers [18] points out, that measurement theory in addition introduces an understanding of variances, intervals, and types of errors in the measurement data through the use of statistics and probabilities.

In the literature, measurement theory for software measurement has been discussed in much detail. For this dissertation, the works of Zuse [8], Fenton [1] and Henderson-Sellers [18] were used to gain an understanding of the aspects of measurement theory for software measurement. In the following sections, key aspects of measurement theory are briefly presented:

- **Empirical relations**

  The aspect of empirical relations deals with the intuitive or empirical understanding of relationships of objects in the real world. The relationships can be expressed in a formal relationship system [18]. For example, one might observe the relation of two people in terms of height. The comparisons “taller than”, “smaller than”, etc. are empirical relations for height [1]. If a term, such as complexity or quality, has different meanings to different people, then defining a formal relationship system becomes close to impossible [1, 18]. The difficulty of measuring software complexity is further discussed in Section 3.4.

- **Rules of mapping**

  There are different approaches for mapping an attribute from the real world to a mathematical system. The mapping has to follow specific rules to make the measures reproducible as well as comparable [1]. As an example, several approaches for quantifying the size of a software system exist however these measures are not comparable unless clear information is presented about the way the measures were produced.

- **Scale types**

  Differences in mapping can limit the possible ways of analysing a given measure [1, 8]. Measurement scale types exist to help identify the level of statistics applicable. The different scale types are further discussed in Section 3.2.2.

- **Direct and indirect measurement**

  Measurement theory categorizes measures into direct and indirect measures [1]. Direct measurement of an attribute does not depend on the mapping of any other attribute. Whereas, indirect measurement of an attribute is measurement which involves the measurement of one or several other attributes. The same classification applies for software metrics.

- **Validity and reliability of measurement**

  Measurement theory helps to validate the transformation of abstract mapping concepts into operational mapping definitions and prove the definitions’ reliability [34, 35].

This means that software metrics definitions can be inspected and validated mathematically. However, “theory and practice are travelling very different roads on the topic of software metrics “ [36]. The theoretical validation of software metrics is often neglected [18, 34]. Several software metrics (such as Cyclomatic Complexity) exist without theoretical proof but with analytical analyses verifying the concept [18]. Furthermore, Henderson-Sellers [18] complains that the number of experiments performed are often not sufficient and the underlying experiment context is too narrow. According to Zuse, theory and the application of statistics should be combined for validating software metrics [8].

- **Measurement error and inaccuracy**
  Measurement errors result in deviations of the measures produced from the true values in the real world [34, 37]. Measurement error and inaccuracy in software measurement is further discussed in Section 3.2.3.

3.2.2 Measurement scales for code analysis

Measurement theory provides conditions for applying statistics to the measures produced. The measures as well as the mappings used to produce the measures can be connected to certain scale levels. This is important for understanding which analyses are appropriate for underlying measurement data [8]. Different scale types allow a different set of applicable statistical methods. In Table 3.1 the scale types are divided into *nominal*, *ordinal*, *interval*, *ratio* and *absolute* scale types. The scale types are each successively supersets of the preceding sets of scales. The set of defined relations and thereby the applicable analyses increase for each such superset [1]. The classification into scale types helps to understand which kinds of measurements may be used to derive meaningful results. Stating that a software project is twice as big as another, by using the number of lines of code (absolute scale), is valid. However, stating that a project is twice as complex with regard to the measures produced is problematic, since complexity measures may be defined on an interval scale [1].

Scale type	Scale definition	Examples	Defined relations
Nominal classes	A set of classes or categories, in which the observed elements are placed.	colors, fruits	equivalence
Ordinal ordered classes	A extension to the nominal scale, as it provides a ordering of the categories.	user ratings	(above), bigger smaller
Interval f(x) = ax+b (a>0)	Uses ordered categories, as the ordinal, and in addition preserves the gap size between the groups.	temperature	(above), difference
Ratio f(x) = ax (a>0)	The ratio scale enhances the interval scale by starting with a zero element, which itself represents the absence of the attribute being measured.	length, mass	(above), ratio
Absolute f(x) = x	Counts number of occurences. Valid values are zero and positive integers.	detected errors	(all)

Table 3.1: Scale types

In this dissertation the interest lies in the static analysis of source code. Therefore the restriction to metrics that produce measures on interval scales seems advisable. The nominal and ordinal scales, as shown in Table 3.1, would seem less suited to the required types of analyses needed to for this project (such as the arithmetic average).

3.2.3 Software measurement error and inaccuracy

Errors may occur during measurement, even in simple measurements [23, 37]. Software measurement error concerns the deviation and inaccuracy of software measures produced from the actual software characteristics in the real world. Several classifications for measurement errors exist [34, 37] and in this dissertation a classification into *instrumental measurement errors* and *personal errors* is made. Instrumental measurement errors are caused by imperfections in the measurement instrument used during the measurement process. Possible instrumental measurement errors in this dissertation project caused by the measurement tools encountered during the measurement process are discussed in Chapter 5.

Personal errors are caused by human mistakes during the measurement process. These errors can include improper selection of software metrics, the incorrect adjustment and application of measurement tools as well as incorrect interpretation of the measures produced. The metric selection for this dissertation project is discussed in Chapters 5 and 6. Measurement results and the interpretation of these follow in Chapter 7.

Measurements are highly dependent on the instruments used and on the accuracy these provide [24, 36, 37]. For software measurement the inaccuracy of available software metrics is an often discussed topic. Some software attributes are harder to quantify more accurately than others. Software complexity for example is difficult to quantify [5, 33] because this software attribute is influenced by several other not always ascertainable factors. The factors themselves can further depend on other aspects, thus making the intended attribute difficult to quantify. The problem of quantifying software complexity and the accuracy of software metrics is further discussed in Section 3.4.

3.3 Software entity code

Software entities can be classified into the three software measurement domains presented in Figure 3.1. Each software entity has specific attributes, which are of interest for the measurement. According to Fenton [1], attributes of software attributes can be classified into internal and external attributes. Internal attributes can be measured directly through its entity, whereas external attributes are measured through the entities environment. For external attributes the behaviour is of interest rather than the entity itself [1]. Figure 3.1 shows a list of example software entities and their corresponding attributes.

ENTITIES	ATTRIBUTES	
	Internal	*External*
Products		
Code	size, reuse, modularity, coupling, cohesiveness, functionality, algorithmic complexity, control-flow structuredness,…	reliability, usability, maintainability, complexity, portability, readability,...
Specification	size, correctness, syntactic…	comprehensibility, …
Processes		
Constructing specification	time, effort, number of requirement changes,…	quality, cost, stability,…
Resources		
Personnel	age, price	productivity, experience, intelligence,…

Figure 3.1: Examples of software entities [1]

Source code is the software product created with the combination of software entities from the processes and resources domain. It consists of a sequence of statements and/or declarations written in one or several programming languages. Source code represents the implementation of a software program. In this dissertation project, the code attributes of different software projects are analyzed to gain meaningful information about the different code characteristics.

3.3.1 Code attributes

Code attributes represent characteristic traits of the source code and will be also classified into internal and external.

Internal code attributes

Internal code attributes hold information about the code itself. The following internal attributes are of special interest for this project: size, modularity, coupling, cohesion, control-flow and other internal aspects that are involved in the structure of the source code. Several software metrics exist to quantify these attributes. The measurement of internal code attributes can be performed objectively [18] due to their independence from other software entities. LOC e.g. is a software metric that quantifies the internal attribute *size* of the software entity *code*. As further explained in Chapter 4, LOC quantifies the size by counting the lines of code.

External code attributes

External code attributes hold information about the code and the code behaviour or characteristics within its environment [1]. These attributes require to be quantified with regards to how the code relates to other entities [1]. Thus, external code attributes are difficult to grasp and since in this dissertation project only the source code itself is analyzed are external attributes not directly of interest. In this dissertation only the code maintainability, as external attribute, is considered.

3.4 Measuring code complexity

Since software complexity was found to be one major contributing factor to the cost of developing and maintaining software [12, 15, 38], the interest in quantifying this characteristic / software attribute arose after the 1970s [18, 39, 40] and a number of complexity metrics have been proposed [5]. In order to measure the complexity the researchers first had to understand what software complexity is and means. However, here the definitions differ in the software engineering literature, indicating that the aspects of software complexity are not yet fully understood

3.4.1 Different perspectives on software complexity

In the software engineering literature a number of definitions for software complexity are given [5, 18, 41].

> For Evangelisti [5] software complexity is "the degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, and the types of data structures"
>
> Basili [18] defined software complexity as "a measure of resources expended by a system (human or other) while interacting with a piece of software to perform a given task".
>
> IEEE [42] defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify"
>
> For Zuse [5] software complexity is "the difficulty to maintain, change and understand software. It deals with the psychological complexity of programs" [5].

Thus the definition of the term software complexity is difficult to specify precisely, but can we still measure the complexity of software? Fenton states that quantifying complexity into a single measure is close to impossible or equally difficult as finding the "holy grail" [33]. Instead complexity metrics focus on certain aspects of complexity, what complexity consists of and what influences the complexity of a system. However, since the term itself is unclear,

opinions differ in the software engineering literature also with regard to different aspects of software complexity.

Fenton classified the aspects of complexity into 4 areas: problem complexity, algorithmic complexity[2], structural complexity and cognitive complexity (cognitive complexity is further discussed in Section 3.4.2) [1]. Jones [43] lists 20 different aspects of software complexity. Here the approach from Henderson-Sellers [18] is discussed in more detail. He classified software complexity as depicted in Figure 3.2 using the following three aspects.

Computational complexity is concerned with the complexity of the computation in terms of required time and hardware resources [18, 43]. Algorithms which are difficult to understand do not necessarily have to be computationally complex [34].

Psychological complexity is related to the difficulty of understanding the software program. This aspect of software complexity is also referred to as cognitive complexity and is further discussed in Section 3.4.2.

Representational complexity deals with the tradeoffs between graphical and textual notations for unambiguous representations of the software system, its environment and the system interactions within [18].

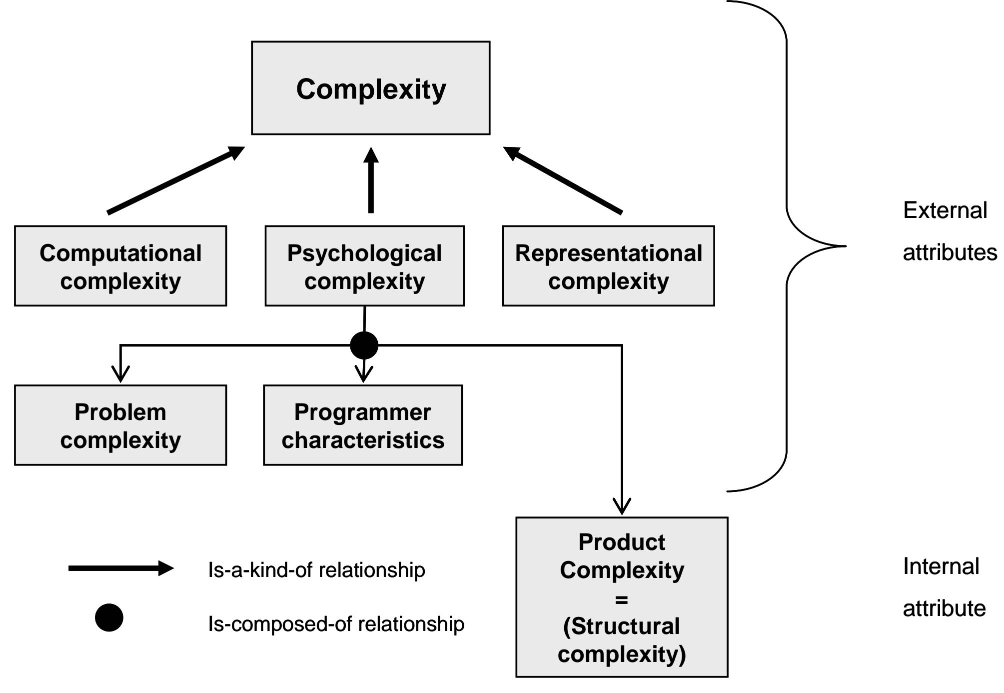


Figure 3.2: Classification of software complexity

Henderson-Sellers considered Zuse's view [5] in this approach to software complexity (presented above) and introduces the psychological complexity as a major aspect in software complexity. As shown in Figure 3.2, psychological complexity now consists of the individual programmer characteristics as well as problem complexity and structural complexity. The latter two were perceived in other publications (such as [1] and [43]) as direct components of software complexity. A model for code complexity applied to product complexity as an internal attribute is presented in Section 3.4.3.

3.4.2 Psychological/cognitive complexity

Psychological (cognitive) complexity refers to the difficulty for a person/programmer of understanding and verifying a software product [5, 18]. This complexity aspect is thus highly dependent on the difficulty of the underlying problem, the product complexity including software characteristics (such as size, control flow and coupling) and the programmer characteristics [18].

Problem complexity refers to the difficulty of the underlying problem. It is assumed that complex problems are more difficult for a person/programmer to comprehend than simple ones. This type of complexity is not controllable and difficult to measure. In this dissertation project, the complexity of the underlying problem is classified in terms of class level of the student. It is assumed that A-level courses provide easier problems than D-level courses.

Product complexity (structural complexity) is concerned with the structure of the software program. Structural complexity is frequently discussed in the software engineering literature [1, 18, 40] and, as it is essential to this dissertation project, will be presented in more detail in the following section.

Programmer characteristics deal with the individual programmer traits that are involved when performing tasks on software products. Quantifying the programmer characteristics is difficult, especially without adding subjective bias [18].

3.4.3 Internal code/product complexity

The structure of a program is measurable through internal attributes of the software code (such as size and coupling). Several metrics [5], called structural complexity metrics, have been proposed in order to achieve this goal. In the software engineering literature, structural

---

[2] Algorithmic complexity deals with the difficulty of implementing a problem solving algorithm

complexity is often referred to as product complexity [1, 18, 34], as the complexity of the product (source code) is highly dependent on the underlying structure. A complex structure makes it more difficult to understand and maintain a software product. The proposed structural complexity metrics offer monitoring of structural complexity through the assessment of a variety of structural aspects. These aspects and thereby the metrics themselves can be classified as follows: size, logical structure, data structure, cohesion, coupling and information flow [18]. Through the combination of these metrics a prediction for the product complexity can be made [44]. However, they ought to be interpreted as indicators. [18, 33]. Chapter 4 presents software metrics that capture structural aspects of the source code. Thus e.g., the LOC metric measures the size of a program and the software metric Cyclomatic Complexity is used to quantify the complexity of the underlying logical structure.

3.5 Code complexity vs. code distribution

In this dissertation the term code distribution is understood as the way the source code of a program, as a whole, is organised over a number of files, classes, and methods. As an example, a program's code can be placed in a single method within one file, whereas for the same program, in a different code distribution approach, the code may be distributed over several methods within different programming files. Figure 3.3 illustrates this point further.

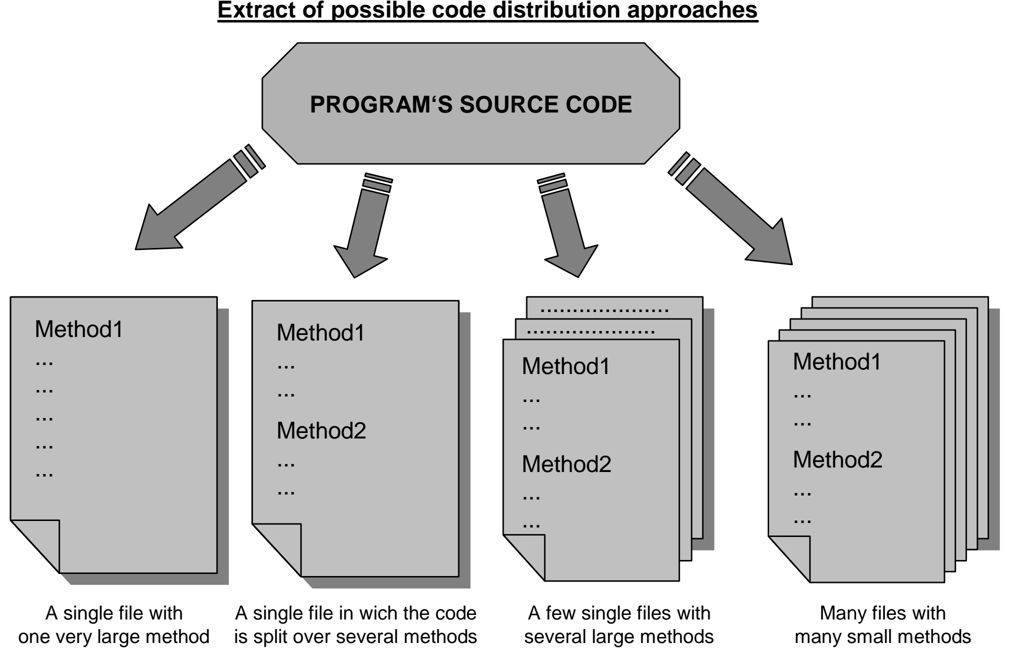


Figure 3.3: Code distribution example

Which code distribution is the most suitable, is highly dependent on the project, the programming language used, and the environment [10]. Thus, there is no clearly defined optimal approach for the textual code length of methods, classes, and files. Nevertheless, different views about an ideal length exist in the software engineering literature [10, 45]. In this dissertation the discussion about a possible ideal or maximum size (length) is not of interest. What is of interest is the actual choice of size and code distribution. Before the relationship between code distribution and code complexity is further explained, two important issues about the code distribution require to be considered:

(1) Long[3] files, classes, and methods (with regard to their size, e.g. lines of code) are potentially difficult to understand [46]. With regard to the capabilities of the human brain, it is easier to understand small pieces of information rather than the full story at once [46]. However, to be able to sub-divide information into suitable pieces, a reasonable overview of the information must be constructed. As an example, scientific books and papers are organised into several chapters and sections. Thus, the authors present the full information in pre-selected units of information, helping the readers to follow their argument. How difficult a chapter or section (or a piece of source code) is to follow and understand also depends on the reader and is thus somewhat subjective.

(2) Using a large number of files, classes, and methods can help to divide the information (a program' source code) into smaller and more easily manageable pieces. The higher this number is the lower the average textual length of the individual pieces will be; thus resulting in small, easy to understand sections of code. However, the more files, classes and methods used, the harder the organisation of the program will become. By this is meant that if code is distributed over just one file, the programmers responsible for maintaining the system know exactly where to look for the code. When a program consists of a large number of files, the task of finding a particular source code file, might not be so easy and will be highly dependent on how the files are managed (subdirectories and labelling). A large number of classes will most likely result in a design, which is difficult to understand. Similarly with a large number of methods within a file or a class.

In summary, a balance between section length and manageability should be found. Long code sections are considered less desirable, but on the other hand, short sections may not always provide an ideal solution [47]. However, to understand the underlying complexity of a code section, the actual code distribution i.e. the organisation of the code between classes and files, has to be known beforehand.

For this dissertation, this last point is of great importance, and thus will be discussed in more detail. Complexity metrics measure particular quantifiable aspects of the source code, and store the quantity measured in a measure for a particular program, file, class or method. Thus, a measure produced can be defined for program, file, class, or method scope. To understand the individual measure, the measure has to be seen in relation to the environment

[3] At which point a file, class, or method is to be considered as long is difficult to decide [47]. As stated above, the goal of the code distribution aspect in this dissertation is not to define an optimal length, but to understand which code distribution is used for a particular program.

of the related scope. By just looking at the weight measure of a unknown person, for example, you will see a value, but to grasp the meaning of the weight measure for this particular person, you most likely either see him/her in person or consider his/her height, age, and gender measures. Thus, you are consulting the person's other physical characteristics.

The same goes for complexity measure of a program (in the example above replaced by the weight measure for a person). Just seeing a value of 500 for this complexity measure will not tell us much, unless we either look at the program in detail or consider other measures describing the program scope environment.

A program's code distribution can be seen as analogous to the person's height information in this matter

3.6 Summary

Aspects of measurement theory and, internal and external code attributes are all part of the foundation for later chapters. The definition and meaning of a particular software metric has to be kept in mind when the measure produced are interpreted. One difficult to measure internal attribute is code complexity. The code complexity can not be captured in a single measure [1, 33]. To describe the underlying complexity, software metrics targeting different aspects of software complexity should be combined. Furthermore aspects about the code distribution of a program require to be considered when complexity measures are interpreted.

4 Software metrics: Static code metrics

This chapter gives an overview of existing static code metrics. Furthermore, the important software metrics for this study are explained and discussed. Not all metrics listed will be applied in the later code measurements (see in Chapter 6 Table 6.1 for an overview of the metrics used).

4.1 Introduction

This section illustrates the most common software metrics in the area of static code analysis. As well as illustrating the metrics, the areas of application and the advantages and disadvantages were taken into consideration. In addition, some examples are provided to clarify these metrics in more detail.

Static code metrics are applied to quantify attributes of source code before run-time; such attributes are size, reuse, coupling, complexity. See Figure 3.1 for further examples. Basic and derived metrics are presented and discussed in this chapter. Basic software metrics are directly applied; whereas computed software metrics are derived using existing basic software metrics.

4.2 Program size

These metrics include the lines of code including comments (LOC), lines of code excluding comments (NCLOC), number of statements, operands and operators.

4.2.1 Lines of code

Lines of code metrics count the number of lines in a program. However, some lines might be of more interest than others with regard to the measurement purpose [1, 10]; i.e. lines of code may need to be further classified.

The content of one physical line of code can be classified as follows [1]:

- Blank line (line with no content)
- Comment line
- Statement line[4]

[4] A statement can be a data declaration, a data definition, a simple statement, or a compound statement.

Line of code metrics can therefore focus on a set of aspects of this content classification. For example the line of code metric CLOC (Comment Lines of Code) focuses only on comment lines, whereas NCLOC (Non-Comment Lines of Code) focuses only on lines containing statements. In this way, several useful metrics can be defined and the following section presents them as well as their usage. A defined standard for measuring the lines of code does not exist [43].

The general positive aspects of the lines of code metrics can be seen as follows. When applied, they can give a quick overview of the size of a program. Furthermore, like other countable metrics, lines of code metrics are easy to compute and to apply. In addition, lines of code metrics are not restricted by programming language.

However, when applying lines of code metrics, it is important to know that the complexity of a program is not considered.

- Physical lines of code (LOC)

  LOC counts all lines with no regard to their content. The advantage of LOC is that it is easy to understand. [10].

- Non-comment lines of code (NCLOC)

  The metric NCLOC counts lines that are not blank and not comment lines. NLOC is also called eLOC (for effective lines of code).

- Comment lines of code (CLOC)

  The metric CLOC counts only lines of code that contain one or more comments. The quality of the comments is not considered. An interesting derived measure in this context is the comment rate (CLOC/LOC) which is described later in Section 4.6.1.

4.2.2 Number of constructs

The program size can also be measured in terms of number of programming constructs used within the code. The number of constructs is independent of the number of lines since several constructs may appear on one line for example writing several statements on one line.

- Numer of statements (#STAT)

  The metric #STAT counts the total number of statements in the program. Since one source code line may contain several statements, the number of statements may be higher

than the number of non comment lines of code (NCLOC) [1]. Counting the number of statements in the code is not as straight forward as counting the lines of code. Here it depends on how a statement is defined.

☐ Number of operators (N1)

Operators are all keywords of a programming language (such as while, for, and switch) and symbols, which represent an action.

☐ Number of operands (N2)

Operands are all code items which represent data (such as variables, constants).

☐ Halstead program length (HL)

The Halstead program length is one of the linguistic software metrics[5] introduced by Maurice Halstead in 1977. The so called Halstead software science metrics were created in order to measure the program size (which is discussed in this section) and the textual code complexity (see Section 4.3) [2, 13].

The Halstead length (HL) is based on counting the total number of operators (N1) and operands (N2). Halstead introduced the Halstead length of a program as an estimation of the length of the program excluding comments as an alternative to NCLOC [13]. The metric definition is as follows:

$$HL = N1 + N2$$

As such HL is thus a well defined metric, defined in terms of the number of instances of operands and operators used. [48] [1] [14].

4.2.3 Comments and examples

Program size metrics (especially the lines of code metrics) are widespread and accepted. Care must be taken when a lines of code metric is discussed, to distinguish between LOC, NCLOC and CLOC. The lines of code metrics are a reasonable indicator for a program's size and also offer potential for derived metrics. Following is an example (Table 4.1) to illustrate how the different program size metrics vary in the measures produced.

---

[5] Linguistic metrics are quantifying properties of program or specification text without the text's semantics. For example : lines of code, number of statements, number of unique operators

	LOC	NCLOC	CLOC	#STAT	HL
// switches the values of two integers	1		1		
function switch(int &a, &b) {	1	1		1	7
Int tmp;	1	1		1	2
tmp = a; a = b; b = tmp; // switch values	1	1	1	3	9
}	1	1			1
	5	4	2	5	19

Table 4.1: Program size example

4.3 Code distribution

Section 3.5 illustrated the importance of considering the code distribution aspects for the interpretation of measures produced. This section gives a brief overview on how the code distribution can be quantified.

4.3.1 Measuring code distribution

The code distribution of a program can be measured by considering the size (i.e. length) of the individual files, classes and methods together with the number of these files, classes and methods. The files, classes and methods are considered to be program/code sections in the program, which hold parts of the code. The following metrics measure for a program section group defined (e.g. all methods) the code distribution among these code sections defined.

4.3.2 Code distribution metrics

- Number of program sections

The Number of program sections metrics count the number of specific program sections in a program. Within this group of metrics are the following metrics:

- The number of file (#files)
- The number of classes (#classes)
- The number of methods (#methods)

Combined with the program size metrics, these metrics can give information about the organisation of the underlying program. For example, LOC/#files gives the average lines of code per file. A pathological case is the monolithic program i.e. all the code is in one file.

☐ Size per program section

The size per program section metrics measure the size (length) for particular code sections (such as methods). As size metric, one of the program size metrics listed can be selected. For example the Statements per method metric measures the number of statements in a method. Here, extremely long methods can be difficult to comprehend. Refer back to Section 3.5 for a discussion on code distribution aspects.

4.3.3 Summary

The code distribution metrics are simple to understand and compute. In order to understand the code distribution of a program, more than the average values (e.g. average LOC per file) require to be considered.

4.4 Halstead code complexity

4.4.1 Halstead's software science

The software science metrics, introduced by Halstead in 1977, were created as means of quantitatifying the complexity of a program [2, 13]. As linguistic software metrics[6], however, the Halstead metrics were found not to capture the complexity of a program, but the textual complexity of the code [14].

The Halstead software metrics are a good example of how a set of derived metrics can be created from a set of simple metrics. Halstead bases his software metrics on counting the number of operators and operands (see Section 4.2.2.). Halstead further classified into total and distinct number of operators and operands. The four resulting simple metrics are as follows [14, 48]:

n1 : number of distinct operators

n2 : number of distinct operands

N1 : number of actual operators

N2 : number of actual operands

In the following section the Halstead software science metrics are explained. The Halstead program length (HL) was previously introduced in Section 4.2.2)

[6] Linguistic metrics are quantifying properties of program or specification text without the text's semantics. For example : lines of code, number of statements, number of unique operators

4.4.2 Halstead's complexity metrics

☐ Halstead vocabulary (Hvoc)

The Halstead vocabulary is an indicator for vocabulary size of a program. It is calculated as the sum of the number of unique operators and operands [13].

$$Hvoc = n1 + n2$$

A small program vocabulary can indicate two possible aspects. (1) In the program measures, the same operands and operators are used frequently, thus making the program less complicated to understand. (2) The program measured uses a small number of operands and operators, as it is small in program size, which consequently results in a small program vocabulary.

☐ Halstead Volume (HV)

Halstead's volume (HV) metric was intended by Halstead to measure the volume of a program.[48]. For the computation of HV, the number of operations performed and operands handled are considered [1]. The computation of V is performed as follows.

$$V = HL * \log_2(Hvoc)$$

The Halstead volume is a "count of the number of mental comparisons required to generate a program" [13]. Halstead believed that the Halstead volume indicates the size of any implemented algorithm.

☐ Halstead Difficulty (HD)

The program difficulty was intended to measure the difficulty to implement the program [13].

$$\mathrm{HD} = (\mathrm{n1} * \mathrm{N2}) / (2 * \mathrm{n2}) = 0.5 * (\mathrm{n1} / \mathrm{n2}) * \mathrm{N2}$$

☐ Halstead Effort (HE)

According to Halstead [13], the program effort, as the ratio of program volume and program level, represents the number of mental decisions (effort) required for the implementation of the program.

$$\mathrm{HE} = \mathrm{HD} * \mathrm{HV}$$

4.4.3 Comments on the Halstead metrics

Halstead's software science metrics are based on countable measures and therefore can be computed easily. However, the rules for identifying operators and operands have to be determined for every programming language. The Halstead difficulty and effort have been much discussed [48]. Halstead intended them as complexity metrics, but they have been under severe criticism since their publication. The main criticism is that the measurements show lexical and textual complexity rather than structural or logical complexity (for more on complexity see Chapter 3.4). Some authors [14, 49] argue that when the drawbacks are known and accounted for, the Halstead software science metrics can deliver good results.

4.5 Control flow complexity

4.5.1 Simple control flow metrics

The metrics presented help in later experiments, in combination with the complexity metrics (such as CC or Halstead difficulty), to gain information about the structure and complexity of a program.

- Nested block depth (NBD)

  The nested block depth metric measures the depth of conditional nesting in a method or module. The nesting depth is indicated by the width of the methods/modules flow graph (see Figure 4.1). Therefore the metric is an indicator of complex control flow within the program. Deeply nested conditional statements increases the conceptual complexity of the code and are more likely to be error-prone [38].

- Brach percentage (BP)

  This metric considers the percentage of statements that cause a break in the sequential execution (i.e. creating a branch in the programs flow graph) over all statements (#STAT). A high branch percentage indicates a complex control flow within the program.

4.5.2 Cyclomatic and Essential Complexity

☐ Cyclomatic Complexity (CC)

The Cyclomatic Complexity, which was introduced by Thomas McCabe in 1976 [12], measures the structural complexity of a module by counting the number of linearly-independent paths through the program module. Cyclomatic Complexity is sometimes referred to as McCabe complexity or simply as program complexity. For a flow graph F, the Cyclomatic Complexity is calculated from the number of arcs (e) and the number of nodes (n) as follows:

$$V(F) = e - n + 2$$

In a given flow graph example shown in Figure 4.1, we have 8 edges and 7 nodes. The Cyclomatic Complexity for this flow graph is calculated as follows:

$$CC = v(F) = e - n + 2$$
$$CC = v(F) = 8 - 7 + 2 = 3$$

The number of linearly-independent paths through the module can be clearly captured with flow graphs. Another way of calculating the Cyclomatic Complexity for a module is to count the number of single/multiway decisions, which in program languages are indicated by keywords (such as if and switch). In Figure 4.1, the intersections illustrate the decision points, which in the source code is presented through if – else constructs.

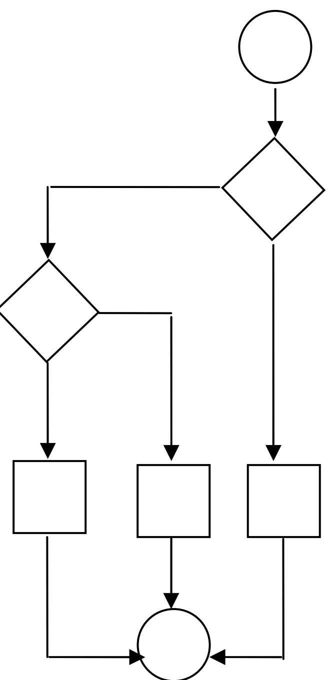


Figure 4.1: Control flow

Because of CC's simple computation and the ability to indicate how difficult a module will be to test or maintain, it is widely used and used it several software measurement tools [1].

The key benefits of Cyclomatic Complexity can be seen as following [5, 50]:

- The metric is independent of source code layout and only loosely bound to the programming language used, as different single/multiway decision statements (such as if, case, switch) might be available. The decision making constructs are common among programming languages but the naming (e.g. switch and case) of these can differ.

- As the cyclomatic number indicates the number of independent paths through the program module, CC itself represents the minimal number of tests needed to confirm a program's preconditions [51][34].

The disadvantage of this metric is that the CC metric gives no representation of relationships between methods. Thus, the metric does not show coherence and coupling of methodss [5].

According to van Doren [52], CC finds its areas of application in code development risk analysis, change risk analysis in maintenance, test planning, and reengineering. When it comes to threshold values for CC, McCabe suggested the value of 10 to be a good indicator for when a module might need to be reviewed [1]. In 1994, Grady concluded that the CC value for a module under maintenance should not exceed 15 [1].

- ☐ Extended Cyclomatic Complexity (ECC)

This metric is an extension of the McCabe Cyclomatic Complexity metric. ECC extends the original CC metric by including Boolean operators in the decision count [53]. I.e. the value of ECC is always higher or equal to the CC value of a module measured. Regarding the computation of ECC, the general CC computation is extended by increasing the complexity count whenever a Boolean operator (And, Or, Xor, Not) is found within a conditional statement (see Cyclomatic Complexity). In software measurement tools Extended Cyclomatic Complexity is often used (see Chapter 5) instead of the CC metric, since a Boolean operator increases [51] the internal complexity of a branch. E.g. the same complexity level can be achieved through sub-dividing the code into sub-conditions without Boolean operators.

- ☐ Essential Complexity (EC)

In 1977, McCabe [12] proposed another complexity metric named Essential Complexity metric [1]. The essential complexity metric measures the level of structured logic in a program [1, 51, 52]. For the computation of EC, the Cyclomatic Complexity of program's reduced flow graph is evaluated. For the control flow graph reduction, structured programming primitives (sequential, selection and iteration nodes) are removed until the graph cannot be reduced further [12, 51]. I.e. the reduced flow graph will not produce a higher CC value then the original flow graph, having EC always less or equal to CC. Thereby the definition of EC looks as follows:

EC = CC – m = v(F) – m

Where m is the number of subgraphs (in the control graph) with sole entry and exit points [1, 12]. Figure 4.2 illustrates a control flow graph reduction to a single statement in the essential complexity computation process. I.e. the corresponding program sequence can be reduced to a program of unit complexity [54].

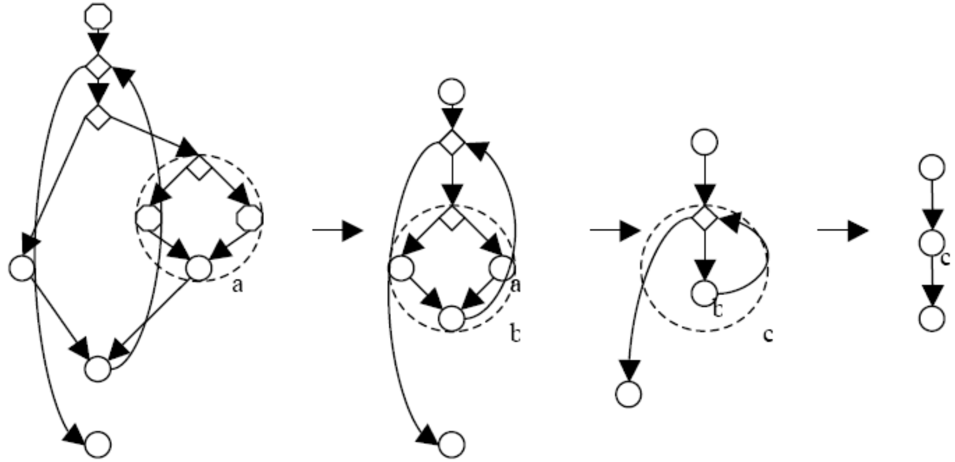


Figure 4.2: Tree reduction [51]

Not all control flow graphs can be reduced in the same manner. Further, McCabe [51] states that even primitive constructs containing module calls can be removed. At this point only a brief introduction to the essential complexity is presented. For further readings on the topic please see the McCabe literature listed under Chapter 10, especially [51].

4.5.3 Comments on the control flow complexity metrics

Further complexity measures have been designed especially for object oriented designs. These are discussed in Section 4.7. So, several complexity metrics that try to quantify software complexity exist. However, as Fenton [33] and other authors [8] have expressed, the complexity metrics have to be treated with caution. The metrics presented can be used as indicators of program complexity, thus, should not be used alone as a basis for the general complexity of a system. Laird [34] states that complexity metrics can be more helpful when used in combination. In Chapter 3.4 software complexity is discussed in more detail.

4.6 Code maintainability

Code maintainability is, as Sommerville [38] sees it, a code attribute that characterizes how easy existing source code can be modified to provide new or changed functionality, correct

faults, improve performance, or adapt to a changed environment. According to ISO 9126, can maintainability be classified into the attributes stability, analyzability, changeability and testability. Since software products should preferably stay maintainable [55], there have been several attempts to quantify the maintainability of a software system. The most widely spread software metric trying to quantify the maintainability is the Maintainability Index which is presented in 4.6.2 [56].

4.6.1 Comment Rate

The Comment Rate (CR) metric measures the ratio between the number of physical lines, LOC, and CLOC and therefore gives an indicator of how much the source code is commented. A high comment rate can be beneficial in terms of code maintainability, however the quality and usefulness of the code has to be considered to determine this. The Comment Rate does not quantify the quality of the comments but the quantity, thus has to be treated with care, when the code maintainability is analyzed.

4.6.2 Maintainability Index (MI)

The Maintainability Index (MI) was introduced by Dr. Paul W. Oman [49] in 1992 as a predictor of maintainability. The MI has been subject of several studies [53, 56, 57], which state a high correlation between a system's maintainability and the MI value. Furthermore, the MI metric is still today a widely used indicator for maintainability [53]. Oman combined in the MI the following four widely used unrelated software metrics [56]:

- The average Halstead Volume (Avg_HV) per program section (e.g. file, class, ..)
- The average Extended Cyclomatic Complexity (Avg_ECC) per program section
- The average number of lines of code (Avg_LOC) per program section
- The percentage of comment lines (CR) per program section

These metrics are discussed in detail in preceding subsections of this chapter.

The Maintainability Index is defined as

$$\begin{aligned} MI = \; & 171 - 5.2 * \ln(\text{avg_HV}) \\ & - 0.23 * \text{avg_ECC} \\ & - 16.2 * \ln(\text{avg_LOC}) \\ & + 50 * \sin(\sqrt{(2.4 * CR)}) \end{aligned}$$

According to [56, 57] the computation of the CR (Comment Rate) in the MI metric is optional. A general understanding is that good commented code is easier to maintain, however the CR does not indicate the quality of the comments written. The coefficients used in the definition were derived throughout numerous calibration tests of the MI metric [56]. On application of the MI metric, measures typically range from 100 to -200 [53]. Here the higher the MI value, the more maintainable [53, 56, 57] a system is considered to be.

The advantages of MI:

The calculation of the MI metric is considered simple and straightforward [56]. In addition, through the combination of the McCabe and Halstead metrics, the MI targets individual aspects of software complexity (structural & lexical complexity).

The disadvantages of MI:

Since the MI is derived from three or optional four unrelated metrics, a change in value in the MI requires examination of all the metrics used in the derivation to furnish a explanation for the change [57]. In object oriented systems the possibility of a high number of methods with low complexity (e.g. getter & setter) may affect the MI via the ECC. The use of an average Extended Cyclomatic Complexity can result in flawed outcomes [57].

4.6.3 Maintainability Index summarized

As van Doren [56] states, there have been several attempts to quantify software maintainability. MI is a good approach to do so as it tries to quantify a program's maintainability through the combination of widely-used and commonly-available metrics. The main drawback of MI, as it produces a composite measure, is problem of identifying the particular cause for a value change.

4.7 Object oriented metrics

The metrics presented in this section were designed to fit the object oriented software approaches. The work of Chidamber and Kemerer [20] is a widely used and frequently cited [58] set of object oriented metrics. In addition, the MOOD metrics set from Abreu [21] is presented and discussed. More object oriented metrics exist and are briefly mentioned. However, for a more complete picture [59], [60] and [18] are suggested.

4.7.1 Chidamber and Kemerer metrics (CK)

The metrics set introduced by Chidamber and Kemerer [20]. In 1994 considers structural characteristics and design of object oriented programs and is considered one of the most widely spread OO metrics [59]. The CK metrics set contains six metrics which are presented and discussed below:

- Weighted Method per Class (WMC)

This metric measures the total complexity of a class. For this, the number of methods and the complexity level of the methods involved are considered. The authors state that the number of methods and the complexity of these are indicators for the time and effort required to develop and maintain the class [20]. The WMC metric is calculated as follows:

$$WMC = \sum_{i=1}^{n} c_i$$

Here c_i is equivalent to the complexity of the respective method and n the total number of methods involved. To construct the WMC metric flexible, the authors intentionally did not specify a method for computing the complexity c_i and thereby left the users to choose the best fitting approach [20]. The available software measurement tools use the Cyclomatic Complexity metric (CC) for computing the complexity of the methods. However other static complexity metrics could be used instead (e.g. ECC) [18]. A high value of the WMC metric indicates a class which should be redesigned into smaller classes [20] [18].

- Depth of Inheritance Tree (DIT)

The Depth of the Inheritance Tree (DIT) is defined as the longest path from the current node back to the root of the tree [20]. With a growing inheritance tree, more classes are involved, the design complexity as a whole grows and it becomes more difficult to

calculate the behaviour of a class [59]. A lower DIT value is worthwhile, however a low average DIT value may suggest the presence of little reuse by inheritance [58].

- Number of Children (NOC)

The Number of Children metric measures the level of reuse in a system by counting the numbers of immediate subclasses of a defined object in the class hierarchy. A growing NOC value indicates a higher level of reuse. However, the higher the NOC value the higher the amount of testing required due to the increase in class responsibilities [59].

- Coupling between Object classes (CBO)

For each object this metric returns the number of objects to which it is coupled. Two objects are seen to be coupled if object methods or instance variables are used from each other. High coupling between classes complicates the reuse of these modules, since the classes will require exactly the same conditions in the new system and are thereby limited in maintainability and reusability. The CBO value should be kept low, since higher values indicate more testing effort [59].

- Response for a Class (RFC)

The RFC metric measures the number of called methods plus the number of class owned methods. The RFC measure reveals the maximum number of methods which can be invoked through a message to an object of this class [20]. A high value is an indicator for high class design complexity [59] and lower values indicate higher reusability due to greater polymorphism.

- Lack of cohesion in Methods (LCOM)

The LCOM metric measures the lack of cohesion within a class by evaluating the dissimilarities between methods of a class. A high dissimilarity between methods indicates that the class fulfils different purposes and should be divided into several subclasses. In addition to the lack of cohesion, the metric is intended by [20] to reveal the complexity of a class and the design quality of the system [61]. CK defined the LCOM metric as follows [20]:

$$LCOM = \begin{cases} |P| - |Q|, & \text{if } |P| < |Q| \\ 0, & \text{otherwise} \end{cases}$$

where

P = all disjoint attribute sets used by class methods

Q = all not disjoint attribute sets used by class methods

However, since the introduction of LCOM, the metric has been under severe criticism and other definitions for LCOM have been proposed. Two suggestions for LCOM by Henry & Li and Henderson-Sellers will be presented later in this chapter.

4.7.1.1 CK summary

The CK metrics present the headstone for object oriented design metrics and are the basis for several other OO metrics, some of which will be presented subsequently in this chapter. Regarding the criticism on the six CK metrics, WCM and LCOM have emerged as the most criticised [1]. The WMC's advantage of leaving the metric adjustable regarding the complexity method used can be seen as a disadvantage from the point of view of lack of precise description, since CK does not indicate which complexity metric to use [58]. In addition, the complexity metrics used need to be adapted to object oriented design approaches. Henderson-Sellers [18] points out that setter and getter methods can bias the WMC metric by considerably lowering the mean complexity measure if a non OO complexity metric is applied, for example the Cyclomatic Complexity (CC). Setter methods are methods with no other purpose than setting a value or state of a class property. Getter methods are used for retrieving a class property. Additionally, [62] highlights that the WMC description altogether lacks the differentiation about which metrics should be considered in the WMC. Regarding the NOC value, Reißing [62] and Sarker [59] point out the ambiguity of striving for a high or low value. Both high and low values have drawbacks. The LCOM metric was criticized heavily and has been target for many suggestions for improvement, because the version proposed by Chidamer and Kemerer does not react sensitively enough to cases of high cohesion and lacks coverage of special cohesion cases [18].

4.7.2 The MOOD metrics set

The MOOD (Metrics for Object Oriented Design) metrics set developed by Abreu et at. [21] can be categorized into the four object oriented paradigms namely encapsulation, inheritance, coupling and polymorphism [63]. Abreu based the six software metrics in the MOOD metric set on two key elements of object oriented programming: object methods (M) and object attributes (A). Furthermore, the visibility (visible/invisible) of methods and attributes outside the class is considered [21]. The six Abreu metrics (MHF, AHF, MIF, AIF, POF and COF) are presented below.

4.7.2.1 Encapsulation

The MHF and AHF metrics measure the invisibility of methods and attributes in classes. The invisibility of a method/attribute is the percentage of the classes/(total number of classes) from which the method/attribute is not visible [58].

☐ Method Hiding Factor (MHF)

The MHF metric measures the invisibility of methods in all classes (TC).

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{j=1}^{TC} M_d(C_j)}, \text{ where } V(M_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(M_{mi}, C_j)}{TC - 1}$$

M_d (C_i) = the number of all methods defined in the $class_i$

A MHF measure of 0% indicates that all methods are defined as visible, whereas a MHF measure of 100% indicates that all methods are hidden. The latter scenario provides little functionality, since the classes in the system cannot communicate well with each other [59].

☐ Attribute Hiding Factor (AHF)

The AHF metric computation is very similar to MHF. Instead of hidden methods in a $class_i$, hidden attributes in a $class_i$ are considered. Thus, the AHF metric measures the invisibility of attributes in all classes (TC).

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{j=1}^{TC} A_d(C_j)}, \text{ where } V(A_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(A_{mi}, C_j)}{TC - 1}$$

According to object oriented design principles, attributes should be kept hidden within a class [55] Therefore, the AHF measure is expected to be high, indicating high attribute invisibility.

4.7.2.2 Inheritance

The MIF and AIF metric measure the inheritance level in a system.

☐ Method Inheritance Factor (MIF)

The MIF metric is computed through the number of inherited methods (M_{inh}) in all classes (TC) and indicates the percentage of inherited methods in a system.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

M_a (C_i) = the number of all methods in the $class_i$

M_a (C_i) = M_d (C_i) + M_{inh} (C_i)

- Attribute Inheritance Factor (AIF)

The AIF metric is computed through the number of inherited attributes (A_{inh}) in all classes (TC) and indicates the percentage of inherited methods in a system.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

A_a (C_i) = the number of all attributes in the $class_i$

A_a (C_i) = A_d (C_i) + A_{inh} (C_i)

4.7.2.3 Polymorphism

Polymorphism can be measured as degree of overriding in the inheritance tree.

- Polymorphism Factor (PF)

The PF metric is computed as the number of methods that redefine inherited methods divided by the actual number of possible different polymorphic situations. The PF is defined as follows:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} \left[M_n(C_i) \times DC(C_i)\right]}$$

M_n (C_i) = the number of new methods defined in $class_i$

M_o (Ci) = the number of overriding methods in $class_i$

DC (Ci) = the descendants count in C_i

TC = total number of classes

4.7.2.4 Coupling

- Coupling Factor (CF)

The CF metric measures the coupling degree excluding inheritance coupling. The computation is done as follows:

$$COF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_client(C_i, C_j)\right]}{TC^2 - TC},$$

where $is_client(C_c, C_s) = \begin{cases} 1 & if\ C_c \Rightarrow Cs \wedge Cc \neq Cs \\ 0 & otherwise \end{cases}$

The numerator represents the actual number of couplings not attributable to inheritance. TC is defined as the total number of classes. The denominator is computed as the maximum number of couplings with TC classes. The CF measures can vary between 0 and 1. High values indicate a very high coupling degree and should be avoided [59].

4.7.2.5 MOOD Summary

The MOOD metrics have become an accepted object oriented metrics set. They cover the object oriented aspects of encapsulation, inheritance, coupling and polymorphism. Studies indicate that the metrics are valid [58, 59, 63] and that only minor problems, such as the imprecise definition of the concept "attribute", exist [63].

4.7.3 Henry and Kafura metrics

In 1981 Henry and Kafura introduced metrics of intermodule coupling [18].

4.7.3.1 Fan-in

The Fan-in (Informational fan-in) metric measures the fan-in of a module. The fan-in of a module A is the number of modules that pass control into module A.

4.7.3.2 Fan-out

The Fan-out metric measures the number of the number of modules that are called by a given module.

4.7.3.3 Information Flow

The Information Flow metric, which was proposed as structural complexity metric, is derived from a modules fan-in and fan-out measures. The metric is calculated as the square of the product of the fan-in and fan-out of a single module.

4.7.3.4 Summary

High values for Fan-out indicates a high complexity of a certain module due to the control logic needed to coordinate the called methods/modules [38]. A high value for Fan-in indicates the particular module is tightly coupled to the system. Tight coupling should be avoided if possible, since in a tightly coupled system a single module change will result in changes to other modules [38].

4.7.4 Other OO metrics

More OO metrics have been proposed but are not as widely spread and analyzed as the CK and MOOD set [58], and are therefore are not discussed in detail.

4.7.4.1 Li and Henry

Li and Henry reused five of the six Chidamber and Kemerer metrics and introduced five new metrics. These five new metrics are listed below:

- Message Passing Coupling (MPC)
- Data Abstraction Coupling (DAC)
- The Number of Methods (NOM)
- The number of semicolons (SIZE1)
- The number of properties (SIZE2)

4.7.4.2 Chen and Lu

Chen and Lu developed an object oriented metric set in 1993 which mainly focuses on class complexity [58]. These metrics are the follwing:

- Operation Complexity (OpCom)
- Operation argument complexity (OAC)
- Attribute complexity (AC)
- Reuse (Re)
- Operation coupling (OpCpl)
- Class coupling (ClCpl)
- Cohesion (Coh)
- Class hierarchy (CH)

4.7.4.3 Moreau and Dominick

- Message vocabulary size (MVS)
- Inheritance complexity (IC)
- Message domain size (MDS)

4.7.4.4 Abbott, Korson and McGreggor

- Interaction level (IL)
- Interface size (IS)

4.7.4.5 Hitz and Montazeri

- Object Level Coupling (OLC)
- Class Level Coupling (CLC)

4.8 Summary

The Cyclomatic Complexity (CC) and the Halstead complexity (HC) metrics are important contributions to the area of software measurement, but should be used be used in combination with other metrics to provide a context in which these two metrics (CC, HC) may be interpreted.

A wide variety of object oriented metrics have been designed to quantify aspects of object oriented systems. The CK metric set focuses on class hierarchy, complexity, coupling and cohesion. Whereas, the MOOD metric set focuses on encapsulation, inheritance, message passing and polymorphism within the system level. Other object oriented metrics that have not been discussed in this chapter, can be reviewed in [18, 19, 58-62].

5 Software measurement tools: Static code analysis

This chapter presents the results from the survey of available static code analysis conducted. A list of available software measurement tools is presented as well as an analysis and comparison of these tools in order to select the tools used in this study.

5.1 Introduction

Several commercial and open source software measurement tools are available, but the majority of the static code analysis tools were found to mainly focus on one specific programming language. Since the implementation of software metrics can differ [2] from one tool to another, the application of several tools for the same metric should be avoided. For this project, tools supporting the analysis of C, C++ and Java programs were required. The following section presents a list of available software measurement tools which focus on code analysis.

5.2 List of tools

Many static code analysis tools exist, both commercial and non commercial. The following list of tools was created during the software measurement tool survey, presenting an overview on the number and the names of the tools available. The tools were found with the help of internet search engines, wikipedia and 'http://www.testingfaqs.org'.

A-C

AccVerify SE for FrontPage, Aivosto Project Analyzer, ASSENT, Axivion Bauhaus Suite, Ccount, CCCC, Cleanscape LintPlus, ClearMaker, CMT++/CMTjava, CodeCompanion, CodeReports, CodeSurfer, Coverity Prevent and Extend

D-K

DeepCover, Dependency Walker, Discover, Eclipse Metrics, Enerjy CQ2, ES1, ES2, Flawfinder, floppy/fflow, ftnchek, FxCop, Hindsight/SQA, HP_Mas, Jtest, JHawk, JMetric, JDepend, Jstyle, Klocwork K7, Krakatau, Essential Metrics

L-P

LDRA Testbed (static analysis), Lind, Logiscope, Malpas, McCabe QA, METRIC, Metrics Tools, NStatic, Optimal Advisor, ParaSoft CodeWizard, Project Analyzer, PC-lint/FlexeLint, PC-Metric, PMD, PolySpace, Plum Hall SQS

Q-Z

QStudio for Java Pro, RSM (Resource Standard Metrics), Safer C Toolset, SCLC, SDMetrics, SemmleCode, Splint, SofAudit, SourceMonitor, SSW Code Auditor, STATIC, Team in a Box, Together ControlCenter

Table 5.1: Static code analysis tools list

Since the above list is long, only a handful of tools were considered for the tool comparison. Here, the tools which passed the given requirements were examined closer. These requirements included the programming languages, the metrics supported, the reliability and the availability. Only tools with C, C++ and Java support were of interest. A mix of software measurement tools can bias the measures produced for different programming languages (as indicated in Section 5.1), thus one tool should cover all three selected programming languages. In addition, the measures produced should be reliable. Thus, the metric computation was expected to come close to originally intended metric definition (listed in Chapter 4). Moreover, for commercial software measurement tools, a sufficient trial version was expected to be available on request.

Finally three commercial and two non commercial software measurement tools were selected based on the requirements listed. In the following sections the selected tools, namely CCCC, CMT, Essential Metrics, Logiscope and SourceMonitor, are introduced. In Section 5.3 the tools are analyzed and compared with regard to the metrics support, the metrics implementation, reliability, and usage.

5.2.1 CCCC

CCCC (C and C++ Code Counter) is a free software measurement tool for static code analysis. Tim Littlefair [35] developed the tool as a part of his PhD software metric project in 2001 [64]. Since then, it has been under continuous maintenance and is available through the open source development and download repository SourceForge.net. The programming

languages supported by CCCC are C, C++ and Java. Earlier versions of CCCC also provided Ada (83, 95) programming language support. This static source code analysis tool supports a variety of software metrics including LOC, Cyclomatic Complexity and metrics proposed by Chidamber & Kemerer and Henry & Kafura. The tool can be found at http://cccc.sourceforge.net/.

5.2.2 CMT

The commercial software measurement toolset CMT (Complexity Measurement Tool), containing the tools CMT++ and CMTjava, is a product of the company Testwell and is distributed through Verisoft. These tools are intended for static source code analysis and mainly focus on code complexity metrics (such as the Halstead metrics or Cyclomatic Complexity). CMT++ supports the analysis of C and C++ programs and CMTjava similarly supports java programs. The traditional software metrics implemented are CC, MI, Halstead metrics and LOC metrics. The tools are advertised as being able to measure many thousand source lines per second. More Information about the tool can be found at the Testwell website at http://www.testwell.fi/.

5.2.3 Essential Metrics

Essential Metrics is a commercial command line software measurement tool which is distributed through Power Software ("http://www.powersoftware.com/"). Essential Metrics supports the static analysis of C, C++ and Java source code. As well as traditional software metrics, such as LOC and Cyclomatic Complexity, Essential Metrics also supports several object oriented metrics (such as the CK and MOOD metrics). More Information about the tool can be obtained from the company website at http://www.powersoftware.com/em/. As well as Essential Metrics, Power Software also offers five other products that are intended for software measurement (such as tools for software project measurement).

5.2.4 Logiscope

Telelogic Logiscope is a commercial software package developed for software quality assurance. Static and dynamic complexity analysis of software code can be performed and a wide variety of software metrics (over 40) are supported in Logiscope. As graphical software measurement tool Logiscope supports metrics for C, C++, Java and Ada. Furthermore,

functions for new custom software metrics can be defined. This tool was provided by Telelogic as a trial version for this dissertation. More information about the tool can be gathered on the Telelogic website at http://www.telelogic.com/.

5.2.5 SourceMonitor

The freeware tool SourceMonitor by Campwood Software is a graphical software measurement tool with focus on static code analysis comparison and code review. This means SM provides functionality for comparing source code versions via checkpoints. Additionally, SourceMonitor supports a wide variety of programming languages including C, C++, C#, Java, VB.NET, Delphi, Visual Basic (VB6) and HTML. The measures produced may be graphed, printed, or exported and analyzed through diagrams. However, only a small selection of software metrics is supported by SourceMonitor. The tool can be found at http://www.campwoodsw.com/.

5.3 Tool comparison

After the introduction to the selected tools in the previous section, a more detailed examination is presented in this section. Here, the focus lies on the metrics support, the metrics implementation, reliability, and usage.

5.3.1 Tools in detail: Usage and the metrics supported

CCCC (version 3.1.4)

The documentation provides a section for the metrics description and a separate section for the metric computation. In addition, the documentation explains the application of the command line tool as well as the options available. However, since CCCC is a command line tool, extra time for understanding and adjusting to its usage was required. Once comfortable with the tool, various programs can be measured in one command line. As an open source project, CCCC was intended to be platform-independent and still is; however a windows installer has been added to remove the duty of compiling from the user. The results produced by running CCCC are exported as HTML and xml files. The output in the html file is well structured and colours indicate measures outwith specified boundaries. As briefly mentioned in Section 5.2, CCCC also provides object oriented metrics from Chidamber & Kemerer (CK) and Henry & Kafura besides the LOC and CC metrics. However, the CK metrics set supported is not complete, since the RFC and LCOM metrics have not been implemented in

the static code analysis tool at present. The Henry & Kafura metrics implemented are: Fan-in, Fan-out, and Information flow.

CMT (version 4.1)

CMT++ and CMTjava (version 2.1) are alike in their usage. The difference between the two tools lies in the programming languages supported. Due to the different programming languages, different programming constructs have to be taken into account. For example, the '#define' construct in C and C++ is not valid for Java which results in minor differences in the metric computation. The differences however, can also be found in tools which combine the analysis of Java, C, and C++. The graphical user interface (GUI) provided (please see the Appendix for screenshots) is very clear and the functionality is self-explanatory. The GUI is, compared to other evaluated software measurement tools like Logiscope or SourceMonitor, rather simple, and due to this simplicity further reading of the documentation is not required. Several report types are supported in CMT, and in this project, the xml output has been used. The xml output file is structured clearly and logical names are used to label the measures produced. A disadvantage of CMT is that it does not display results using graphs and diagrams. On the other hand this deficiency is acceptable, regarding the simplicity of the tools. Furthermore, the graphs required can be build manually (e.g. in excel) from the output files. The full name of CMT (Complexity Measurement Tool) indicates the integration of complexity metrics. However, the tool does not take into consideration the complexity of object oriented designs. Neither CK and Mood, nor HK metrics are implemented. The metrics covered are: LOCs, Halstead, block depth, CC and MI. Table 5.2 presents the metrics support in more detail.

Essential Metrics (version 1.2)

The available documentation for Essential Metrics is somewhat insufficient. The documentation provided after installing the software is available in the form of a 46-line long README text file, most of which is used for company contact information and acknowledgments. The essential point in the README file is to explain how to create a file list via the DOS dir command. Since more emphasis was placed on how the product can be licensed, the user needs to figure out the application of the command line tool through trial and error approaches. Information and a detailed guide for licensing the software can be found both online on the company website as well as in the installation path of Essential Metrics.

Once the use of the tool has been mastered, the lack of progress messages becomes noticeable. When running the tool, no information is given about which source code file is currently being processed or which metric is applied; the command line screen stays almost blank. Only a short start text and three characters are shown ("abc"), for which the intended meaning is unknown. The lack of error message coverage of the tool is also quite noticeable; however, this aspect is more discussed in Section 5.4.

Compared to the other tools analyzed, Essential Metrics supports a wide variety of software metrics. These range from LOC and comment ratio to Halstead metrics and Cyclomatic Complexity to the object oriented metric sets CK and MOOD. However, the Henry and Kafura metrics are not implemented. Table 5.1 presents which metrics in detail are supported by Essential Metrics.

The tool does not provide an option for selecting which metrics should be applied and therefore all metrics are applied automatically. This can create unnecessarily complex output-files. The results can be stored in html, xml, or csv file format.

Logiscope (version 6.2.30)

As indicated in the previous section, Telelogic Logiscope is a graphical software measurement package. The graphical user interface was found to support ease of use. However, not all computed metrics are accessible over the GUI (e.g. object oriented metrics are not displayed). Additionally, measuring several projects via the GUI is very time consuming. Therefore, the usage of the provided command line tools is of interest. The full functionality of each command line tool is explained in separate documentation.

The information about the tools is extensive and the variety of metrics supported is not clearly explained. Unfortunately, the list of metrics supported and information regarding the metrics is not available through the company's official webpage. Whereas, for the other software measurement tools evaluated, the software metrics supported were mostly listed on the specific company website and sufficient information about the product was given.

After getting acquainted with Telelogic and its command line tools, it was noted that the metrics supported were different for different programming languages. Thus, for C++ and Java, both being object oriented programming languages, a major difference in metric support was detected. This meant in detail that the object oriented metrics described in Chapter 4 (such as CK and MOOD) were supported for C++, but not for Java. This difference resulted in reconsidering the tools application for the code measurements planned.

SourceMonitor (version 2.3.6.1)

The build-in Documentation in SourceMonitor explains the metric computation for the different programming languages supported. Each metric is mentioned in a small paragraph. In addition, chapters on “Getting started”, explanation of the user interface, and how to export the data are provided. All in all, the documentation is complete, but the computation could have been explained in more detail. The metric implementation is not entirely clear and therefore test scenarios are required.

Another drawback of the tool is that object oriented metrics are not fully supported. By this, is meant that neither the CK nor the MOOD metrics are included and the ones which are, only barely capture OO design aspects. The two OO metrics included are the number of classes and interfaces and the number of methods per class. The metrics which are supported mainly focus on flow graphs and can be classified into complexity (cyclomatic complexity) and branch level (nested block depth) metrics. Regarding complexity aspects, the following measures can be produced:

- The line number of the most complex method, the name of the most complex method, the maximum complexity and the average complexity

All complexity measures produced are not based on the CC metric as the SM documentation indicates, but actually on the ECC metric (as later shown in Section 5.3.2). For the branch level the following measures are producible:

- The percentage of branch statements, the line number of the deepest block, the maximum block depth and the average block depth

In addition, some more measures such as the lines of code in a program are included (see Table 5.2 for a complete list).

		CCCC	CMT	Source Monitor	Essential Metrics	Logiscope
Program size & Code distribution						
LOC	Lines Of Code	-	+	+	+	+
NCLOC	Non Comment LOC	+	+	-	+	-
#STAT	Number of Statements	-	-	+	-	+
HL	Halstead Length	-	+	-	+	+
# Files	Number of Files	-	-	+	+	+
# Methods	Number of Methods	+	-	+	+	+
Textual code complexity						
Hvoc	Halstead Vocabulary	-	+	-	+	+
HV	Halstead Volume	-	+	-	+	+
HD	Halstead Difficulty	-	+	-	+	+
HE	Halstead Effort	-	+	-	+	+
Control flow complexity						
NBD	Nested Block Depth	-	+	+	-	+
BP	Branch Percentage	-	-	+	-	+
CC	Cyclomatic Complexity	-	-	-	+	+
ECC	Extended CC	+	+	+	-	-
ESC	Essential Complexity	-	-	-	-	-
Maintainability						
CR	Comment Rate	+	-	+	+	-
MI	Maintainability Index	-	+	-	+	-
Object Orientation						
WMC	Weighted Methods	+	-	-	+	+
DIT	Depth of Inheritance Tree	+	-	-	+	+
NOC	Number Of Children	+	-	-	+	+
CBO	Coupling Between Objects	+	-	-	+	+
RFC	Response for a class	-	-	-	+	-
LCOM	Lack of cohesion	-	-	-	+	+
MHF, AHF	Hiding factor	-	-	-	+	+/-
MIF, AIF	Inheritance factor	-	-	-	+	+/-
PF	Polymorphism factor	-	-	-	+	+/-
CF	Coupling factor	-	-	-	+	+/-
FI	Fan-in	+	-	-	-	-
FO	Fan-out	+	-	-	-	-
IF4	IFlow	+	-	-	-	-

Legend:

+	The tool does support the metric
-	The tool does not support the metric
+/-	Metric is for ether C++ or Java not supported

Table 5.2: Tools' metric support

5.3.2 How the original metrics are implemented

In this section a brief overview is given on the interpretation and implementation metrics provided by these tools. If a software metric can be applied by several tools, the tool chosen is that which computes the metric in the manner closest to the definition given in Chapter 4. Due to difficulties experienced, Essential Metrics and Telelogic Logiscope are not included in this comparison (more detail on this issue is presented in Section 5.4.). In order for the measures produced to be comparable, the same tool must be used.

In the following paragraphs, the metrics implementation is analyzed and compared. With regard to the LOC metrics, SourceMonitor and CMT produce different results. CMT and SourceMonitor produced different results for measuring the lines of code in a program. Although, one might expect that different tools count the number of physical lines of a program in the same way, this was found not to be the case. For a set of programs, lower values were produced in some cases by CMT and in other cases by SourceMonitor. Neither the documentation nor test cases revealed the source of the differences.

Both CCCC and CMT support the NCLOC metric, though CCCC counts per module rather than per file and not all lines are considered during the measurement.

The computation of the cyclomatic complexity needs also to be reviewed in more detail. The tools' product-feature lists indicate that the tools support the Cyclomatic Complexity (CC) metric. However, on closer inspection it was revealed that they actually have the Extended Cyclomatic Complexity (ECC) metric implemented. This feature is not necessarily a drawback, but it has to be known before the measures are interpreted. Once again, as indicated in Chapter 4, the difference between ECC and CC is that ECC considers Boolean operators in the decision count, whereas CC does not. For the three tools mentioned (CCCC, CMT and SourceMonitor), the Extended Cyclomatic Complexity is implemented as follows:

- if statements increase the complexity count by 1
- boolean operations increase the count by 1
- for and while loop increase the count by 1
- switch statements and switch cases increase the count by 1,
  however CMT does not increase the count for switch default cases
- methods increase the count by 1, however not for CCCC
- else statements do not increase the count, however in SourceMonitor they increase the count by 1

Within the scope of a file, CMT computes the Extended Cyclomatic Complexity as follows:

$$\mathit{file\,scope\,ECC} = 1 + \sum_{i=1}^{\#methods} ECC(method_i) - 1$$

Using this method, getter and setter functions (with intentionally low complexity) do not influence the complexity measure.

Only CMT supports the application of the Halstead metrics. The implementations of Halstead length, vocabulary, volume, difficulty, level and effort, which all rely on the operator and operand count, follow the equations presented in Chapter 4. Tokens in the following categories are all counted as operands by CMT: identifiers, type names, type specifiers, constants. Tokens in the following categories are all counted as operators by CMT: storage class specifiers, type qualifiers, and operators (+,-, &&, {,}…}, reserved words, pre-processors directives.

CMT has the Maintainability Index implemented according to the definition presented (see Chapter 4). On file-level CMT does not calculate the MI measures via the averages of its participating modules as suspected, but directly through averaged file-level V, ECC, and LOC measures. If the module count is zero the number 1 is taken as average divider for the three stated metrics.

The WMC metric is interpreted and implemented by CCCC as the number of methods in the class. Thereby the integrated complexity function *in this tool* (see 4.7.1) is not represented by the cyclomatic complexity of the method but by the value of 1. Since this interpretation does not seem to be that useful, this metric will not be used for the code measurements.

Other metric implementations will not be discussed further at this point. The documentation provided by the tool can be consulted for further understanding.

5.3.3 How to compare the results

Since the software measurement tools listed, computed the software metrics differently, only one specific tool per metric was chosen for all three programming languages. Using different tools for the same metric can result in different measures. In this dissertation, the application and interpretation of these metrics were of interest rather than the reasons for computational differences. Therefore, the most suitable software measurement tool for each metric was selected at the author's discretion. The list, explaining the metrics tool selection, is given in the measurement description in Table 6.2.

5.4 Difficulties experienced while comparing the tools

During the tool comparison, difficulties with the Essential Metrics and Logiscope tools were experienced. In this section these problems are presented.

Essential Metrics worked as advertised for smaller software projects, such as the test code examples and the students' code. However, for six of the larger code programs measured (7Zip, Emule, FileZilla, Gimp, Miranda and VIM) the tool did not complete the code measurements. For these larger projects, the tool became literally 'stuck' (see the Appendix for the screenshots in A.5.4) and no measures were produced. Other software measurement tools required only a few seconds, whereas Essential Metrics was not finished after several hours. The problem was not system dependent, because the same problem occurred on other computers. Since the insufficient support for process and error message handling (see Section 5.3.1) did not help in finding the source of the problem, the issue was brought up with the vendor. However, the vendor did not show much interest in fixing the problem and a later request for a debug mode version (displaying encountered problems) of the tool was left unanswered. Therefore, the decision was made to omit the tool from the later code measurements and to forgo the metrics supported.

Also with the Logiscope tool, difficulties were encountered. These difficulties were related to a lack of clarity in the vendor product description. The initial tool selection was made due to the wide variety of software metrics and the programming language support as well as to previous studies [28] in which the tool has been used .

The product description did not distinguish between the metric support for Java and C++, neither on the official website nor in the PDF product description (which was available for download). Therefore, the author assumed that metric support for the two object oriented programming languages C++ and Java were very similar with regard to the available object oriented metrics. That this was not the case is indicated in Table 5.2. The difference was discovered while studying the manuals provided. The manuals did not state this difference directly, but the missing software metrics were simply not listed in the Java manual. As one of the requirements for this dissertation was to avoid the mixing software measurement tools for different programming languages (see Section 5.1) the Logiscope tool was not selected for the later code measurements.

5.5 Summary

As presented in this chapter, a large number of software measurement tools exist. The analysis and comparison of the selected tools indicated a difference in the metric implementation and in the metric sets supported. Three software measurement tools, namely CCCC, CMT and SourceMonitor, were found helpful for this dissertation project and were used for the student and industry code measurements. For the measurements, one of the tools was selected for each software metric of interest. The selection process is described in the following chapter.

6 Code analysis: Measuring

To avoid any misconception at this point, the distinction made in Chapter 3 between the measurement process, the measurement stage and the individual code measurements is reiterated. The measurement process is split up into two parts, the measurement stage and the analysis stage. This chapter is about the measurement stage, and thus deals with the preparations taken for the code measurements as well as with the individual code measurements themselves.

The following chapter will then deal with the second part of the measurement process, the measure analysis. There, the code measurement results are presented and analyzed.

6.1 Introduction

In preparation for the measurement, software metrics of interest were selected. This selection and the metric interpretation for this dissertation project are discussed in Section 6.2. Then in Section 6.3, the measurement stage is described. The first part of Section 6.3 summarises the tool selection process performed and explains the metric tools assignment taken. Additionally, the different code programs measured (sample programs) are briefly introduced. The later sections of 6.3 then describe the preparatory steps required, as well as present a brief summary of the code measurements. Expected differences between students, open source and closed code are discussed and expectations with regard to the results are presented.

6.2 Metrics of interest

Chapter 4 gave a broad overview on what software metrics are available. This section presents the metrics of interest for the code measurements performed and explains the metrics interpretation. Furthermore, aspects about metrics combinations are considered.

6.2.1 Metrics interpretation

Software metrics are in the best case only indicators for a particular code attribute [1] and therefore cannot describe this attribute fully, thus interpretations of the measures produced should not be over emphasized. Table 6.1 lists the metrics of interest and explains how low and high values for the measures produced will be interpreted in this dissertation.

Low values indicate	Software metrics	High values indicate
small project	**LOC, NCLOC**	large project
small project that is based on few statements / operators and operands	**HN, #Stat**	large projects based on many statements / operators and operands
code is distributed over a small number of files. For big projects this should be avoided.	**#Files**	code distributed over a large number of files
the program actions are based on a small number of methods. For big projects this should be avoided	**#Methods**	the program actions are based on a large number of methods.
small underlying functionality	**HVoc**	large underlying token vocabulary
low textual complexity	**HV, HL, HD, HE**	high textual complexity
simple control flow structure of the program	**NBD**	wide control flow structure
simple control flow structure of the program	**BP**	complex control flow throughout the program
method contains a simple control flow	**ECC per method**	method contains a complex control flow
the file is small and contains a simple control flow	**ECC per file**	the file is presumably big and / or complex
rather difficult to maintain	**MI**	rather easy to maintain
code is not highly commented	**CR**	code is commented, but the comment quality is unknown
low polymorphism	**DIT**	complex inheritance tree
low polymorphism	**NOC**	high polymorphism but maybe too complex inheritance tree
low coupling and thus flexible to requirement changes	**CBO**	high coupling, changes might be difficult to perform
low communication within the program	**IFlow**	high communication and thus coupling of programming parts (modules)

Table 6.1: Metrics interpretation

When the measures produced are analysed, metric combinations have to be considered to fully understand a measure produced. For many software metrics, just looking at a single

measure will not tell you much other than the value itself. It is important to understand what caused the measure to be high or low. Recall the person's weight example from Chapter 3, where not only the weight but also the size and age measures are required for the interpretation of the weight measure. This metric combination aspect is discussed further in the next section.

6.2.2 Metrics combination

As seen in Chapter 4, software metrics can be combined pre application to form new metrics (such as the MI which is a combination of LOC, HV and ECC) [56] or to normalise existing metrics (such as dividing HE by NCLOC) [65].

Post metrics application, the metric measures produced can be combined to understand those aspects which influenced the measures in reaching a specific value and thus understand what this value indicates. The possible combinations of post application metric performed in this dissertation are discussed in the list below.

- When considering the ECC measures produced, the size measures for a particular program section indicate whether the ECC has a high value due to control flow complexity or due to length. In addition, the branch percentage (BP), nested block depth (NDB) and Cyclomatic Complexity must be combined to understand the underlying control flow complexity.
- Similar to the ECC metric, the textual code complexity metrics also can be combined with the size metrics by using normalization. Long program sections are likely to result in a higher textual code value, simply because they hold more text. Thus, the Halstead Difficulty, normalized by the size metric NCLOC, can indicate how high the textual code complexity is per line.
- The Maintainability Index (MI) can be combined with the metrics it is derived from to understand why a certain value of a MI measure was produced. For example, a low MI measure can indicate low code maintainability. In order to understand which aspect requires improvement, analysing each of the composite metrics of the MI can help in determining the reason.
- When average measures (such as the average extended CC per file) are used, the code distribution (see Section 3.5) plays an important role for understanding these average measures. An average value alone is not enough to determine whether a value is high or low.

6.3 Measurement stage description

This section takes a closer look at the steps taken before the individual code measurements were performed.

6.3.1 Measurement tools used

Before the metrics of interest were applied, suitable software measurement tools were selected. The tool comparison and selection is described in Chapter 5 and now in a summarized form displayed in Figure 6.1 below.

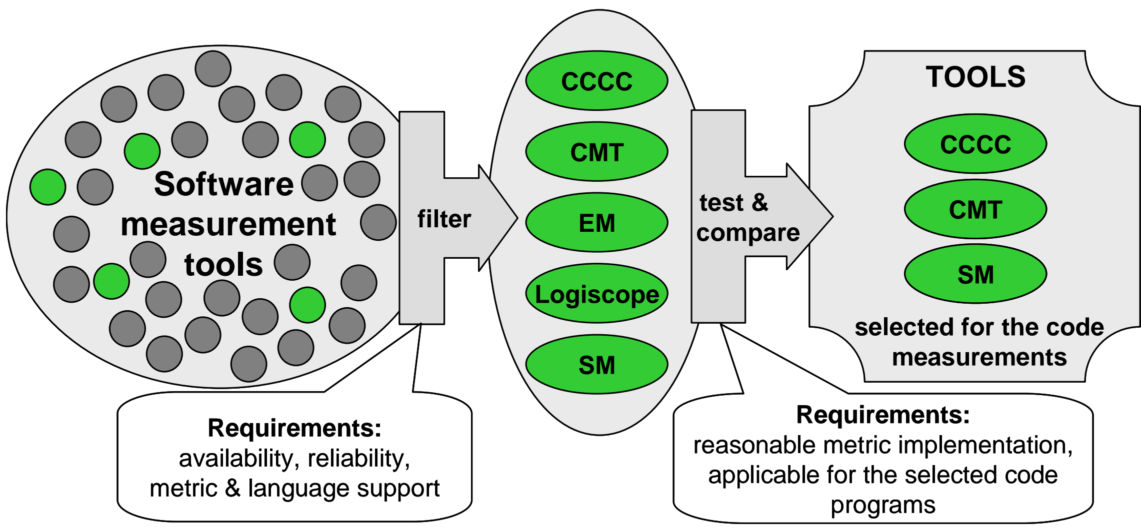


Figure 6.1: Tool selection process

A wide variety of software measurement tools is available. After filtering by and testing for the requirements (see Chapter 5), the tools CCCC, CMT and SourceMonitor were selected for the code measurements. Since, as depicted in Table 5.2, the tools supported a common set of metrics, one of the 3 tools was chosen for each software metric. This choice is presented in the following section.

6.3.2 The choice of tool for each metric

The table below (Table 6.2) shows the selected tool/metric pairing. The specific decisions, for or against a tool metric combination, are presented and discussed.

Metrics		CCCC	CMT	Source Monitor
Program size & Code distribution				
LOC	Lines Of Code	-	+	**+**
NCLOC	Non Comment LOC	+	**+**	-
#STAT	Number of Statements	-	-	**+**
HL	Halstead Length	-	**+**	-
# Files	Number of Files	-	**+***	+
# Methods	Number of Methods	-	**+***	+
Textual code complexity				
Hvoc	Halstead Vocabulary	-	**+**	-
HV	Halstead Volume	-	**+**	-
HD	Halstead Difficulty	-	**+**	-
HE	Halstead Effort	-	**+**	-
Control flow complexity				
NBD	Nested Block Depth	-	+	**+**
BP	Branch Percentage	-	-	**+**
ECC	Extended CC	+	**+**	+
Maintainability				
CR	Comment Rate	+	+*	**+**
MI	Maintainability Index	-	**+**	-
Object Orientation				
DIT	Depth of Inheritance Tree	**+**	-	-
NOC	Number Of Children	**+**	-	-
CBO	Coupling Between Objects	**+**	-	-
FI	Fan-in	**+**	-	-
FO	Fan-out	**+**	-	-
IF4	IFlow	**+**	-	-

Legend:

+	The tool does support the metric
-	The tool does not support the metric
+*	Tool's output used for computation
+ (highlighted)	This tool is selected for the metric

Table 6.2: Tools and metrics selection

For the measure of NCLOC, the CMT tool was chosen over CCCC. Since the majority of metrics were computed per file and method rather than per module, the idea was to leave room for later metric combinations (see Chapter 5). The #Files, #Methods metrics were applied with CMT. As only CMT out of the three tools selected supports the Halstead metrics and the Maintainability Index, the decision was simple. The same was the case for CCCC and the object oriented metrics. The ECC metric was computed by CMT as the implementation

came close to the original definition (see Chapter 5). For the Cyclomatic Complexity, CMT was selected, since the other tools were not as suitable with regard to the metric computation (see Section 5.3.2).

6.3.3 The set of source programs measured

For the code measurement, several software programs were selected. These can be categorized into student, open source and closed source programs.

The student programs were collected from volunteer students at Karlstad University. Their code was made anonymous and thus is referred to as for example: Student1 code. The university courses, for which the code was produced, range from A- (first year) to D-level (final year) courses. One of the student courses (OODM – Object Oriented Design Methods) especially focused on the design and program structure of the student programs. Thus extra attention will be given to measurement result difference between this and the other courses in Chapter 7. The following is a brief introduction to the different courses.

First year course (A-Level)

- **A.OOPJ** (Object Oriented Programming with Java)

  The course gives elementary mechanisms for object oriented programming (such as inheritance, polymorphism, cohesion and data abstraction). Additionally object oriented design and design patterns are introduced.

  <u>Measured:</u> 5 different java lab assignments from the same student

Second year course (B-Level)

- **B.OS** (Operating System)

  This course describes the elementary principles of how an operating system is constructed and functions and how the principles are implemented in today's operating systems.

  <u>Measured:</u> 5 different lab assignments programmed in C by a single student

Third year course (C-Level)

- **C.PL** (Programming languages)

  The course provides an awareness of different kinds of programming languages as well as an understanding of how syntax and semantics are described. The course further aims to give a deeper knowledge of imperative languages and their design. The functional, logic and object-oriented paradigms are introduced.

Measured: 7 programs from different students, implemented for the same lab assignment
The task in this single lab assignment was to implement a program that can parse code using a given grammar (a reduced Pascal grammar). For the lab, the students had the choice of using C or C++. Four out of the seven programs are written in C++, the other three in C.

Fourth year courses (D-level)

- **D.CC** (Compiler Construction)

The goals of the course are to give an introduction to programming languages and compiler implementation. A background in the history and development of computer languages and compiler construction theory is presented. Further course goals are the construction and implementation of a compiler for a simpler imperative language.
Measured: 15 programs from 4 students and the course teacher for 3 lab assignments
The task for the first lab was to implement a recursive descent parser with symbol table support and basic syntax and semantic error handling. Lab2 built on Lab1 and required the students to add the functionality of generating 3 address code (3ac) or stack machine code to their programs. The final lab assignment was to implement an interpreter for the in lab2 produced 3ac or stack machine code.

- **D.OODM** – Object Oriented Design Methods

The goal of this course was to teach object oriented design methods and patterns, as well as their practical application to the students. Several design patterns, such as singleton, factory, decorator, composite, etc. were discussed in great detail.
Measured: 5 java programs from different students for the same lab assignment
The lab assignment was to design and implement a game factory to configure and arrange board games (such as chess and checkers). In this lab assignment special attention had to be given to the design and in the course presented design patterns (such as abstract factory, singleton and factory method)

The open source code was collected via the open source code development and download repository Sourceforge (http://www.sourceforge.net). The selection was made in accordance to the number of previous downloads and the programming language used to implement the project. High download frequency counts were seen as rough indicators for good and typical open source projects with several developers involved. For each programming language six

open source projects were selected. Two of the 18 projects were selected, since a previous software measurement study distinguished them as "good" and "bad" regarding their design and programming background [65]. These are namely FitNesse and BonForum. The open source programs are presented sorted by programming language. The following programs descriptions were gathered through Sourceforge and [65].

C Projects:

- **GIMP** – version 2.2.15

GIMP (GNU Image Manipulation Program) is a raster graphics editor, which is used to process digital graphics and photographs.

- **Panda** – version 0.5.4

Panda is A PDF generation application programming interface.

- **Miranda** – version 0.6.8

Miranda is an instant messenger with support for multiple protocols including AIM, Jabber, ICQ, IRC, MSN, Yahoo.

- **SDCC** – version 2.7.0

SDCC (Small Device C Compiler) is an ANSI - C compiler.

- **Null Webmail** – version 0.9.0

Null Webmail is POP3/SMTP Webmail Common Gateway Interface.

- **VIM** – version 7.1

VIM is a text editor, which originated from 'vi' one of the standard text editors on UNIX systems.

C++ projects:

- **7-Zip** – version 4.5.3

7-Zip is a file archiver tool with support of several compression formats (including 7z, ZIP, RAR, ARJ, and so on…)

- **FileZilla** – version 3.0.1

FileZilla is a file transfer protocol and simple file transfer protocol client for Windows.

- **DiskCleaner** – version 1.5.7

DiskCleaner is a tool to free disk space that is used by temporary files (such as temporary system files and internet cookies and cache).

- **HexView** – version unknown

HexView is a standalone multiple document hexadecimal viewer for windows that can display and print a file as a hex dump.

- **eMule** – version 0.4.7.c
eMule is a filesharing client which is based on the eDonkey2000 network.

- **Notepad++** – version 4.2.2
Notepad++ is a source code editor and intended as windows Notepad re-placement.

Java projects:

- **BonForum** – version 0.5.2
BonForum is a chat application, that was developed for technology demonstration purposes as part of the book “XML, XSLT, Java and JSP” and not much attention was given to design and implementation [65].

- **FitNesse** – version 20050301
FitNesse is a software development colla-boration tool, that is administrated by Robert Cecile Martin, an author of several books in the area of object oriented design patterns and design principles, and CEO of a worldwide acting company specialist in object-oriented programming and software design consulting. [65]

- **HTML Unit** – version 1.13
HTML Unit is a unit testing frame-work intended for html based web site testing.

- **JSS** – version 0.1
Java Internet Spreadsheet is a spreadsheet application intended for online use.

- **JSettlers** – version 1.0.6
Java Settlers is a web-based version of the board game ‘Settlers of Catan’. JSettlers was developed as part of a doctor degree dissertation in the area of artificial intelligence.

- **Nekohtml** – version 0.95
NekoHTML is a simple HTML scanner and tag balancer that enables application programmers to parse HTML documents and access the information using standard XML interfaces.

The industry code (closed source code) was collected at Sogeti Sverige AB in Karlstad. Sogeti, as a wholly owned subsidiary of Cap Gemini, is a consultancy specialized in local professional IT services with locations in 13 different countries and revenue of EUR 1,309 million in 2006. The measured source code belongs to project Production Planing System (PPS). As the name indicates, the system can be used for production planning. The program is written in Java and will be referred to as CS.PPS (closed source production planning system) in this dissertation.

6.3.4 Code measurement preparations

For each of the above mentioned software programs the software metrics of interest (see Section 6.2) had to be applied and the measurement data be properly stored for later comparison. Since three different software measurement tools were used, the analysis could not be done with the tools itself but the sets of measures produced needed to be exported and then united in order to facilitate comparison (Figure 6.2).

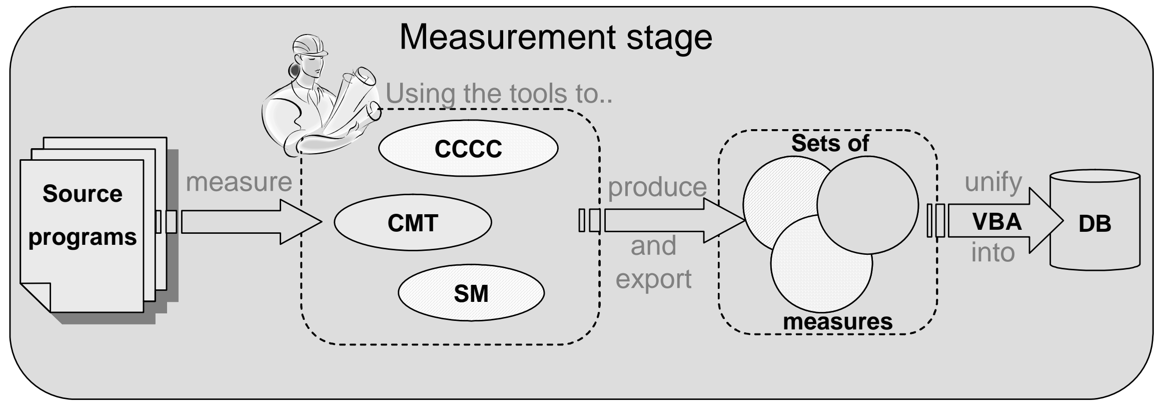


Figure 6.2: Measurement stage

In order to automate the collection of the individual code measurements and the measure export, batch files were implemented that triggered the tools' measurement processes with the specific measurement options. Thus, the code measurements were easily reproducible.

In order to unite the measures exported, Visual Basic Application modules were created that read the data into an MS Access database. Since more than one software measurement tool was used, the export file layout had to be considered and covered by the implemented import modules as well. In addition, as the export file layout also differed depending on the programming language measured, the VBA modules were adjusted further. This last step, however, was not initially expected to be required. The previous setups steps can be figuratively followed by means of Figure 6.2.

The database created contained 5 tables, as depicted in Figure 6.3, which were used to store the measures with regards to their scope (project scope, file scope, class scope and method scope).

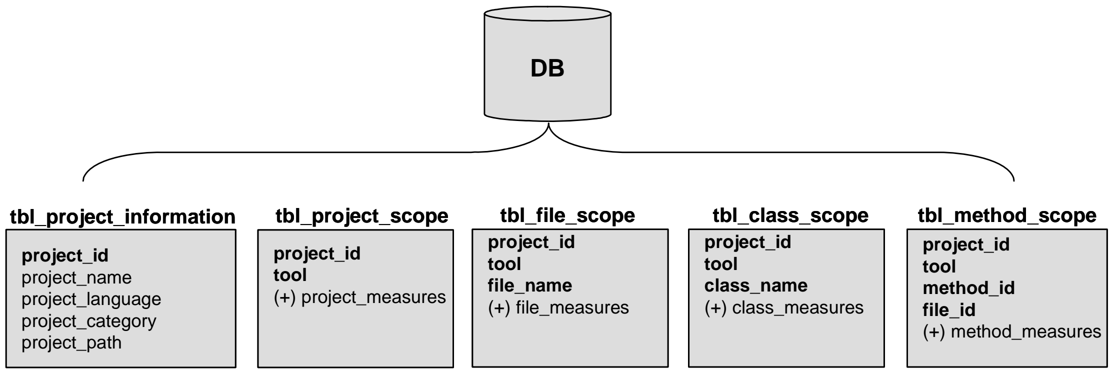


Figure 6.3: Database tables

One table was used for storing project information, such as name, programming language, and whether it is student, open source, or industry code. The other four tables were covering project-, file-, class-, and method scope measures. This table setup was found to be a sufficient basis for allowing tool independent and flexible measure analysis.

6.3.5 The code measurements performed

The code measurements were then, after these extensive measurement preparations, performed rather quickly. The batch files were executed to produce all necessary measures for CCCC and CMT. As SourceMonitor did not include a command line access, the measures here had to be produced for each program individually. After all measurement data was exported, the VBA import process was manually triggered to automatically read the data into the prepared and specified database tables. Thus, except for the code measurement with SourceMonitor, the code measurements and data union was quickly reproducible. For all measurements, reproducibility should be a determining factor [1, 2]. Reproducibility is very important, especially for continuous code measurements (such as in industry for different software realises) and for multiple code measurements (such as in education measuring a large number of different student programs).

6.4 Expected differences

Before taking a look at the empirical differences in Chapter 7, let us consider expected differences between the different areas of student programming, open source programming and industry programming.

6.4.1 Students vs. industry

One immediately noticeable difference is that computer science students deal with smaller projects and problems than professionals do in industry [66]. According to [66], programmers and software engineers should get involved in bigger programming projects during their education to gain necessary experience for industrial scale programming.

Not only the size of the programs produced differs but also the way the two groups of programmers approach problems. Students tend to use a trial and error approach [67]. In order to learn a programming language, this method is initially very effective, however students tend to use this method for too long [67]. This means that students tend to use insufficient time to analyze the underlying problem and design a solution [68], but start programming right away. If no clear plan towards the solution is known beforehand, more time is needed to implement the program, as new problems are encountered. Thus the program is updated in an ad hoc manner to solve the current problem or difficulty. This not only results in low productivity [69] but also most likely results in complex and poor code structure [29] and program design [70].

Without a good design and simple implementation, the code maintenance can become difficult [70], as the solution is likely to be too specialized on a specific problem [69]. According to [69], students are usually not graded on program structure, program design and code readability, but it "is expected that design effort and quality will indirectly benefit students, with better (more accurate) programs and reduced time spent on projects" [69]. However a majority of graduate students still cannot create good program designs [70]. Next to a good design, code readability and maintainability are required factors in industry. Students tend not to focus enough on the readability [71] and maintainability aspects of their code [70, 72] and rather direct their efforts towards creating a correct program [72].

Common poor programming practices among students can be listed as follows [29, 72]:

- Too many loop and conditional statements
- Not enough methods
- Use of global variables rather than parameters to a method
- Use of too many parameters for a single function

- Too large methods
- Improper identifier-naming
- Unused variables
- Perform unnecessary checking with Boolean expression
- Un-initialised variables
- Inappropriate access modifiers

Some of these programming practices can be identified in the code with the software metrics used in this dissertation. More on this aspect, however, will be presented in Section 6.5.

6.4.2 Open source vs. closed source

Open Source Software is computer software whose source code is freely available together with the executable program (e.g. via sourceforge). Furthermore, OSS includes a license allowing anyone to modify and redistribute the software [73]. As stated by Aberdour [74], OSS projects are characterized by highly distributed teams of volunteer programmers who contribute to and maintain the existing code and write documentation for the product [75].

Advocates of OSS argue that through the open available source code, reliability and security are increased as more pairs of eyes are reviewing the code; thus finding errors and bugs [76]. An additional benefit of the potentially unlimited number of people involved is seen in the inevitable requirement of modular, self-contained and self explanatory code [73, 76] which itself would result in high readability, maintainability and flexibility of the program [76].

The disadvantages of OSS development lie in the area of task coordination. [74]. Here, Stamelos [73] and Berry [76] find that OSS projects can suffer from weak project management and development processes which are not well defined. This is reflected by activities such as system testing and documentation, which are often ignored [73], improper requirement definition performed by the programmers themselves, no formal risk assessment process, too much effort dedicated to programming rather than a detailed design [73, 76].

Despite differences in the development processes and quality assurance, is it seen that OSS and CSS are comparable in terms of software quality [74, 76] and also in terms of structural code quality [73].

In this dissertation project, the single industry project will be placed in the same group as the OSS projects for the later code measurement.

6.4.3 Expected measurement results

With regard to the different project backgrounds and the expected differences between student, open source and closed source code, several assumptions about the expected measurement results can be made. In what intervals will the results lie and what are the major differences between the three areas are typical questions.

Since students have to deal with rather small problems compared to the ones in industry [66], it is expected that the program size measures for the latter are clearly higher. It is expected that the student projects will be within the range of 100 lines to 5000 lines of code. Whereas, the OSS and CSS projects are expected to range from 4000 lines and upward.

The potentially poor programming practices present among students, might be reflected by high NLOC per method measures (large methods) and high NBD, BP, ECC measures (not enough methods, and to many loop and conditional statements). With regard to the design of student programs, the CBO measure might indicate too tight coupling.

For the open source projects high Maintainability Index measures are expected, since the code (indicated in Section 6.4.2) is supposedly self-contained and self explanatory. However, all in all the differences between OSS and CSS should not be highly noticeable as the two are comparable (see Section 4.4.2). If major differences are found, then no solid conclusions should be drawn from these, as only a single CSS project was available for this dissertation project. Here the difference between student and industry code is more of interest.

6.5 Summary

When applying different software measurement tools for code measurements, special attention has to be paid to the aggregation of the data produced and the automation of the code measurements. Without any automation in the code measurements the results are costly to produce and reproduce for single and especially multiple programs. Certainly teachers and tutors would prefer running the code measurements for all selected students via a single click, rather than performing the code measurement for each student individually.

As presented in this chapter, the areas of student, open source and closed source show disparities. The following chapter will take a closer look on whether these differences are reflected by the code measurement results.

7 Code analysis: Measure interpretation

7.1 Introduction

At this point, the programs introduced in Chapter 6 are presented in summarized and numbered form. The industry projects, which are sorted by programming language, are numbered from 1 to 19 (the closed source program from Sogeti is listed last in this group). The student programs, which are sorted by course level (A-D) and course name, range from the number 20 to 56. Rather than the full program titles, the program numbers are used in the diagrams in this chapter (Table 7.1 represents a look-up table in this matter).

Industry Programs

ID	Program name	Lan
01	GIMP	C
02	Miranda	C
03	NullWebmail	C
04	Panda	C
05	SDCC	C
06	VIM	C
07	7-Zip	C++
08	DiskCleaner	C++
09	Emule	C++
10	FileZilla	C++
11	HexView	C++
12	Notepad++	C++
13	BonForum	Java
14	FitNesse	Java
15	HTML Unit	Java
16	NekoHTML	Java
17	Jsettlers	Java
18	JSS	Java
19	Sogeti.PPS	Java

Student Programs

ID	Program name	Lan	ID	Program name	Lan
20	A.OOPJ	Java	39	D.CC_L2_st2	C
21	A.OOPJ	Java	40	D.CC_L2_st3	C
22	A.OOPJ	Java	41	D.CC_L2_st4	C
23	A.OOPJ	Java	42	D.CC_L3_teacher	C
24	A.OOPJ	Java	43	D.CC_L3_st1	C
25	B.OS_L1_st1	C	44	D.CC_L3_st2	C
26	B.OS_L2_st1	C	45	D.CC_L3_st3	C
27	B.OS_L3_st1	C	46	D.CC_L3_st4	C
28	B.OS_L4_st1	C	47	D.CC_L4_teacher	C
29	B.OS_L5_st1	C	48	D.CC_L4_st1	C
30	C.PL_L1_st1	C+	49	D.CC_L4_st2	C
31	C.PL_L1_st2	C	50	D.CC_L4_st3	Java
32	C.PL_L1_st3	C	51	D.CC_L4_st4	Java
33	C.PL_L1_st4	C+	52	D.OODM_L4_st1	Java
34	C.PL_L1_st5	C+	53	D.OODM_L4_st2	Java
35	C.PL_L1_st6	C	54	D.OODM_L4_st3	Java
36	C.PL_L1_st7	C++	55	D.OODM_L4_st4	Java
37	D.CC_L2_teac	C	56	D.OODM_L4_st5	Java
38	D.CC_L2_st1	C			

Table 7.1: List of programs measured

7.2 Program size

The program size was measured in terms of lines of code, number of statements and number of tokens (operands and operators).

7.2.1 Measurement results

The lines of code (both LOC and NCLOC), the number of statements (#STAT) and the Halstead length (HL) indicate the size of the programs. The following table (Table 7.2) illustrates the results for these aspects.

	Total values			
Open Source	**LOC**	**NCLOC**	**#STAT**	**HL**
HexView	1856	1121	721	5131
JSS	3029	1198	1762	8521
DiskCleaner	7470	5430	3765	25748
BonForum	*8584	4077	3086	21545
NullWebmail	9912	8925	6961	63583
NekoHTML	11482	6478	4890	34247
Panda	*26049	13848	6706	148989
Miranda	34029	28823	23892	196620
FitNesse	35347	27122	20399	138937
HTML Unit	*42482	14188	11203	70533
Notepad++	48233	35677	27285	195363
JSettlers	56768	30724	20801	154519
FileZilla	112208	81934	58600	450582
7-Zip	117789	95889	64469	469066
SDCC	207690	144892	96970	757764
Emule	221641	169635	124427	947371
VIM	320531	239693	123416	1111573
GIMP	732157	526279	290981	2755645

	Total values			
Closed Source	**LOC**	**NCLOC**	**#STAT**	**HL**
CS.PPS	120778	76338	65109	493258

	Average Values for the course			
Student	**LOC**	**NCLOC**	**#STAT**	**HL**
B.OS	228	172	123	760
D.CC	869	673	585	3438
C.PL	886	679	562	3416
A.OOPJ	**1034	578	382	2513
D.OODM	*1487	594	449	2855

* -> LOC values pushed up by a high CR > 40%
** -> percent of blank lines ~10% and CR ~ 30%
CR = comment rate

Table 7.2: Different program sizes

The measures in Table 7.2 are sorted in ascending order by the LOC column. For the open source programs, a major difference between lowest (1856, HexView) and highest (732157, GIMP) LOC measure was detected. The student code measures are presented grouped by the course for which they were produced. They lie in the same size range and their highest average LOC measure is below the lowest LOC measure found in the industry group.

7.2.2 Definition revisions and correlations

The observed measures for NCLOC and #STAT follow the ascending order of the LOC measures (see the scatter charts in Appendix A.3.2). While this relationship will in general be

expected to be true, it need not necessarily hold in all cases, for example when (say) 90% of the program text is composed of comments. Note that the LOC value is highly influenced by the number of blank lines and the comment rate (see Chapter 4) which itself differs from programmer to programmer and from program to program. The correlation values in Table 7.3 confirm that the trend depends on the range measured. Student programs which have a much lower LOC value show less correlation with the other program size metrics than the larger Industry programs.

	total_NCLOC	total_#STAT	total_HL
Student	0,811	0,741	0,783
Industry	0,997	0,989	0,997
ALL	0,999	0,991	0,998

Table 7.3: Correlations[7] with the total_LOC measures

The other three program size metrics show a high correlation for small and large ranged groups (see Table 7.4). Since a statement is a collection of operands and operators [2], the Halstead length (as the total number of operands and operators in the code, see Chapter 4) correlates to the total number of statements (see Table 7.4 and scatter plot in the Appendix). Both #STAT and HL, however, are dependent on the programming language used by their definition (see Chapter 4). By definition NCLOC is not dependent on the programming language used, as the metric counts non comment lines of code rather than language specific constructs.

	NCLOC vs. #STAT	NCLOC vs. HL	#STAT vs. HL
Student	0,969	0,952	0,958
Industry	0,991	0,997	0,994
ALL	0,992	0,998	0,995

Table 7.4: Correlations[7], NCLOC, #STAT, HN

7.2.3 Defining lines of code size groups

Since for the industry programs the range of the program size measures is rather large, a division into size groups seemed reasonable. For this dissertation the author finds the NCLOC metric the most suitable. NCLOC was chosen over LOC, #STAT and HN since this metric is not affected by the comment rate and its computation is not affected by the programming language used.

[7] Spearman correlation

The industry programs were divided into the following three groups: small, medium and large.

- The small group contains programs with a NLOC measure < 10.0000
- The medium group ranges from 10.000 NCLOC to 100.000 NLCOC
- The large group contains the programs above 100.000 NCLOC

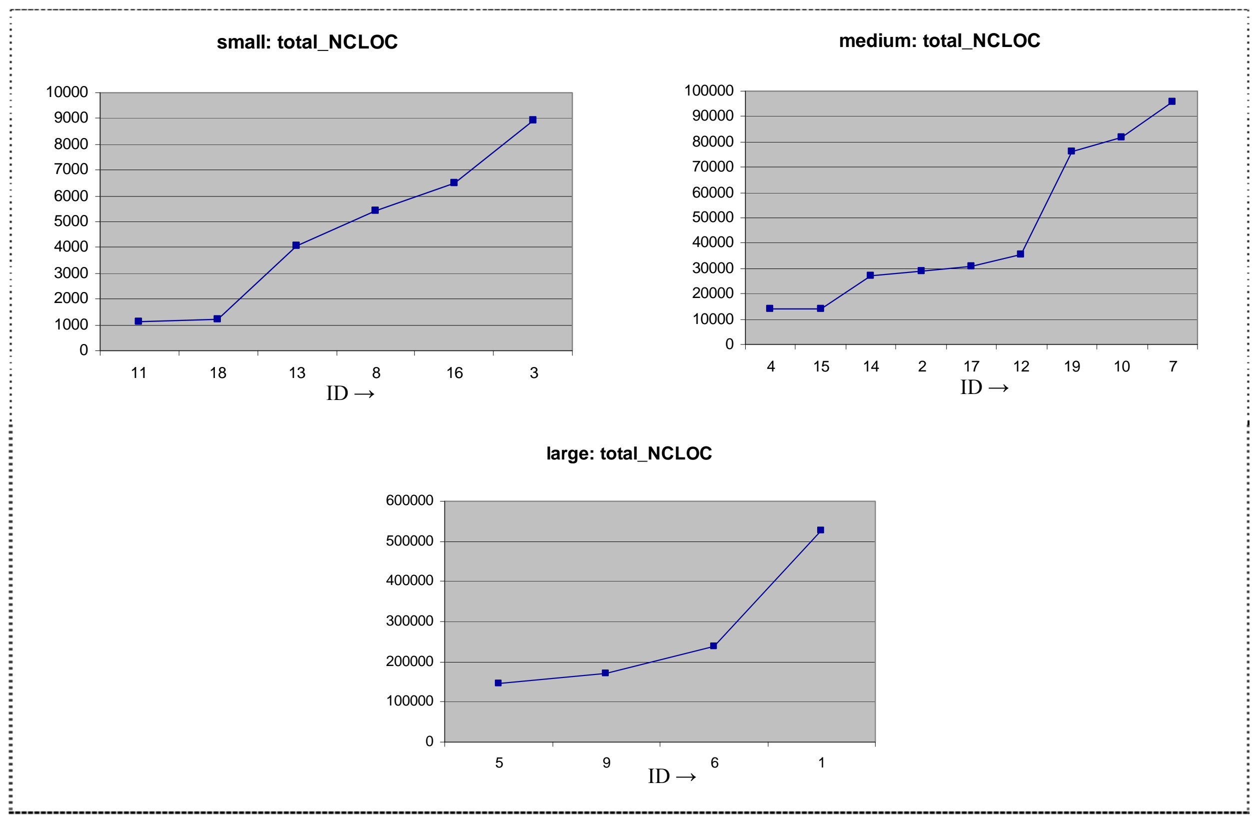


Figure 7.1: Industry NCLOC groups

Since the program size measures for the student programs lie within a much tighter range, the programs are not sub-divided into size groups. The values for these programs can be seen in Figure 7.2 below.

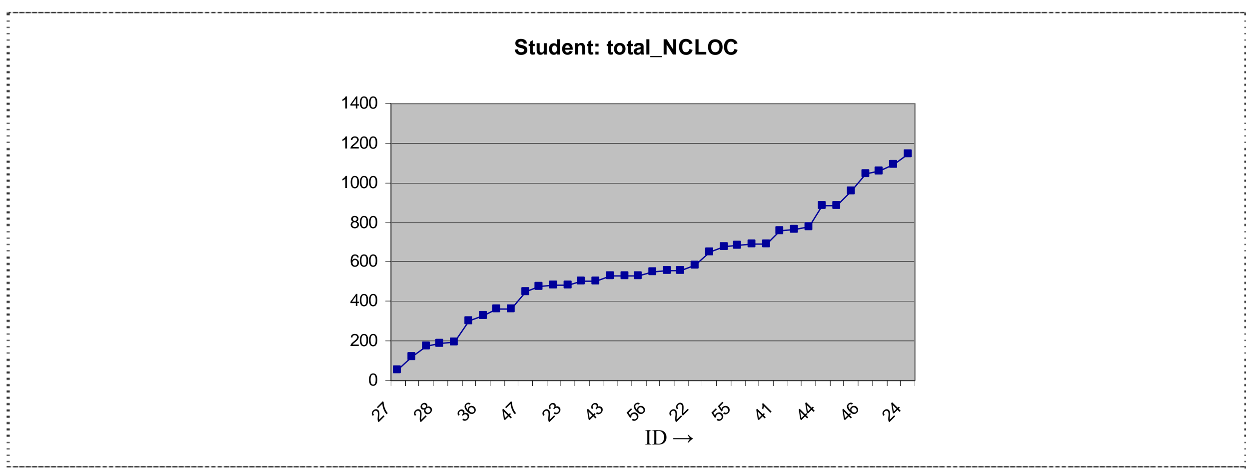


Figure 7.2: Student NCLOC values

7.3 Code distribution

As discussed in Chapter 3 (Section 3.5), program source code can be distributed in several ways among files, classes and methods. In this section, the results for the code distribution metrics are presented and discussed. The code distribution is presented and discussed for a set of sample programs. The complete set of measures is given in Appendix A.3.

7.3.1 Measure ranges and measure outliers

Table 7.5 presents the ranges measured for the code distribution measures in file scope.

	NCLOC measures per file							
	average		max		min		#files	
Group	From	to	from	to	from	to	from	to
Student	17	960	52	960	4	960	1	29
Small	70	307	230	2298	1	33	15	58
Medium	55	294	386	16786	1	10	70	915
Large	255	2282	4723	16512	0	2	104	2036

Table 7.5: Code distribution measure ranges (file scope)

Both the average and minimum NCLOC measures per file seem rather high for the student group, when compared to the ranges of the three industry groups. These high values belong to the same student program (D.CC_L3_st3) which only consists of one file; thus the average, min, max and total measures per file are equal for this particular program. This example indicates the importance of combining measures to understand their meaning. When looking at a single measure, other measures should be used to gain an understanding of what this value indicates (more on this aspect is presented in the next section and Section 7.4.6).

The number of files used increases noticeably for larger programs. However, as indicated in Chapter 3.5, this need not necessarily have to be the case.

The maximum NCLOC measures per file for both the large and the medium length group were unexpectedly high. Figure 7.3 illustrates these aspects further. In the next section (Section 7.3.2) the outliers are investigated in more detail.

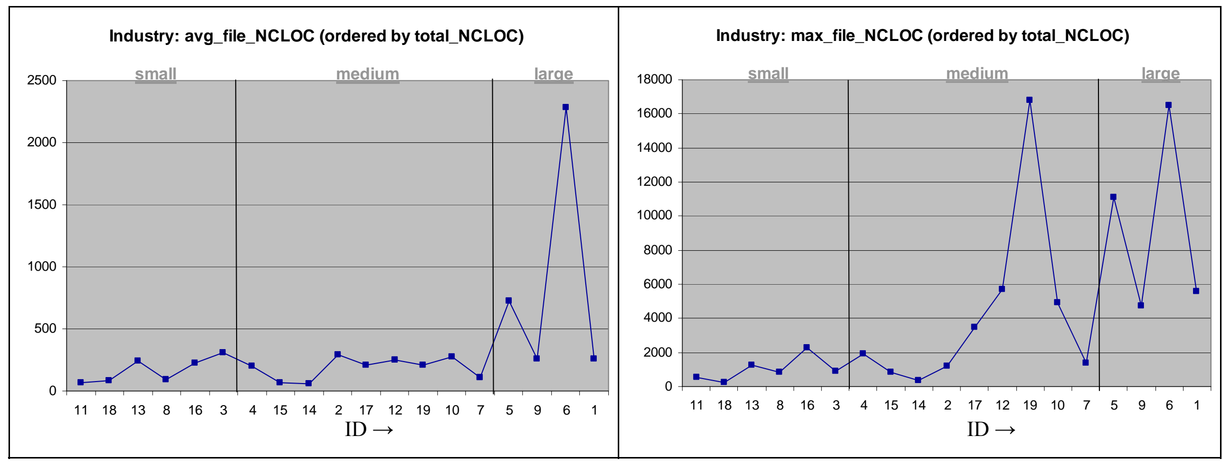


Figure 7.3: Industry NCLOC per file (avg, max)

Figure 7.3 shows that three of the programs measured have source files with more than 10000 non comment lines of code. Understanding the content of such a programming file can possibly be rather difficult (see Chapter 3.5 for the discussion on length per file).

The measures for the method scope show values within a similarly high range. Before these measures are discussed, the full measure ranges are presented in Table 7.6.

	NCLOC measures per method							
	average		max		min		#methods	
Group	From	to	from	to	from	to	from	to
Student	4.1	49.7	23	131	1	10	3	100
Small	13.5	39,8	112	505	1	4	66	414
Medium	6.1	37.0	86	1691	1	2	261	4279
Large	23.4	33.4	922	2326	1	2	1592	14795

Table 7.6: Code distribution measure ranges (method scope)

The average NCLOC measures for the method scope lie within similar ranges for the four groups (see the corresponding charts in the appendix A.3 for more detail). As with the number of files, the number of methods also increases for each size groups (small vs. medium vs. large). The maximum method size measures indicate the use of methods which are presumably too large. Here, 9 out of the 19 industry programs have at least one method with a code length above 700 NCLOC. Three out of the 19 have at least one method with a code length above 1600 NCLOC. As stated in Chapter 3.5, defining a clear optimal length or limit is difficult; however methods with a NCLOC measure above 700 should evidently be refactored. A programmer would need to read 46 to 77 screen pages (if 30 to 50 lines fit on one page) to see the full content of the longest method measured (2326 NLOC long in

program 9). Understanding this content would be a totally different task (see chapter 3 for the discussion on code distribution and optimal method and file size).

The class scope has been left out in this presentation, since the measures produced for the code distribution were very similar to the measures produced on file scope. One class is stored in one file and a file does not contain more than one class except for a few cases.

Since the individual code distributions can not be captured by the measure ranges presented, a closer look at individual example programs will be taken in the following section.

7.3.2 A closer look and measure combination

When comparing different programs or program versions, the code distribution can help to identify problem areas. In this matter, the results for the C level course C.PL showed interesting results in terms of number of files. Two of the student programs differed greatly from the course average of 10 files. Student1 of the C.PL course decided to keep all his code in a single file, whereas Student5 divided his code over 22 files. As stated in Chapter 3, defining an optimal code distribution for a program is difficult. However, when a group of programs designed for the same problem differ greatly, it is advisable to double check with the programmers (in this case the two students) to understand what they did and why.

In order to grasp the code distribution of a program, the measures produced must be combined. Just looking at single measures (like the number of files) is useful to capture the group outliers, but does not indicate much about the individual code distribution.

At this point a set of sample programs will be used to show the differences in code distribution. The three file size outliers from Figure 7.3 and the program FitNesse (ID 14) are used as samples in this matter. Let us now consider the combined code distribution measures in Table 7.8 and Table 7.9 together with the total program size measures in Table 7.7.

	SDCC (5)	VIM (6)	FitNesse (14)	CS.PPS (19)
Total_NCLOC	144,892	239,693	27,122	76,338

Table 7.7: Program size (ID 5,6,14,19)

	NCLOC measures per file					
ID	Mean	median	max	min	stdev	#files
5	728	123	11,077	1	1,681	199
6	2,282	1420	16,512	1	2,714	104
14	55	37	386	4	54	493
19	207	88	16,786	3	925	368

Table 7.8: Code distribution sample group (file scope)

	NCLOC measures per method					
ID	Mean	median	max	min	stdev	#methods
5	33,4	17	1763	1	58,38	3874
6	25,9	13	1070	1	50,87	1592
14	7,5	6	86	1	5,00	3254
19	17,1	4	716	1	42,15	4279

Table 7.9: Code distribution sample group (method scope)

Combined, the measures help build a picture about the actual code distribution. Thus, program 5 and 6 with respect to program size, have rather few files and consequently a high average file length. Especially program 6 consists of mainly long files (average per file of 2,282 NCLOC and mean per file of 1,420 NCLOC). In method scope, both programs seem to have a couple of very long methods, but the majority of methods lie within a fairly narrow range. Program 19 shows a similar code distribution for file and method scope (a few very long file and methods, but the majority short). For the FitNesse (program 14) the code for both file and method scopes lies within a fairly narrow range with low values.

It is very important to understand the underlying code distribution scenario for the interpretation of other measures produced as we will find out later.

7.3.3 The big picture

As described in the previous section, a variety of measures have to be combined in order to understand the code distribution of an underlying program. It would be an advantage to have a single measure describing a program's code distribution. However, the more information is combined in a single measure, the greater is the risk of loss of information about particular aspects [8]. Even with looking at 6 measures per scope, not all aspects of the code distribution are considered. In order to see the distribution to the full extent, either the individual files and lengths of methods or the graphical representation of these lengths should be inspected.

Below in Figure 7.4 this is done for the four sample programs, for which the code distribution measures were presented in detail.

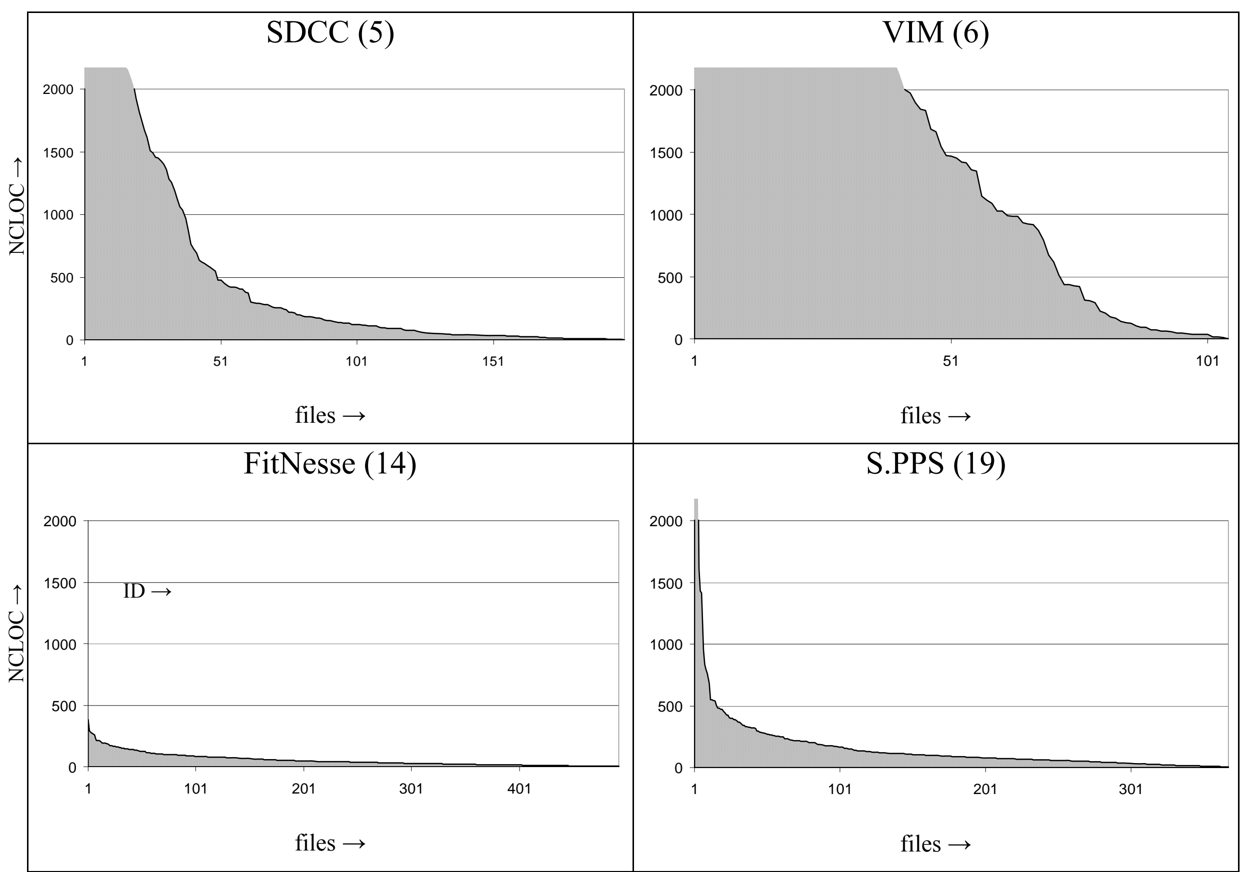


Figure 7.4: Code distribution charts on file scope (5, 6, 14, 19)

The charts combined with the measures produced capture the different code distributions. In the following sections the code distribution of a program will be used for understanding and explaining the value of particular measures. Measures (such as complexity measures) have to be combined with the information about the underlying measure scope, in a similar way to considering a person's physical aspects (such as height, age and gender) before any decision is made, i.e. whether a certain weight measure is too high, low or acceptable.

7.4 Textual code complexity

Selected examples of the textual code complexity measures produced will be displayed in this section. See the corresponding section in the appendix (A.2.3) for a full view on the textual code complexity measures.

7.4.1 Problematic aspects

The total measure for the Halstead vocabulary (total_Hvoc), as the sum over all file Hvoc measures, is not useful for this dissertation since the information about the actual vocabulary size gets lost through summing up. By this is meant that the vocabulary for a program (by Halstead defined as the distinct number of operands and operators) will contain double entries when several presumably non disjunctive vocabularies are joined. This is not due to an error in the metric definition, but to deficiencies in the software measurement tool applied.

For the total Halstead Difficulty (total_HD) a similar aspect came to light. The more files the code was distributed over, the higher the total_HD measures found. However, since the overall number of operands and operators is independent to the number of files used, this should not be the case. Thus, a similar problem as for the total_Hvoc was experienced.

To what extent the total formation of the HV and HE measures biased the corresponding total values is not clear[8]. Here, however, no strong indicators for a measure distortion were found. Nevertheless the total values of these should be treated with care.

The mean values are highly dependent on the code distribution aspects. Programs with a high mean program size showed high textual code complexity measures. Here again, the trend is especially due to the high size range of the underlying program sample.

7.4.2 Results & normalisations

In this section the total values of HV and HE will be presented and discussed. Furthermore, as suggested by [14] and [65], the total values for the textual code complexity measures will be presented in combination with the underlying code size; thus normalized.

[8] As described in chapter 4 the Halstead Difficulty (HD) and Halstead Effort (HE) metrics are based on the Halstead vocabulary aspects. However, the total measures produced for these metrics are not affected by the total_Hvoc. Even though the total measures are again the sum of the file scope measures, the computation for these file scope measures was performed on the file scope level as well, thus unbiased Halstead vocabulary measures were applied.

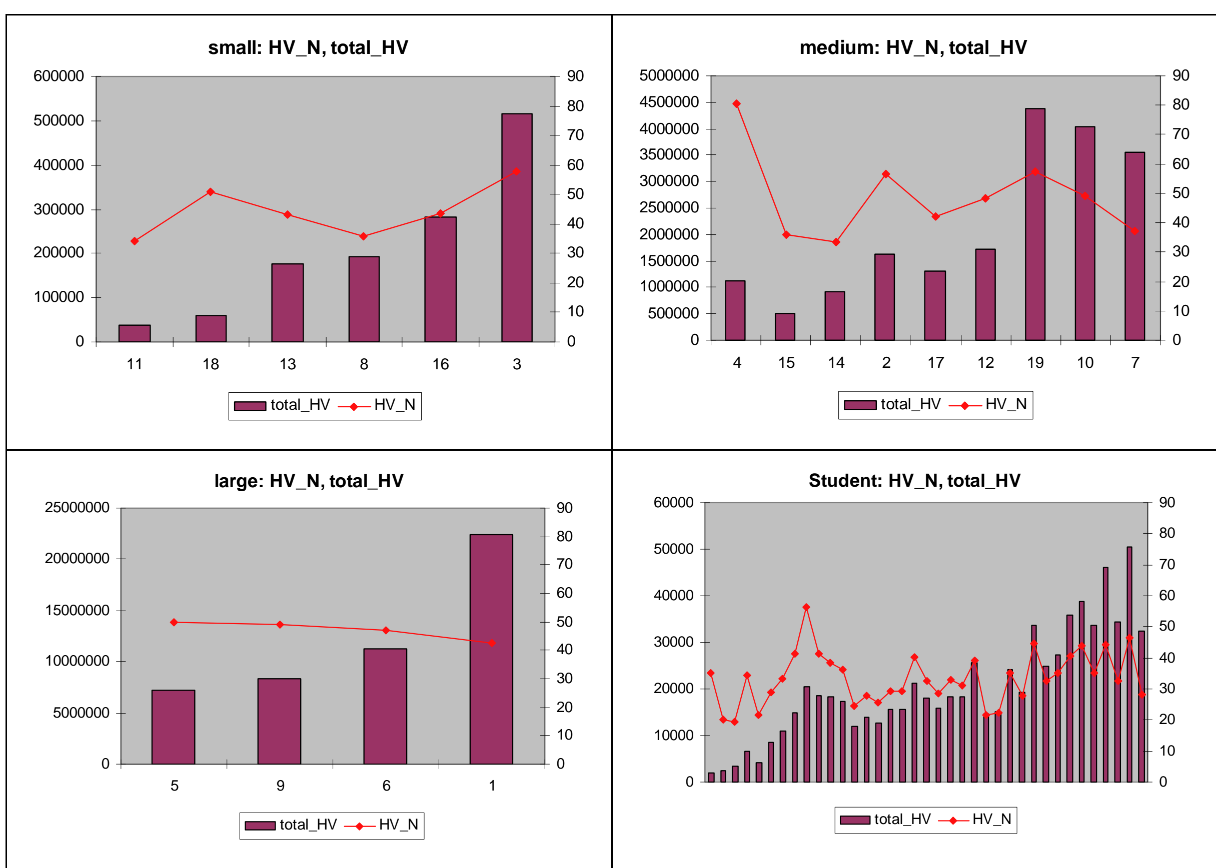


Figure 7.5: Halstead Volume results (industry & student)

Figure 7.5 displays the total_HV measures combined with the normalized values (HV_N[9]) and ordered by the program size (NCLOC value). The different program size groups show clear range disparities for the total values measured. However the normalized values range between 19.2 and 80.6 (or 57.7 if the one outlier is ignored) for all groups displayed. Another observation is that, for the large group just a minimal fluctuation of the normalized measure can be noticed. Since only 4 programs are in the group, this might be due to coincidence. Also possible is that for very large programs, the normalized measure runs towards a constant value. However, this will not be analyzed at this point and thus will be left for future work.

The normalized Halstead Effort measures[9], on the other hand, show a very high fluctuation and range (from 19.9 to 10,103.5) for the group of small programs such as the student programs. The industry programs fluctuate less and less with increasing program size (small: 43.3 - 276.6, medium: 2 - 59.9 [without program 4: 582.3], large: 7.2 – 101.4). The comparison between student and industry programs in this matter is depicted in Figure 7.6.

[9] The Halstead measures were normalized by dividing the underlying NCLOC measure.

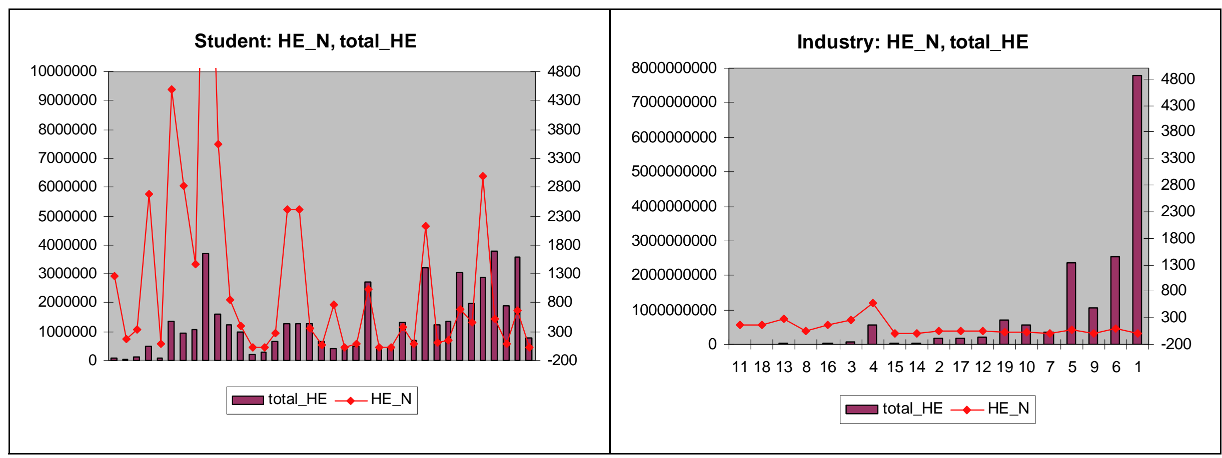


Figure 7.6: Halstead Effort results (industry & student)

7.4.3 Result interpretation

As described above, the normalized Halstead Volume measures lie within a tight range and seems for this specific set of programs to be independent of program size and the number of files. Since the HV metric is intended to measure the information content of a program (see Chapter 4) the normalized Halstead Volume could be interpreted as compactness and density of the information content per lines of code. Whether this is a valid conclusion may be shown by later program comparisons (presented in Section 7.7.2).

7.4.4 Control flow complexity results

At this point a summary of the group results is given. See the corresponding section in the appendix (A.2.4) for a full view on the control flow complexity measures.

	total_ECC		avg_file_ECC		avg_method_ECC		max_method_ECC	
	from	to	from	to	from	to	from	to
Student	7	253	2,0	253,0	1,33	11,5	4	41
Small	118	2421	7,4	83,5	2,27	13,41	19	111
Medium	1580	14212	3,2	63,9	1,33	7,89	25	474
Large	28756	59548	24,6	567,1	4,04	9,00	143	515

Table 7.10: ECC range measures (industry & student)

	BP		avg_NBD		max_NBD		ECC_N[10]	
	from	to	from	to	from	to	from	to
Student	0,03	0,41	0,80	3,51	2	8	0,034	0,264
Small	0,10	0,25	1,32	2,86	6	10	0,105	0,271
Medium	0,05	0,28	0,79	3,14	6	10	0,058	0,217
Large	0,18	0,30	1,57	2,25	10	10	0,097	0,248

Table 7.11: BP, NBD, ECC_N range measures (industry & student)

Table 7.10 and Table 7.11 display the measure ranges for the control flow complexity measures produced. Except for four outliers, the average ECC per file (avg_file_ECC) ranged between 2 and 84 for all program groups (see the corresponding chart in the appendix). These four outlying programs are discussed in the following section. Additionally, the normalized Extended Cyclomatic Complexity measures will be used, as well as the code distribution aspects, to help understand the full meaning behind the ECC measures produced. The normalized ECC metric seems to hold good potential in this respect, since the metric quantifies the cyclomatic complexity per line of code. To what extent the normalized ECC contributes will be indicated by the following comparisons.

The measures for the branch percentage (BP), the average nesting depth (avg_NBD) and the average Extended Cyclomatic Complexity per method (avg_method_ECC) were found to be similar for the different groups (see the appendix A.3.1 for the corresponding charts).

[10] ECC_N stands for the normalized Extended Cyclomatic Complexity for the program. The measure is normalized with the program size metric NCLOC.

However, the programs require to be compared on an individual basis in order to explore the full potential behind the measures.

7.4.5 Individual differences

No large group differences were noted for the structural complexity measures produced, apart from increasing total_ECC measures, which as total measures are related to program size groups. Thus, at this point, the comparison level is moved to the level of individual programs.

The four outliers for the average file ECC measures mentioned in Section 7.4.4, are two student programs (ID 40 and 45) and two industry programs (ID 5 and 6). These student programs were written for the course D.CC by the same student (student 3). The industry programs, which are written in C, both belong to the size group large and namely are, SDCC and VIM. The code distributions for programs SDCC (5) and VIM (6) were already presented in Section 7.3.2; thus their high average file ECC, resulted most likely from their long code length per file. For the two student programs a similar situation was found (the average textual code length per file was high). In order to better understand these high ECC values, Table 7.12 presents measures used for the interpretation.

ID	Avg_file_NCLOC	Avg_file_ECC	ECC_N	BP	Avg_NBD
5	728	161	0,22	0,30	2,19
6	2,282	567	0,25	0,28	2,25
40	557	147	0,26	0,33	1,24
45	960	253	0,26	0,33	1,54

Table 7.12: Control flow complexity results (5, 6, 40, 45)

With this table (Table 7.12) and the ranges measured (Table 7.11), it becomes clear that the high average file ECC measures are not just caused by long code files. Both the branch percentage and the normalized ECC measures have high values. Even though the average nesting block depth measures seem still acceptable for the two student programs, the other two measures (BP and ECC_N) indicate a very complex control flow throughout the program. As the student programs are rather short, a very detailed look at the full source code was possible. The programs indeed show a very complex control flow. The majority of the other students in the course, managed to hold the BP and ECC_N values significantly lower (see Figure 7.7). Program 48, however shows similar results and an even higher branch

percentage. Due to a relatively low average file ECC (=46), this program was not detected before. Thus to find the outliers, all aspects require to be considered and combined.

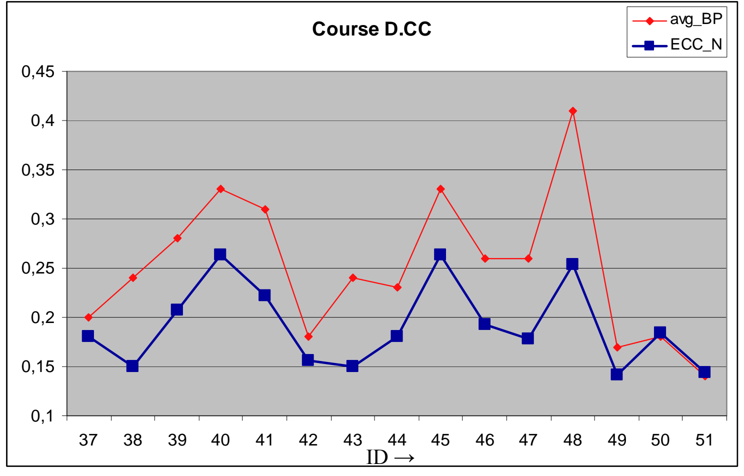


Figure 7.7: BP and ECC_N for D.CC

7.4.6 The importance of measure combination

As the previous section reveals, measures produced have to be combined to understand their full potential. The measures should not only be combined with the code distribution aspects, but also with measures which quantify the same code characteristic (such as the control flow complexity) in a different approach. The measures require to be viewed from different angles to understand their meaning for the particular program [44].

7.5 Maintainability

In this section the results for the Maintainability Index (MI) and the comment rate (CR) are presented and discussed.

7.5.1 Maintainability measure results

The Maintainability Index metric, as pointed out in Chapter 4, measures the maintainability of a program. High measures indicate a good maintainability whereas low measures indicate the opposite [56]. The results for the program selection are presented in the line diagrams below (Figure 7.8).

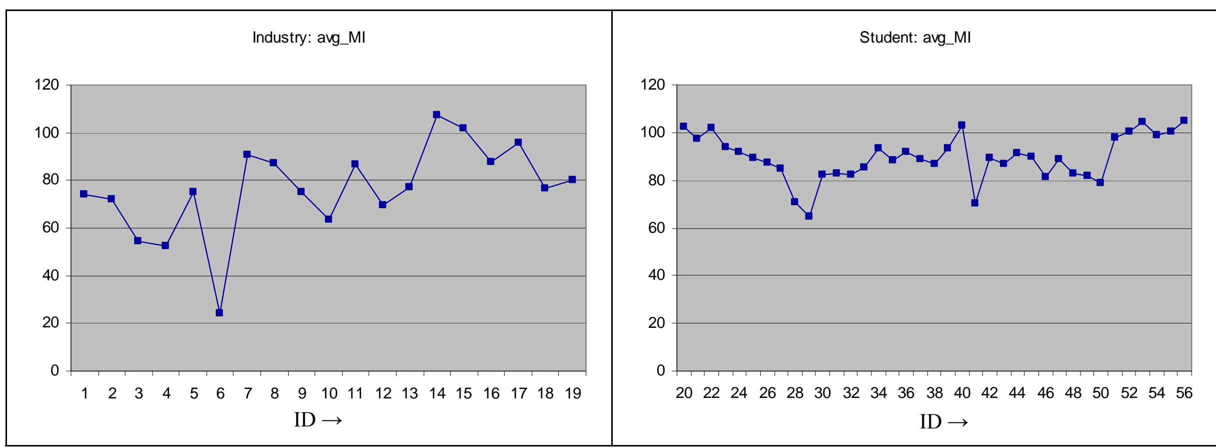


Figure 7.8: Maintainability Index results

Another definition (see Chapter 4) for the Maintainability Index includes the comment rate. Thus the comment rate results are presented (Figure 7.9) but are not used to draw conclusions about the maintainability of particular programs. Although, a well documented code can be easy to maintain [56], the comment rate does not convey the quality of the comments found in a program code.

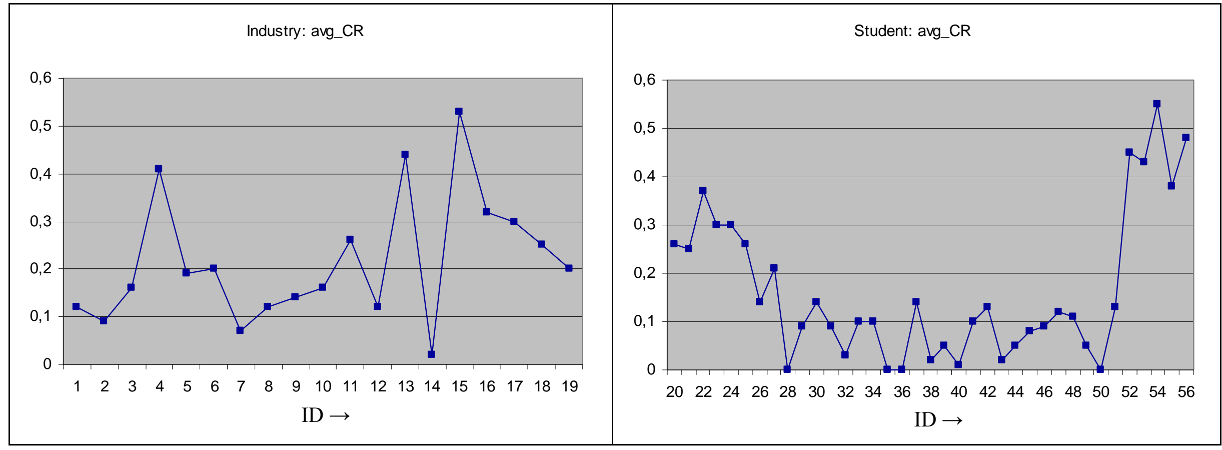


Figure 7.9: Comment rate results

As seen above in Figure 7.8, the MI measures ranged from 65 to 104 for student programs and from 24 to 107 (small: 54 - 87, medium: 52 - 107, large: 24 -75) for the industry programs. For the group of student programs the majority (89%) showed results above an MI measure of 80, whereas in the group of industry programs only 42% went above this value. The next section takes a closer look at the outlying measures seen in Figure 7.8 (program 6 and 14).

7.5.2 Investigating group outliers

For the individual investigation two extreme points within the group of industry programs were chosen. The industry programs are VIM (ID: 6, MI: 24.0) and FitNesse (ID: 14, MI:

107.2). Since the Maintainability Index metric is composed of the Cyclomatic Complexity, the Halstead Volume and the average size (see Chapter 4), these metrics have to be considered when the MI measures are interpreted.

	File scope measures					
ID	avg_MI	avg_NCLOC	Avg_HV	avg_ECC	HV_N	ECC_N
6	24.0	2282	107,662	567	47.2	0.25
14	107.2	55	1,851	3	33.6	0.06

Table 7.13: MI measure comparison (6, 14)

The definition for the Maintainability Index used in this dissertation is as follows (see Chapter 4 for more information about the MI metric):

$$MI = 171 - 5.2 * \ln(avg(HV)) - 0.23 * avg(ECC) - 16.2 * \ln(avg(NCLOC))$$

With this definition in mind, we now consider the measures produced for the sample programs listed in Table 7.13. The two extreme MI measures for programs 6 and 14 can be easily explained. VIM (program 6), as shown in the previous sections about code distribution and control flow complexity, consists of mainly long and complex files, and consequently is rather difficult to maintain (low MI measure). FitNesse (program 14) is this matter the complete opposite to VIM, and shows the highest Maintainability Index measure of all programs measured.

7.5.3 Interpretation

The Maintainability Index measures can be used to quickly determine a program's maintainability [65]. However, it would not be wise to rely only on the results for this metric in order to understand the maintainability of a program (as the previous section illustrates). Here the important issue should again be the combination of measures produced.

7.6 Object oriented aspects

This section presents and discusses the results for the object oriented metrics. Out of the 56 measured programs, 29 were written in object oriented programming languages (10 in C++, 19 in Java). As described in Chapter 5, the tool selected for the object oriented aspect, had some difficulties measuring all lines of code for a program. Thus, not too much emphasis will be placed on the measures produced.

7.6.1 Information flow and object coupling

For the information flow and object coupling the following result ranges were measured:

		Total		Mean		Max	
	Group	From	to	from	to	from	to
IFlow	Student	0	6061	0	275	0	3136
	Small	0	11699	0	111	0	9216
	Medium	37098	6775496	116	21441	9216	6285049
	Large	783824	783824	1139	1139	492804	492804
CBO	Student	6	192	1	5	3	18
	Small	42	500	1	4	6	30
	Medium	952	4120	5	7	43	312
	Large	4734	4734	6	6	122	122

Table 7.14: Information flow and object coupling measure ranges

How much information should flow, is unknown and highly related to the program task and problem size. The information flow measures produced can, however, help to filter out students who might have to reconsider their program design (see Figure 7.10 left and right, and for example program 56 which exhibits high values for those measures) and compare different programs for the same task with each other. In order to compare two independent programs, the information flow value requires to be normalized (similarly to the ECC_N and HV_N), but the question here is which denominator should be chosen.

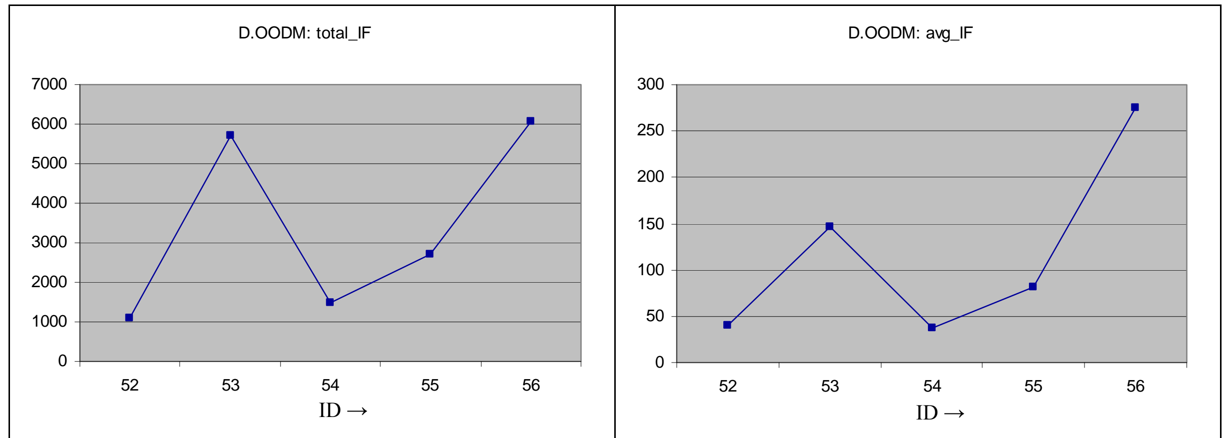


Figure 7.10: Information flow D.OODM

Similar to the IF measures, the measures produced with the CBO metric, should be compared within a group of programs written for the same problem. Otherwise, understanding which measures are too high becomes very difficult.

7.6.2 Inheritance and polymorphism

The DIT (Depth of the Inheritance Tree) metric was found to present no valuable information for this particular set of programs (student and industry programs). The corresponding line diagram charts are to be found in the appendix A.3.1. Also, the Uppsala study [65] did not find the metric that useful for their set of programs. The depth of the inheritance tree ranged from 0 to 7 for the programs measured. Thus, no clear indicators for a too long and complex inheritance tree were found. Often programs have only an indirectly deep inheritance tree, due to the reuse of existing libraries (such as Java libraries). However these libraries are not and also should not be considered in the DIT metric [65].

The NOC metric, similar to the DIT metric, did not produce useful measures for the set of programs measured in this dissertation. The industry programs were found to have a high number of children, but this is very likely for extremely large programs.

7.6.3 Is high good or bad?

One problem for most object oriented measures produced in this dissertation was that without analyzing the programs in depth (meaning understanding and mentally reconstructing the underlying object oriented design), the measures do not provide much information. Whether a measure is high or low can at this point only be identified by comparing this particular measure to measures of programs which were created for the same task (such as student programs for the same assignment).

7.7 Individual comparisons

In this section, the comparison is made from a group view down to individual programs. In order to conclude the group view comparison, the group results from the previous sections in Chapter 7 are presented in summarized form.

7.7.1 Group differences summarized

The results from the Student vs. Industry comparison revealed major differences in the program size and code distribution. The average non comment lines of code per program was 583 NCLOC for the student and 79,593 NCLOC for the industry programs. With regard to the number of files and methods the code was distributed over, these numbers differed greatly as well. Besides these, also the maximum values of lines per file and per method showed large differences between the two groups.

Apart from these major differences, a few minor differences were found in three of the other code characteristic areas. Here the textual code complexity, the control flow complexity (in terms of nesting block depth) and the maintainability results differed slightly for the industry and student programs measured. Since in both groups good and not so good candidate programs were found, the comparison will be made at the individual level.

Considering the programming languages used, the differences can be summarized as follows:

With regard to the code distribution, the java programs showed the same tendencies in code as the C++ programs. The results for the programs written in C indicate longer files and methods than the object oriented languages. Nevertheless, just because the language is object oriented does not mean the programmers follow this paradigm [77]. Thus for Java and C++, similar long cases were found. In general, the Java programs measured showed especially good results for the Maintainability Index (which however is influenced by the code distribution aspect). Apart from these characteristics, no major differences were spotted.

7.7.2 Individual comparison: BonForum vs. FitNesse

In this section the measures produced are combined in full detail for the two industry programs BonForum and FitNesse. Even though they differ in program size (BonForum: 8584 NCLOC, FitNesse: 35347 NCLOC), the two programs may be compared.

	NCLOC measures per file					
Program	max	min	mean	median	stdev	#files
BonForum	1287	9	239,82	100	367,00	17
FitNesse	386	4	55,01	37	53,96	493

Table 7.15: File scope code distribution (BonForum vs. FitNesse)

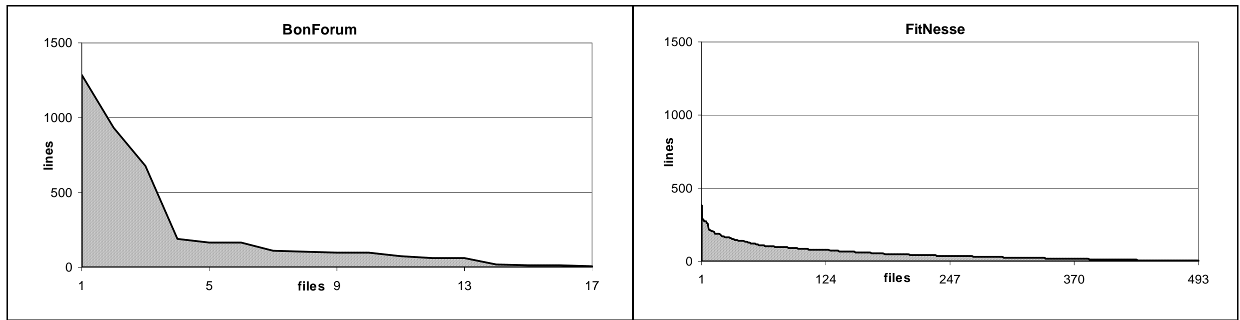


Figure 7.11: File scope code distribution (BonForum vs. FitNesse)

It is important to notice for the following charts that, BonForum contains 3 files above than 500 NCLOC, whereas the remaining 14 files are below 250 NCLOC (see Table 7.15 and Figure 7.11). For FitNesse only a small number of files (<10) are slightly above 250 non comment lines of code.

	NCLOC measures per method					
Program	max	min	mean	median	stdev	#methods
BonForum	505	3	21,8	9	50,34	180
FitNesse	86	1	7,5	6	5,00	3254

Table 7.16: Method scope code distribution (BonForum vs. FitNesse)

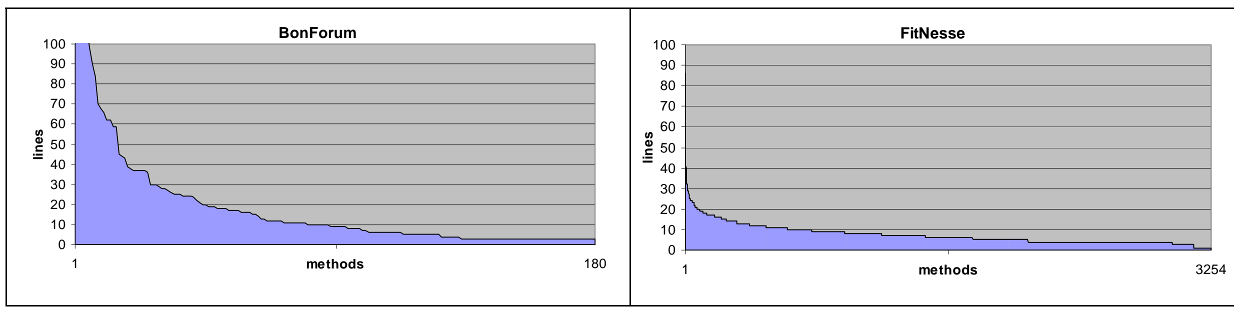


Figure 7.12: Method scope code distribution (BonForum vs. FitNesse)

As Table 7.16 and Figure 7.12 indicate, BonForum has on average methods which are almost 3 times longer than FitNesse. About one quarter of BonForum's methods are longer than 20 non comment lines of code. The code distribution on file and method scope requires to be considered for the following measure comparisons.

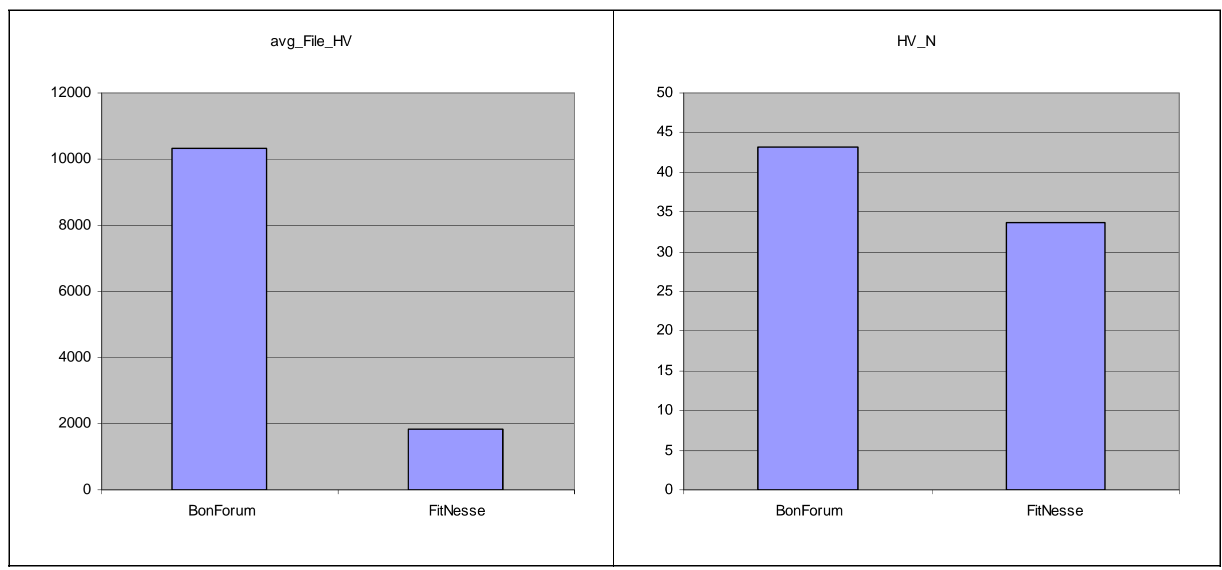


Figure 7.13: Textual code complexity (BonForum vs. FitNesse)

The average Halstead Volume per file is about 5 times larger for BonForum (see Figure 7.13 left), but this result has to be seen in relation to the number of lines per file, as the code distribution differs greatly. Also, since the normalized Halstead Volume is higher (see the right chart in see Figure 7.13), the results suggest that FitNesse has a lower textual code complexity.

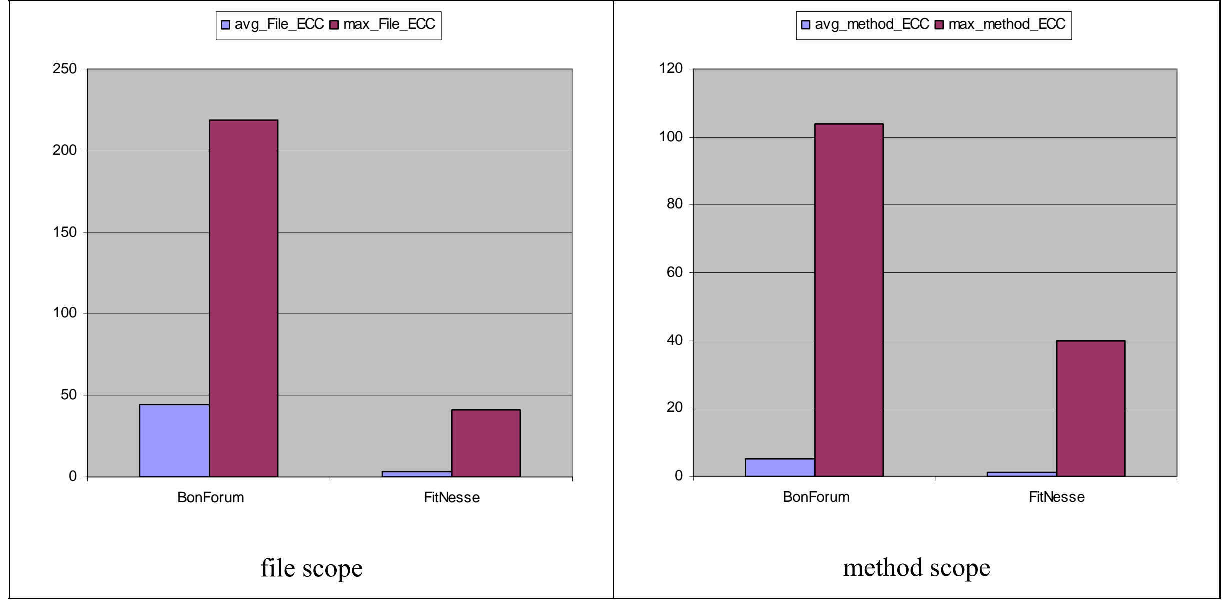


Figure 7.14: ECC file & method scope (BonForum vs. FitNesse)

The control flow complexity measures (ECC measures, BP and NBD presented in Figure 7.14, Figure 7.15 and Figure 7.16) require to be combined to obtain an overview of the underlying control flow. The average file and method complexity was found to be high for BonForum, but since the code distribution affected the measures, the normalized ECC measure had to be taken into consideration as well.

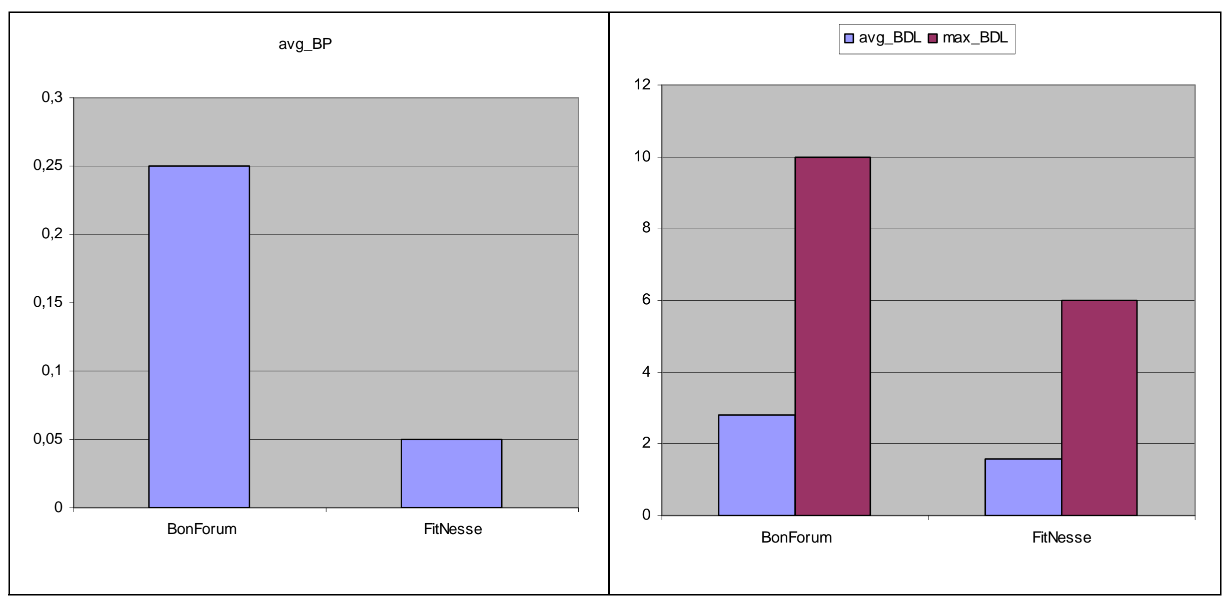


Figure 7.15: Branch percentage & nesting level (BonForum vs. FitNesse)

The Branch Percentage (BP) and the nesting block depth confirm the difference in control flow complexity for the two programs. Thus, together the charts indicate a much more complex control flow for BonForum than for FitNesse. Besides the control flow complexity, also the Maintainability Index is noticeably lower for BonForum (see the right chart in Figure 7.16). FitNesse reached the best Maintainability Index results among all programs measured.

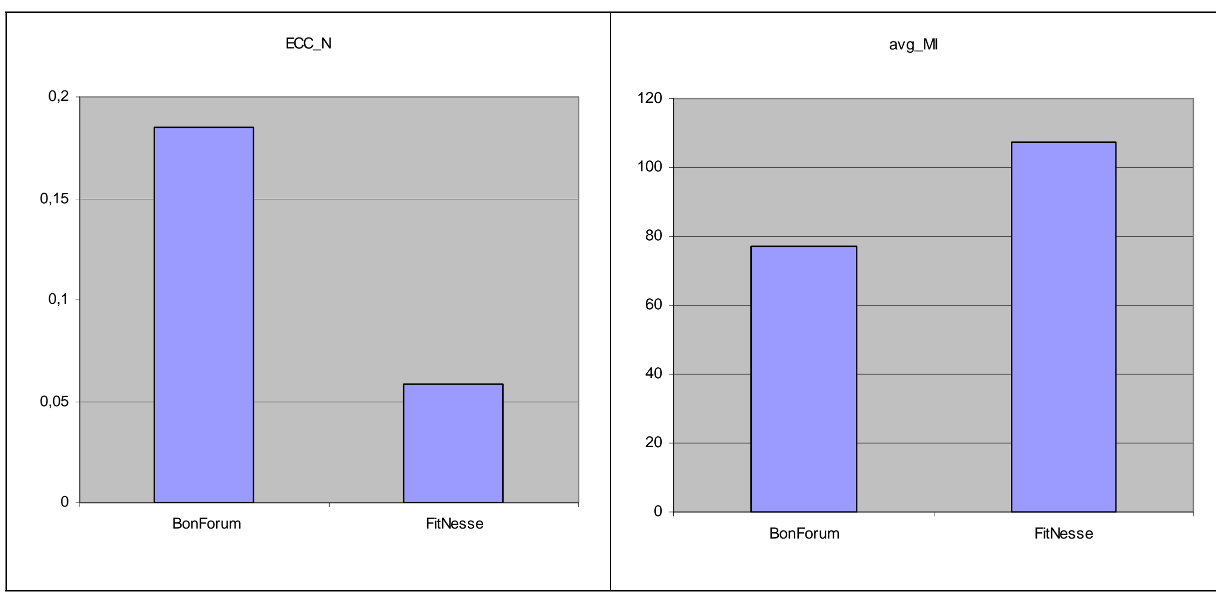


Figure 7.16: ECC_N and Maintainability Index (BonForum vs. FitNesse)

A previous study [65] of BonForum and FitNesse, indicated a poor object oriented design quality for BonForum and good object oriented design quality for FitNesse. The measures produced show that this difference in quality is not just to be found for the design but also for the source code. Here, BonForum was behind FitNesse in all aspects measured.

7.7.3 Creating an individual code profile

As seen in the previous section (7.7.2), comparing measures for two or more projects can be an extensive task. A better solution might be to generate a code profile for each program, which allows the programs to be compared quickly and easily. The next chapter deals with this aspect of generating a code profile for the programs. The programs BonForum and FitNesse can help understand a good choice of profile metrics, since the two programs have now been discussed in detail.

7.8 Summary

When software projects are analyzed by means of software measurements, a considerable amount of data is produced. The task here is to organize the data and draw meaningful information from the measures produced [8]. At this point, the importance of measure combination should be reviewed. Measures produced need to be combined, as illustrated in this chapter, to avoid drawing hasty decisions about a particular code characteristic measured.

8 Code profiling

This chapter deals with the generation of code profiles in general and for the programs measured.

8.1 Introduction

As suggested in Chapter 7, the comparison should be brought to the individual level due to large amounts of data produced. In Section 7.7.2, this individual comparison was illustrated by the example BonForum vs. FitNesse. Even if only two programs are compared, the comparison is not simple and the number of measures, tables and charts can still feel overwhelming. The creation of code profiles can improve the comparison process, as a summary of data is compared rather than the full number of measures produced (in this dissertation, for each program 66 measures were produced).

Before the topic of code profiling is discussed in more depth, the definition of profile listed in Chapter 2 is presented again:

A **profile** is *"a formal summary or analysis of data, often in the form of a graph or table, representing distinctive features or characteristics"* [7].

Thus code profiling is the activity of creating a formal summary of distinctive characteristics of a program's source code. A literature survey revealed that mostly data concerning the (dynamic) performance of software is profiled such as in [53] and [78], rather than the static characteristics of the code. Nevertheless in this dissertation the static code characteristics (such as the control flow complexity) are profiled.

8.2 Reasoning and usage

A program code profile can help to present a quick overview about code characteristics. Instead of reviewing a large number of measures, a reduced set of summarizing measures can be sufficient to determine whether changes have to be made. If the code profile indicates problematic code characteristics, these particular characteristics can be inspected further by consulting the remaining measures, which were not included in the profile, or by going directly to the source code level (see Figure 8.1).

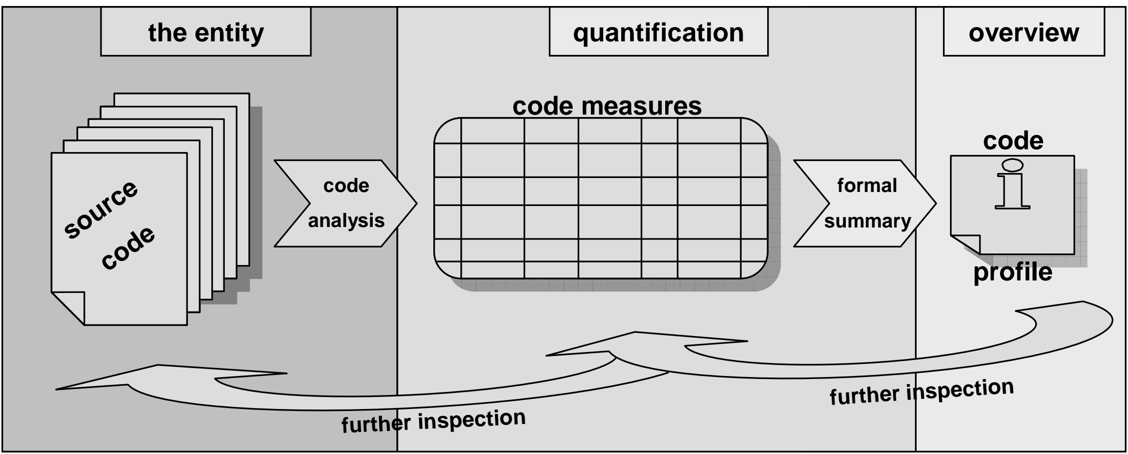


Figure 8.1: Code profiling illustrated

Several scenarios of application seem reasonable for both industry and education. Four of these scenarios are presented in the following sections.

8.2.1 For industry

Scenario 1:

The use of code profiles allows a quick comparison of different releases for the same program. A code profile for a particular program could be generated each time a new release is made. Over time, when the current release profile indicates a large change compared to the previous profiles the areas of change can be inspected and through refactoring adjusted. Thus the program profile history will be stored, and used in order to have a deeper basis for answering questions similar to the following ones:

- Did the project code become more complex, when the last change was made?
- Is a trend visible? Does the maintainability become better or worse?
- If worse, then why and how can it be improved?

Scenario 2

Before acquiring a new software product, the source code can quickly be checked in terms of maintainability. As the code of the software is possibly unknown beforehand, a profile can give a quick overview about the code characteristics. If the profile indicates a very complex source code, an alternative product with better maintainability aspects might be considered instead.

8.2.2 For education

Scenario 1:

Profiles for different student programs in the same course can be compared. When one profile is significantly different to the ones of the other student programs, the code of this particular program should be inspected further. Inspecting all student programs in detail is rarely performed and the more students who are in course, the more difficult it becomes to conduct. With code profiles, students who might need help can be detected rather quickly, without having the teacher look through all student programs in detail.

Scenario 2:

Every time a student hands in a programming assignment a code profile for the particular program can be generated. These profiles can then according to date and course level be stored in a profile portfolio for the student. This way the student's programming improvement can be tracked from the first year assignments to the final year assignments. Whether the profile indicate better programs over the years would be interesting to see.

8.3 The code profile

8.3.1 Defining areas of importance

In this dissertation the following areas of code characteristics will be considered for the code profiling:

- Code distribution
- Textual code complexity
- Control flow complexity
- Maintainability aspects

Since the results for the object oriented metrics applied in this dissertation were not found very useful for the underlying program, these metrics will be covered to a lesser extent in the code profiles. The maintainability aspects can be seen as combination of code distribution textual code complexity and control graph complexity (the MI metric covers these three areas).

8.3.2 Selecting metrics of importance

In this section, two sets of metrics (comparative and descriptive) will be presented and discussed. A distinction was made to guarantee a low complexity of the profile and thus allowing a quick comparison of two individual profiles. The measures for the comparative metrics (comparative measures) will receive, as means of comparison, the main attention in the profile. When anomalies in those measures are detected, the measures for the descriptive metrics (descriptive measures) can be consulted to further interpret the program's code characteristics.

Since the student and industry programs differ greatly in their textual length, the comparative metrics where chosen to be program length independent. In detail, 8 comparative metrics were selected[11]. The selected comparative metrics cover the defined areas of importance and are listed and explained below:

- **Code distribution**
  - Since it is difficult to say which code distribution is to be preferred for a single program (see Chapter 3), the metrics are more to be seen as supportive metrics and thus are mostly within the set of descriptive metrics. Nevertheless, one adjusted code distribution metric was included: the lines of code per method. The average lines per method were not measured, but instead the percentage of large methods determined by the number of non comment lines of code used. This adjusted metric was labelled:

    - **LPM>X** (Lines per Method greater than a defined limit)
- **Textual code complexity**
  - As representative for the textual code complexity the normalized Halstead Volume (see Section 7.4) was chosen:

    - **HV_N** (normalized Halstead Volume | code distribution independent)
- **Control flow complexity**
  - The control flow complexity was measured through the branch percentage, the nesting block level and the Extended Cyclomatic Complexity (ECC). As Chapter 7 shows these three aspects require to be combined to grasp the complexity of the control flow. Here 5 representative metrics were selected (from which one was adjusted similar to LPM>X):

[11] Any number of comparative metrics can be chosen. The number 8 was found to be a good balance between high granularity and lucidity.

- **avg_BP** (average Branch Percentage)
- **avg_NBD** (average Nesting Block Depth)
- **ECC_N** (normalized ECC | code distribution independent)
- **avg_file_ECC** (average ECC per file | file scope code distribution dependent)
- **ECC>X** (percentage of methods with an ECC above X | method scope code distribution dependent)

- **Maintainability**
    - The maintainability aspects are covered in the set of primary metrics by the combination of the metrics above and the Maintainability Index (MI).

      - **MI** (Maintainability Index | file scope code distribution dependent)

The set of descriptive metrics cover code the textual length of a program and more aspects of the above stated four code characteristic areas (such as code distribution and control flow). Additionally, object oriented aspects are integrated in form of the coupling between object metric. The full list of descriptive metrics can be found in the corresponding section of the appendix (see A.4.1).

8.3.3 Defining measure intervals

The two newly defined metrics (LPM>X, and ECC>X) are required to be adjusted further, as the value of X was not set. For LPM, this value was set to 20 and for ECC to 10. Consequently, methods fulfilling the following conditions were counted:

- Lines per method > 20
- ECC per method > 10

As discussed in Chapter 3, no clear understanding about how many lines a method should have exists. In this dissertation, methods with a length above 20 NCLOC are not essentially considered to be fault prone and do not require to be redesigned without consulting other measures beforehand. However, knowing the percentage of long methods in a program can improve understanding about the code distribution and thus supports the interpretation of other metrics (such as ECC>X).

Long methods are not essentially complex [51]. Thus, the complexity measures for a method have to be considered together with the method length. For the complexity per method (i.e. ECC per method) the counting limit of 10 was chosen in conformance with the selected LPM limit.

8.3.4 Profile visualisation

The visualisation of measurement data has been well researched, thus several different visualization methods exist [79], [80]. Figure 8.2 displays two out of many of those methods within the area of software measurement.

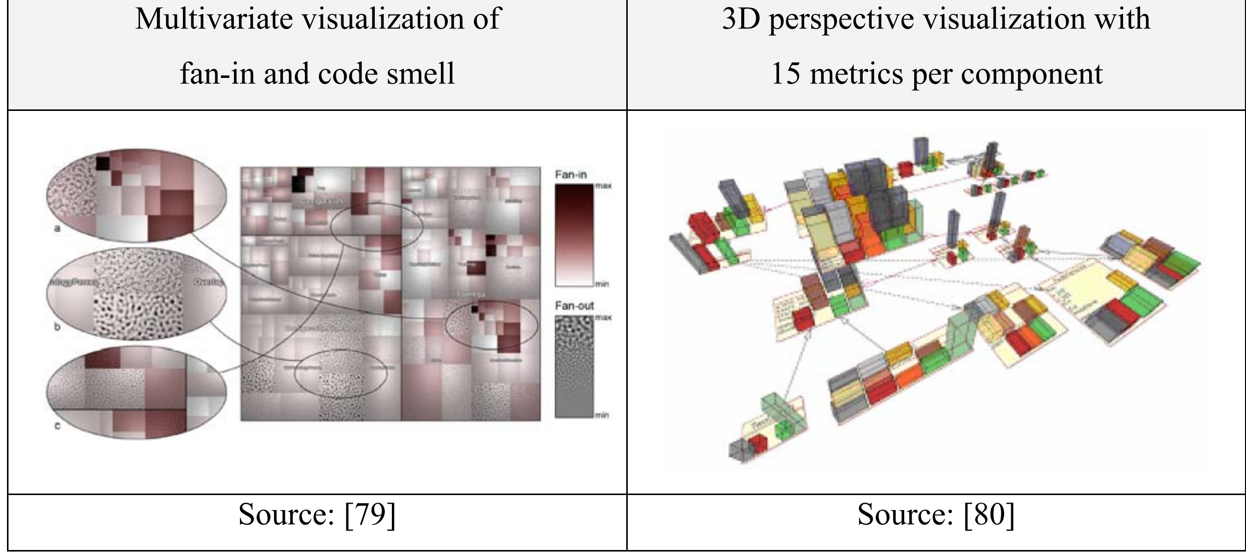
Multivariate visualization of fan-in and code smell	3D perspective visualization with 15 metrics per component
	
Source: [79]	Source: [80]

Figure 8.2: Examples for data visualization

In this dissertation, a rather less complex visualisation form is chosen. When comparing the profiles of different students in class, a very detailed profile can reduce the clarity and usability. Thus, here the ease of comparison and simplicity is chosen over the granularity level of the representation.

The measures for the comparative metrics will be visualized by using a kiviat diagram (also called spider or radar diagram), which is presented in Figure 8.3. The file scope code distribution will be presented in visualized form as well (see 7.3.3). The measures for other descriptive metrics will be listed in textual form next to the kiviat diagram.

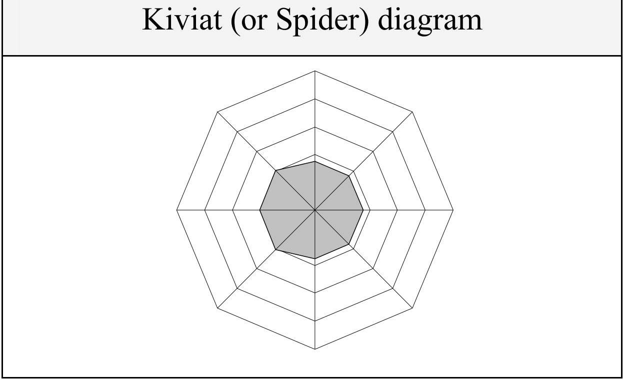
Kiviat (or Spider) diagram


Figure 8.3: Example kiviat diagram

6 Therefore, the layout for the code profile in this dissertation is a combination of literal and figurative elements. A brief explanation of the profile layout follows in the next section.

8.3.5 Profile revision

Figure 8.4 shows the layout selected and explains the different sections briefly

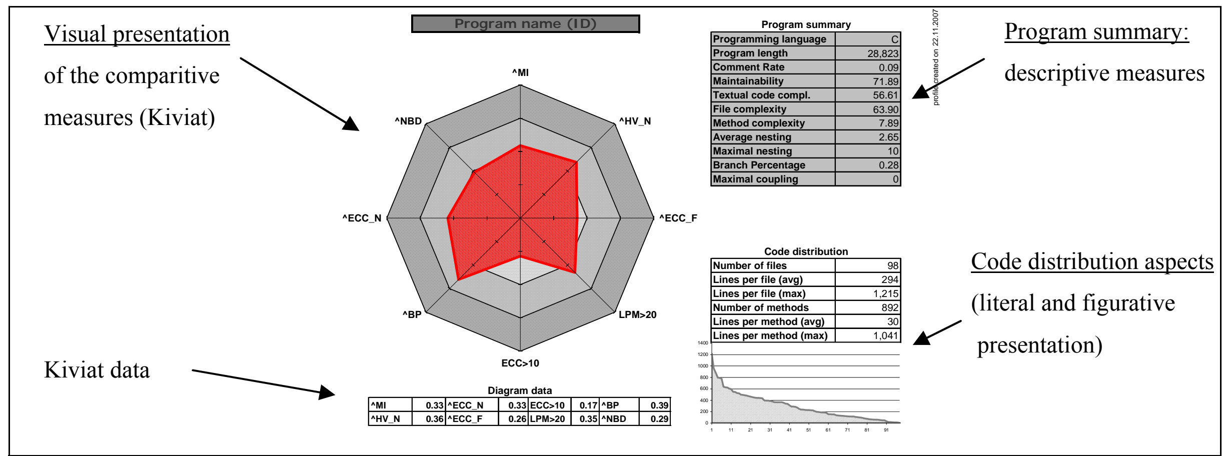


Figure 8.4: Profile layout explanation

Six of the comparative metrics were normalized and renamed (these are indicated with a '^' symbol), for the purpose of display (the new computation can be found in Appendix A.4.1). In order to keep the original values, the measures for these adjusted metrics were included in the program summary section (see Figure 8.4). In this way the adjusted Maintainability Index[12] (^MI) is used for the kiviat diagram and the kiviat data section, whereas the original MI measure is included in the project summary. Table 8.1 gives a brief overview of the selected primary metrics (adjusted and not adjusted).

Name	description	name	description
^MI	Maintainability Index	^BP	Branch percentage
^HV_N	Normalized Halstead Volume	^ECC_N	Normalized Extended CC
^NBD	Nesting Block Depth per file	LPM>20	Lines (NCLOC) per method > 20
^ECC_F	Cyclomatic complexity per file	ECC>10	Extended CC per method > 10

Table 8.1: List of comparative metrics

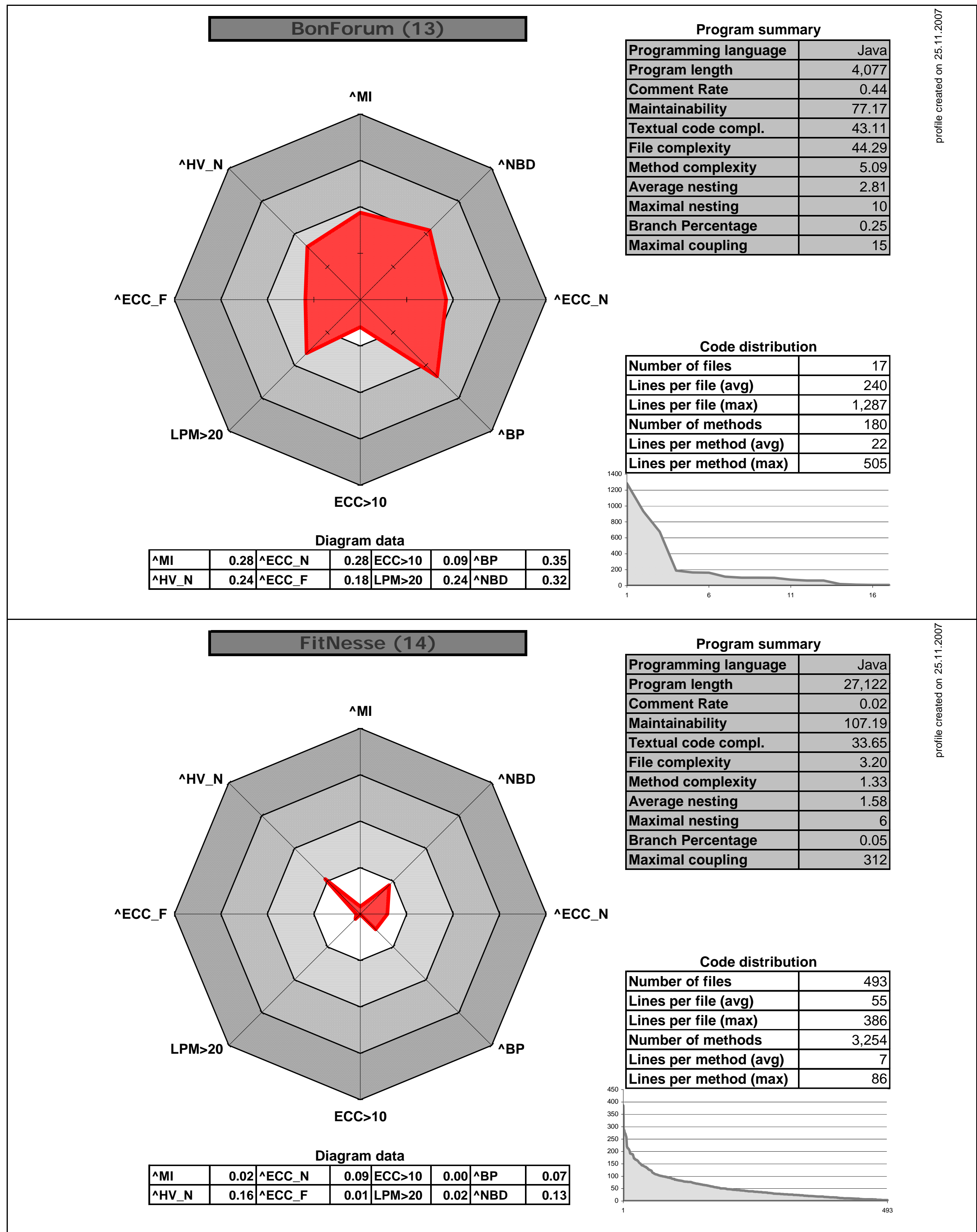
The measures for these comparative metrics are presented as surface on the kiviat diagram. In terms of interpretation, single spikes on the kiviat diagram indicate possible problems in particular code characteristics (such as textual code complexity). Here the smaller the area on kiviat diagram (face) the better the program code. The code distribution aspects are displayed in figurative and literal form in the bottom right corner.

[12] The Maintainability Index did not only required to be adjusted due to the measure range, but also was reversed as for this metric originally high values indicate a good maintainability and low values a poor maintainability [57].

8.4 Profiling the programs

8.4.1 BonForum vs. FitNesse

Consider the following profiles for BonForum and FitNesse.



A detailed comparison between the programs BonForum and FitNesse was performed in Chapter 7. At this point, with the differences of the two programs in mind, the profile generation can be examined.

The first immediately visible difference between the two profiles is the size of the shaded area on the kiviat diagram (surface), thus indicating an overall worse outcome for the program BonForum. The surface is in all 8 directions (the eight selected comparative metrics) bigger than the surface for FitNesse. The following sectors on the surface should to be viewed in combination as these can quickly reveal information about the code characteristics (see Figure 8.5):

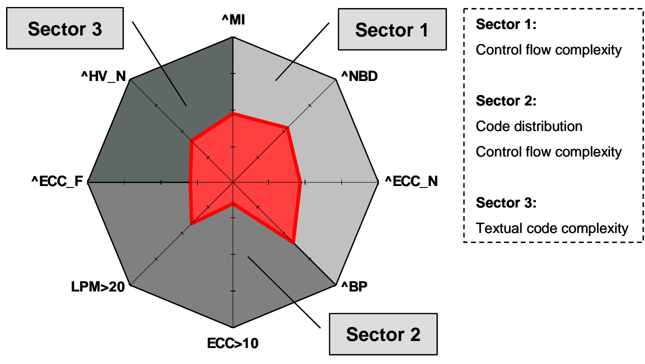


Figure 8.5: Kiviat diagram section example

- Sector 1 (^MI - ^BP) presents the control flow complexity measured independent from the code distribution
- Sector 2 (^BP - ^ECC_F) represents a combination of code distribution and control flow complexity
- Sector 3 (^ECC_F - ^MI) shows the textual code complexity of the program

With regard to the information in these 3 sectors and the two program profiles, the conclusion about these programs which was drawn in the previous chapter, can quickly be confirmed. In the following section the elements within the three sections will be considered closer.

8.4.2 Student vs. Student

In Figure 8.6, the kiviat diagrams are shown for the programs of the course D.CC and programming assignment L4. The full code profiles are presented in the corresponding section in Appendix A.4.2.

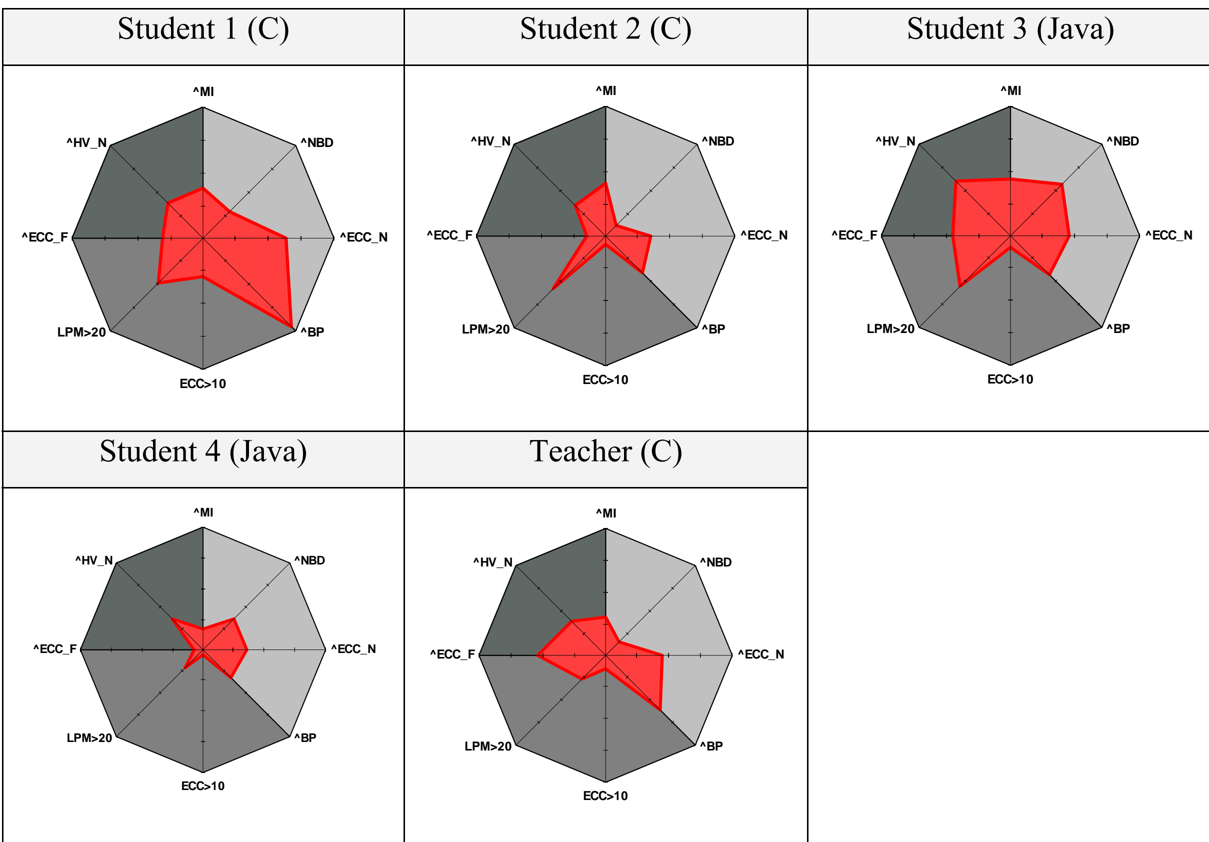


Figure 8.6: kiviat diagrams for course D.CC_L4

The kiviat surfaces spike out for student 1 and 3. The code from student 1 in particular shows a high control flow complexity. The results for Student 3's program indicate overall high values for the textual code complexity and the control flow complexity. As the teacher version indicates, the programming assignment required a rather complex control flow. Nevertheless, Student 2 and 4 showed, with this in mind, good results. The spike in the bottom right corner (LPM>20) for Student 2 together with the minimal value for the method complexity (ECC>10) indicate that the code is distributed over predominantly long but rather non complex methods. In conclusion, the 5 kiviat diagrams can be compared quickly; and the comparison points to further inspection of the programs of Student 1 and 3, by considering the full code profile (1), the full measures (2), and finally the code itself (3) (see Figure 8.1).

8.4.3 General comparison

The course, in which the design and implementation was graded (D.OODM), the code profiles show, compared to the other profiles generated, good results. These programs were written in Java. That Java programs do not necessarily produce good profiles is shown by the BonForum vs. FitNesse comparison.

When comparing the industry with the student programs (see Appendices A.4.2 and A.4.3), no clear group difference is visible. Both large and small kiviat surfaces can be found among student and industry programs. As indicated in Chapter 7, the difference is mostly to be found in the areas of program size and code distribution.

8.5 Profile drawbacks

Code profiles allow a quick overview about code characteristics, as pointed out in Section 8.2. However, as not the source code itself, but only a set of metrics quantifying code characteristics are considered, the profiles should not be over emphasized. When a set of metrics is chosen, a balance between high granularity level and clarity has to be found. A profile high in granularity can capture more aspects about the code, but will be lacking in ease of comparison and analysis.

In this dissertation, four areas of importance were selected (see Section 8.3.1); the object oriented aspects were not included in the group of comparable metrics. Thus, the profile does not indicate possible issues in the design, but focuses more on the implementation and the code itself.

8.6 Summary

For code profiling several application scenarios seem reasonable, in both education and industry. The definition of a code profile is straight forward, though several decisions (such as the granularity level, the areas of importance, and the metrics of choice) have to be made in the definition process. The programs profiled in this dissertation show, with regards to their area (education vs. industry), that apart from program size and code distribution, there were only minor differences. However, the profiles generated were used to find individual differences. Here, the results found through a detailed result analysis of the industry programs BonForum and FitNesse (see 7.7.2), were confirmed by the profiles generated for these two programs. In conclusion, using code profiles to quickly compare (without an intensive and in

depth measure analysis) software programs in terms of code characteristics was found to be very reasonable.

9 Conclusion

In this chapter conclusions from the theoretical and practical part of this dissertation are presented together with suggestions for related studies and future work.

9.1 Review

In this dissertation, a survey on software metrics and software measurement tools was conducted. A set of tools was selected and compared in terms of metric support, implementation and interpretation. Three tools (CCCC, CMT and SourceMonitor), which fulfilled previously defined requirements, were used for measuring student and industry programs with regard to the source code characteristics. The measures produced were compared and analyzed in terms of programming language groups, textual length groups, area groups (student vs. education) and individually. Since an individual comparison was found to be quite an extensive task, the idea of code profiling was used and implemented to reduce the complexity from the individual program comparisons (such as BonForum vs. FitNesse). Finally, the student and industry code profiles generated were for a subset of programs which were discussed and analyzed in detail.

9.2 Project evaluation

In the first part of the evaluation, the three key questions of the dissertation are answered and the goal achievement evaluated. The following section discusses problematic aspects during the dissertation and proposes suggestions for related work within this area.

(1) Is it useful to perform software measurements in software engineering education?

Yes, the project and the source code measurements involved show that software measurement in education is useful. The software measurement process can be, as experienced, realized in almost fully automated form, thus allowing the investigator to quickly produce and to reproduce measurement results for a group of students. Nevertheless, the preparations for an automated software measurement system can be extensive. Especially when different software measurement tools are used, extra effort is needed to unify the measurement results. Learning

to understand what the measures produced indicate and how these should be interpreted is feasible, though practical experience is needed.

- Which software metrics are available and useful for the analysis of source code?

The survey conducted (see Chapter 4) illustrates that a variety of software metrics can be used and applied in the software measurement process to measure code characteristics. In this dissertation the metrics were classified into the following six areas of code characteristics:

1. Program size
2. Code distribution
3. Textual code complexity
4. Control flow complexity
5. Code maintainability
6. Object oriented aspects

Here no single metrics were found to be useful for the analysis of source code but instead a combination of several metrics (such as control flow complexity metrics supported by code distribution metrics). A distinction between code distribution (how the source code is distributed over a number of files and methods) dependent and independent metrics was made. While interpreting measures produced by the former, code distribution aspects have to be considered. One area of code characteristics (namely object oriented aspects) was found not to be particularly useful for this dissertation and the profile generation, since determining whether a measure produced is too high or low was found to be very difficult for this area; especially if the program and the program design is not known.

The variety of software metrics available can be an issue; using and applying software metrics just because these are available can result in an overwhelming amount of data (measures) produced [2]. Depending on the scope level of interest one single metric can result in a handful of measures for one particular program measured (measures for program scope, file scope, class scope, and method scope).

- Which software measurement tools are available and how do they differ?

The survey of software measurement tools in the area of static code analysis conducted (presented in Chapter 5) points out that here a great variety also exists. For the tools surveyed, a difference in programming language support as well as in software metric support was noted. No clearly defined set of metrics was observed; each tool vendor defined their own set of interest.

None of the chosen tools presented a metrics set covering the list of metrics selected, thus a combination of tools was used. For further studies in the same area, the selection of a single tool is however advisable, since the measure unification process can present an extensive task which should be avoided if possible. The issue is that the tools export the measures produced in different formats, as no clear standard exists.

- Are the measures produced by different tools comparable?

The interpretation and computation of a single metric was found to differ somewhat from tool to tool. Thus, even for small programs noticeable differences in the values of the measures produced were found. The quality of the tools' documentation varied. In this dissertation, the creation and application of test cases was found useful to determine the metric implementation and the tool's documentation was rather seen as support in this area. A difference in metric computation results in comparison problems. Measures produced by different tools require to be transformed before any comparison. For instance, one has to transform Fahrenheit measures to Celsius measures or visa versa, for making valid comparisons of temperatures. As the metric computation can be difficult to grasp and also differ from tool to tool (see chapter 5), measure comparisons became very difficult and almost meaningless. In order to avoid these problematic aspects, each metric was computed by just one tool for all three programming languages.

(2) How does student source code differ from industry source code? Are students prepared for industrial size programming?

Since the set of source samples was small (56 programs) and not completely randomly chosen (it was acquired from what was available), this question is difficult to answer. For the set of sample programs, complex and non complex cases were found for both industry and student programs. A large difference was nevertheless found in the program size and code distribution. Thus the idea that students in software engineering education rarely face industrial scale problems was supported. Here the student assignments considered dealt with the creation of new solutions rather than with the maintenance of existing ones. Since the development of larger software system takes several months to years, students never become, when not assigned with maintenance aspects, involved with the life cycle process of these systems in their courses. The dilemma is that university courses are generally short and thus do not allow large scale assignments. Nevertheless, different computer science courses could be coordinated to have students work on the same large scale program in different courses. One course can focus on the design aspects, another on the program extension, yet another

one on the program maintenance and quality assurance. This way the students become closely familiar with the system and gain experience with and understanding of large scale programs.

- What code characteristics can be compared?

A variety of code characteristics (such as size) can be compared. The set of these is depending on the software metrics used.

- When is a code complex, and what does complex in this sense mean?

The definition of the term complex is difficult to specify precisely. Chapter 3 illustrated that different views about the related aspect of code complexity exist in the software measurement literature. Thus, it is not easy to determine what code complexity actually means. In this dissertation, code complexity was measured in terms of the control flow complexity and the textual code complexity.

(3) Is student code profiling feasible in means of quick program comparison?

For the set of programs samples, code profiling was found useful for observing a program's code characteristics and for means of quick program comparison. The profiles generated are to be seen as overview of the program's program size, code distribution, textual code complexity, control flow complexity, and maintainability. Once the preparations for the profile generation are completed, the profile generation itself is literally one click away; thus allowing such use in the education sector. Teachers and tutors can quickly browse over and compare student profiles on the fly, or save them for later comparison. In order to understand the profiles generated, surface patterns require to be interpreted. Since the profile visualization was chosen to be rather simple and immediate, the understanding and interpretation of the profiles can be taught quickly. Furthermore the profiles can be used to for different programming language, as the measures and profiles generated for programs of different programming languages were found to be comparable. Nevertheless, code distribution tendencies of the different programming languages should be noted.

- What is a code profile?

A code profile is in this dissertation defined as a formal summary of distinctive characteristics of a program source code.

- What is needed for a code profile?

In order to profile code, code characteristics of interest and software metrics which can quantify these require to be defined. In addition, a form of representation has to be selected and a balance between granularity level and overview be found.

- Can code profiling be used to improve the students programming knowledge?

Since the students were not measured continuously in this dissertation, this question can not be answered to the full extent at this point. However, since the results indicate the possibility of quick program comparisons, it seems reasonable that code profiling can used to determine areas which can be improved.

9.3 Future Work

With regard to future work the following aspects can be noted:

A continuous measurement and profiling of student programs may be started in order to collect historical data and analyse which code characteristic areas showed improvement. Once problematic and well understood areas are detected, teaching can be adjusted accordingly.

Since only a small set of sample programs (56 programs) were used, a study on student code and industry code differences may be conducted on much larger scale.

Another interesting aspect may be, to further investigate the use of object oriented metrics for code profiling. Due to difficulties experienced with the tools, the set of object oriented metrics applied was reduced and the ones which could be used were found not that useful for this study. The difficulties experienced with the tools as well as with the tool selection point out that a project regarding the creation of a software measurement tool for education may be of interest. The support for all taught programming languages and the automated generation of code profile could be seen as possible requirement for this tool.

10 References

[1] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach.* Boston 1997.

[2] J. C. Munson, *Software Engineering Measurement.* Boca Raton, FL: Auerbach Publications, 2003.

[3] Wikipedia, "Metrics," http://www.wikipedia.org/Metrics, accessed 2007-06-10.

[4] Die-net, "Measure," http://dict.die.net/measure/, accessed 2007-06-10.

[5] H. Zuse, *Software Complexity: Measures and Methods.* New York: Walter de Gruyter, 1991.

[6] Wikipedia, "Information," http://www.wikipedia.org/Information, accessed 2007-06-10.

[7] TheFreeDictionary, "Profile," http://www.thefreedictionary.com/profile, accessed 2007-11-13.

[8] H. Zuse, *A Framework of Software Measurement.* Berlin: Walter de Gruyter, 1998.

[9] H. Zuse, "History of Software Measurement," 14. September 1995, http://irb.cs.tu-berlin.de/~zuse/metrics/3-hist.html, accessed 2007-06-01.

[10] S. H. Khan, *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2002.

[11] R. J. Rubey and R. D. Hartwick, "Quantitative measurement program quality," in *ACM, National Computer Conference,* pp. 671-677, 1968.

[12] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering,* vol. 2, pp. 308-320, 1976.

[13] M. H. Halstead, *Elements of Software Science.* New York: Elsevier, 1977.

[14] E. v. Doren, "Halstead Complexity Measures," January 1997, http://www.sei.cmu.edu/str/descriptions/halstead_body.html, accessed 2007-06-01.

[15] B. W. Boehm, Clark, et al., *Software Cost Estimation with Cocomo II with Cdrom.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[16] R. E. Park, W. B. Goethert,W.A.Florac, "Goal-Driven Software Measurement – A Guidebook," 1996.

[17] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc, 1987.

[18] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity.* Upper Saddle River, N.J.: Prentice Hall, pp. 234, 1996.

[19] S. Purao and V. Vaishnavi, “Product metrics for object-oriented systems,” *ACM Comput. Surv.,* vol. 35, pp. 191-221, 2003.

[20] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Tansactions on Software Engineering 20,* pp. 476 – 493, 1994.

[21] F. B. Abreu, “The MOOD metrics set,” in *ECOOP'95,* 1995,

[22] N. E. Fenton and M. Neil, “Software Metrics: Successes, Failures and New Directions,” *Journal of Systems and Software,* vol. 47, pp. 149-157, 1999.

[23] C. R. Pandian, *Software Metrics: A Guide to Planning, Analysis, and Application.* Boca Raton, FL, USA: Auerbach Publishers, 2003.

[24] Lionel Briand, Khaled El Emam,Sandro Morasca, “On the application of measurement theory in software engineering,” International Software Engineering Research Network, Tech. Rep. #ISERN-95-04, 1996.

[25] D. R. McAndrews, “Establishing a software measurement process,” Software Engineering Institute, Tech. Rep. CMU/SEI-93-TR-16, 1993.

[26] W. A. Florac, R. E. Park and et al., “Practical software measurement: Measuring for process management and improvement,” Tech. Rep. CMU/SEI-97-HB-003, 1997.

[27] S. A. Mengel and Y. Yerramilli, “A case study of the static analysis of the quality of novice student programs,” *ACM SIGCSE Bulletin*, vol. 31, pp. 78-82, 1999.

[28] S. A. Mengel and J. Ulans, “Using Verilog LOGISCOPE to analyze student programs,” *Frontiers in Education Conference, 1998. FIE '98. 28th Annual,* vol. 3, pp. 1213-1218, 1998.

[29] N. Truong, P. Roe and P. Bancroft, “Static analysis of students' Java programs,” in *ACE '04: The sixth conference on Australasian computing education*, pp. 317-325, 2004.

[30] A. L. Patton and M. McGill, “Student portfolios and software quality metrics in computer science education,” *Journal of Computing Sciences in Colleges,* vol. 21, pp. 42-48, 2006.

[31] Wikipedia, “Static code analysis,” http://en.wikipedia.org/wiki/Static_code_analysis, accessed 2007-07-10.

[32] P. Pohjolainen, “Software testing Tools,” Department of Computer Science and Applied Mathematics, University of Kuopio, 2002.

[33] N. Fenton, “Software Measurement: A Necessary Scientific Basis,” *IEEE Trans. Software Eng.,* vol. 20, pp. 199-206, 1994.

[34] L. M. Laird and M. C. Brennan, *Software Measurement and Estimation: A Practical Approach (Quantitative Software Engineering Series).* Wiley-IEEE Computer Society Pr, 2006.

[35] T. Littlefair, “An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment,” Ph.D. thesis, 2001.

[36] R. L. Glass, “The relationship between theory and practice in software engineering,” *Commun ACM,* vol. 39, pp. 11-13, 1996.

[37] S. Rabinovich, *Measurement Errors and Uncertainties: Theory and Practice.* Springer, 2005.

[38] I. Sommerville, *Software Engineering.* 7th ed., Pearson Addison Wesley, 2004.

[39] L. Arockiam, U. L. Stanislaus, P. D. Sheba and Kasmir Raja, S. V., "An analysis of method complexity of object-oriented system using statistical techniques," in *Conference Proceeding SMEF 2005,* 2005.

[40] D. P. Tegarden, S. D. Sheetz and D. E. Monarchi, “A software complexity model of object-oriented systems,” *Decis. Support Syst.,* vol. 13, pp. 241-262, 1995.

[41] A. Abran, M. Lopez and N. Habra, “An analysis of the McCabe cyclomatic complexity number,” in *IWSM Proceedings*, Berlin, 2004.

[42] IEEE Computer Society, “IEEE Glossary of Software Engineering Terminology,” 1990.

[43] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality.* 2nd ed., New York: McGraw-Hill, pp. 618, 1997.

[44] M. Bauer, “Analyzing software systems using combinations of metrics,” in *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering,* 1999,

[45] K. E. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis and S. N. Rai, “The Optimal Class Size for Object-Oriented Software,” *IEEE Trans. Softw. Eng.,* vol. 28, pp. 494-509, 2002.

[46] T. Klemola, "A cognitive model for complexity metrics," in *4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering,* 2000.

[47] S. McConnell, *Code Complete, Second Edition.* Redmond, WA, USA: Microsoft Press, 2004.

[48] H. Zuse, “Resolving the Mysteries of the Halstead Measures,” *Metrikon 2005 - Software-Messung in Der Praxis,* 2005.

[49] P. Oman, “HP-MAS: A tool for software maintainability, software engineering,” Moscow, ID: Test Laboratory, University of Idaho, Tech. Rep. (#91-08-TR), 1991.

[50] E. Li, "On the cyclomatic metric of program complexity," California Polytechnic State University School of Buisness, Quality DATA PROCESSING, 1987.

[51] A. H. Watson and T. J. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," National Institute of Standards and Technology, 1996.

[52] E. v. Doren, "Cyclomatic Complexity," January 1997, http://www.sei.cmu.edu/str/descriptions/cyclomatic.html, accessed 2007-06-01.

[53] D. Spinellis, *Code Quality: The Open Source Perspective.* Addison Wesley, 2006.

[54] A. Krusko, "Complexity analysis of real time software: using software complexity metrics to improve the quality of real-time-software," MS.c thesis, Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Sweden, vol. 04032, pp. 90, 2004.

[55] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* 5th ed., Boston: McGraw-Hill, pp. 860, 2001.

[56] E. v. Doren, "Maintainability Index Technique for Measuring Program Maintainability," January 1997, http://www.sei.cmu.edu/str/descriptions/mitmpm.html, accessed 2007-06-01.

[57] T. Kuipers and J. Visser, "Maintainability index revisited," in *11th European Conference on Software Maintenance and Reengineering,* 2007.

[58] J. R. Abounader and D. A. Lamb, "A data model for object oriented design metrics," Department of Computing and Information Science Queen's University, Kingston, Ontario, Canada, Tech. Rep. ISSN-0836-0227-1997-409, 1997.

[59] M. Sarker, "An overview of Object Oriented Design Metrics," MS.c thesis, Department of Computer Science, Umeå University, Sweden, 2005.

[60] Xenos, M. et al., "Object oriented metrics – A survey," in *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations,* 2000,

[61] R. Harrison, S. J. Counsell and R. V. Nithi, "An Investigation into the Applicability and Validity of Object-Oriented Design Metrics," *Empirical Software Engineering: An International Journal,* vol. 3, pp. 255-273, 1998.

[62] R. Reißing, "Towards a model for object-oriented design measurement," in *5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering,* 2001.

[63] R. Harrison, S. J. Counsell and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Eng.,* vol. 24, pp. 491-496, 1998.

[64] Information Society and Media, "Software quality observatory for open source software," Tech. Rep. IST-2005-33331, 2007.

[65] M. Andersson and P. Vestergren, “Object-Oriented Design Quality Metrics,” MS.c thesis, Computing Science Department, Uppsala University, Sweden, 2004.

[66] N. Clark, “Evaluating student teams developing unique industry projects,” in *ACE '05: Proceedings of the 7th Australasian Conference on Computing Education,* pp. 21-30, 2005.

[67] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *SIGCSE Bull,* vol. 36, pp. 26-30, 2004.

[68] J. Tenenberg, S. Fincher, et al., “Students Designing Software: a Multi-National, Multi-Institutional Study,” *Informatics in Education,* vol. 4, pp. 143-162, 2005.

[69] J. B. Raley, “Factors affecting the programming performance of computer science students,” MS.c thesis, Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Va., 1996.

[70] A. Eckerdal, R. McCartney, J. E. Mostrom, M. Ratcliffe and C. Zander, “Can graduating students design software systems?” in *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pp. 403-407, 2006.

[71] G. T. Leavens, A. L. Baker, V. Honavar, S. Lavalle and G. Prabhu, “Programming is Writing: Why Student Programs must be Carefully Evaluated,” *Mathematics and Computer Education,* vol. 32, pp. 284-295, 1998.

[72] P. Relf, “Achieving software quality through source code readability,” in *QUALCON 2004: Quality - the Key to Organisational Prosperity,* 2004.

[73] I. Stamelos, L. Angelis, A. Oikonomou and G. L. Bleris, “Code quality analysis in open source software development,” *Information Systems Journal 12*, pp. 43-60, 2002.

[74] M. Aberdour, “Achieving Quality in Open Source Software,” *IEEE Software,* vol. 24, pp. 58-64, 2007.

[75] A. MacCormack, J. Rusnak and C. Y. Baldwin, “Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code,” *Manage. Sci.,* vol. 52, pp. 1015-1030, 2006.

[76] D. Berry, “Open source software,” *Parliamentary Office of Science and Technology*, vol. Number 242, London, 2005.

[77] S. R. Schach, *Object-Oriented and Classical Software Engineering.* McGraw-Hill Pub. Co, 2001.

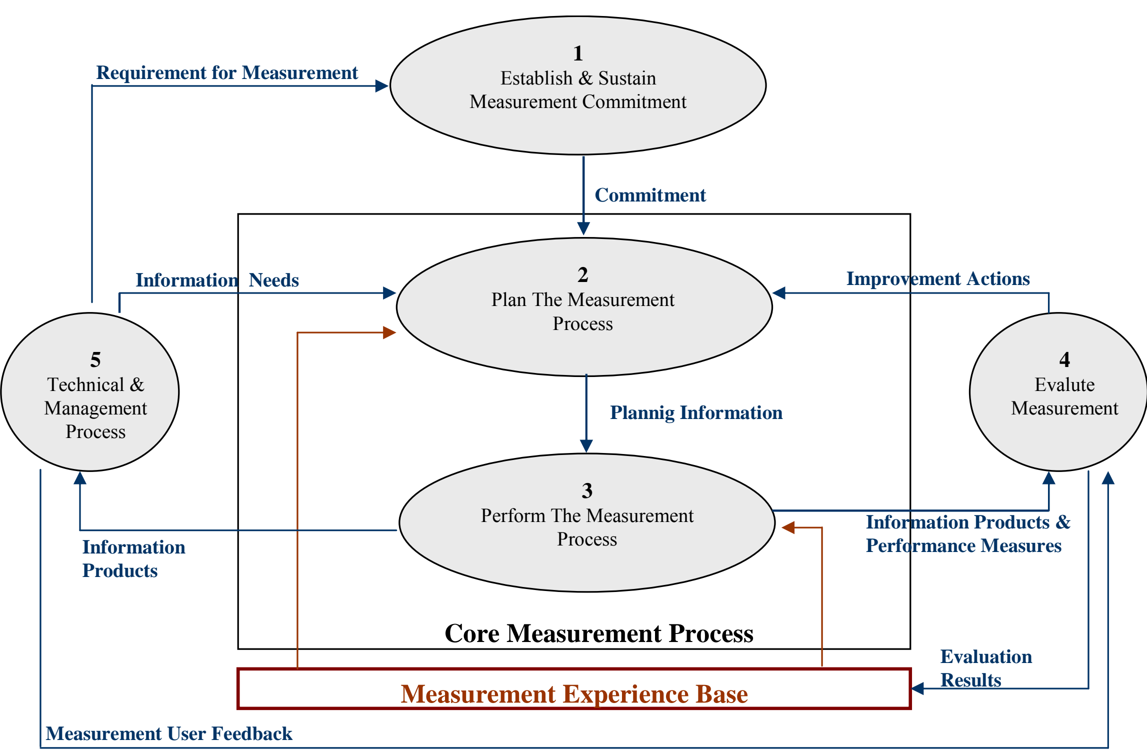
[78] A. Kent, *Encyclopedia of Computer Science and Technology.* CRC, 1997.

[79] D. Holten, R. Vliegen and van Wijk, Jarke J., “Visualization of software metrics using computer graphics techniques,” in *Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging,* 2006,

[80] M. Termeer, C. F. J. Lange, A. Telea and M. R. V. Chaudron, "Visual exploration of combined architectural and metric information." in *VISSOFT,* pp. 21-26, 2005.

11 Appendix

A.1 Software measurement process



Software measurement process: Scope of ISO/IEC 15939

A.2 Measurement result tables

A.2.1 List of software metrics applied

		Scope applied in…			
Abb.	**Full name**	**Program**	**File**	**Class**	**Method**
LOC	(Physical) Lines of Code	X	X		X
NCLOC	Non Comment Lines of Code	X	X	X	X
#STAT	Number of statements	X	X		X
HL	Halstead Length	X	X		
LPM	Lines (of code) per Method	X	X		
SPM	Statements per Method	X	X		
#files	Number of files	X			
#methods	Number of methods	X	X		
LPM>20	Percentage of methods with LPM bigger than 20	X			
Hvoc	Halstead vocabulary	X	X		
HV	Halstead Volume	X	X		
HD	Halstead Difficulty	X	X		
HE	Halstead Effort	X	X		
HV_N	Normalized Halstead Volume	X			
HE_N	Normalized Halstead Effort	X			
CR	Comment Rate	X	X		
MI	Maintainability Index (without comments)	X	X		
MIwc	Maintainability Index (with comments)	X	X		
NBD	Nesting Block Depth	X	X		X
BP	Branch Percentage	X	X		
ECC	Extended Cyclomatic Complexity	X	X	X	X
ECC_N	Normalized ECC	X			
ECC>8	Percentage of methods with ECC bigger than 8	X			
ECC>10	Percentage of methods with ECC bigger than 10	X			
WMC	Weighted Methods per Class	X		X	
DIT	Depth of Inheritance Tree	X		X	
NOC	Number of Children	X		X	
CBO	Coupling Between Objects	X		X	
FI	Fan In	X		X	
FO	Fan Out	X		X	
IF4	Information Flow	X		X	

A.2.2 Program size & code distribution

pid	lan	name	total_LOC	total_NCLOC	total_#STAT	total_HL	avg_NCLOC	min_NCLOC	max_NCLOC	stdev_NCLOC	avg_HL	avg_LOC	avg_#STAT	avg_LPM	max_LPM	#files	#methods	LPM>20
1	C	GIMP	732157	526279	290981	2755645	255.1	2	5586	429.95	1335.74	354.89	141.04	27.78	922	2063	14795	0.42
2	C	Miranda	34029	28823	23892	196620	294.11	1	1215	236.92	2006.32	347.23	243.79	30.01	1041	98	892	0.35
3	C	NullWebmail	9912	8925	6961	63583	307.75	33	918	250.04	2192.51	341.79	240.03	39.78	255	29	179	0.58
4	C	Panda	26049	13848	6706	148989	197.82	1	1895	284.21	2128.41	372.12	95.8	18.90	326	70	261	0.22
5	C	SDCC	207690	144892	96970	757764	728.1	1	11077	1681.26	3807.85	1043.66	487.28	33.44	1763	199	3874	0.44
6	C	VIM	320531	239693	123416	1111573	2282.79	0	16512	2713.82	10586.4	3082.02	1186.69	25.86	1070	104	1592	0.31
7	C++	7-Zip	117789	95889	64469	469066	104.34	1	1376	169.52	510.4	128.73	70.45	19.93	459	915	4063	0.25
8	C++	DiskCleaner	7470	5430	3765	25748	93.62	1	824	158.94	443.93	128.79	64.91	21.53	134	58	216	0.32
9	C++	Emule	221641	169635	124427	947371	258.19	0	4723	472.18	1441.96	337.86	189.67	23.37	2326	656	6395	0.28
10	C++	FileZilla	112208	81934	58600	450582	278.68	1	4922	595.32	1532.59	381.65	199.31	29.94	1691	294	1881	0.34
11	C++	HexView	1856	1121	721	5131	70.06	1	528	127.32	320.68	116	45.06	14.47	112	16	66	0.23
12	C++	Notepad++	48233	35677	27285	195363	251.24	2	5711	591.51	1375.79	339.66	192.14	37.02	1444	142	893	0.45
13	Java	BonForum	8584	4077	3086	21545	239.82	9	1287	367.01	1267.35	504.94	181.52	21.81	505	17	180	0.24
14	Java	FitNesse	35347	27122	20399	138937	55.01	4	386	53.96	281.81	71.69	41.37	7.45	86	493	3254	0.02
15	Java	HTML_Unit	42482	14188	11203	70533	65.08	3	843	117.33	323.54	194.87	51.38	6.08	96	218	2156	0.05
16	Java	NekoHTML	11482	6478	4890	34247	223.37	5	2298	431.68	1180.93	395.93	168.62	13.54	181	29	414	0.17
17	Java	Jsettlers	56768	30724	20801	154519	207.59	10	3465	498.15	1044.04	383.56	140.54	17.27	1338	148	1682	0.17
18	Java	JSS	3029	1198	1762	8521	79.86	10	230	73.46	568.06	201.93	117.46	17.49	157	15	95	0.27
19	Java	CS.PPS	120778	76338	65109	493258	207.44	3	16786	925.08	1340.37	328.2	176.92	17.13	716	368	4279	0.22
20	Java	A.OOPJ_L1_st1	265	172	96	529	86	28	144	82.02	264.5	132.5	48	10.19	44	2	16	0.06
21	Java	A.OOPJ_L2_st1	758	502	298	1868	100.4	12	302	119.56	373.6	151.6	59.6	13.94	53	5	34	0.26
22	Java	A.OOPJ_L3_st1	1206	586	387	2921	58.6	7	158	39.40	292.1	120.6	38.7	6.63	23	10	81	0.01
23	Java	A.OOPJ_L4_st1	918	318	341	1358	39.75	23	55	12.43	169.75	70.61	26.23	9.28	23	13	29	0.07
24	Java	A.OOPJ_L5_st1	2021	1148	786	5218	54.66	7	148	31.66	248.47	96.23	37.42	10.40	43	21	97	0.14
25	C	B.OS_L1_st1	183	120	80	407	40	7	68	30.81	135.66	61	26.66	15.17	36	3	6	0.17
26	C	B.OS_L2_st1	306	197	169	746	49.25	45	54	4.03	186.5	76.5	42.25	11.42	25	4	12	0.08
27	C	B.OS_L3_st1	76	52	42	298	52	52	52	-	298	76	42	14.33	26	1	3	0.33
28	C	B.OS_L4_st1	219	190	132	977	190	190	190	-	977	219	132	35.40	94	1	5	0.40
29	C	B.OS_L5_st1	358	301	193	1373	301	301	301	-	1373	358	193	49.67	92	1	6	0.67
30	C++	C.PL_L1_st1	1084	763	620	3884	54.5	8	281	69.52	277.42	77.42	44.28	12.30	54	14	54	0.11
31	C	C.PL_L1_st2	983	692	564	2973	62.9	10	259	77.98	270.27	89.36	51.27	14.94	51	11	34	0.24
32	C	C.PL_L1_st3	837	652	549	3435	163	17	427	190.04	858.75	209.25	137.25	19.83	131	4	29	0.24
33	C++	C.PL_L1_st4	1025	780	694	4260	65	12	188	53.16	355	85.41	57.83	10.34	46	12	65	0.11
34	C++	C.PL_L1_st5	1326	1058	900	5318	48.09	6	260	59.78	241.72	60.27	40.9	10.03	123	22	93	0.09
35	C	C.PL_L1_st6	556	479	381	2508	95.8	17	259	100.25	501.6	111.2	76.2	8.04	36	5	45	0.04
36	C++	C.PL_L1_st7	392	331	227	1532	331	331	331	-	1532	392	227	13.87	46	1	23	0.22
37	C	D.CC_L2_teacher	1332	884	831	4808	176.8	5	633	262.43	961.6	266.4	166.2	9.48	38	5	75	0.12
38	C	D.CC_L2_st1	620	526	386	2017	526	526	526	-	2017	620	386	18.04	40	1	28	0.39
39	C	D.CC_L2_st2	825	689	586	3153	137.8	6	560	238.44	630.6	165	117.2	17.63	40	5	35	0.34
40	C	D.CC_L2_st3	625	557	549	2026	557	557	557	-	2026	625	549	8.69	35	1	58	0.09
41	C	D.CC_L2_st4	986	755	662	4097	377.5	80	675	420.73	2048.5	493	331	14.79	35	2	42	0.29
42	C	D.CC_L3_teacher	1596	1089	1010	5954	217.8	5	838	352.18	1190.8	319.2	202	9.61	42	5	94	0.14
43	C	D.CC_L3_st1	620	526	386	2017	526	526	526	-	2017	620	386	18.04	40	1	28	0.39
44	C	D.CC_L3_st2	1091	883	761	4498	176.6	6	716	304.64	899.6	218.2	152.2	18.49	48	5	43	0.37
45	C	D.CC_L3_st3	1059	960	883	4007	960	960	960	-	4007	1059	883	14.93	92	1	59	0.25
46	C	D.CC_L3_st4	1374	1044	883	5901	149.14	9	661	232.59	843	196.28	126.14	15.63	41	7	52	0.29
47	C	D.CC_L4_teacher	620	448	375	2319	448	448	448	-	2319	620	375	12.39	50	1	31	0.16
48	C	D.CC_L4_st1	440	363	304	2119	181.5	118	245	89.80	1059.5	220	152	19.53	53	2	17	0.29
49	C	D.CC_L4_st2	587	475	383	2581	158.33	43	299	129.87	860.33	195.66	127.66	15.73	46	3	26	0.35
50	Java	D.CC_L4_st3	540	365	336	2858	365	365	365	-	2858	540	336	19.39	117	1	18	0.33
51	Java	D.CC_L4_st4	716	528	435	3214	75.42	15	207	69.71	459.14	102.28	62.14	11.80	77	7	41	0.12
52	Java	D.OODM_L4_st1	1376	676	401	2583	30.72	7	126	29.69	117.4	62.54	18.22	6.77	26	22	87	0.05
53	Java	D.OODM_L4_st2	1368	500	450	2594	17.24	4	61	15.82	89.44	47.17	15.51	4.14	23	29	100	0.01
54	Java	D.OODM_L4_st3	1858	558	488	3234	21.46	7	75	17.78	124.38	71.46	18.76	4.79	29	26	100	0.03
55	Java	D.OODM_L4_st4	1448	684	481	2781	23.58	5	94	22.85	95.89	49.93	16.58	7.78	67	29	77	0.13
56	Java	D.OODM_L4_st5	1386	552	427	3085	29.05	11	150	31.30	162.36	72.94	22.47	4.99	30	19	98	0.06

A.2.3 Textual code complexity and code maintainability

pid	lan	name	total_Hvoc	avg_Hvoc	total_HV	avg_HV	total_HD	avg_HD	total_HE	avg_HE	HE_N	HV_N	avg_CR	avg_MI	stdev_MI	avg_MIwc
1	C	GIMP	283860	137.59	22448069	10881.27	95796	46.43	7778062667	3770267.9	14779.35	42.65	0.12	73.94	15.72	105.25
2	C	Miranda	23453	239.31	1631597	16648.94	7201	73.47	169110062	1725612.9	5867.19	56.61	0.09	71.89	18.96	97.79
3	C	NullWebmail	6524	224.96	515362	17771.1	2379	82.03	64653276	2229423.3	7244.06	57.74	0.16	54.37	19.90	81.75
4	C	Panda	11188	159.82	1115848	15940.68	3380	48.28	564495518	8064221.7	40763.69	80.58	0.41	52.45	36.94	90.84
5	C	SDCC	46368	233	7232712	36345.28	15393	77.35	2373544889	11927361.3	16381.48	49.92	0.19	75.01	21.65	103.47
6	C	VIM	75169	715.89	11304576	107662.62	14013	133.45	2552986794	24314159.9	10651.07	47.16	0.20	23.96	45.05	56.81
7	C++	7-Zip	80773	87.89	3545186	3857.65	30737	33.44	347877457	378539.1	3627.92	36.97	0.07	90.76	23.64	109.47
8	C++	DiskCleaner	4917	84.77	193799	3341.36	1413	24.36	13622036	234862.7	2508.66	35.69	0.12	87.44	19.87	113.31
9	C++	Emule	126459	192.47	8307287	12644.27	32321	49.19	1056524874	1608104.8	6228.22	48.97	0.14	75.26	19.49	104.75
10	C++	FileZilla	60856	206.99	4039110	13738.46	12751	43.37	553839334	1883807.3	6759.58	49.30	0.16	63.78	25.50	98.23
11	C++	HexView	1157	72.31	38222	2388.87	328	20.50	3012977	188311.1	2687.76	34.10	0.26	86.81	13.90	126.81
12	C++	Notepad++	25448	179.21	1726844	12160.87	6768	47.66	214834473	1512918.8	6021.65	48.40	0.12	69.48	19.94	99.04
13	Java	BonForum	2459	144.64	175778	10339.88	793	46.64	19169846	1127638.0	4701.95	43.11	0.44	77.17	20.34	114.23
14	Java	FitNesse	31875	64.65	912566	1851.04	9000	18.25	26315393	53378.1	970.26	33.65	0.02	107.19	10.52	109.02
15	Java	HTML_Unit	14540	66.69	510462	2341.56	4096	18.78	25085323	115070.3	1768.07	35.98	0.53	101.72	8.28	145.99
16	Java	NekoHTML	3919	135.13	281373	9702.51	1400	48.27	33038474	1139257.7	5100.10	43.44	0.32	87.58	17.34	128.48
17	Java	Jsettlers	14687	99.23	1298697	8774.97	5677	38.35	189110608	1277774.4	6155.14	42.27	0.30	95.56	14.74	135.70
18	Java	JSS	1528	101.86	60938	4062.53	450	30.00	2908658	193910.5	2427.93	50.87	0.25	76.53	10.56	111.80
19	Java	CS.PPS	52302	142.12	4371178	11878.2	15403	41.85	693377956	1884179.2	9083.00	57.26	0.20	80.26	24.84	103.98
20	Java	A.OOPJ_L1_st1	118	59	3308	1654	47	23.50	114137	57068.5	663.59	19.23	0.26	102.50	13.44	140.00
21	Java	A.OOPJ_L2_st1	327	65.4	12765	2553	144	28.80	673042	134608.4	1340.72	25.43	0.25	97.20	10.71	136.80
22	Java	A.OOPJ_L3_st1	608	60.8	18205	1820.5	186	18.60	490289	49028.9	836.67	31.07	0.37	101.90	7.03	139.10
23	Java	A.OOPJ_L4_st1	478	59.75	8171	1021.37	134	16.75	147749	18468.6	464.62	25.69	0.30	93.87	5.30	132.75
24	Java	A.OOPJ_L5_st1	1311	62.42	32345	1540.23	427	20.33	791571	37693.9	689.52	28.18	0.30	91.76	11.92	130.19
25	C	B.OS_L1_st1	139	46.33	2433	811	45	15.00	58424	19474.7	486.87	20.28	0.26	89.33	17.16	126.33
26	C	B.OS_L2_st1	205	51.25	4243	1060.75	68	17.00	73254	18313.5	371.85	21.54	0.14	87.50	1.91	115.50
27	C	B.OS_L3_st1	71	71	1833	1833	35	35.00	65039	65039.0	1250.75	35.25	0.21	85.00	-	117.00
28	C	B.OS_L4_st1	102	102	6519	6519	78	78.00	509594	509594.0	2682.07	34.31	0.00	71.00	-	78.00
29	C	B.OS_L5_st1	79	79	8655	8655	156	156.00	1350194	1350194.0	4485.69	28.75	0.09	65.00	-	87.00
30	C++	C.PL_L1_st1	790	56.42	24827	1773.35	384	27.42	1243100	88792.9	1629.23	32.54	0.14	82.57	11.93	109.14
31	C	C.PL_L1_st2	646	58.72	19178	1743.45	244	22.18	719334	65394.0	1039.50	27.71	0.09	82.63	12.77	105.72
32	C	C.PL_L1_st3	430	107.5	25638	6409.5	194	48.50	2706760	676690.0	4151.47	39.32	0.03	82.25	14.38	90.75
33	C++	C.PL_L1_st4	789	65.75	27265	2272.08	394	32.83	1352322	112693.5	1733.75	34.96	0.10	85.50	14.90	102.66
34	C++	C.PL_L1_st5	1186	53.9	34311	1559.59	607	27.59	1874811	85218.7	1772.03	32.43	0.10	93.59	15.15	114.00
35	C	C.PL_L1_st6	409	81.8	17288	3457.6	160	32.00	970672	194134.4	2026.46	36.09	0.00	88.40	14.60	94.60
36	C++	C.PL_L1_st7	143	143	10969	10969	85	85.00	936159	936159.0	2828.27	33.14	0.00	92.00	-	99.00
37	C	D.CC_L2_teacher	663	132.6	38767	7753.4	122	24.40	1982214	396442.8	2242.32	43.85	0.14	89.00	14.02	123.20
38	C	D.CC_L2_st1	205	205	15490	15490	82	82.00	1263944	1263944.0	2402.94	29.45	0.02	87.00	-	100.00
39	C	D.CC_L2_st2	447	89.4	24157	4831.4	133	26.60	1304809	260961.8	1893.77	35.06	0.05	93.40	13.81	101.20
40	C	D.CC_L2_st3	231	231	15908	15908	27	27.00	427354	427354.0	767.24	28.56	0.01	103.00	-	112.00
41	C	D.CC_L2_st4	457	228.5	33763	16881.5	113	56.50	3204983	1602491.5	4245.01	44.72	0.10	70.50	21.92	94.50
42	C	D.CC_L3_teacher	783	156.6	50412	10082.4	146	29.20	3591075	718215.0	3297.59	46.29	0.13	89.40	14.01	123.40
43	C	D.CC_L3_st1	205	205	15490	15490	82	82.00	1263944	1263944.0	2402.94	29.45	0.02	87.00	-	100.00
44	C	D.CC_L3_st2	532	106.4	35949	7189.8	178	35.60	3042870	608574.0	3446.06	40.71	0.05	91.60	16.61	98.80
45	C	D.CC_L3_st3	342	342	33730	33730	85	85.00	2862660	2862660.0	2981.94	35.14	0.08	90.00	-	112.00
46	C	D.CC_L3_st4	786	112.28	46096	6585.14	219	31.28	3766006	538000.9	3607.29	44.15	0.09	81.42	15.90	100.42
47	C	D.CC_L4_teacher	259	259	18591	18591	85	85.00	1587256	1587256.0	3542.98	41.50	0.12	89.00	-	115.00
48	C	D.CC_L4_st1	266	133	14988	7494	136	68.00	1063796	531898.0	2930.57	41.29	0.11	83.00	5.66	100.50
49	C	D.CC_L4_st2	340	113.33	18226	6075.33	154	51.33	1220358	406786.0	2569.17	38.37	0.05	81.66	14.01	93.33
50	Java	D.CC_L4_st3	148	148	20605	20605	179	179.00	3687786	3687786.0	10103.52	56.45	0.00	79.00	-	84.00
51	Java	D.CC_L4_st4	486	69.42	21166	3023.71	245	35.00	1276613	182373.3	2417.83	40.09	0.13	98.00	13.89	124.00
52	Java	D.OODM_L4_st1	729	33.13	14674	667	361	16.40	468349	21288.6	692.82	21.71	0.45	100.50	5.89	144.95
53	Java	D.OODM_L4_st2	859	29.62	13987	482.31	345	11.89	288701	9955.2	577.40	27.97	0.43	104.20	8.58	148.00
54	Java	D.OODM_L4_st3	959	36.88	18306	704.07	387	14.88	466933	17959.0	836.80	32.81	0.55	98.84	7.61	145.42
55	Java	D.OODM_L4_st4	806	27.79	15161	522.79	386	13.31	439734	15163.2	642.89	22.17	0.38	100.24	10.40	142.96
56	Java	D.OODM_L4_st5	792	41.68	18016	948.21	361	19.00	638224	33590.7	1156.20	32.64	0.48	104.94	8.28	150.00

A.2.4 Structural code complexity

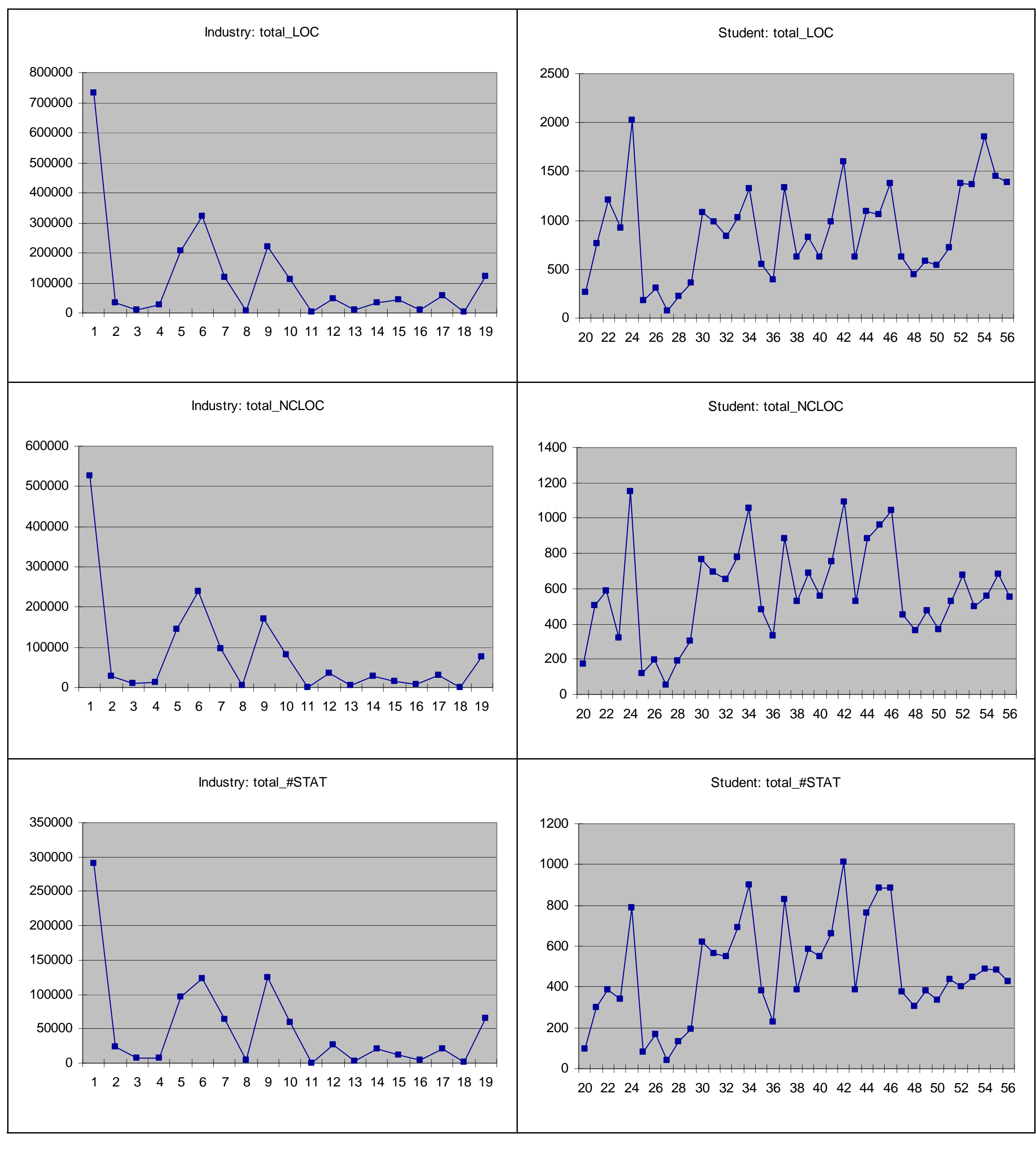
pid	lan	name	avg_NBD	max_NBD	avg_BP	avg_method_ECC	max_method_ECC	total_ECC	avg_File_ECC	max_File_ECC	stdev_ECC	ECC_N	ECC>8	ECC>10
1	C	GIMP	1.57	10	0.18	4.04	143	50842	24.64	901	52.53	0.10	0.11	0.08
2	C	Miranda	2.65	10	0.28	7.89	277	6263	63.9	313	61.35	0.22	0.22	0.17
3	C	NullWebmail	1.84	8	0.23	13.41	111	2421	83.48	306	89.66	0.27	0.42	0.38
4	C	Panda	0.79	10	0.07	3.15	51	1751	25.01	326	56.21	0.13	0.09	0.07
5	C	SDCC	2.19	10	0.30	8.99	466	32043	161.02	2259	367.28	0.22	0.28	0.22
6	C	VIM	2.25	10	0.28	6.56	352	59548	567.12	3582	721.56	0.25	0.18	0.14
7	C++	7-Zip	1.65	10	0.19	3.96	94	14212	15.46	278	30.36	0.15	0.10	0.08
8	C++	DiskCleaner	1.65	8	0.18	3.85	22	721	12.43	131	23.29	0.13	0.12	0.09
9	C++	Emule	2.03	10	0.21	5.18	515	28756	43.76	1284	105.34	0.17	0.13	0.10
10	C++	FileZilla	1.71	10	0.17	6.50	474	11932	40.58	1248	125.35	0.15	0.18	0.15
11	C++	HexView	1.32	6	0.15	2.27	19	118	7.37	59	14.14	0.11	0.03	0.03
12	C++	Notepad++	2.29	10	0.25	7.25	268	5836	41.09	1198	123.12	0.16	0.21	0.15
13	Java	BonForum	2.81	10	0.25	5.09	104	753	44.29	219	68.05	0.18	0.11	0.09
14	Java	FitNesse	1.58	6	0.05	1.33	40	1580	3.2	41	4.40	0.06	0.00	0.00
15	Java	HTML_Unit	1.77	8	0.13	1.64	25	1600	7.33	157	16.96	0.11	0.01	0.01
16	Java	NekoHTML	2.86	10	0.21	3.81	44	1205	41.55	557	104.08	0.19	0.11	0.07
17	Java	Jsettlers	3.14	10	0.22	3.59	386	4506	30.44	613	92.57	0.15	0.06	0.05
18	Java	JSS	2.57	8	0.10	3.37	20	177	11.8	44	11.66	0.15	0.11	0.08
19	Java	CS.PPS	2.61	10	0.10	2.85	72	8256	22.43	1369	81.21	0.11	0.06	0.04
20	Java	A.OOPJ_L1_st1	2.21	5	0.31	2.00	10	18	9	17	11.31	0.10	0.06	0.00
21	Java	A.OOPJ_L2_st1	2.56	8	0.24	2.44	13	54	10.8	36	14.79	0.11	0.03	0.03
22	Java	A.OOPJ_L3_st1	1.56	5	0.03	1.12	4	20	2	7	2.16	0.03	0.00	0.00
23	Java	A.OOPJ_L4_st1	1.6	4	0.09	1.66	4	27	3.37	7	1.77	0.08	0.00	0.00
24	Java	A.OOPJ_L5_st1	1.92	6	0.11	1.84	8	102	4.85	16	4.05	0.09	0.00	0.00
25	C	B.OS_L1_st1	0.88	3	0.16	3.33	9	17	5.66	13	6.35	0.14	0.17	0.00
26	C	B.OS_L2_st1	0.8	2	0.07	2.08	5	17	4.25	5	0.50	0.09	0.00	0.00
27	C	B.OS_L3_st1	1.02	3	0.14	3.00	5	7	7	7	-	0.13	0.00	0.00
28	C	B.OS_L4_st1	2.02	6	0.22	7.40	19	33	33	33	-	0.17	0.20	0.20
29	C	B.OS_L5_st1	3.51	7	0.28	10.50	19	58	58	58	-	0.19	0.50	0.50
30	C++	C.PL_L1_st1	1.22	4	0.16	2.78	10	118	8.42	53	13.73	0.15	0.02	0.00
31	C	C.PL_L1_st2	1.14	4	0.21	4.15	15	124	11.27	59	18.22	0.18	0.09	0.09
32	C	C.PL_L1_st3	1.65	6	0.22	4.90	41	119	29.75	65	26.89	0.18	0.10	0.10
33	C++	C.PL_L1_st4	1.14	5	0.13	2.48	18	114	9.5	43	12.75	0.15	0.02	0.02
34	C++	C.PL_L1_st5	1.34	5	0.19	2.65	26	187	8.5	61	14.98	0.18	0.02	0.02
35	C	C.PL_L1_st6	1.06	4	0.19	2.38	10	69	13.8	37	15.94	0.14	0.02	0.00
36	C++	C.PL_L1_st7	1.5	5	0.18	3.26	11	54	54	54	-	0.16	0.04	0.04
37	C	D.CC_L2_teacher	1.23	5	0.20	3.07	13	160	32	132	56.58	0.18	0.04	0.01
38	C	D.CC_L2_st1	1.61	4	0.24	3.79	11	79	79	79	-	0.15	0.04	0.04
39	C	D.CC_L2_st2	1.36	4	0.28	4.89	10	143	28.6	123	52.97	0.21	0.14	0.00
40	C	D.CC_L2_st3	1.24	3	0.33	3.41	9	147	147	147	-	0.26	0.03	0.00
41	C	D.CC_L2_st4	1.59	4	0.31	4.93	33	168	84	166	115.97	0.22	0.10	0.07
42	C	D.CC_L3_teacher	1.2	5	0.18	2.76	13	170	34	142	61.00	0.16	0.03	0.01
43	C	D.CC_L3_st1	1.61	4	0.24	3.79	11	79	79	79	-	0.15	0.04	0.04
44	C	D.CC_L3_st2	1.37	4	0.23	4.53	13	159	31.8	135	58.04	0.18	0.14	0.09
45	C	D.CC_L3_st3	1.54	5	0.33	5.17	25	253	253	253	-	0.26	0.15	0.14
46	C	D.CC_L3_st4	1.54	4	0.26	4.65	34	201	28.71	158	57.86	0.19	0.10	0.06
47	C	D.CC_L4_teacher	1.32	4	0.26	3.55	16	80	80	80	-	0.18	0.10	0.06
48	C	D.CC_L4_st1	1.83	5	0.41	6.29	19	92	46	47	1.41	0.25	0.24	0.18
49	C	D.CC_L4_st2	1.16	3	0.17	3.42	19	67	22.33	45	21.59	0.14	0.04	0.04
50	Java	D.CC_L4_st3	2.97	7	0.18	4.67	37	67	67	67	-	0.18	0.06	0.06
51	Java	D.CC_L4_st4	2.12	5	0.14	2.68	25	76	10.85	39	13.17	0.14	0.02	0.02
52	Java	D.OODM_L4_st1	1.85	5	0.24	1.77	9	89	4.04	17	5.01	0.13	0.01	0.00
53	Java	D.OODM_L4_st2	1.51	5	0.14	1.57	21	86	2.96	24	4.58	0.17	0.01	0.01
54	Java	D.OODM_L4_st3	1.48	5	0.10	1.47	7	73	2.8	10	2.28	0.13	0.00	0.00
55	Java	D.OODM_L4_st4	2.09	6	0.19	2.29	25	128	4.41	33	7.62	0.19	0.06	0.04
56	Java	D.OODM_L4_st5	1.79	6	0.18	1.84	30	101	5.31	32	9.53	0.18	0.02	0.01

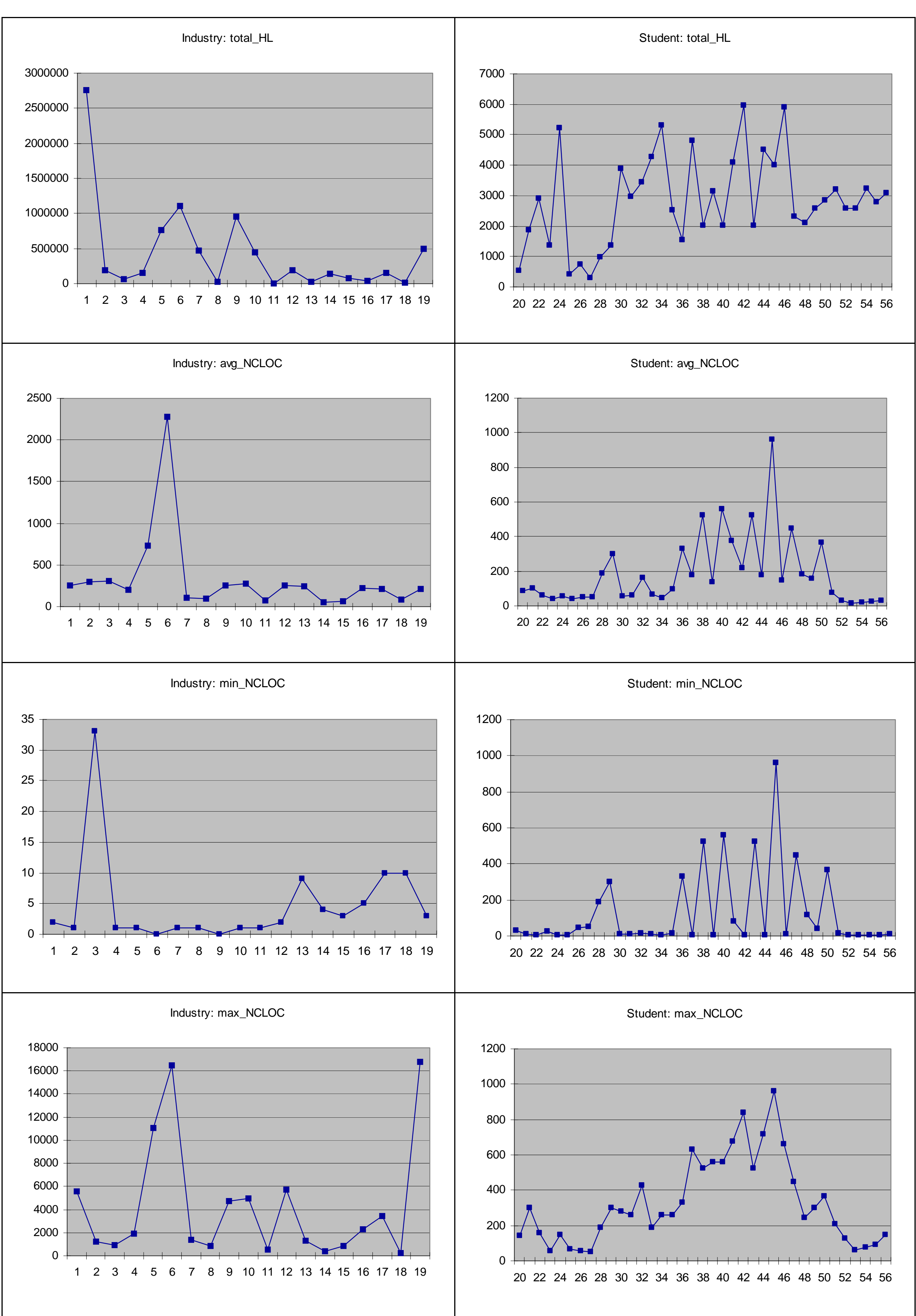
A.2.5 Object oriented aspects

pid	lan	name	total_WMC	avg_WMC	total_DIT	avg_DIT	max_DIT	total_NOC	avg_NOC	max_NOC	total_CBO	avg_CBO	max_CBO	total_FI/FO	avg_FI/FO	max_FI	max_FO	total_IF	avg_IF	max_IF
1	C	GIMP	13024	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	C	Miranda	680	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	C	NullWebmail	359	119	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	C	Panda	517	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	C	SDCC	2520	504	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	C	VIM	472	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	C++	7-Zip	3815	6	219	0	4	323	0	71	3530	5	124	1765	2	124	53	815041	1358	404496
8	C++	DiskCleaner	144	5	7	0	1	7	0	4	42	1	6	21	0	6	6	0	0	0
9	C++	Emule	6837	9	250	0	6	213	0	16	4734	6	122	2367	3	122	38	783824	1139	492804
10	C++	FileZilla	4674	14	114	0	2	91	0	21	1624	5	64	812	2	64	37	37098	116	9216
11	C++	HexView	59	1	8	0	1	8	0	2	54	1	8	27	0	3	8	0	0	0
12	C++	Notepad++	1434	7	101	0	3	64	0	20	952	5	43	476	2	43	39	37365	198	11664
13	Java	BonForum	180	3	13	0	1	13	0	4	154	3	15	77	1	15	11	2850	60	1936
14	Java	FitNesse	3237	5	690	1	6	466	0	69	4120	7	312	2060	3	312	15	2808403	4833	1703025
15	Java	HTML_Unit	2164	6	661	2	7	214	0	46	2348	7	184	1174	3	184	30	6775496	21441	6285049
16	Java	NekoHTML	362	3	37	0	3	27	0	6	500	4	30	250	2	30	25	11699	111	9216
17	Java	Jsettlers	1674	7	199	0	2	155	0	70	1582	7	110	791	3	110	72	321570	1468	82944
18	Java	JSS	100	1	17	0	1	17	0	6	190	3	14	95	1	10	8	2112	39	1600
19	Java	CS.PPS	4220	8	483	0	3	298	0	175	3144	6	222	1572	3	222	58	5042803	10249	4120900
20	Java	A.OOPJ_L1_st1	16	2	0	0	0	0	0	0	10	1	3	5	0	1	3	4	0	4
21	Java	A.OOPJ_L2_st1	34	3	0	0	0	0	0	0	22	2	4	11	1	3	4	9	0	4
22	Java	A.OOPJ_L3_st1	82	4	11	0	2	8	0	4	84	4	12	42	2	10	12	428	25	256
23	Java	A.OOPJ_L4_st1	31	1	6	0	1	13	0	3	84	3	11	42	1	4	10	378	17	144
24	Java	A.OOPJ_L5_st1	98	2	13	0	2	15	0	5	192	4	13	96	2	13	13	1013	22	576
25	C	B.OS_L1_st1	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	C	B.OS_L2_st1	6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	C	B.OS_L3_st1	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	C	B.OS_L4_st1	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	C	B.OS_L5_st1	6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	C++	C.PL_L1_st1	56	5	0	0	0	0	0	0	28	2	7	14	1	4	7	19	1	9
31	C	C.PL_L1_st2	34	34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	C	C.PL_L1_st3	29	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	C++	C.PL_L1_st4	62	6	0	0	0	0	0	0	24	2	5	12	1	3	5	35	3	16
34	C++	C.PL_L1_st5	78	7	0	0	0	0	0	0	28	2	7	14	1	5	4	124	12	100
35	C	C.PL_L1_st6	44	44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
36	C++	C.PL_L1_st7	22	4	0	0	0	0	0	0	8	1	4	4	0	1	4	0	0	0
37	C	D.CC_L2_teacher	75	75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
38	C	D.CC_L2_st1	28	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
39	C	D.CC_L2_st2	6	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	C	D.CC_L2_st3	11	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
41	C	D.CC_L2_st4	41	41	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
42	C	D.CC_L3_teacher	94	94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
43	C	D.CC_L3_st1	28	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
44	C	D.CC_L3_st2	25	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	C	D.CC_L3_st3	11	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	C	D.CC_L3_st4	54	54	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
47	C	D.CC_L4_teacher	31	31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
48	C	D.CC_L4_st1	17	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
49	C	D.CC_L4_st2	13	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
50	Java	D.CC_L4_st3	18	4	0	0	0	0	0	0	6	1	3	3	0	1	3	0	0	0
51	Java	D.CC_L4_st4	41	3	0	0	0	0	0	0	46	4	7	23	2	5	6	216	19	64
52	Java	D.OODM_L4_st1	69	2	15	0	1	15	0	7	112	4	13	56	2	11	7	1089	40	484
53	Java	D.OODM_L4_st2	101	2	37	0	3	23	0	6	172	4	15	86	2	15	8	5712	146	2025
54	Java	D.OODM_L4_st3	87	2	19	0	1	19	0	7	150	3	12	75	1	11	7	1469	37	1024
55	Java	D.OODM_L4_st4	77	2	36	1	2	23	0	6	136	4	15	68	2	15	5	2709	82	1600
56	Java	D.OODM_L4_st5	98	4	12	0	1	12	0	4	130	5	18	65	2	14	6	6061	275	3136

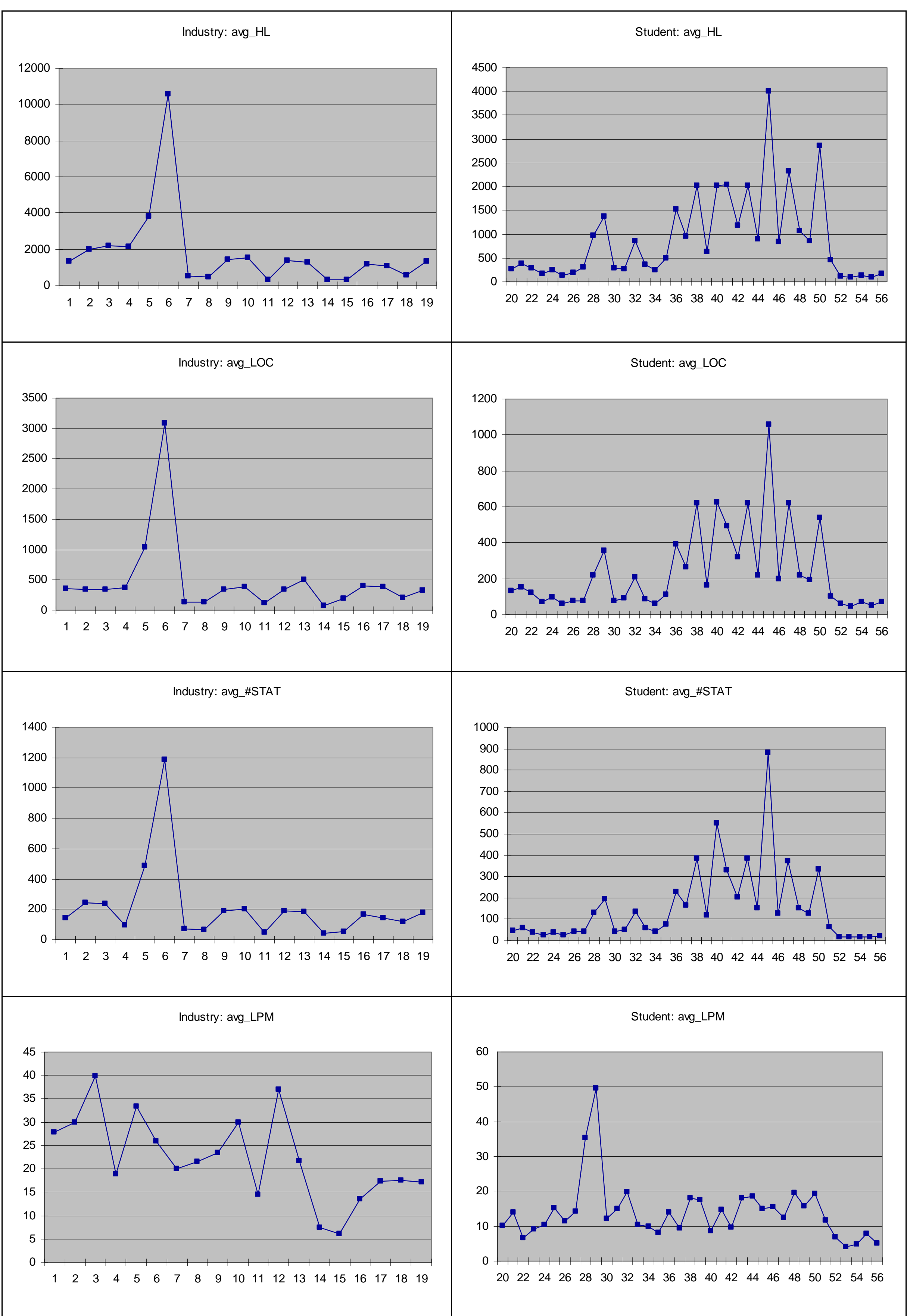
A.3 Measurement result diagrams

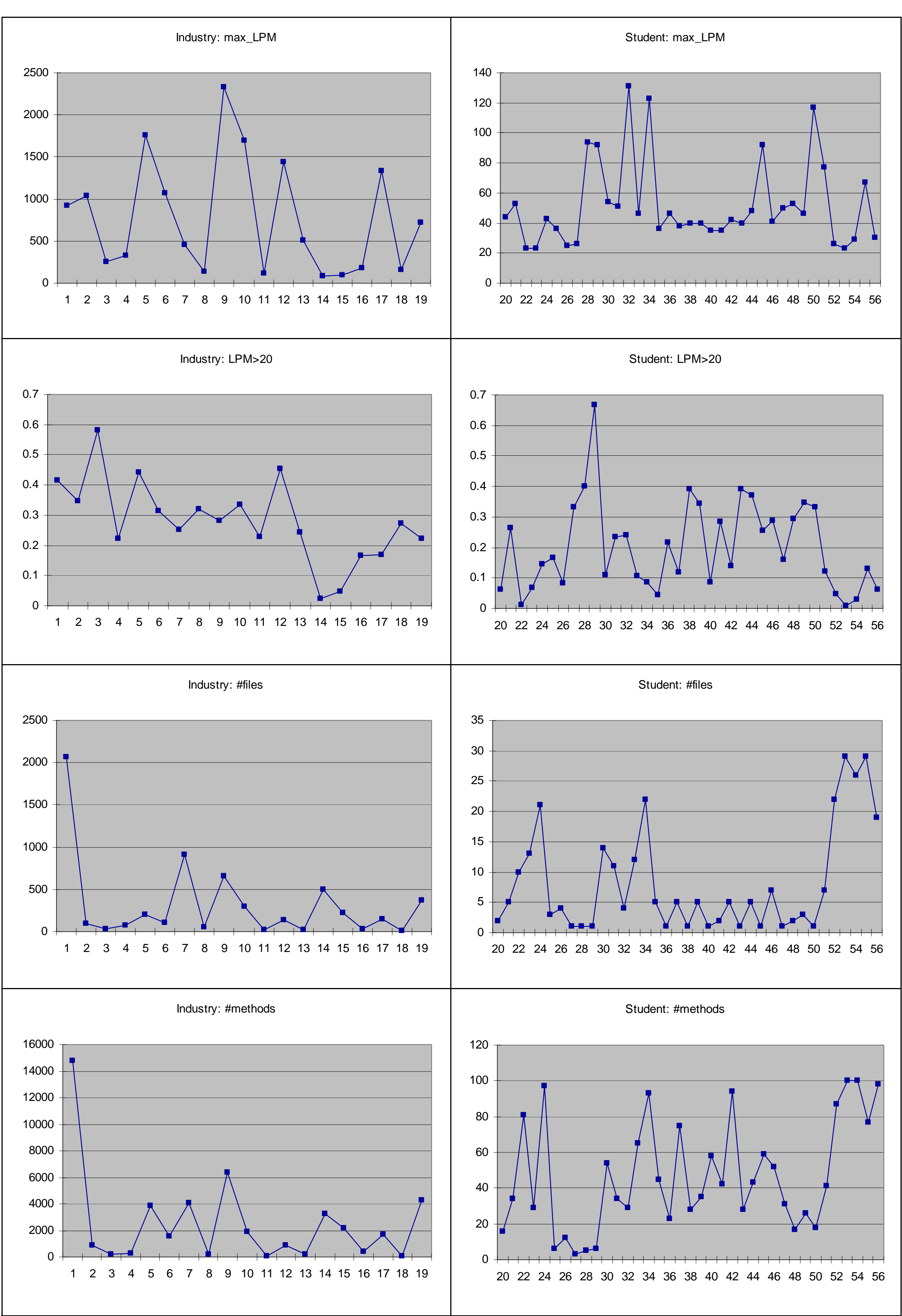
A.3.1 Line diagrams

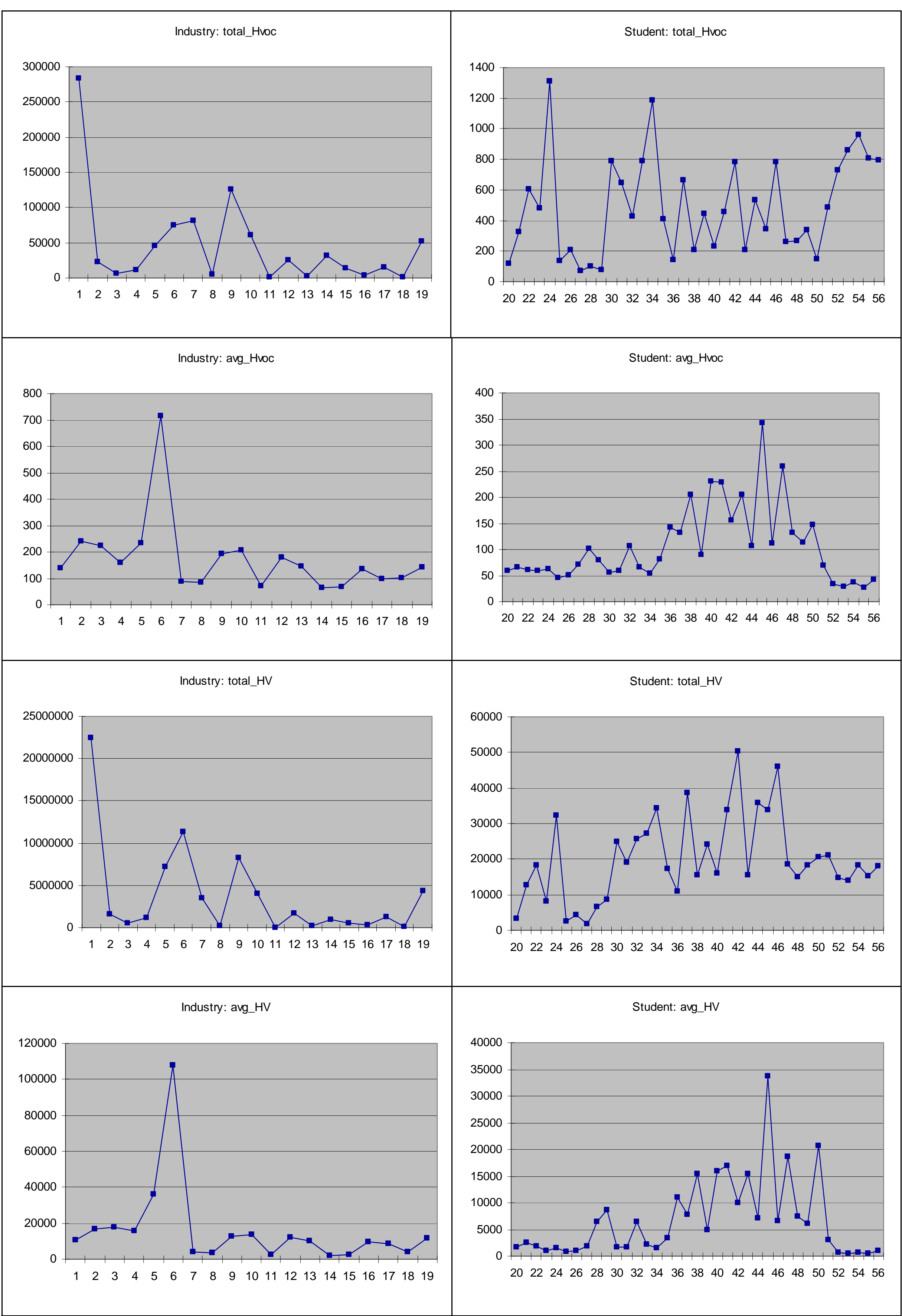


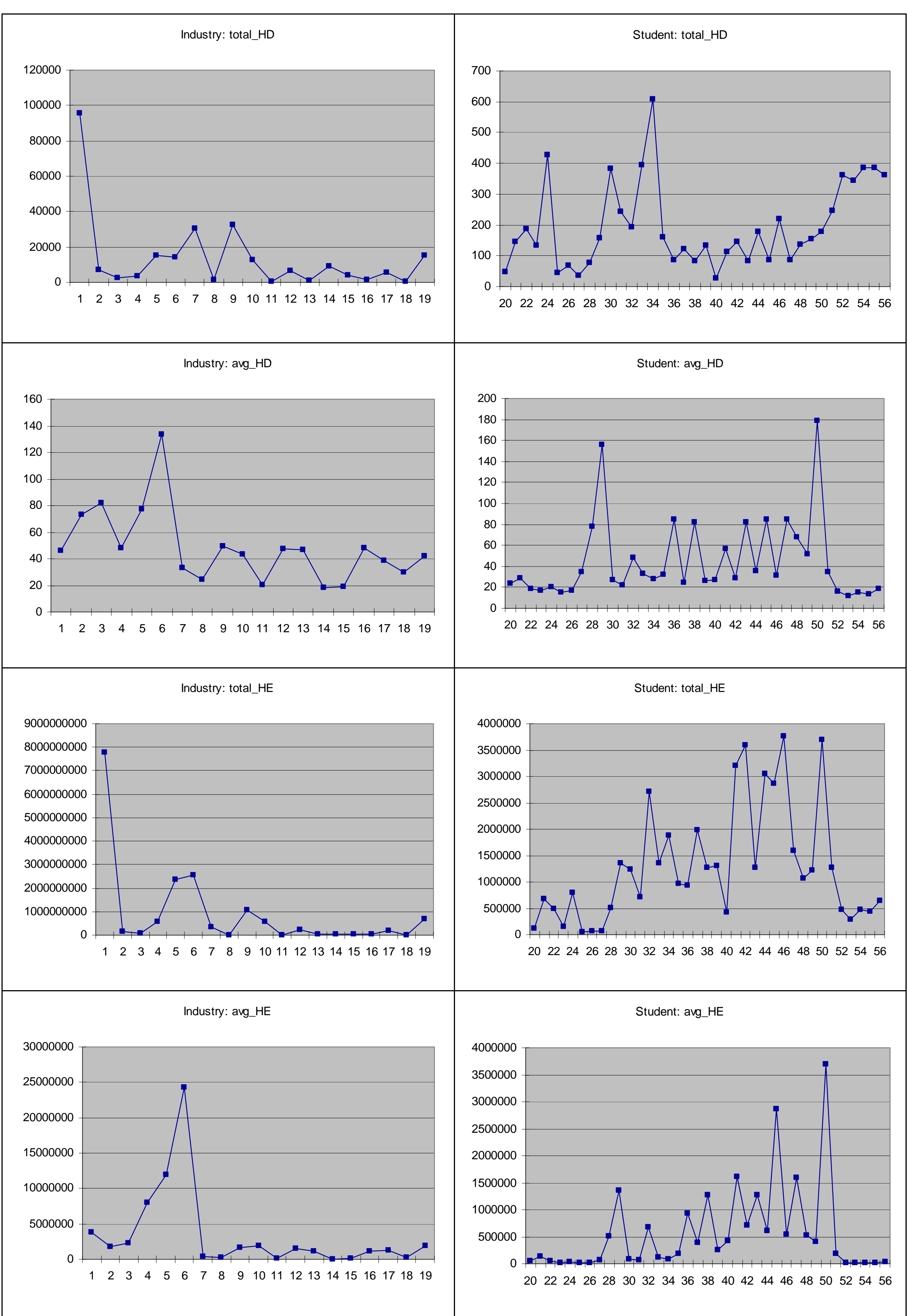


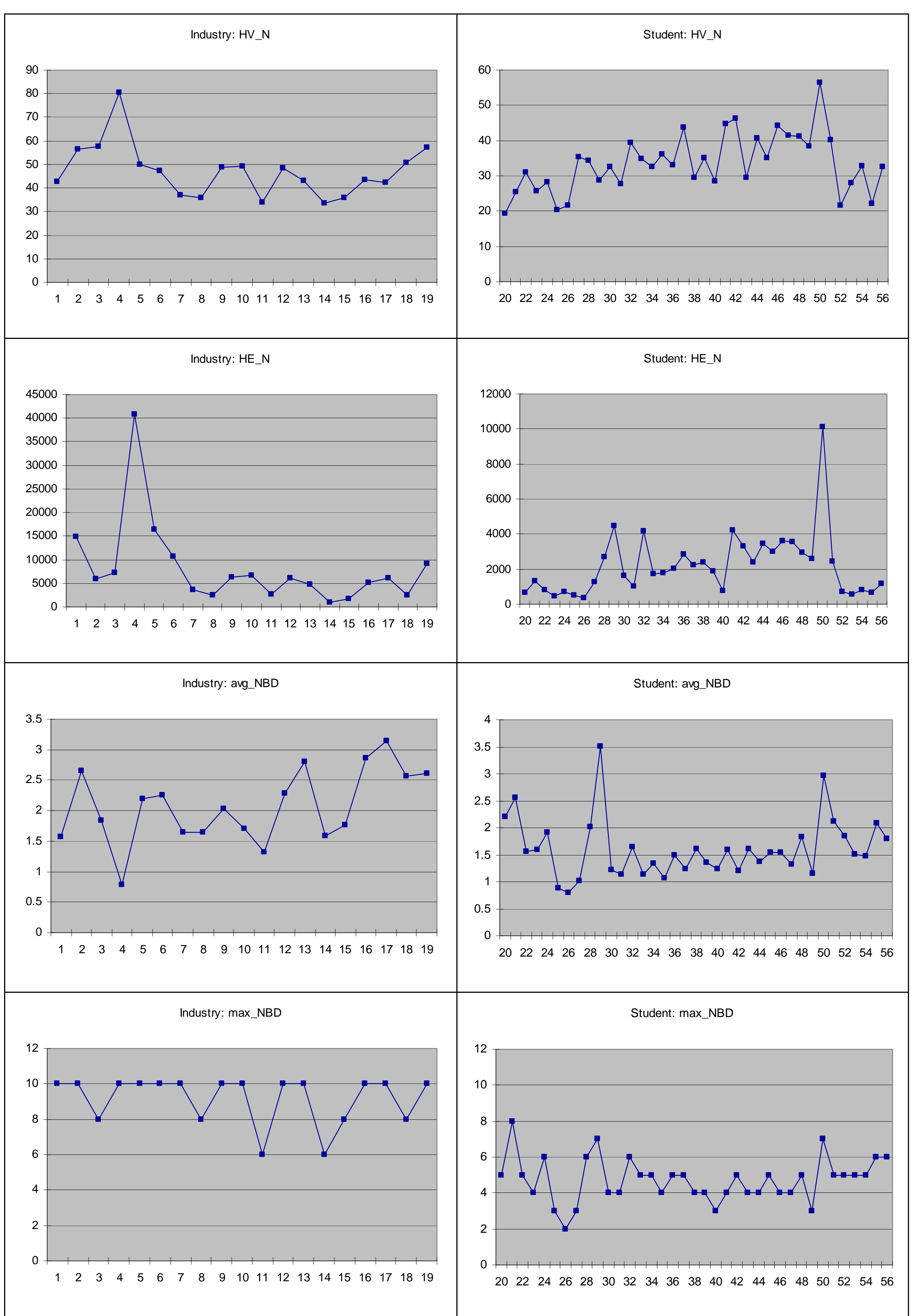
Industry: total_HL
3000000
2500000
2000000
1500000
1000000
500000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: total_HL
7000
6000
5000
4000
3000
2000
1000
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: avg_NCLOC
2500
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: avg_NCLOC
1200
1000
800
600
400
200
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: min_NCLOC
35
30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: min_NCLOC
1200
1000
800
600
400
200
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: max_NCLOC
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: max_NCLOC
1200
1000
800
600
400
200
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56

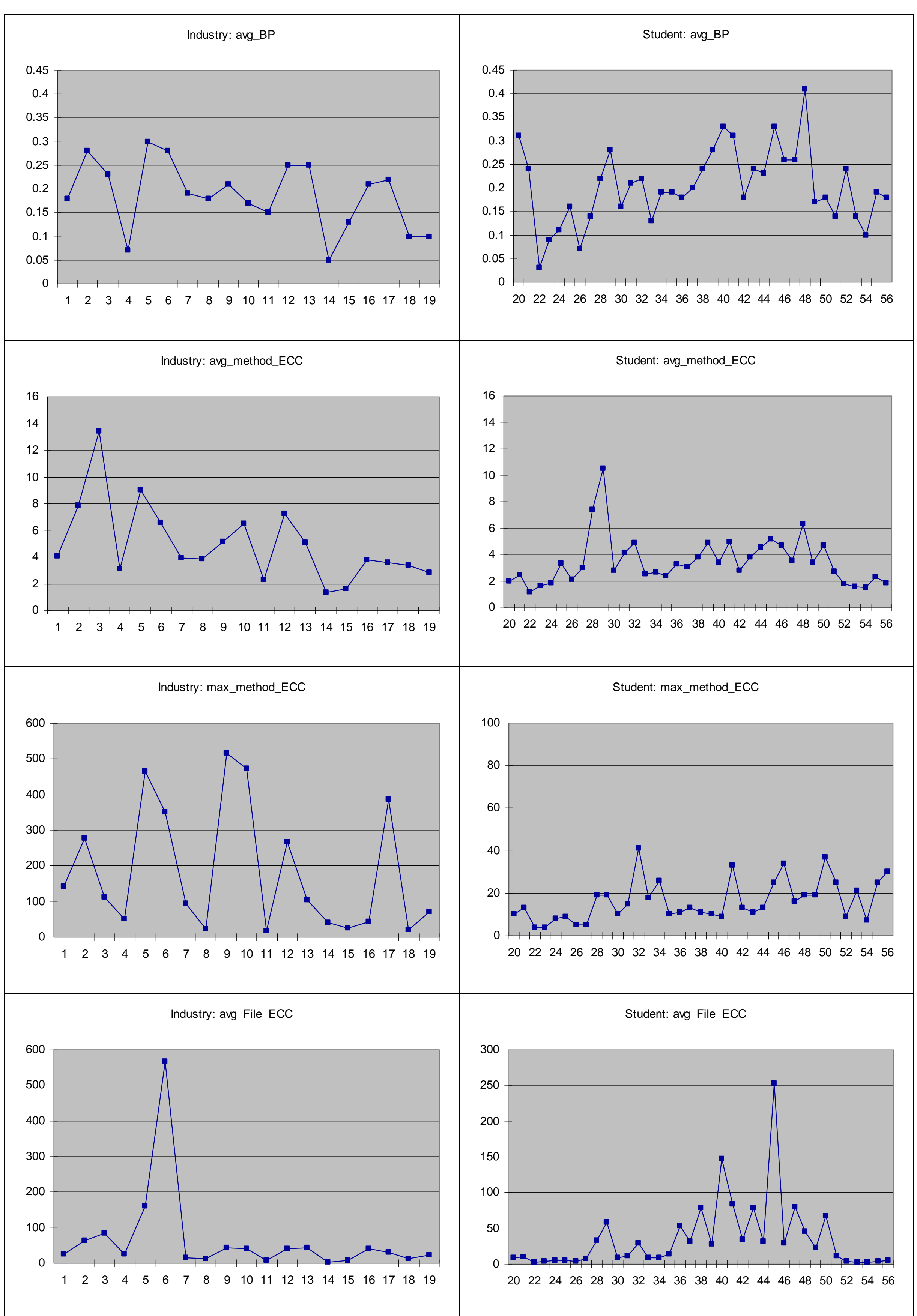

Industry: avg_HL
Student: avg_HL
Industry: avg_LOC
Student: avg_LOC
Industry: avg_#STAT
Student: avg_#STAT
Industry: avg_LPM
Student: avg_LPM

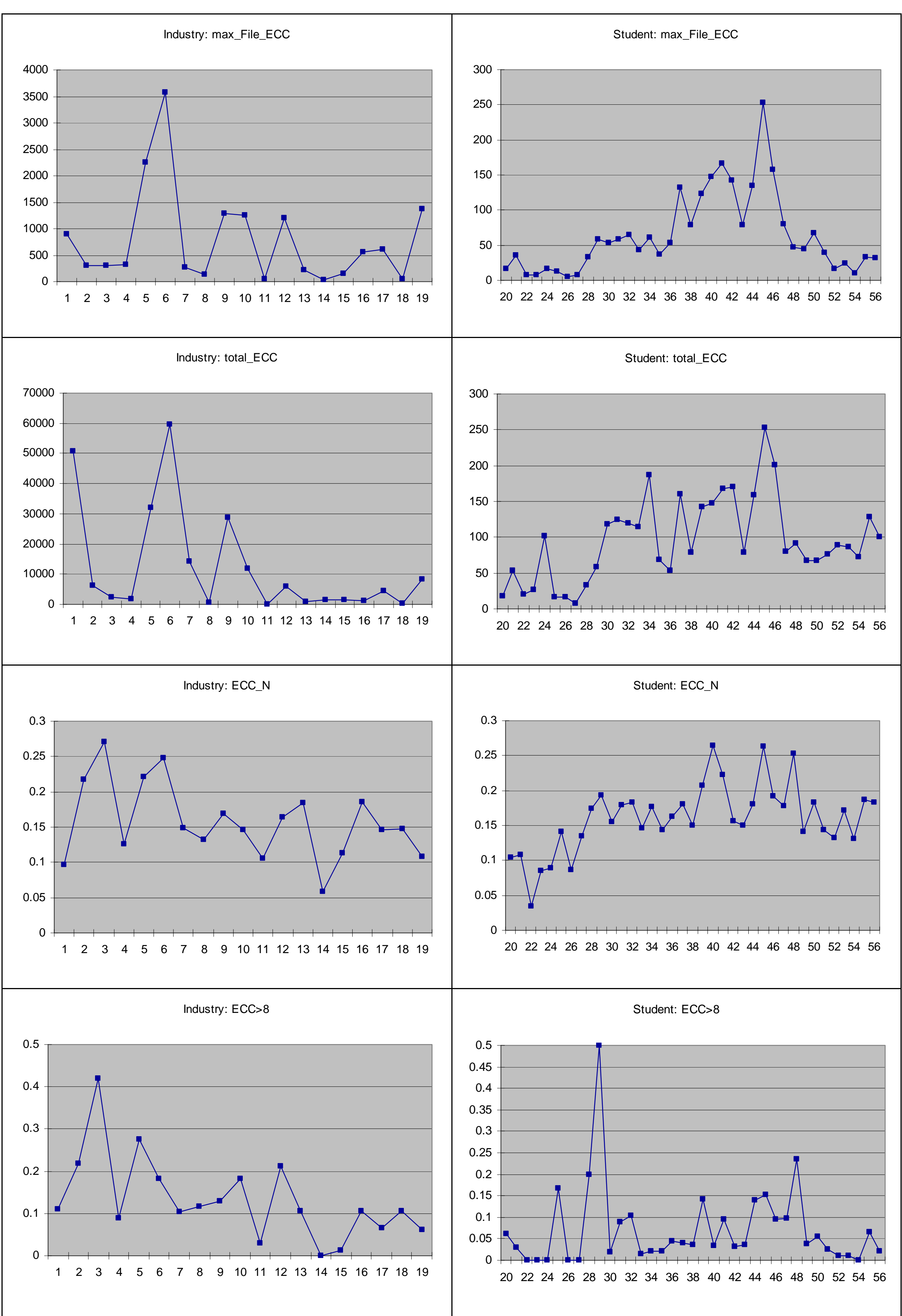

Industry: max_LPM
2500
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: max_LPM
140
120
100
80
60
40
20
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: LPM>20
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: LPM>20
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: #files
2500
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: #files
35
30
25
20
15
10
5
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: #methods
16000
14000
12000
10000
8000
6000
4000
2000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: #methods
120
100
80
60
40
20
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56

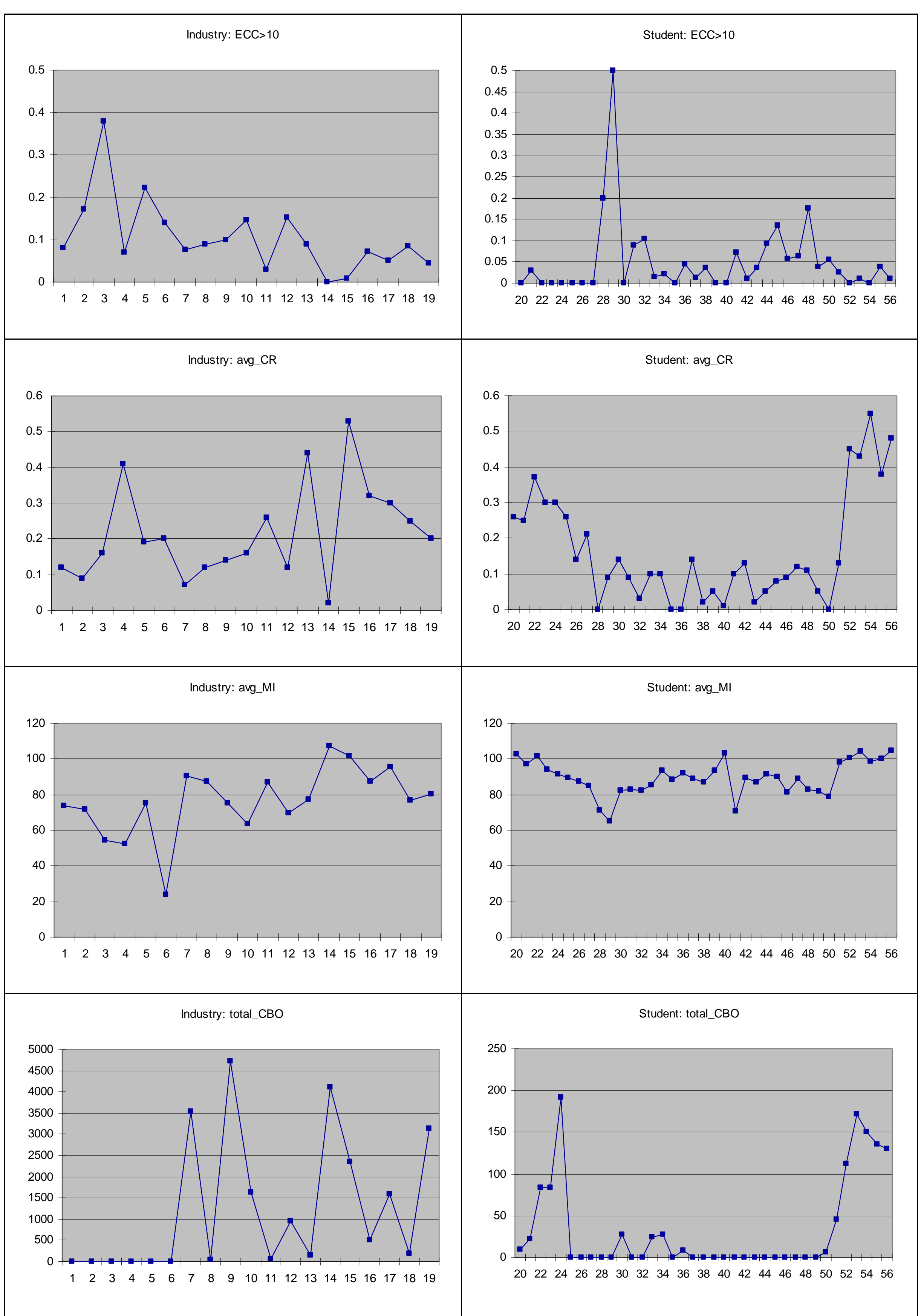

Industry: total_Hvoc
300000
250000
200000
150000
100000
50000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: total_Hvoc
1400
1200
1000
800
600
400
200
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: avg_Hvoc
800
700
600
500
400
300
200
100
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: avg_Hvoc
400
350
300
250
200
150
100
50
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: total_HV
25000000
20000000
15000000
10000000
5000000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: total_HV
60000
50000
40000
30000
20000
10000
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56
Industry: avg_HV
120000
100000
80000
60000
40000
20000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Student: avg_HV
40000
35000
30000
25000
20000
15000
10000
5000
0
20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56


Industry: total_HD
Student: total_HD
Industry: avg_HD
Student: avg_HD
Industry: total_HE
Student: total_HE
Industry: avg_HE
Student: avg_HE


Industry: HV_N
Student: HV_N
Industry: HE_N
Student: HE_N
Industry: avg_NBD
Student: avg_NBD
Industry: max_NBD
Student: max_NBD

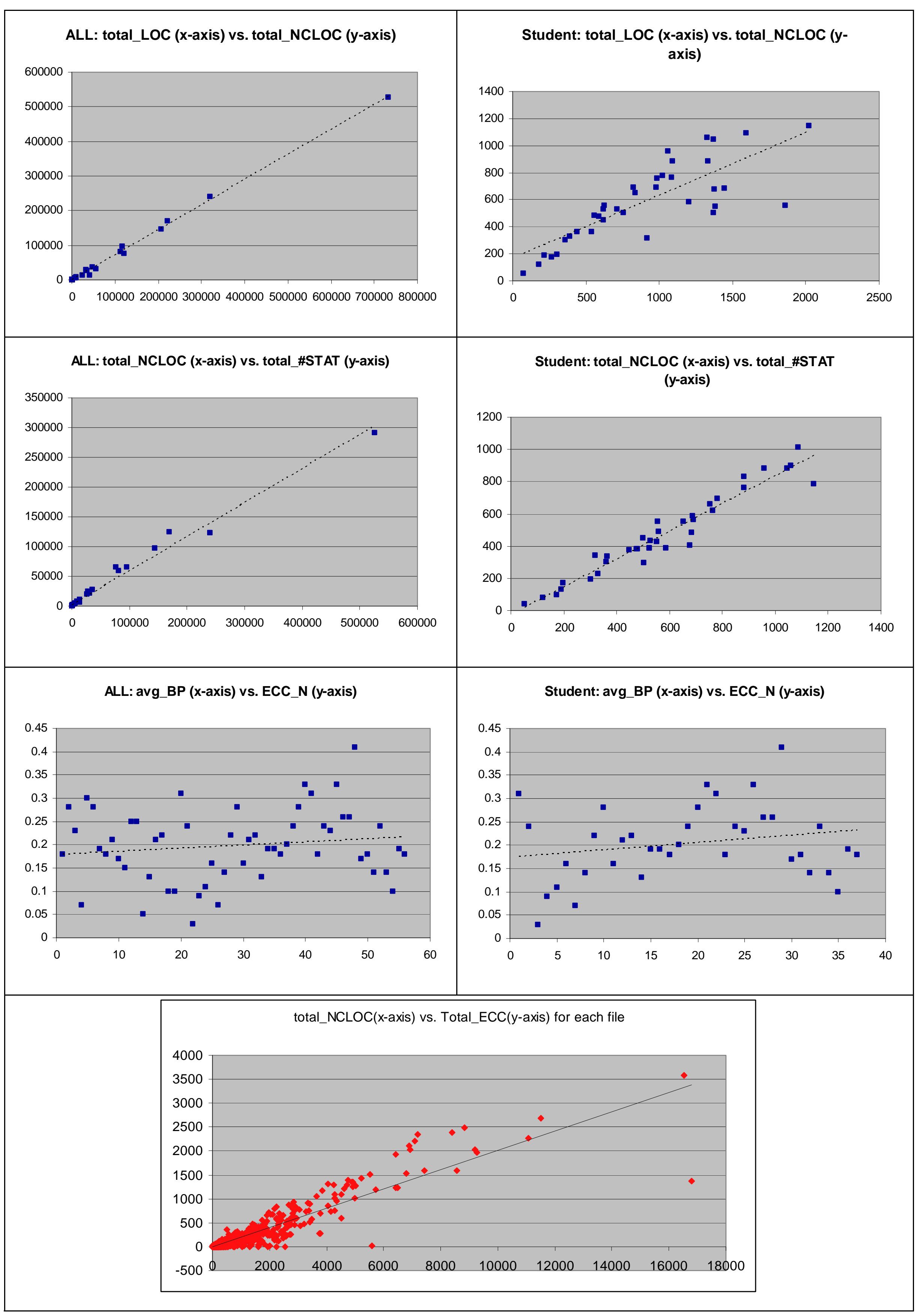

Industry: avg_BP
Student: avg_BP
Industry: avg_method_ECC
Student: avg_method_ECC
Industry: max_method_ECC
Student: max_method_ECC
Industry: avg_File_ECC
Student: avg_File_ECC


Industry: max_File_ECC
Student: max_File_ECC
Industry: total_ECC
Student: total_ECC
Industry: ECC_N
Student: ECC_N
Industry: ECC>8
Student: ECC>8



Industry: ECC>10
Student: ECC>10
Industry: avg_CR
Student: avg_CR
Industry: avg_MI
Student: avg_MI
Industry: total_CBO
Student: total_CBO

A.3.2 Scatter plots



A.4 Program profiles

A.4.1 Metric information

Competitive metrics

• Reversed MI • Nested Block Depth • Normalized ECC • Branch Percentage	• Method ECC > 10 • Lines per method > 20 • Average File ECC • Normalized Halstead Volume

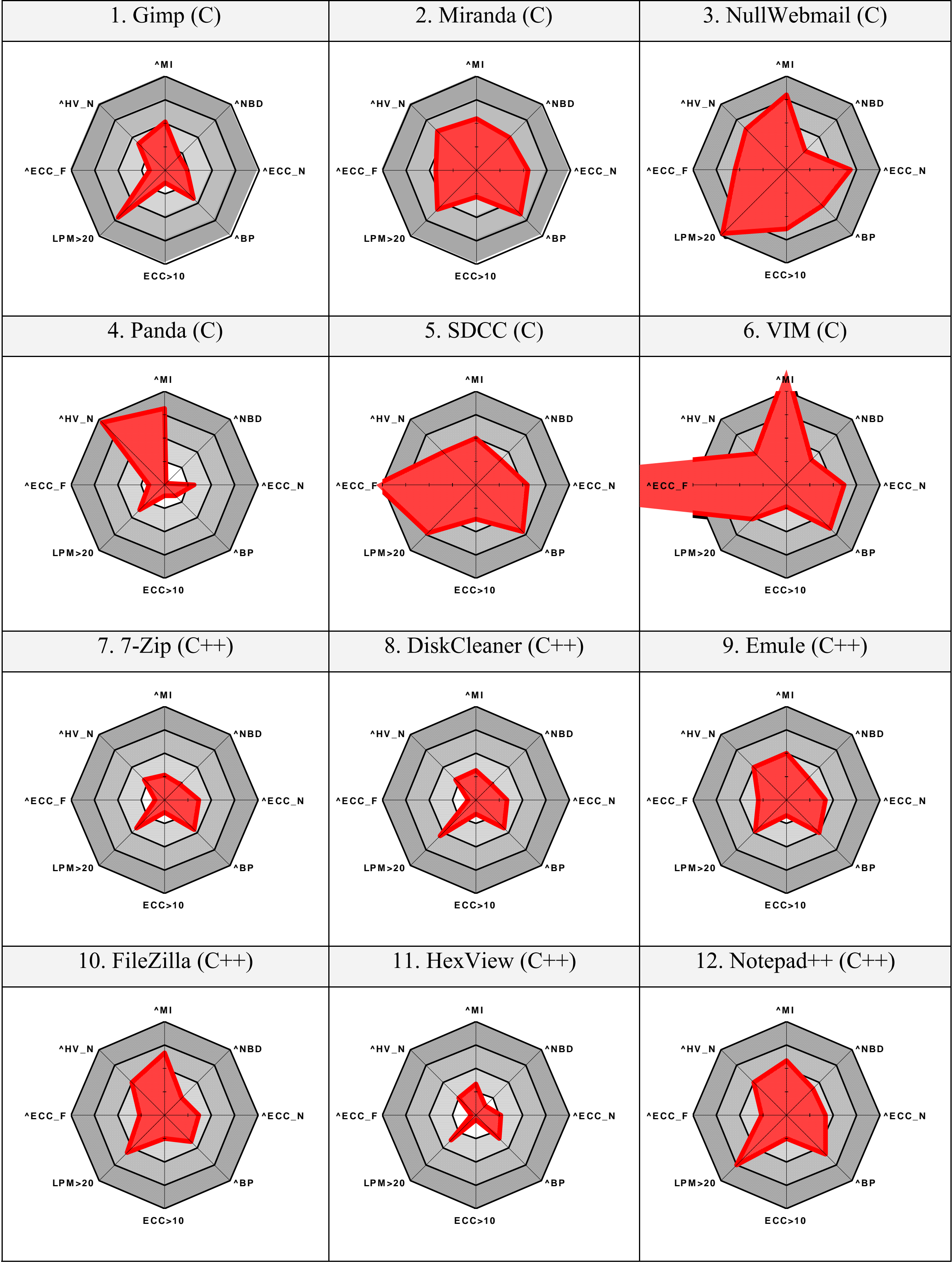
Descriptive metrics

• Program length • Comment rate • Maintainability Index • Average file complexity • Average method complexity • Maximal coupling	• Code distribution aspects o Number of files & methods o Average Lines per file & method o Maximal lines per file & method o Full code distribution on file scope

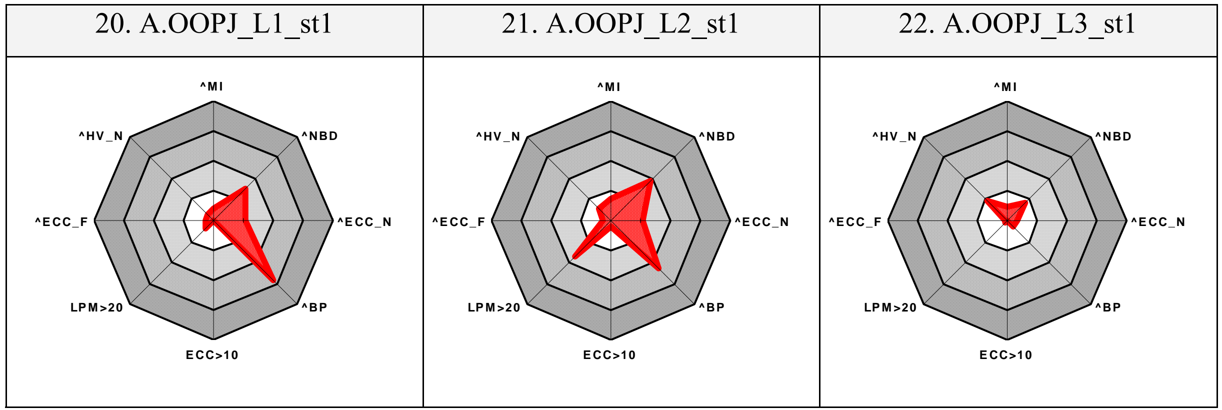
Metric adjustment

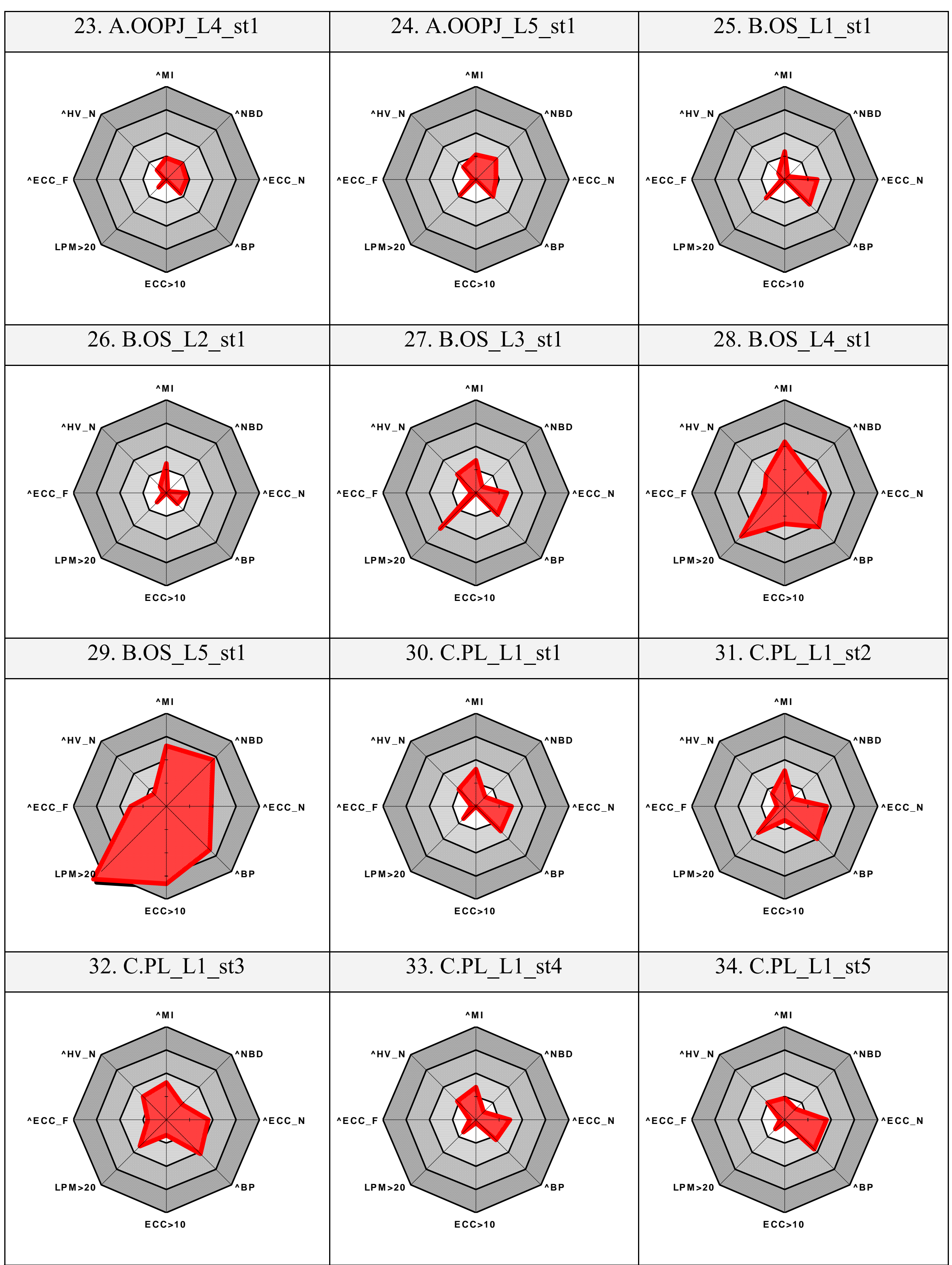
Metric	Computation
^MI	(1 - (MI - 40)/70) * 0.6
^ECC_F	(avg_File_ECC/150)*0.6
^HV_N	((HV_N-15)/70)*0.6
^NBD	((avg_NDB-0.7)/4)*0.6
^BP	(avg_BP*1.4)
^ECC_N	(ECC_N*1.5)

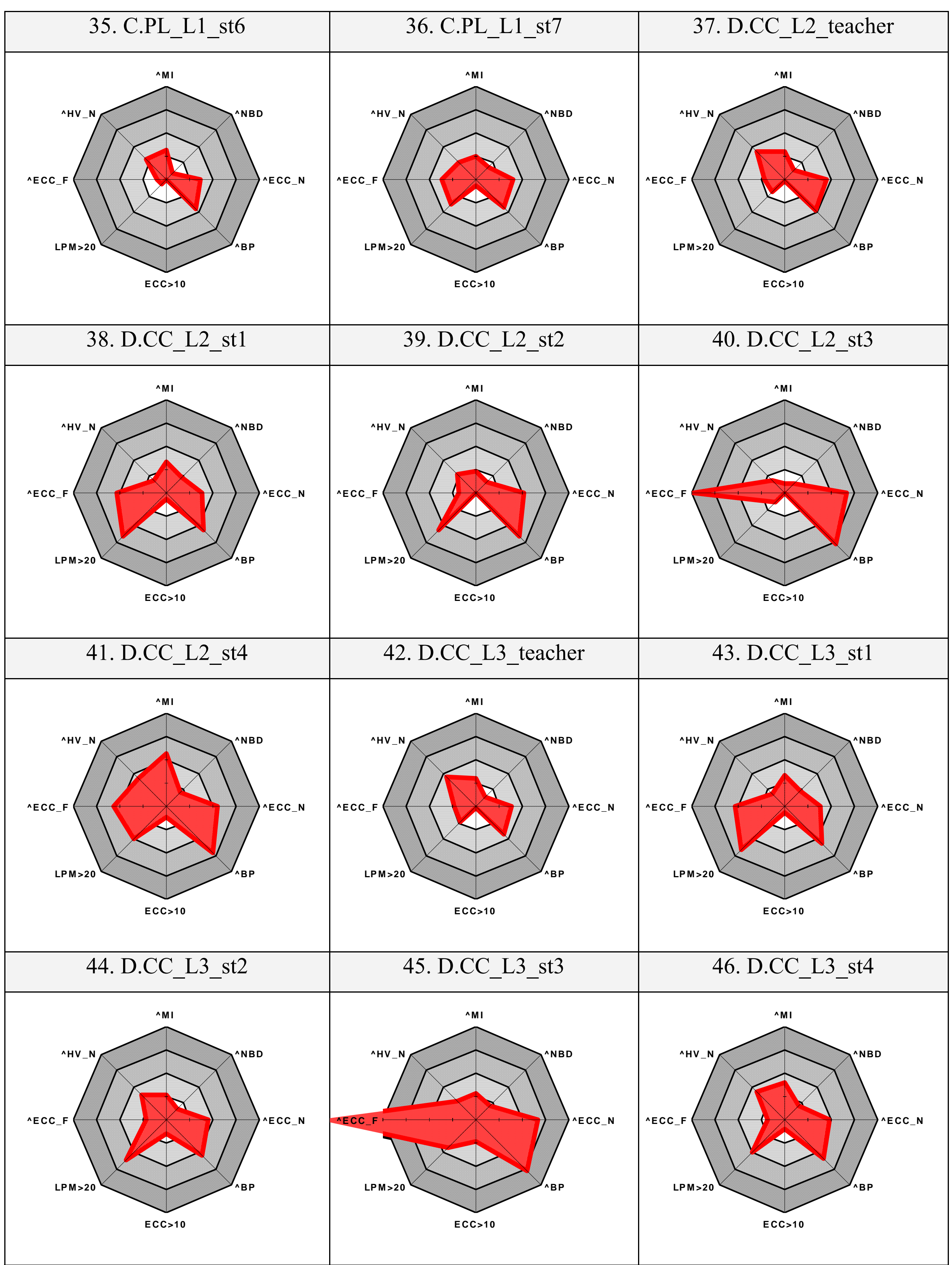
A.4.2 Kiviat diagrams (competitive measures)

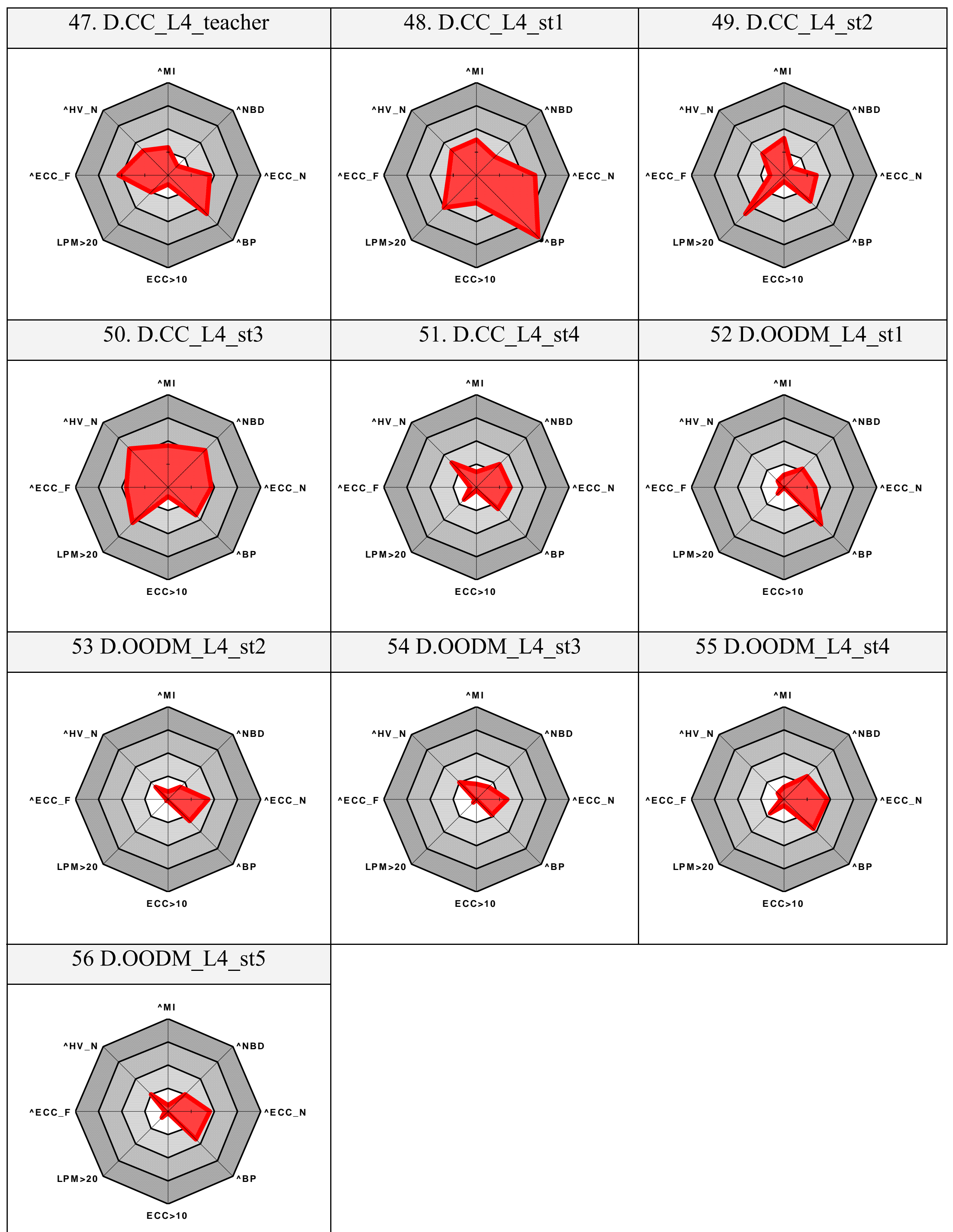


13. BonForum (Java)	14. FitNesse (Java)	15. HTML_Unit (Java)
^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N	^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N	^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N
16. NekoHTML (Java)	17. Jsettlers (Java)	18. JSS (Java)
^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N	^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N	^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N
19. CS.PPS (Java)		
^MI ^NBD ^ECC_N ^BP ECC>10 LPM>20 ^ECC_F ^HV_N		

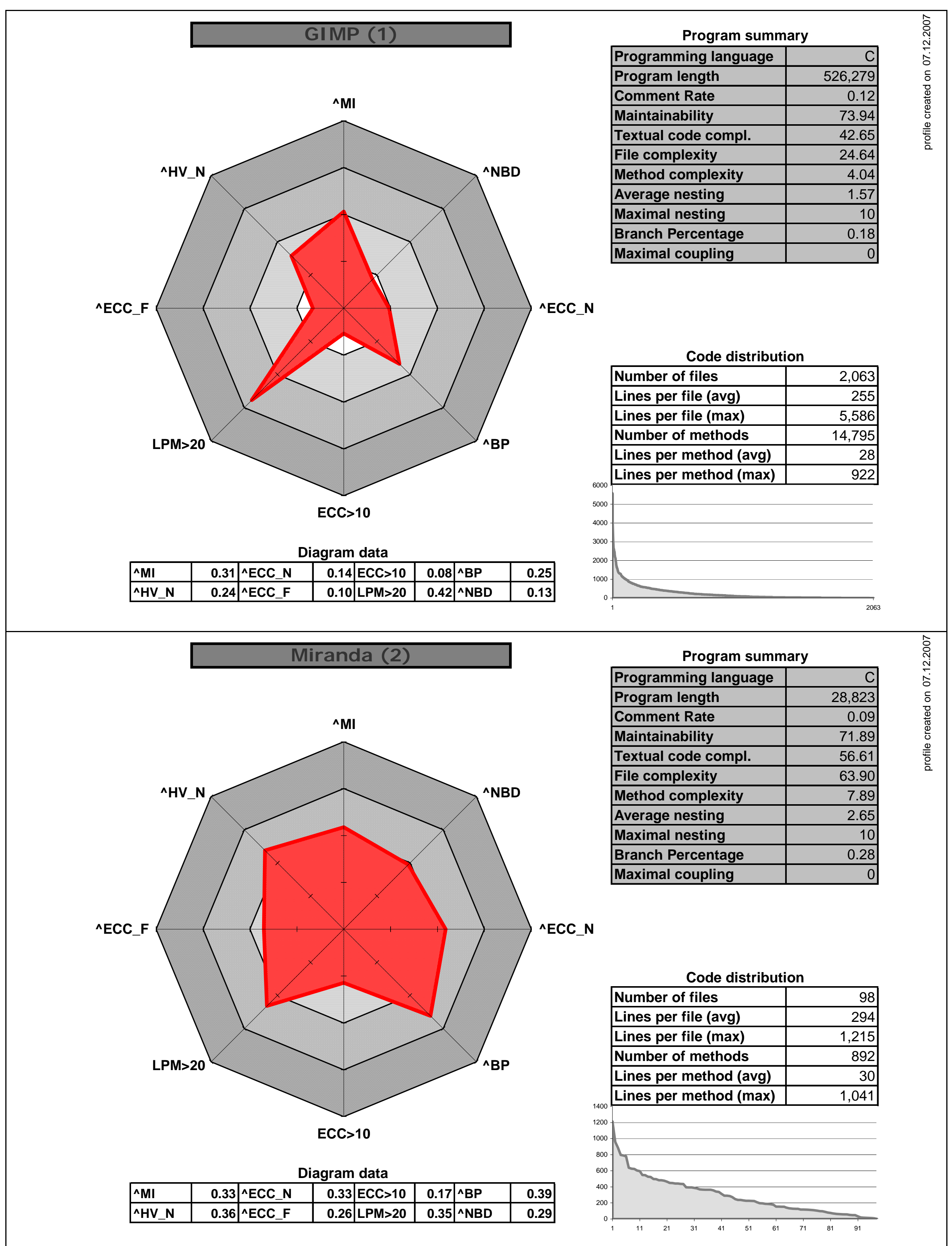


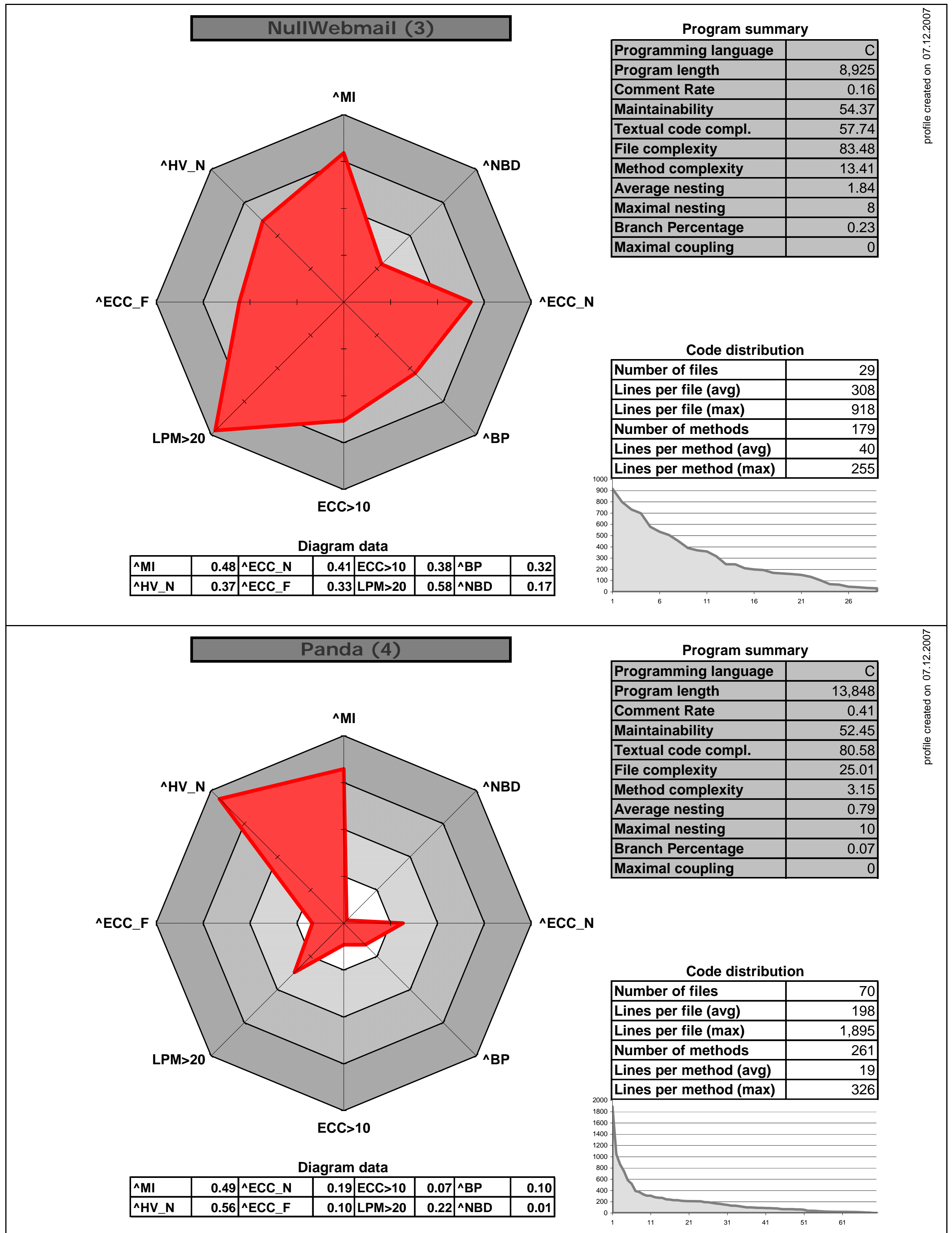

23. A.OOPJ_L4_st1
24. A.OOPJ_L5_st1
25. B.OS_L1_st1
26. B.OS_L2_st1
27. B.OS_L3_st1
28. B.OS_L4_st1
29. B.OS_L5_st1
30. C.PL_L1_st1
31. C.PL_L1_st2
32. C.PL_L1_st3
33. C.PL_L1_st4
34. C.PL_L1_st5
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N


35. C.PL_L1_st6
36. C.PL_L1_st7
37. D.CC_L2_teacher
38. D.CC_L2_st1
39. D.CC_L2_st2
40. D.CC_L2_st3
41. D.CC_L2_st4
42. D.CC_L3_teacher
43. D.CC_L3_st1
44. D.CC_L3_st2
45. D.CC_L3_st3
46. D.CC_L3_st4
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N


47. D.CC_L4_teacher
48. D.CC_L4_st1
49. D.CC_L4_st2
50. D.CC_L4_st3
51. D.CC_L4_st4
52 D.OODM_L4_st1
53 D.OODM_L4_st2
54 D.OODM_L4_st3
55 D.OODM_L4_st4
56 D.OODM_L4_st5
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N

A.4.3 Program profiles (full view)





Program summary

Programming language	C
Program length	8,925
Comment Rate	0.16
Maintainability	54.37
Textual code compl.	57.74
File complexity	83.48
Method complexity	13.41
Average nesting	1.84
Maximal nesting	8
Branch Percentage	0.23
Maximal coupling	0

Code distribution

Number of files	29
Lines per file (avg)	308
Lines per file (max)	918
Number of methods	179
Lines per method (avg)	40
Lines per method (max)	255

Diagram data

^MI	0.48	^ECC_N	0.41	ECC>10	0.38	^BP	0.32
^HV_N	0.37	^ECC_F	0.33	LPM>20	0.58	^NBD	0.17

Program summary

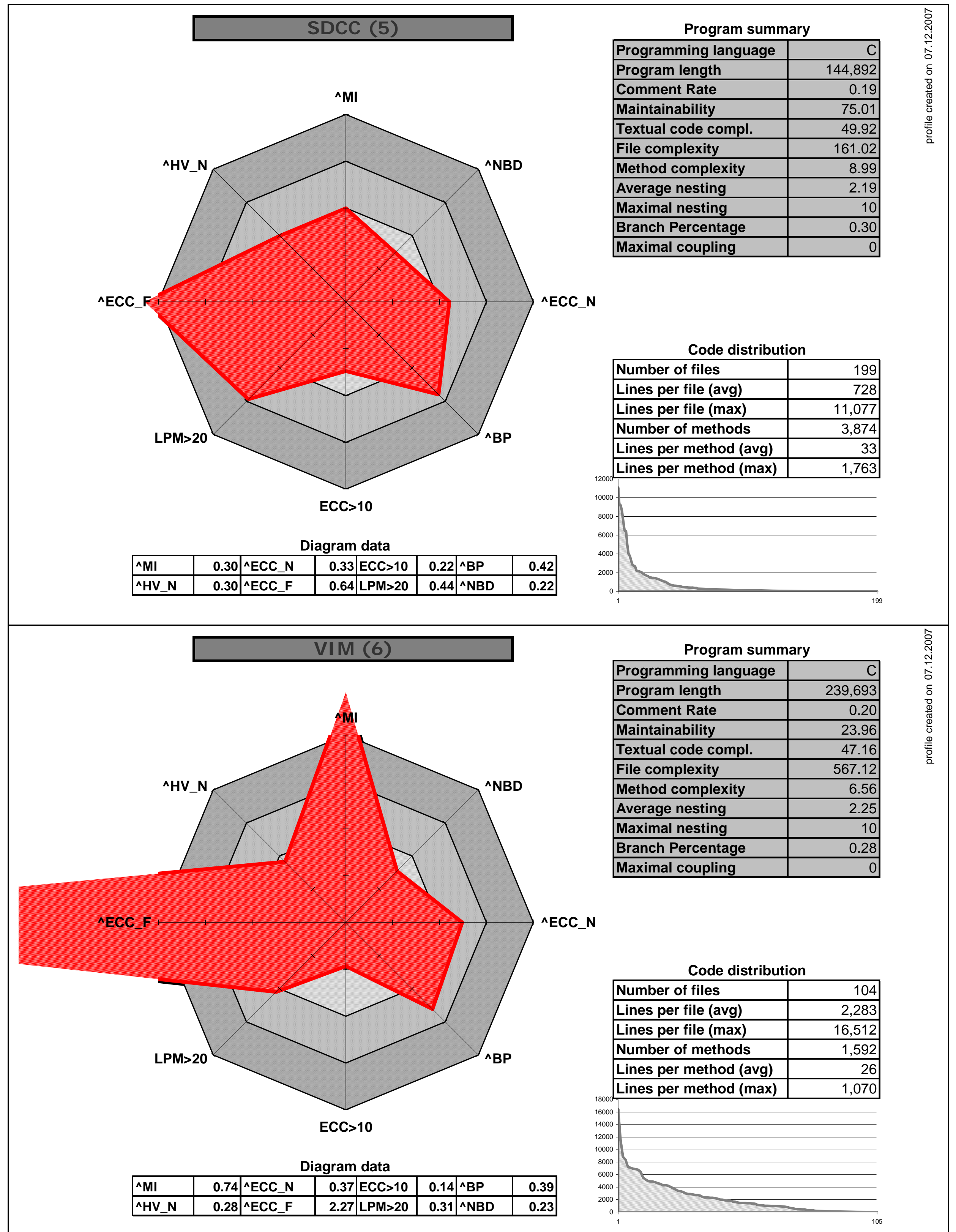
Programming language	C
Program length	13,848
Comment Rate	0.41
Maintainability	52.45
Textual code compl.	80.58
File complexity	25.01
Method complexity	3.15
Average nesting	0.79
Maximal nesting	10
Branch Percentage	0.07
Maximal coupling	0

Code distribution

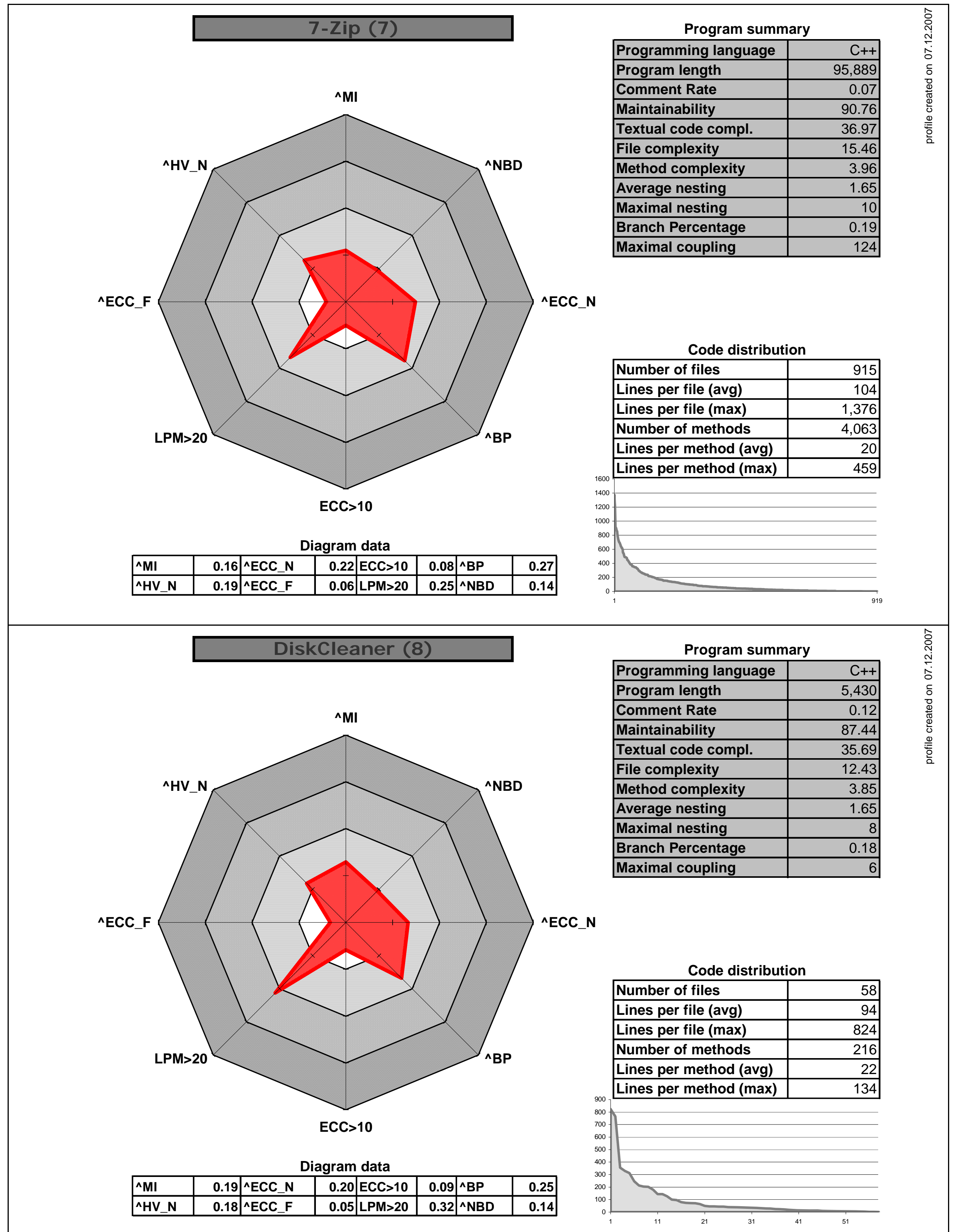
Number of files	70
Lines per file (avg)	198
Lines per file (max)	1,895
Number of methods	261
Lines per method (avg)	19
Lines per method (max)	326

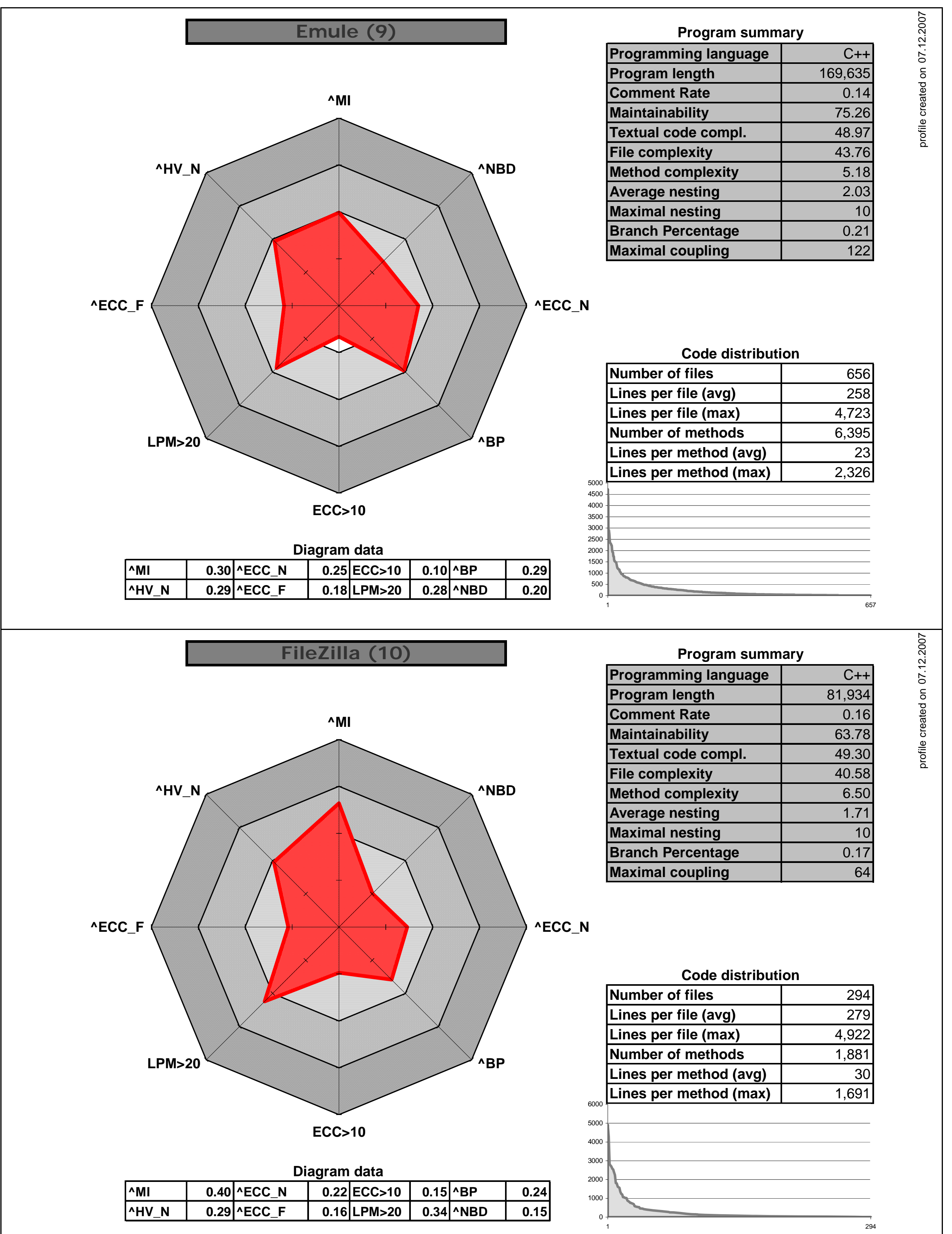
Diagram data

^MI	0.49	^ECC_N	0.19	ECC>10	0.07	^BP	0.10
^HV_N	0.56	^ECC_F	0.10	LPM>20	0.22	^NBD	0.01



SDCC (5)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.30 ^ECC_N 0.33 ECC>10 0.22 ^BP 0.42
^HV_N 0.30 ^ECC_F 0.64 LPM>20 0.44 ^NBD 0.22
Program summary
Programming language C
Program length 144,892
Comment Rate 0.19
Maintainability 75.01
Textual code compl. 49.92
File complexity 161.02
Method complexity 8.99
Average nesting 2.19
Maximal nesting 10
Branch Percentage 0.30
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 199
Lines per file (avg) 728
Lines per file (max) 11,077
Number of methods 3,874
Lines per method (avg) 33
Lines per method (max) 1,763
VIM (6)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.74 ^ECC_N 0.37 ECC>10 0.14 ^BP 0.39
^HV_N 0.28 ^ECC_F 2.27 LPM>20 0.31 ^NBD 0.23
Program summary
Programming language C
Program length 239,693
Comment Rate 0.20
Maintainability 23.96
Textual code compl. 47.16
File complexity 567.12
Method complexity 6.56
Average nesting 2.25
Maximal nesting 10
Branch Percentage 0.28
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 104
Lines per file (avg) 2,283
Lines per file (max) 16,512
Number of methods 1,592
Lines per method (avg) 26
Lines per method (max) 1,070


7-Zip (7)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.16 ^ECC_N 0.22 ECC>10 0.08 ^BP 0.27
^HV_N 0.19 ^ECC_F 0.06 LPM>20 0.25 ^NBD 0.14
Program summary
Programming language C++
Program length 95,889
Comment Rate 0.07
Maintainability 90.76
Textual code compl. 36.97
File complexity 15.46
Method complexity 3.96
Average nesting 1.65
Maximal nesting 10
Branch Percentage 0.19
Maximal coupling 124
profile created on 07.12.2007
Code distribution
Number of files 915
Lines per file (avg) 104
Lines per file (max) 1,376
Number of methods 4,063
Lines per method (avg) 20
Lines per method (max) 459
DiskCleaner (8)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.19 ^ECC_N 0.20 ECC>10 0.09 ^BP 0.25
^HV_N 0.18 ^ECC_F 0.05 LPM>20 0.32 ^NBD 0.14
Program summary
Programming language C++
Program length 5,430
Comment Rate 0.12
Maintainability 87.44
Textual code compl. 35.69
File complexity 12.43
Method complexity 3.85
Average nesting 1.65
Maximal nesting 8
Branch Percentage 0.18
Maximal coupling 6
profile created on 07.12.2007
Code distribution
Number of files 58
Lines per file (avg) 94
Lines per file (max) 824
Number of methods 216
Lines per method (avg) 22
Lines per method (max) 134



Emule (9)

Program summary

Programming language	C++
Program length	169,635
Comment Rate	0.14
Maintainability	75.26
Textual code compl.	48.97
File complexity	43.76
Method complexity	5.18
Average nesting	2.03
Maximal nesting	10
Branch Percentage	0.21
Maximal coupling	122

Code distribution

Number of files	656
Lines per file (avg)	258
Lines per file (max)	4,723
Number of methods	6,395
Lines per method (avg)	23
Lines per method (max)	2,326

Diagram data

^MI	0.30	^ECC_N	0.25	ECC>10	0.10	^BP	0.29
^HV_N	0.29	^ECC_F	0.18	LPM>20	0.28	^NBD	0.20

FileZilla (10)

Program summary

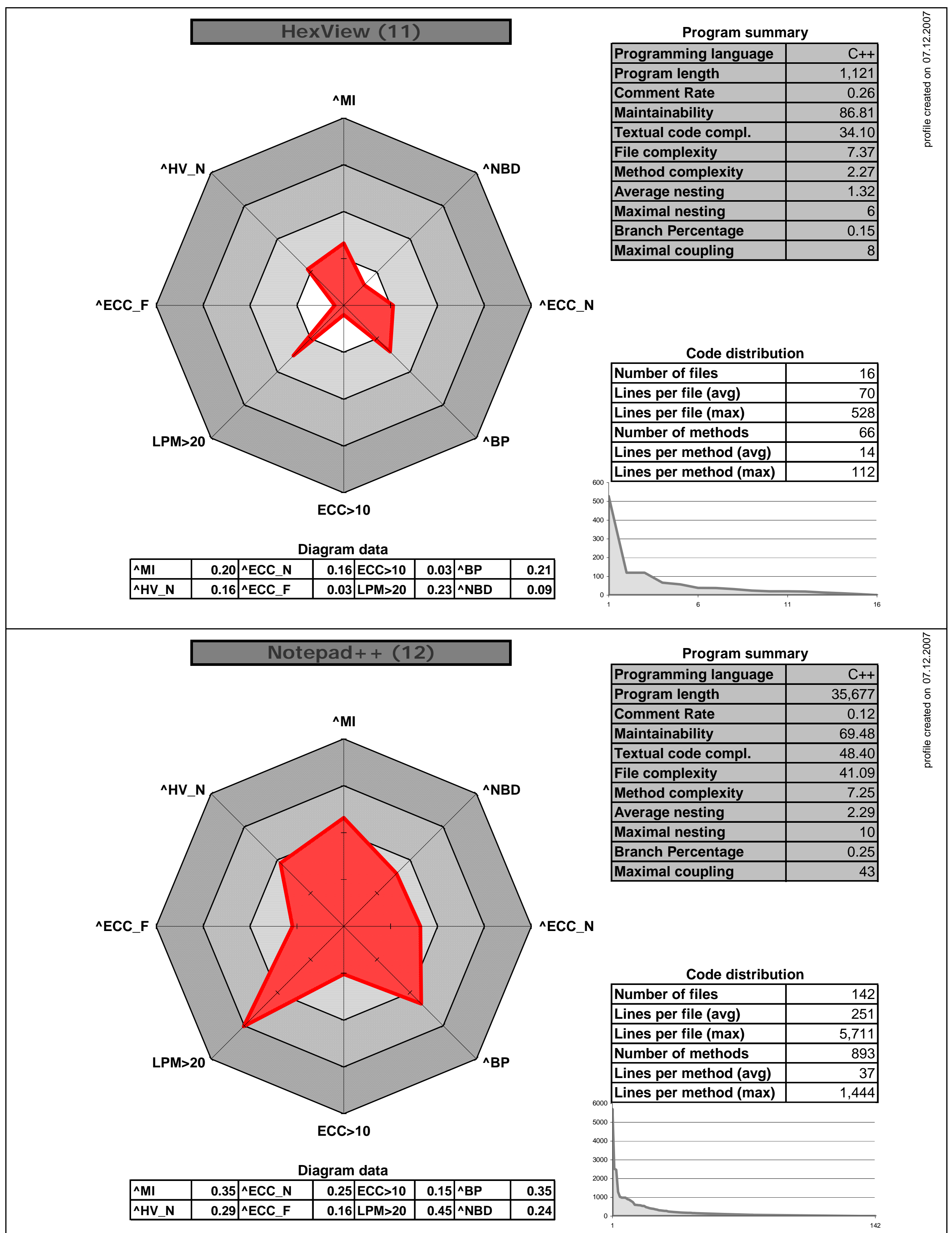
Programming language	C++
Program length	81,934
Comment Rate	0.16
Maintainability	63.78
Textual code compl.	49.30
File complexity	40.58
Method complexity	6.50
Average nesting	1.71
Maximal nesting	10
Branch Percentage	0.17
Maximal coupling	64

Code distribution

Number of files	294
Lines per file (avg)	279
Lines per file (max)	4,922
Number of methods	1,881
Lines per method (avg)	30
Lines per method (max)	1,691

Diagram data

^MI	0.40	^ECC_N	0.22	ECC>10	0.15	^BP	0.24
^HV_N	0.29	^ECC_F	0.16	LPM>20	0.34	^NBD	0.15



HexView (11)

Program summary

Programming language	C++
Program length	1,121
Comment Rate	0.26
Maintainability	86.81
Textual code compl.	34.10
File complexity	7.37
Method complexity	2.27
Average nesting	1.32
Maximal nesting	6
Branch Percentage	0.15
Maximal coupling	8

Code distribution

Number of files	16
Lines per file (avg)	70
Lines per file (max)	528
Number of methods	66
Lines per method (avg)	14
Lines per method (max)	112

Diagram data

^MI	0.20	^ECC_N	0.16	ECC>10	0.03	^BP	0.21
^HV_N	0.16	^ECC_F	0.03	LPM>20	0.23	^NBD	0.09

Notepad++ (12)

Program summary

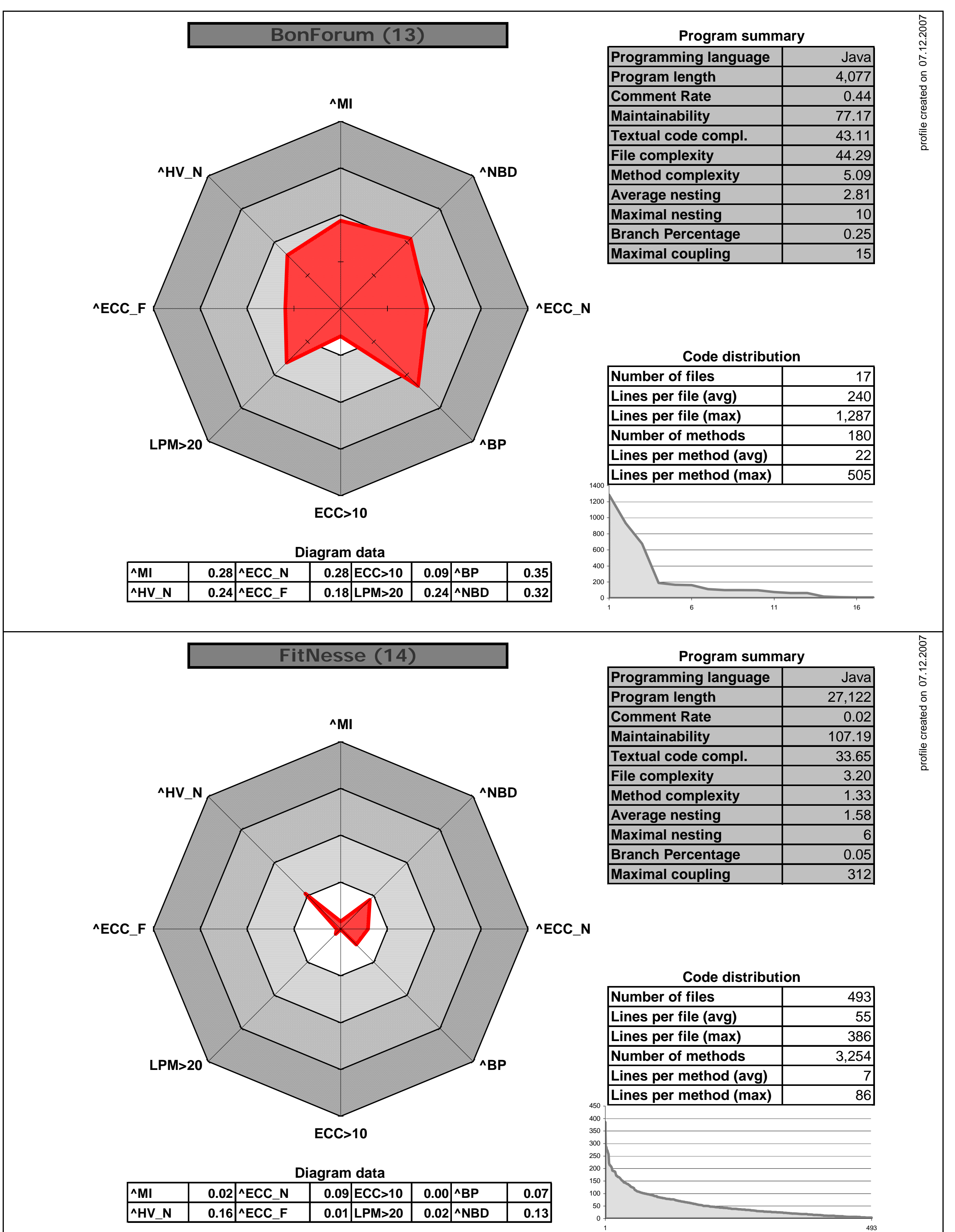
Programming language	C++
Program length	35,677
Comment Rate	0.12
Maintainability	69.48
Textual code compl.	48.40
File complexity	41.09
Method complexity	7.25
Average nesting	2.29
Maximal nesting	10
Branch Percentage	0.25
Maximal coupling	43

Code distribution

Number of files	142
Lines per file (avg)	251
Lines per file (max)	5,711
Number of methods	893
Lines per method (avg)	37
Lines per method (max)	1,444

Diagram data

^MI	0.35	^ECC_N	0.25	ECC>10	0.15	^BP	0.35
^HV_N	0.29	^ECC_F	0.16	LPM>20	0.45	^NBD	0.24



BonForum (13)

Program summary

Programming language	Java
Program length	4,077
Comment Rate	0.44
Maintainability	77.17
Textual code compl.	43.11
File complexity	44.29
Method complexity	5.09
Average nesting	2.81
Maximal nesting	10
Branch Percentage	0.25
Maximal coupling	15

Code distribution

Number of files	17
Lines per file (avg)	240
Lines per file (max)	1,287
Number of methods	180
Lines per method (avg)	22
Lines per method (max)	505

Diagram data

^MI	0.28	^ECC_N	0.28	ECC>10	0.09	^BP	0.35
^HV_N	0.24	^ECC_F	0.18	LPM>20	0.24	^NBD	0.32

FitNesse (14)

Program summary

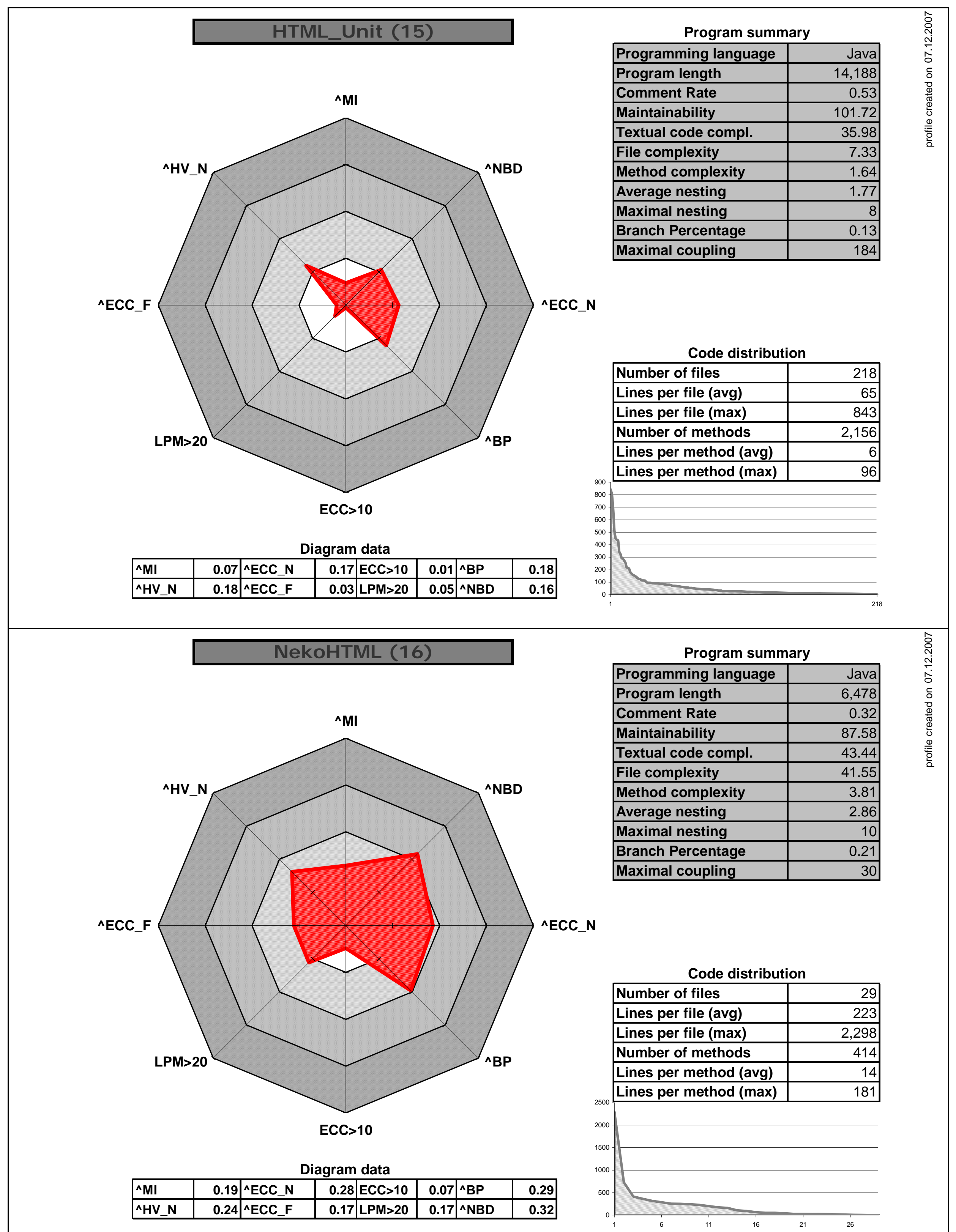
Programming language	Java
Program length	27,122
Comment Rate	0.02
Maintainability	107.19
Textual code compl.	33.65
File complexity	3.20
Method complexity	1.33
Average nesting	1.58
Maximal nesting	6
Branch Percentage	0.05
Maximal coupling	312

Code distribution

Number of files	493
Lines per file (avg)	55
Lines per file (max)	386
Number of methods	3,254
Lines per method (avg)	7
Lines per method (max)	86

Diagram data

^MI	0.02	^ECC_N	0.09	ECC>10	0.00	^BP	0.07
^HV_N	0.16	^ECC_F	0.01	LPM>20	0.02	^NBD	0.13



HTML_Unit (15)

Program summary

Programming language	Java
Program length	14,188
Comment Rate	0.53
Maintainability	101.72
Textual code compl.	35.98
File complexity	7.33
Method complexity	1.64
Average nesting	1.77
Maximal nesting	8
Branch Percentage	0.13
Maximal coupling	184

Code distribution

Number of files	218
Lines per file (avg)	65
Lines per file (max)	843
Number of methods	2,156
Lines per method (avg)	6
Lines per method (max)	96

Diagram data

^MI	0.07	^ECC_N	0.17	ECC>10	0.01	^BP	0.18
^HV_N	0.18	^ECC_F	0.03	LPM>20	0.05	^NBD	0.16

NekoHTML (16)

Program summary

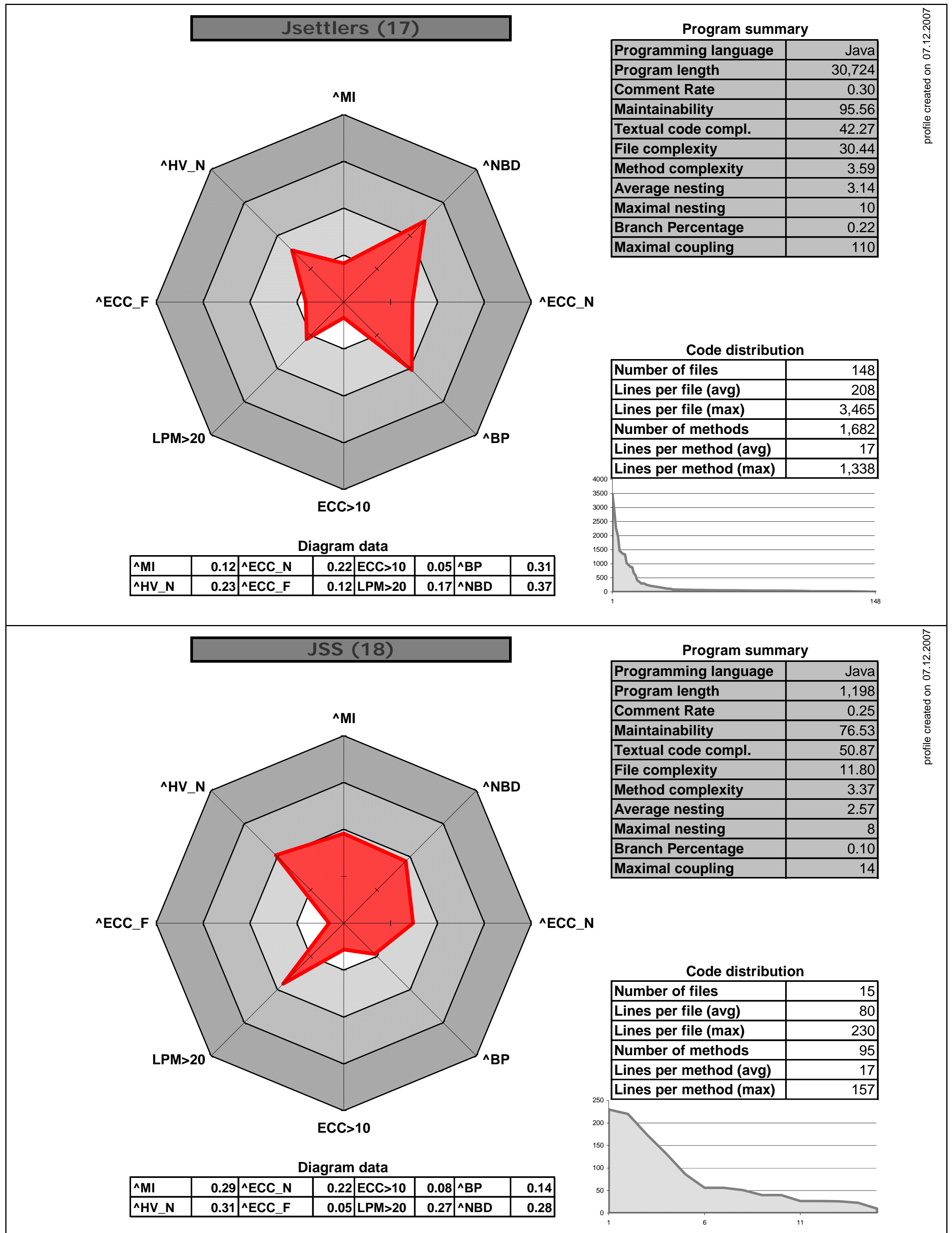
Programming language	Java
Program length	6,478
Comment Rate	0.32
Maintainability	87.58
Textual code compl.	43.44
File complexity	41.55
Method complexity	3.81
Average nesting	2.86
Maximal nesting	10
Branch Percentage	0.21
Maximal coupling	30

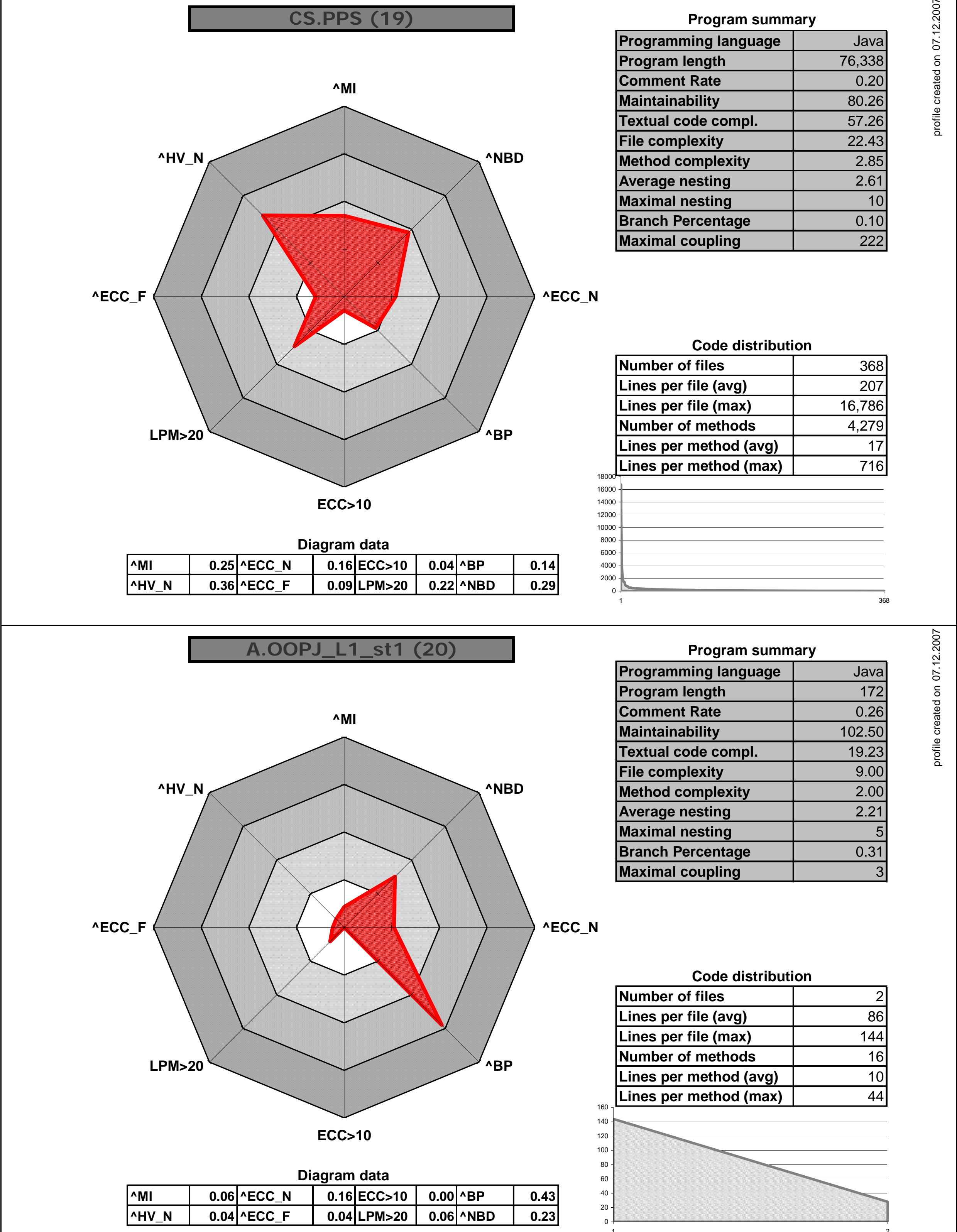
Code distribution

Number of files	29
Lines per file (avg)	223
Lines per file (max)	2,298
Number of methods	414
Lines per method (avg)	14
Lines per method (max)	181

Diagram data

^MI	0.19	^ECC_N	0.28	ECC>10	0.07	^BP	0.29
^HV_N	0.24	^ECC_F	0.17	LPM>20	0.17	^NBD	0.32


Jsettlers (17)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.12 ^ECC_N 0.22 ECC>10 0.05 ^BP 0.31
^HV_N 0.23 ^ECC_F 0.12 LPM>20 0.17 ^NBD 0.37
Program summary
Programming language Java
Program length 30,724
Comment Rate 0.30
Maintainability 95.56
Textual code compl. 42.27
File complexity 30.44
Method complexity 3.59
Average nesting 3.14
Maximal nesting 10
Branch Percentage 0.22
Maximal coupling 110
profile created on 07.12.2007
Code distribution
Number of files 148
Lines per file (avg) 208
Lines per file (max) 3,465
Number of methods 1,682
Lines per method (avg) 17
Lines per method (max) 1,338
JSS (18)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.29 ^ECC_N 0.22 ECC>10 0.08 ^BP 0.14
^HV_N 0.31 ^ECC_F 0.05 LPM>20 0.27 ^NBD 0.28
Program summary
Programming language Java
Program length 1,198
Comment Rate 0.25
Maintainability 76.53
Textual code compl. 50.87
File complexity 11.80
Method complexity 3.37
Average nesting 2.57
Maximal nesting 8
Branch Percentage 0.10
Maximal coupling 14
profile created on 07.12.2007
Code distribution
Number of files 15
Lines per file (avg) 80
Lines per file (max) 230
Number of methods 95
Lines per method (avg) 17
Lines per method (max) 157



CS.PPS (19)

Program summary

Programming language	Java
Program length	76,338
Comment Rate	0.20
Maintainability	80.26
Textual code compl.	57.26
File complexity	22.43
Method complexity	2.85
Average nesting	2.61
Maximal nesting	10
Branch Percentage	0.10
Maximal coupling	222

Code distribution

Number of files	368
Lines per file (avg)	207
Lines per file (max)	16,786
Number of methods	4,279
Lines per method (avg)	17
Lines per method (max)	716

Diagram data

^MI	0.25	^ECC_N	0.16	ECC>10	0.04	^BP	0.14
^HV_N	0.36	^ECC_F	0.09	LPM>20	0.22	^NBD	0.29

A.OOPJ_L1_st1 (20)

Program summary

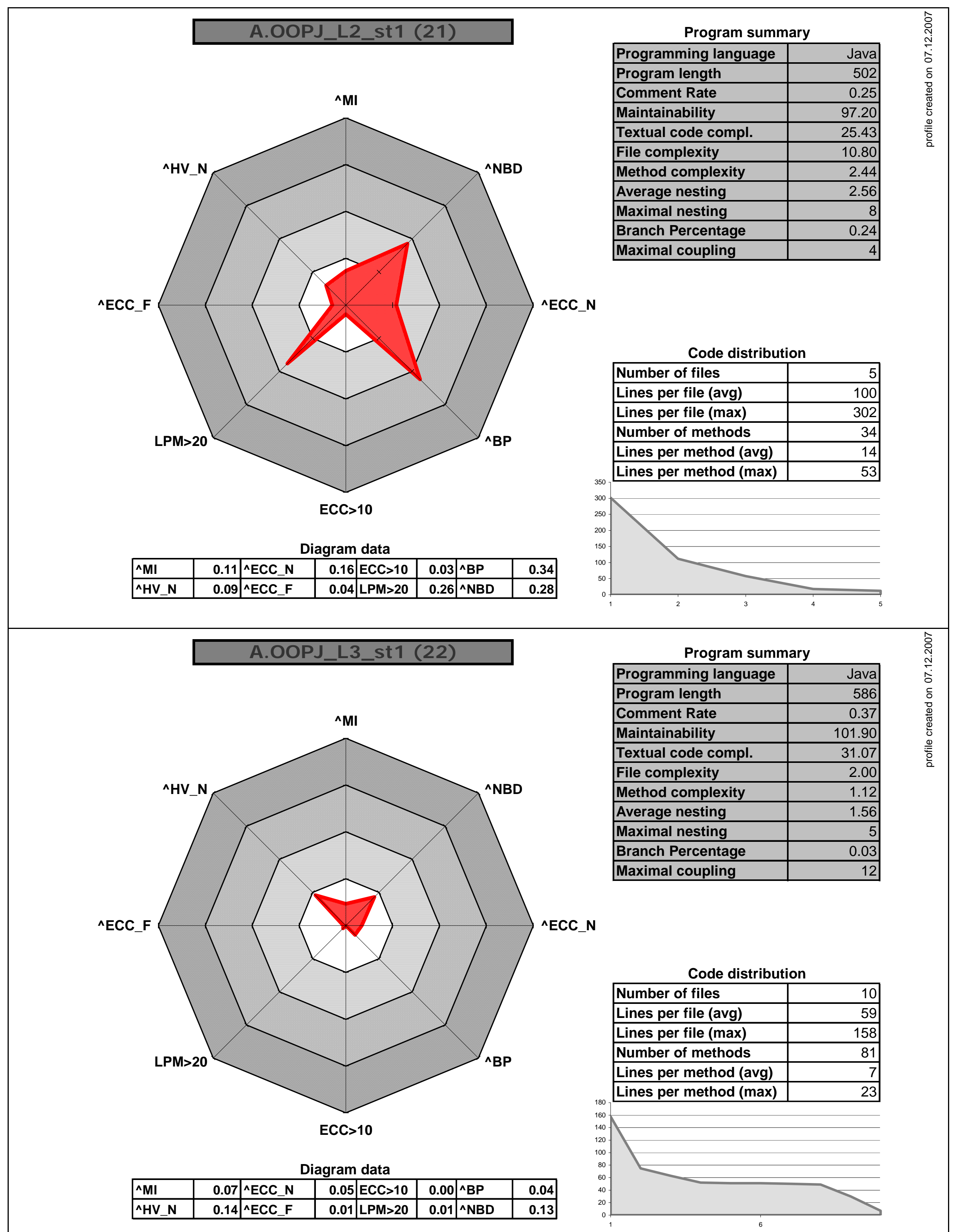
Programming language	Java
Program length	172
Comment Rate	0.26
Maintainability	102.50
Textual code compl.	19.23
File complexity	9.00
Method complexity	2.00
Average nesting	2.21
Maximal nesting	5
Branch Percentage	0.31
Maximal coupling	3

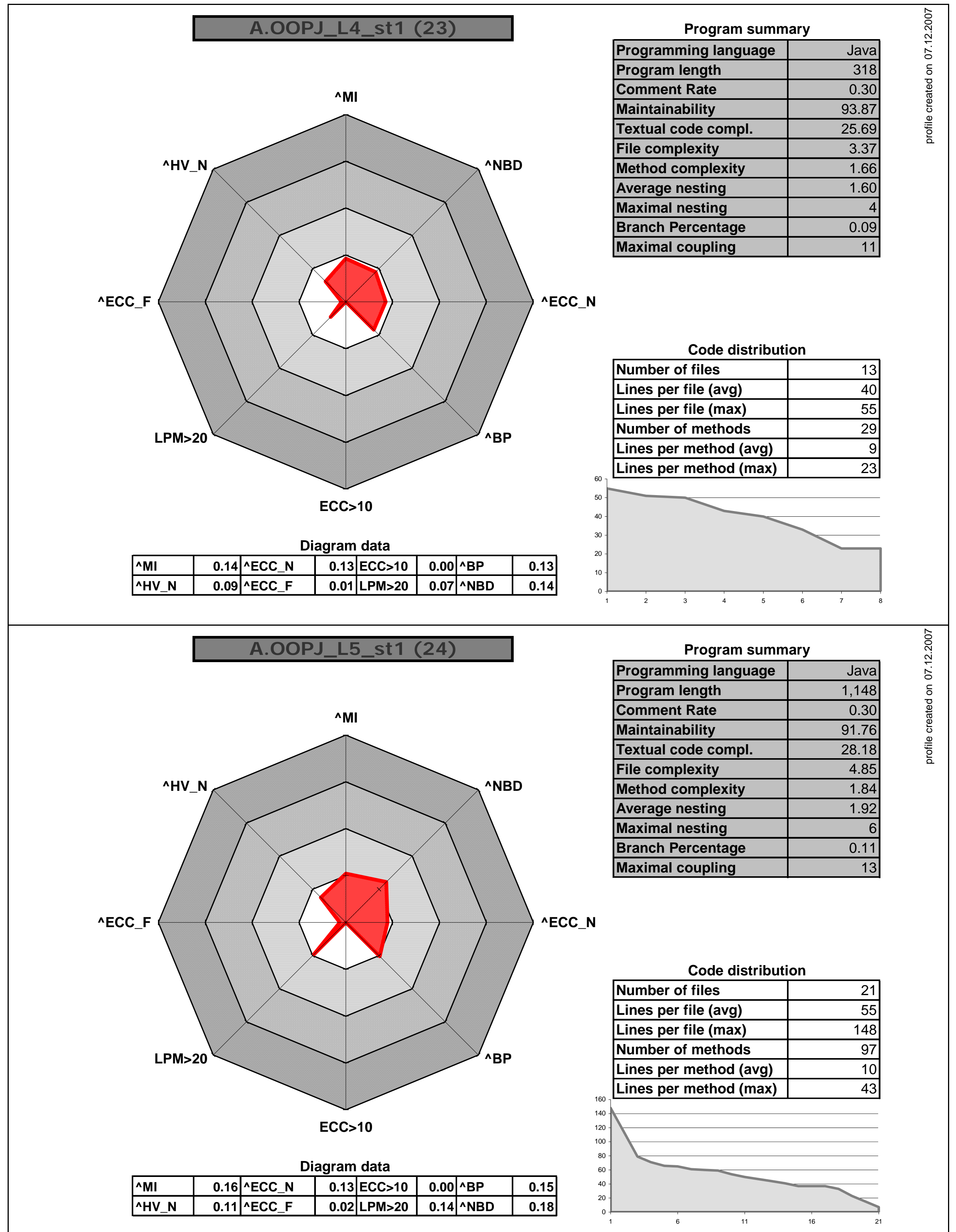
Code distribution

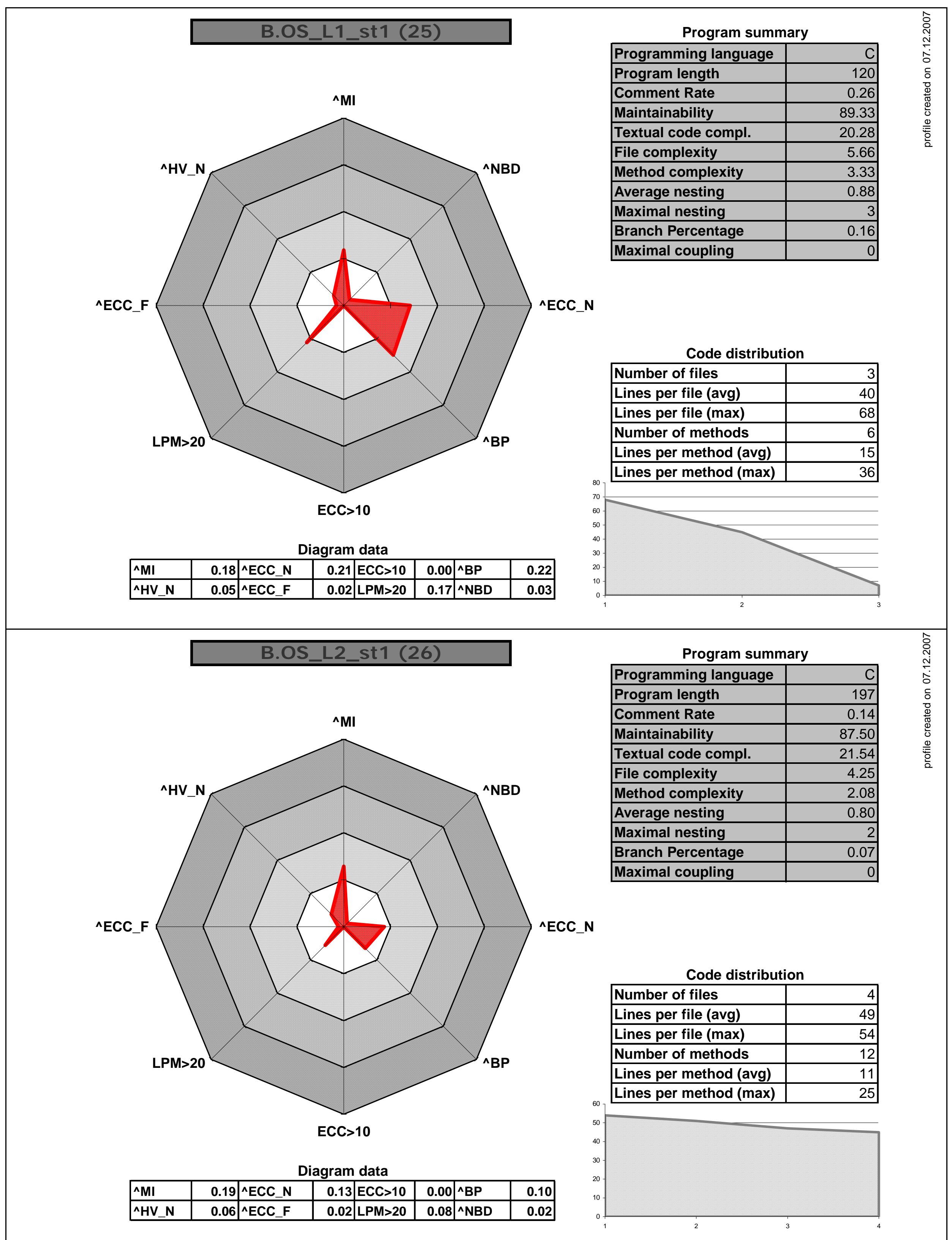
Number of files	2
Lines per file (avg)	86
Lines per file (max)	144
Number of methods	16
Lines per method (avg)	10
Lines per method (max)	44

Diagram data

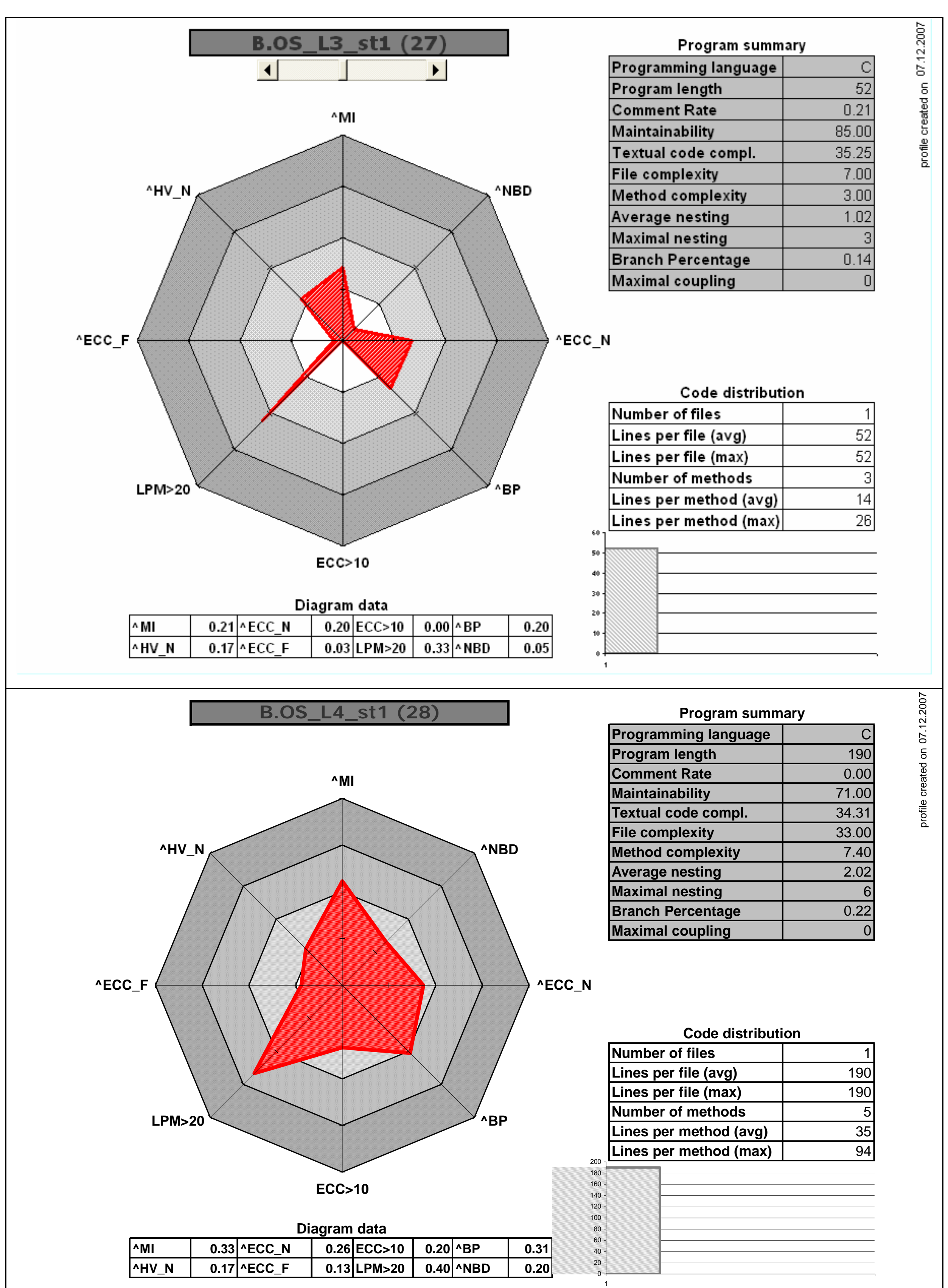
^MI	0.06	^ECC_N	0.16	ECC>10	0.00	^BP	0.43
^HV_N	0.04	^ECC_F	0.04	LPM>20	0.06	^NBD	0.23

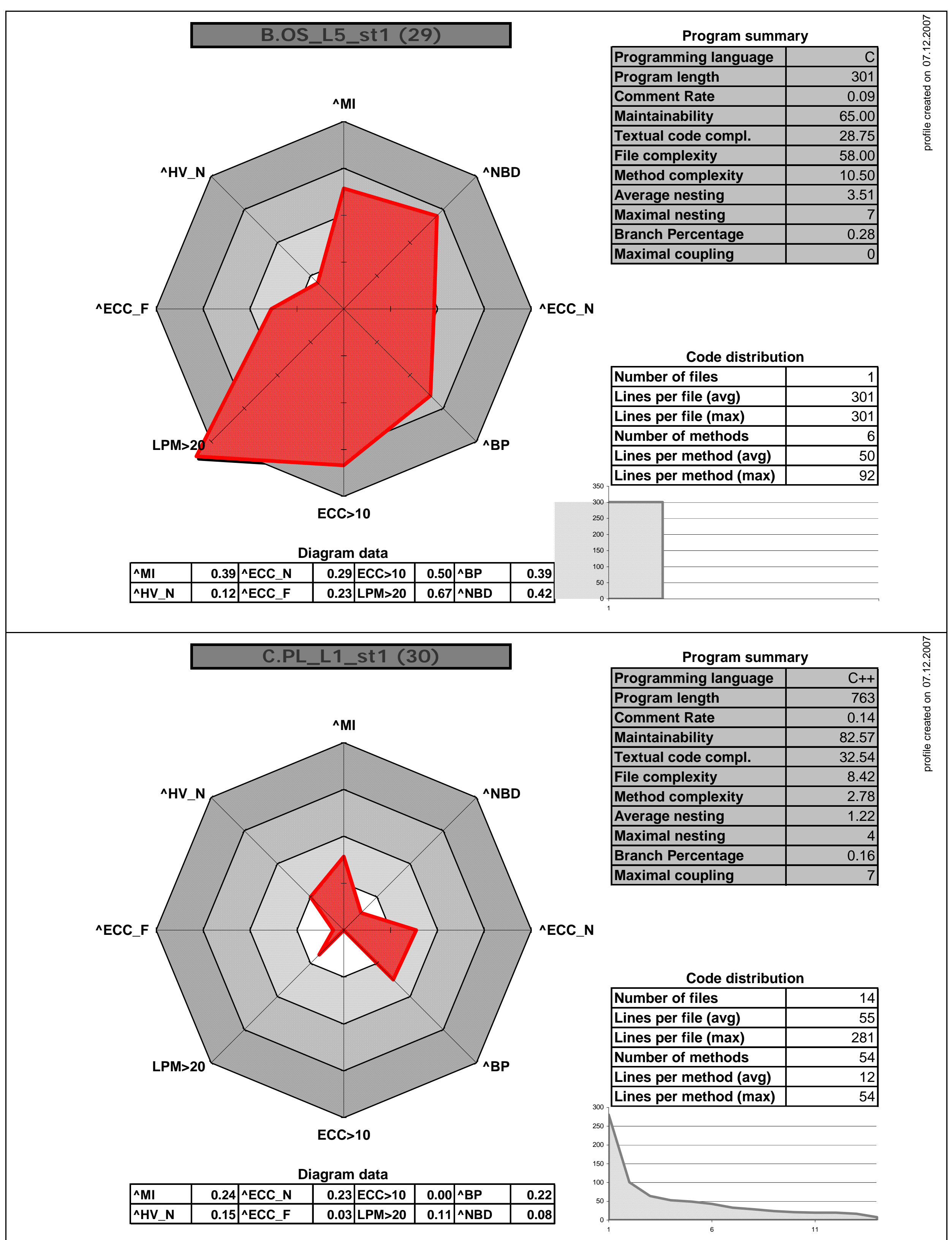

A.OOPJ_L2_st1 (21)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.11 ^ECC_N 0.16 ECC>10 0.03 ^BP 0.34
^HV_N 0.09 ^ECC_F 0.04 LPM>20 0.26 ^NBD 0.28
Program summary
Programming language Java
Program length 502
Comment Rate 0.25
Maintainability 97.20
Textual code compl. 25.43
File complexity 10.80
Method complexity 2.44
Average nesting 2.56
Maximal nesting 8
Branch Percentage 0.24
Maximal coupling 4
profile created on 07.12.2007
Code distribution
Number of files 5
Lines per file (avg) 100
Lines per file (max) 302
Number of methods 34
Lines per method (avg) 14
Lines per method (max) 53
A.OOPJ_L3_st1 (22)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.07 ^ECC_N 0.05 ECC>10 0.00 ^BP 0.04
^HV_N 0.14 ^ECC_F 0.01 LPM>20 0.01 ^NBD 0.13
Program summary
Programming language Java
Program length 586
Comment Rate 0.37
Maintainability 101.90
Textual code compl. 31.07
File complexity 2.00
Method complexity 1.12
Average nesting 1.56
Maximal nesting 5
Branch Percentage 0.03
Maximal coupling 12
profile created on 07.12.2007
Code distribution
Number of files 10
Lines per file (avg) 59
Lines per file (max) 158
Number of methods 81
Lines per method (avg) 7
Lines per method (max) 23

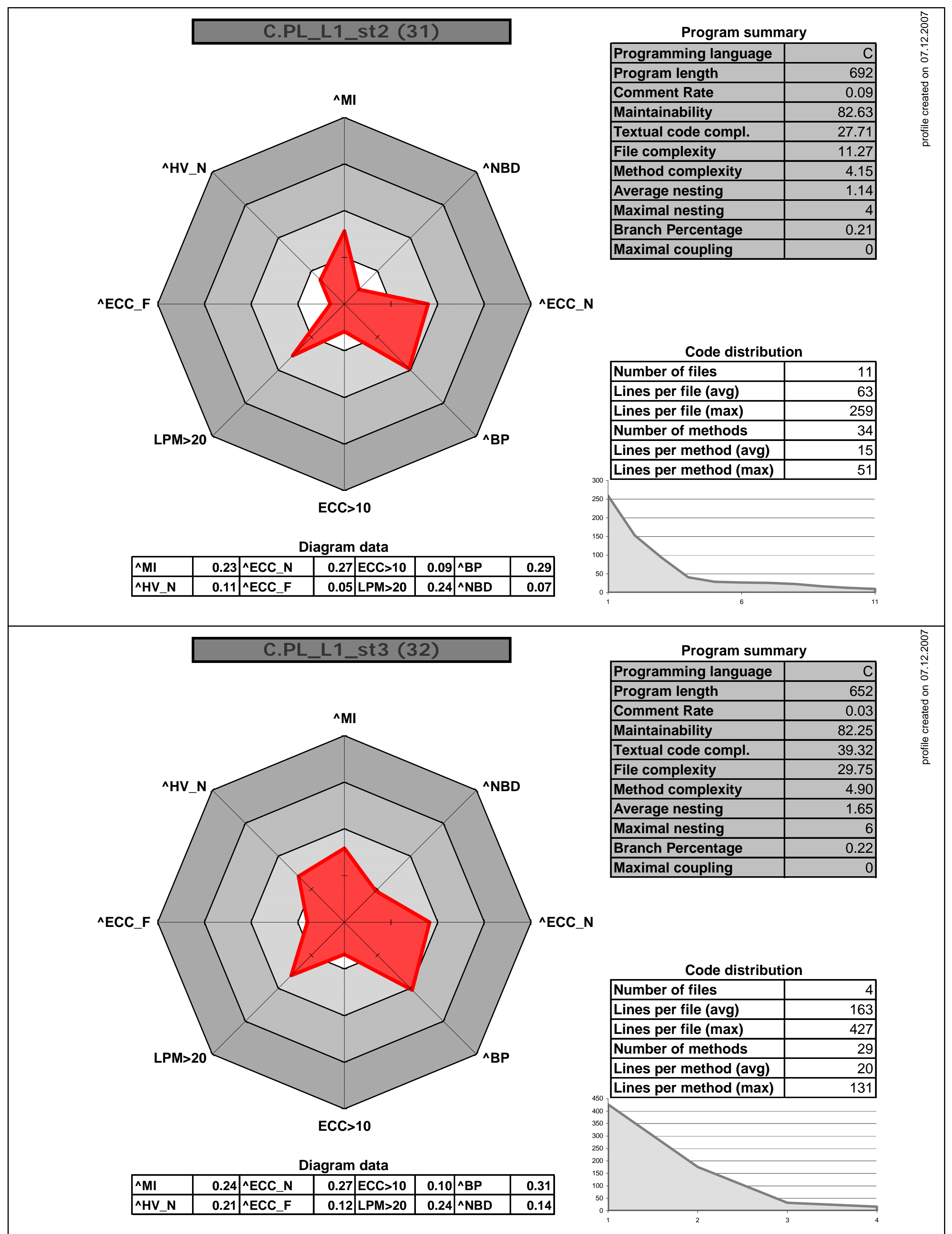

A.OOPJ_L4_st1 (23)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language Java
Program length 318
Comment Rate 0.30
Maintainability 93.87
Textual code compl. 25.69
File complexity 3.37
Method complexity 1.66
Average nesting 1.60
Maximal nesting 4
Branch Percentage 0.09
Maximal coupling 11
profile created on 07.12.2007
Code distribution
Number of files 13
Lines per file (avg) 40
Lines per file (max) 55
Number of methods 29
Lines per method (avg) 9
Lines per method (max) 23
Diagram data
^MI 0.14 ^ECC_N 0.13 ECC>10 0.00 ^BP 0.13
^HV_N 0.09 ^ECC_F 0.01 LPM>20 0.07 ^NBD 0.14
A.OOPJ_L5_st1 (24)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language Java
Program length 1,148
Comment Rate 0.30
Maintainability 91.76
Textual code compl. 28.18
File complexity 4.85
Method complexity 1.84
Average nesting 1.92
Maximal nesting 6
Branch Percentage 0.11
Maximal coupling 13
profile created on 07.12.2007
Code distribution
Number of files 21
Lines per file (avg) 55
Lines per file (max) 148
Number of methods 97
Lines per method (avg) 10
Lines per method (max) 43
Diagram data
^MI 0.16 ^ECC_N 0.13 ECC>10 0.00 ^BP 0.15
^HV_N 0.11 ^ECC_F 0.02 LPM>20 0.14 ^NBD 0.18

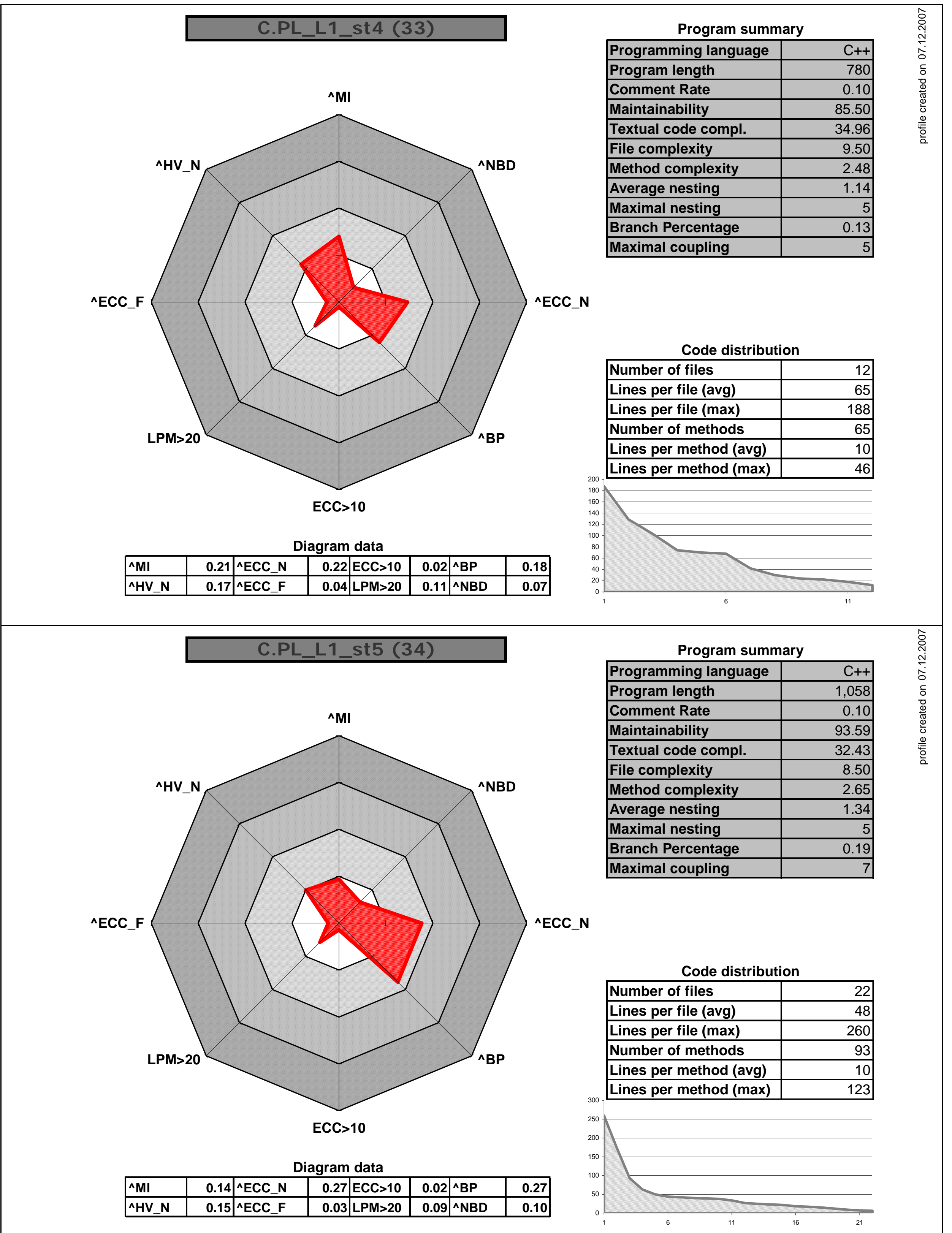


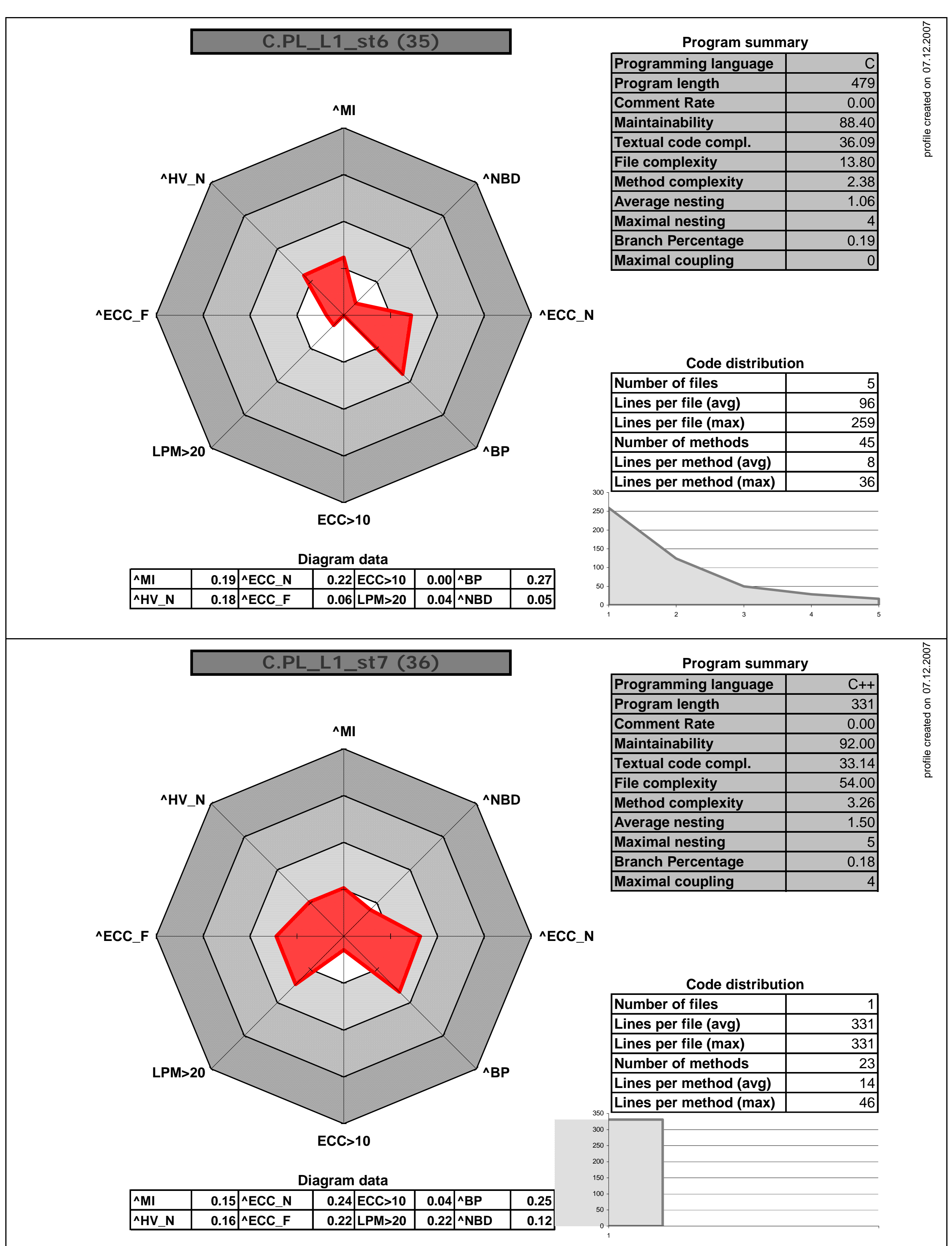
B.OS_L1_st1 (25)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 120
Comment Rate 0.26
Maintainability 89.33
Textual code compl. 20.28
File complexity 5.66
Method complexity 3.33
Average nesting 0.88
Maximal nesting 3
Branch Percentage 0.16
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 3
Lines per file (avg) 40
Lines per file (max) 68
Number of methods 6
Lines per method (avg) 15
Lines per method (max) 36
Diagram data
^MI 0.18 ^ECC_N 0.21 ECC>10 0.00 ^BP 0.22
^HV_N 0.05 ^ECC_F 0.02 LPM>20 0.17 ^NBD 0.03
B.OS_L2_st1 (26)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 197
Comment Rate 0.14
Maintainability 87.50
Textual code compl. 21.54
File complexity 4.25
Method complexity 2.08
Average nesting 0.80
Maximal nesting 2
Branch Percentage 0.07
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 4
Lines per file (avg) 49
Lines per file (max) 54
Number of methods 12
Lines per method (avg) 11
Lines per method (max) 25
Diagram data
^MI 0.19 ^ECC_N 0.13 ECC>10 0.00 ^BP 0.10
^HV_N 0.06 ^ECC_F 0.02 LPM>20 0.08 ^NBD 0.02


B.OS_L3_st1 (27)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 52
Comment Rate 0.21
Maintainability 85.00
Textual code compl. 35.25
File complexity 7.00
Method complexity 3.00
Average nesting 1.02
Maximal nesting 3
Branch Percentage 0.14
Maximal coupling 0
Code distribution
Number of files 1
Lines per file (avg) 52
Lines per file (max) 52
Number of methods 3
Lines per method (avg) 14
Lines per method (max) 26
Diagram data
^MI 0.21 ^ECC_N 0.20 ECC>10 0.00 ^BP 0.20
^HV_N 0.17 ^ECC_F 0.03 LPM>20 0.33 ^NBD 0.05
profile created on 07.12.2007
B.OS_L4_st1 (28)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 190
Comment Rate 0.00
Maintainability 71.00
Textual code compl. 34.31
File complexity 33.00
Method complexity 7.40
Average nesting 2.02
Maximal nesting 6
Branch Percentage 0.22
Maximal coupling 0
Code distribution
Number of files 1
Lines per file (avg) 190
Lines per file (max) 190
Number of methods 5
Lines per method (avg) 35
Lines per method (max) 94
Diagram data
^MI 0.33 ^ECC_N 0.26 ECC>10 0.20 ^BP 0.31
^HV_N 0.17 ^ECC_F 0.13 LPM>20 0.40 ^NBD 0.20
profile created on 07.12.2007


B.OS_L5_st1 (29)
Program summary
Programming language C
Program length 301
Comment Rate 0.09
Maintainability 65.00
Textual code compl. 28.75
File complexity 58.00
Method complexity 10.50
Average nesting 3.51
Maximal nesting 7
Branch Percentage 0.28
Maximal coupling 0
profile created on 07.12.2007
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Code distribution
Number of files 1
Lines per file (avg) 301
Lines per file (max) 301
Number of methods 6
Lines per method (avg) 50
Lines per method (max) 92
Diagram data
^MI 0.39 ^ECC_N 0.29 ECC>10 0.50 ^BP 0.39
^HV_N 0.12 ^ECC_F 0.23 LPM>20 0.67 ^NBD 0.42
C.PL_L1_st1 (30)
Program summary
Programming language C++
Program length 763
Comment Rate 0.14
Maintainability 82.57
Textual code compl. 32.54
File complexity 8.42
Method complexity 2.78
Average nesting 1.22
Maximal nesting 4
Branch Percentage 0.16
Maximal coupling 7
profile created on 07.12.2007
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Code distribution
Number of files 14
Lines per file (avg) 55
Lines per file (max) 281
Number of methods 54
Lines per method (avg) 12
Lines per method (max) 54
Diagram data
^MI 0.24 ^ECC_N 0.23 ECC>10 0.00 ^BP 0.22
^HV_N 0.15 ^ECC_F 0.03 LPM>20 0.11 ^NBD 0.08


C.PL_L1_st2 (31)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 692
Comment Rate 0.09
Maintainability 82.63
Textual code compl. 27.71
File complexity 11.27
Method complexity 4.15
Average nesting 1.14
Maximal nesting 4
Branch Percentage 0.21
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 11
Lines per file (avg) 63
Lines per file (max) 259
Number of methods 34
Lines per method (avg) 15
Lines per method (max) 51
Diagram data
^MI 0.23 ^ECC_N 0.27 ECC>10 0.09 ^BP 0.29
^HV_N 0.11 ^ECC_F 0.05 LPM>20 0.24 ^NBD 0.07
C.PL_L1_st3 (32)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 652
Comment Rate 0.03
Maintainability 82.25
Textual code compl. 39.32
File complexity 29.75
Method complexity 4.90
Average nesting 1.65
Maximal nesting 6
Branch Percentage 0.22
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 4
Lines per file (avg) 163
Lines per file (max) 427
Number of methods 29
Lines per method (avg) 20
Lines per method (max) 131
Diagram data
^MI 0.24 ^ECC_N 0.27 ECC>10 0.10 ^BP 0.31
^HV_N 0.21 ^ECC_F 0.12 LPM>20 0.24 ^NBD 0.14


C.PL_L1_st4 (33)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.21 ^ECC_N 0.22 ECC>10 0.02 ^BP 0.18
^HV_N 0.17 ^ECC_F 0.04 LPM>20 0.11 ^NBD 0.07
Program summary
Programming language C++
Program length 780
Comment Rate 0.10
Maintainability 85.50
Textual code compl. 34.96
File complexity 9.50
Method complexity 2.48
Average nesting 1.14
Maximal nesting 5
Branch Percentage 0.13
Maximal coupling 5
profile created on 07.12.2007
Code distribution
Number of files 12
Lines per file (avg) 65
Lines per file (max) 188
Number of methods 65
Lines per method (avg) 10
Lines per method (max) 46
C.PL_L1_st5 (34)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.14 ^ECC_N 0.27 ECC>10 0.02 ^BP 0.27
^HV_N 0.15 ^ECC_F 0.03 LPM>20 0.09 ^NBD 0.10
Program summary
Programming language C++
Program length 1,058
Comment Rate 0.10
Maintainability 93.59
Textual code compl. 32.43
File complexity 8.50
Method complexity 2.65
Average nesting 1.34
Maximal nesting 5
Branch Percentage 0.19
Maximal coupling 7
profile created on 07.12.2007
Code distribution
Number of files 22
Lines per file (avg) 48
Lines per file (max) 260
Number of methods 93
Lines per method (avg) 10
Lines per method (max) 123



Program summary

Programming language	C
Program length	479
Comment Rate	0.00
Maintainability	88.40
Textual code compl.	36.09
File complexity	13.80
Method complexity	2.38
Average nesting	1.06
Maximal nesting	4
Branch Percentage	0.19
Maximal coupling	0

Code distribution

Number of files	5
Lines per file (avg)	96
Lines per file (max)	259
Number of methods	45
Lines per method (avg)	8
Lines per method (max)	36

Diagram data

^MI	0.19	^ECC_N	0.22	ECC>10	0.00	^BP	0.27
^HV_N	0.18	^ECC_F	0.06	LPM>20	0.04	^NBD	0.05

Program summary

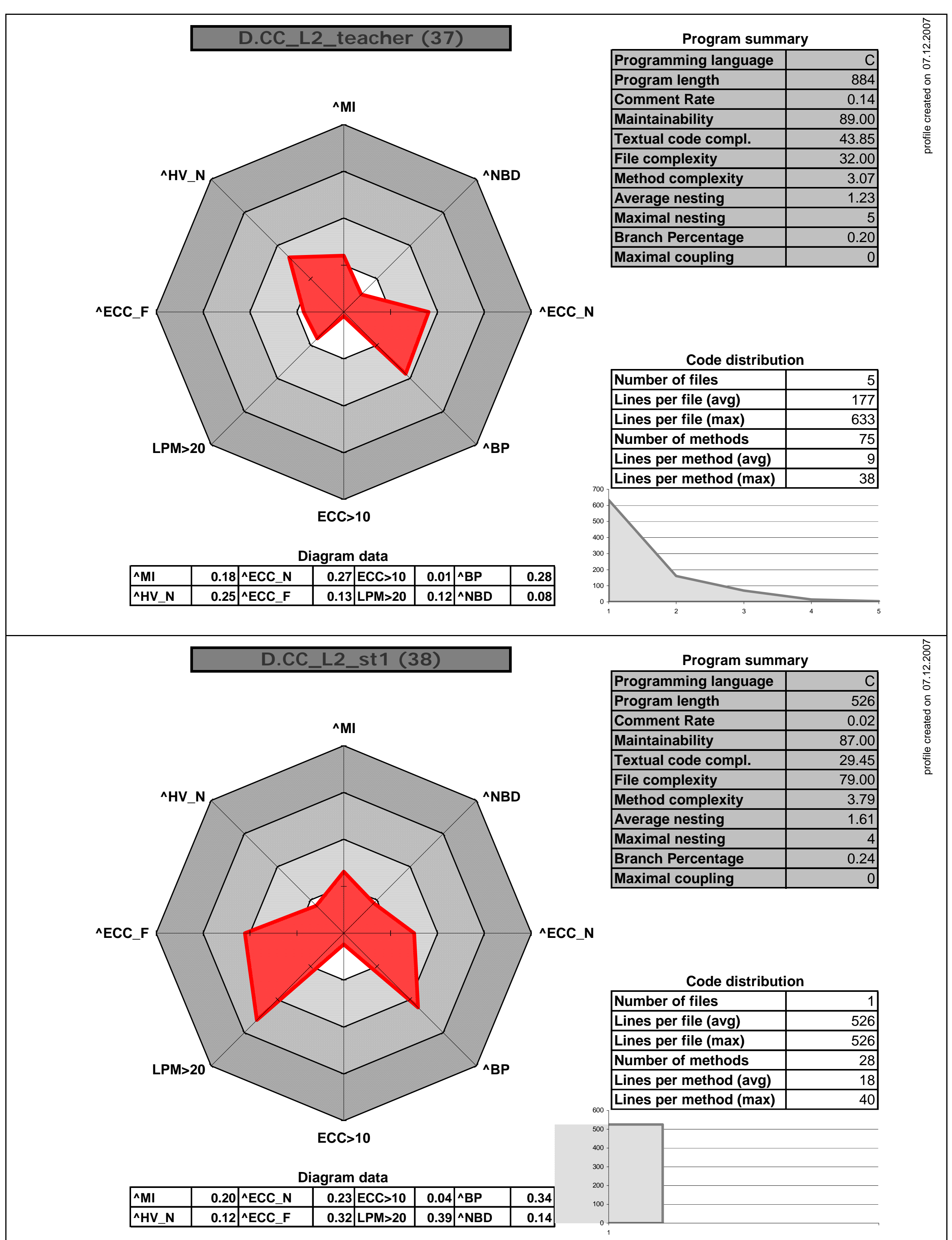
Programming language	C++
Program length	331
Comment Rate	0.00
Maintainability	92.00
Textual code compl.	33.14
File complexity	54.00
Method complexity	3.26
Average nesting	1.50
Maximal nesting	5
Branch Percentage	0.18
Maximal coupling	4

Code distribution

Number of files	1
Lines per file (avg)	331
Lines per file (max)	331
Number of methods	23
Lines per method (avg)	14
Lines per method (max)	46

Diagram data

^MI	0.15	^ECC_N	0.24	ECC>10	0.04	^BP	0.25
^HV_N	0.16	^ECC_F	0.22	LPM>20	0.22	^NBD	0.12



Program summary

Programming language	C
Program length	884
Comment Rate	0.14
Maintainability	89.00
Textual code compl.	43.85
File complexity	32.00
Method complexity	3.07
Average nesting	1.23
Maximal nesting	5
Branch Percentage	0.20
Maximal coupling	0

Code distribution

Number of files	5
Lines per file (avg)	177
Lines per file (max)	633
Number of methods	75
Lines per method (avg)	9
Lines per method (max)	38

Diagram data

^MI	0.18	^ECC_N	0.27	ECC>10	0.01	^BP	0.28
^HV_N	0.25	^ECC_F	0.13	LPM>20	0.12	^NBD	0.08

Program summary

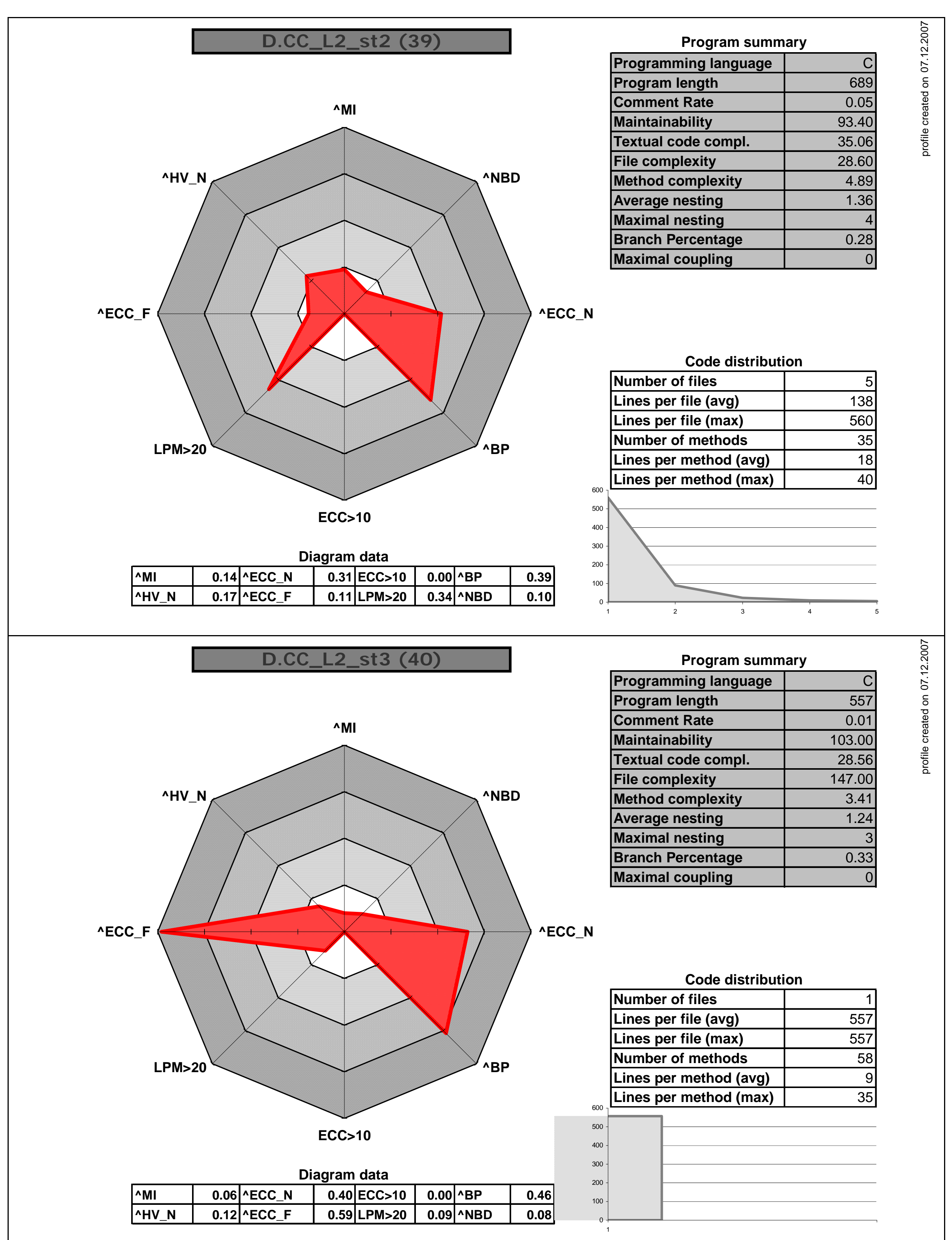
Programming language	C
Program length	526
Comment Rate	0.02
Maintainability	87.00
Textual code compl.	29.45
File complexity	79.00
Method complexity	3.79
Average nesting	1.61
Maximal nesting	4
Branch Percentage	0.24
Maximal coupling	0

Code distribution

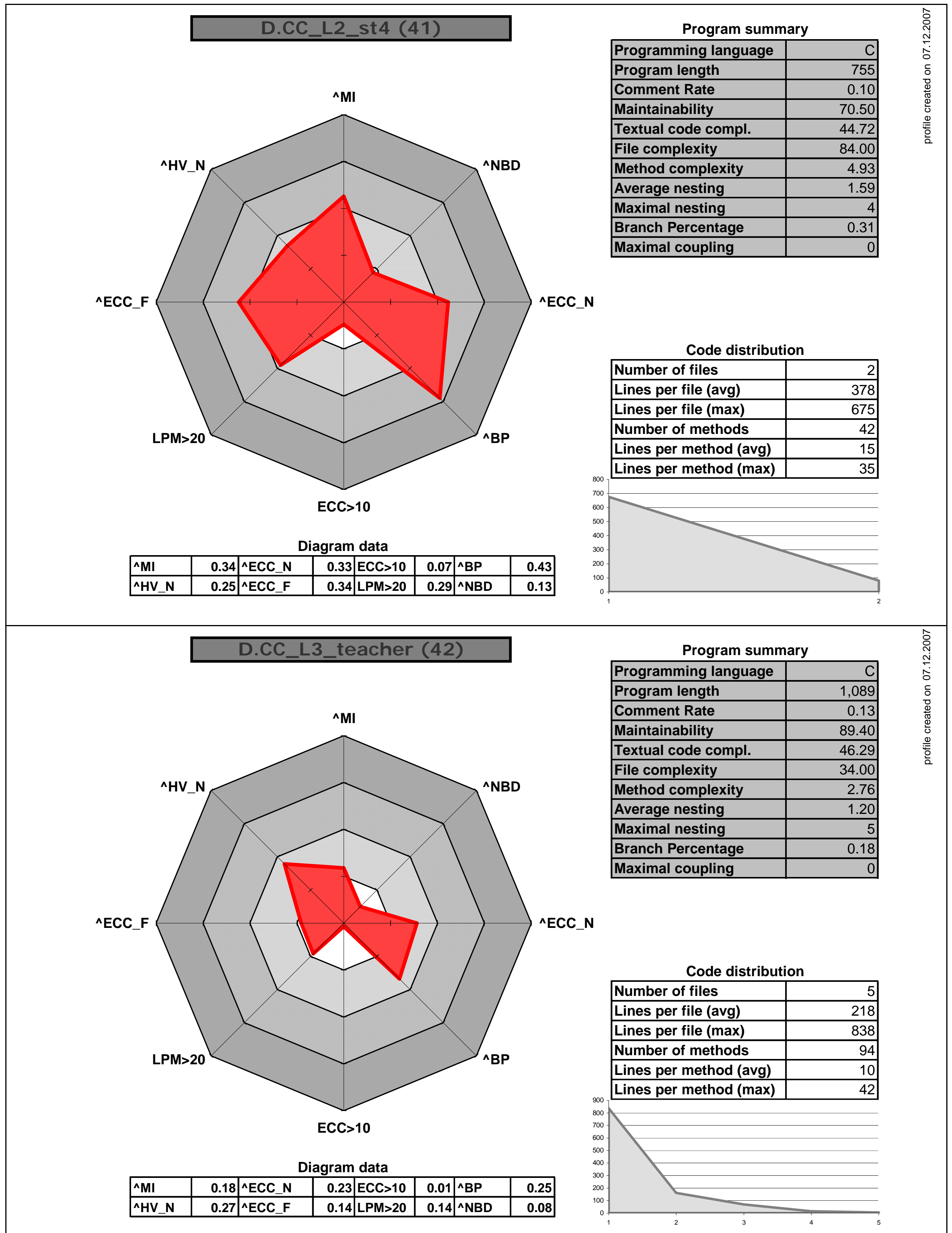
Number of files	1
Lines per file (avg)	526
Lines per file (max)	526
Number of methods	28
Lines per method (avg)	18
Lines per method (max)	40

Diagram data

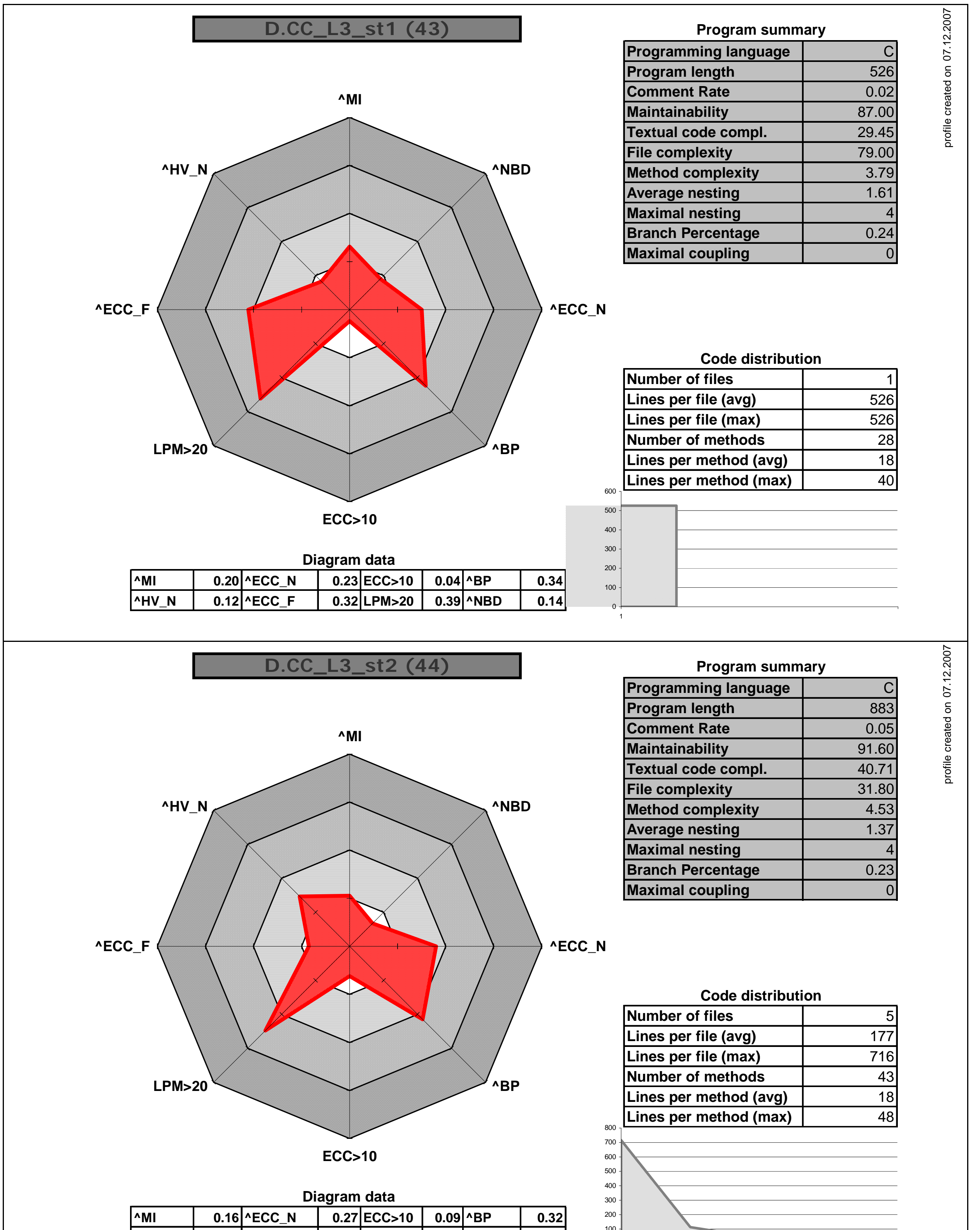
^MI	0.20	^ECC_N	0.23	ECC>10	0.04	^BP	0.34
^HV_N	0.12	^ECC_F	0.32	LPM>20	0.39	^NBD	0.14



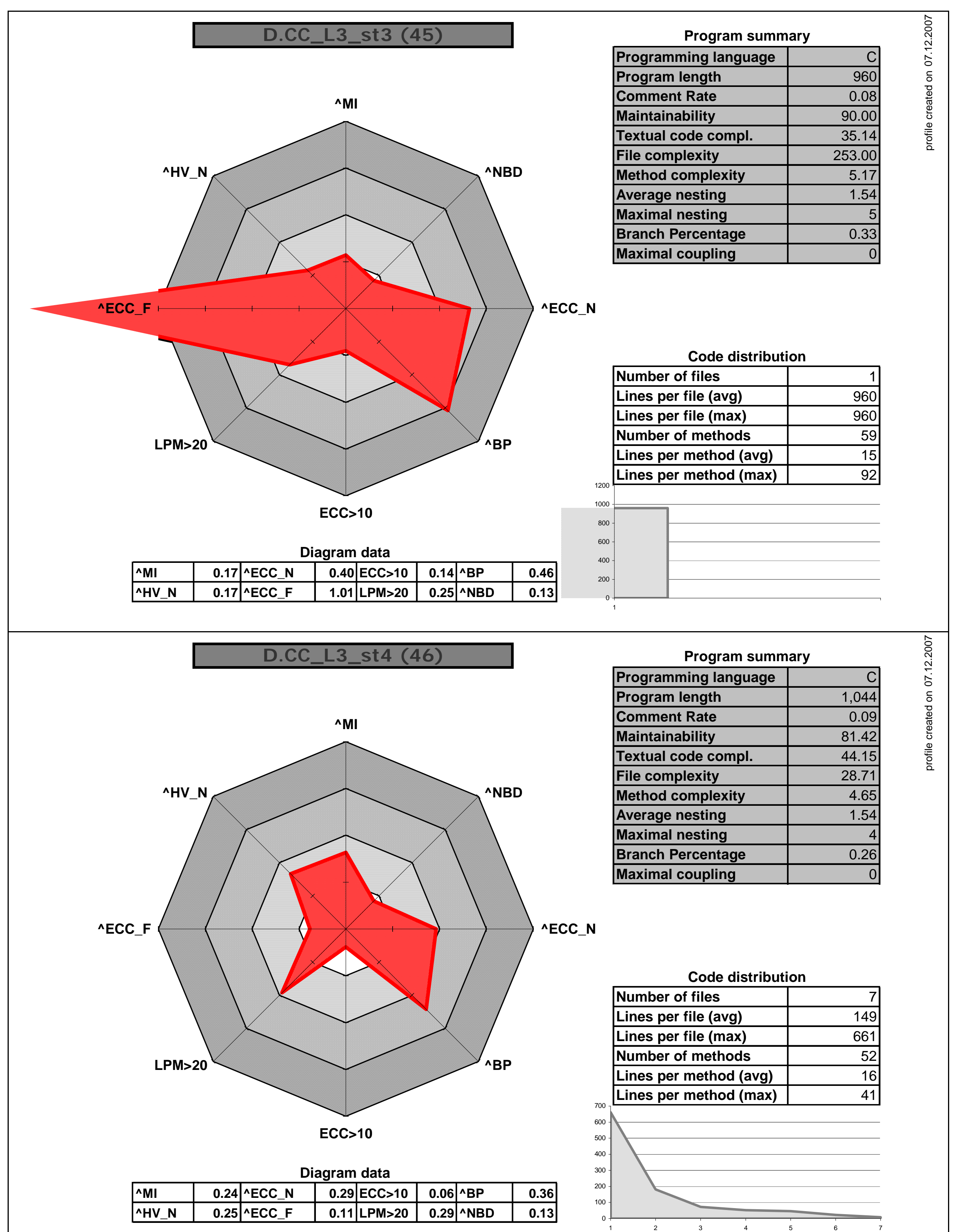
D.CC_L2_st2 (39)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 689
Comment Rate 0.05
Maintainability 93.40
Textual code compl. 35.06
File complexity 28.60
Method complexity 4.89
Average nesting 1.36
Maximal nesting 4
Branch Percentage 0.28
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 5
Lines per file (avg) 138
Lines per file (max) 560
Number of methods 35
Lines per method (avg) 18
Lines per method (max) 40
Diagram data
^MI 0.14 ^ECC_N 0.31 ECC>10 0.00 ^BP 0.39
^HV_N 0.17 ^ECC_F 0.11 LPM>20 0.34 ^NBD 0.10
D.CC_L2_st3 (40)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 557
Comment Rate 0.01
Maintainability 103.00
Textual code compl. 28.56
File complexity 147.00
Method complexity 3.41
Average nesting 1.24
Maximal nesting 3
Branch Percentage 0.33
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 1
Lines per file (avg) 557
Lines per file (max) 557
Number of methods 58
Lines per method (avg) 9
Lines per method (max) 35
Diagram data
^MI 0.06 ^ECC_N 0.40 ECC>10 0.00 ^BP 0.46
^HV_N 0.12 ^ECC_F 0.59 LPM>20 0.09 ^NBD 0.08



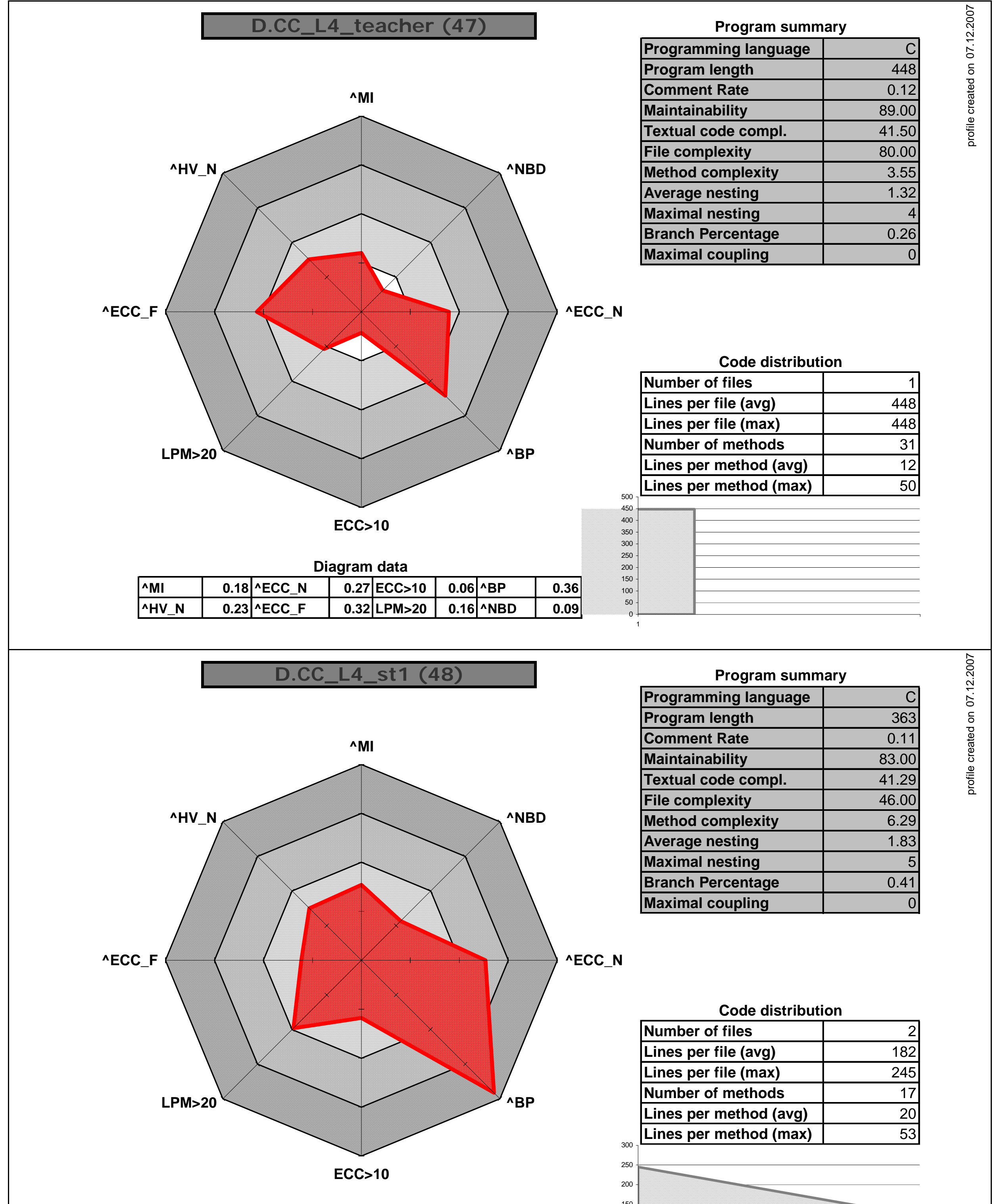
D.CC_L2_st4 (41)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 755
Comment Rate 0.10
Maintainability 70.50
Textual code compl. 44.72
File complexity 84.00
Method complexity 4.93
Average nesting 1.59
Maximal nesting 4
Branch Percentage 0.31
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 2
Lines per file (avg) 378
Lines per file (max) 675
Number of methods 42
Lines per method (avg) 15
Lines per method (max) 35
Diagram data
^MI 0.34 ^ECC_N 0.33 ECC>10 0.07 ^BP 0.43
^HV_N 0.25 ^ECC_F 0.34 LPM>20 0.29 ^NBD 0.13
D.CC_L3_teacher (42)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 1,089
Comment Rate 0.13
Maintainability 89.40
Textual code compl. 46.29
File complexity 34.00
Method complexity 2.76
Average nesting 1.20
Maximal nesting 5
Branch Percentage 0.18
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 5
Lines per file (avg) 218
Lines per file (max) 838
Number of methods 94
Lines per method (avg) 10
Lines per method (max) 42
Diagram data
^MI 0.18 ^ECC_N 0.23 ECC>10 0.01 ^BP 0.25
^HV_N 0.27 ^ECC_F 0.14 LPM>20 0.14 ^NBD 0.08



D.CC_L3_st1 (43)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 526
Comment Rate 0.02
Maintainability 87.00
Textual code compl. 29.45
File complexity 79.00
Method complexity 3.79
Average nesting 1.61
Maximal nesting 4
Branch Percentage 0.24
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 1
Lines per file (avg) 526
Lines per file (max) 526
Number of methods 28
Lines per method (avg) 18
Lines per method (max) 40
Diagram data
^MI 0.20 ^ECC_N 0.23 ECC>10 0.04 ^BP 0.34
^HV_N 0.12 ^ECC_F 0.32 LPM>20 0.39 ^NBD 0.14
D.CC_L3_st2 (44)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 883
Comment Rate 0.05
Maintainability 91.60
Textual code compl. 40.71
File complexity 31.80
Method complexity 4.53
Average nesting 1.37
Maximal nesting 4
Branch Percentage 0.23
Maximal coupling 0
profile created on 07.12.2007
Code distribution
Number of files 5
Lines per file (avg) 177
Lines per file (max) 716
Number of methods 43
Lines per method (avg) 18
Lines per method (max) 48
Diagram data
^MI 0.16 ^ECC_N 0.27 ECC>10 0.09 ^BP 0.32
^HV_N 0.22 ^ECC_F 0.13 LPM>20 0.37 ^NBD 0.10



D.CC_L3_st3 (45)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 960
Comment Rate 0.08
Maintainability 90.00
Textual code compl. 35.14
File complexity 253.00
Method complexity 5.17
Average nesting 1.54
Maximal nesting 5
Branch Percentage 0.33
Maximal coupling 0
Code distribution
Number of files 1
Lines per file (avg) 960
Lines per file (max) 960
Number of methods 59
Lines per method (avg) 15
Lines per method (max) 92
Diagram data
^MI 0.17 ^ECC_N 0.40 ECC>10 0.14 ^BP 0.46
^HV_N 0.17 ^ECC_F 1.01 LPM>20 0.25 ^NBD 0.13
profile created on 07.12.2007
D.CC_L3_st4 (46)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Program summary
Programming language C
Program length 1,044
Comment Rate 0.09
Maintainability 81.42
Textual code compl. 44.15
File complexity 28.71
Method complexity 4.65
Average nesting 1.54
Maximal nesting 4
Branch Percentage 0.26
Maximal coupling 0
Code distribution
Number of files 7
Lines per file (avg) 149
Lines per file (max) 661
Number of methods 52
Lines per method (avg) 16
Lines per method (max) 41
Diagram data
^MI 0.24 ^ECC_N 0.29 ECC>10 0.06 ^BP 0.36
^HV_N 0.25 ^ECC_F 0.11 LPM>20 0.29 ^NBD 0.13
profile created on 07.12.2007



Program summary

Programming language	C
Program length	448
Comment Rate	0.12
Maintainability	89.00
Textual code compl.	41.50
File complexity	80.00
Method complexity	3.55
Average nesting	1.32
Maximal nesting	4
Branch Percentage	0.26
Maximal coupling	0

Code distribution

Number of files	1
Lines per file (avg)	448
Lines per file (max)	448
Number of methods	31
Lines per method (avg)	12
Lines per method (max)	50

Diagram data

^MI	0.18	^ECC_N	0.27	ECC>10	0.06	^BP	0.36
^HV_N	0.23	^ECC_F	0.32	LPM>20	0.16	^NBD	0.09

Program summary

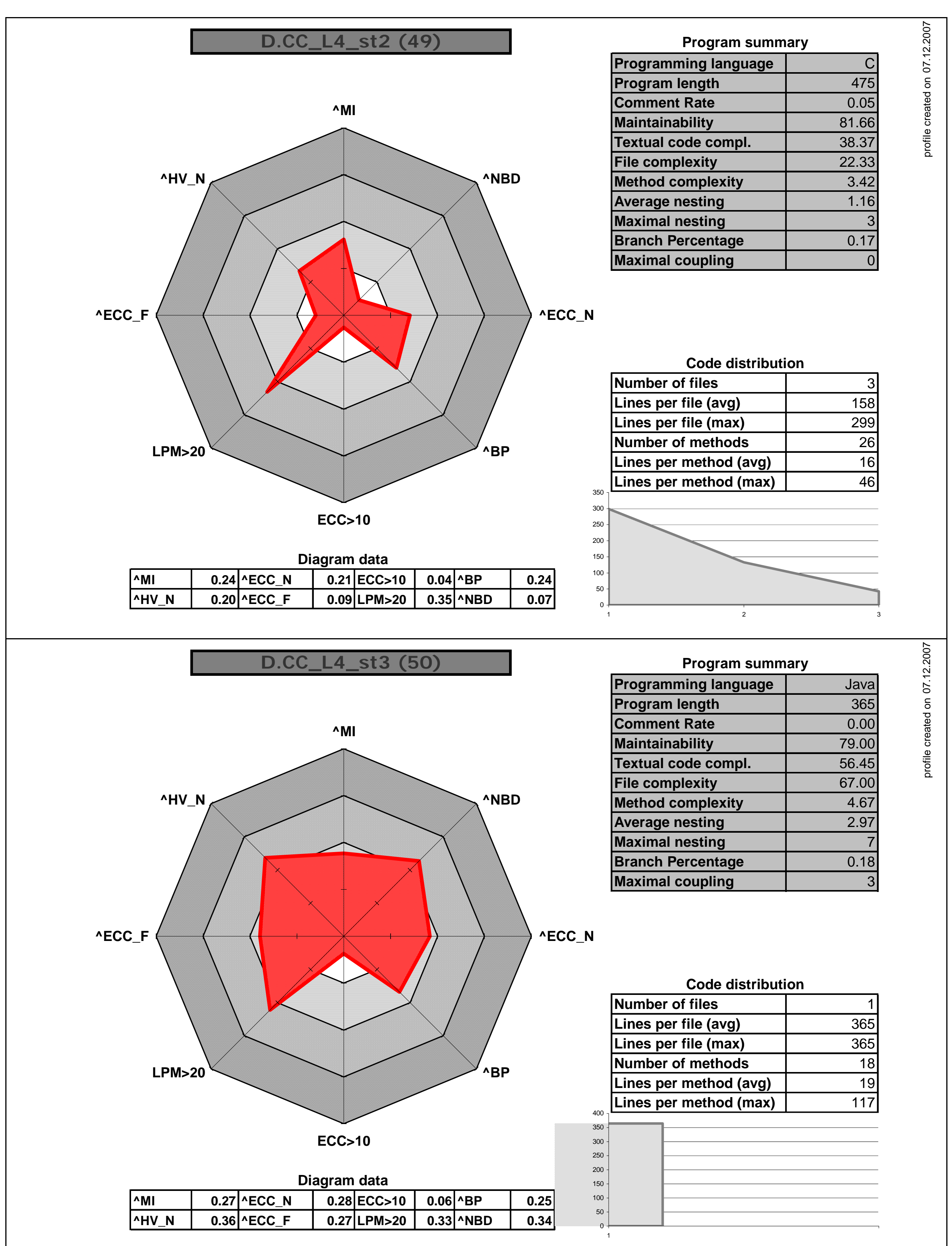
Programming language	C
Program length	363
Comment Rate	0.11
Maintainability	83.00
Textual code compl.	41.29
File complexity	46.00
Method complexity	6.29
Average nesting	1.83
Maximal nesting	5
Branch Percentage	0.41
Maximal coupling	0

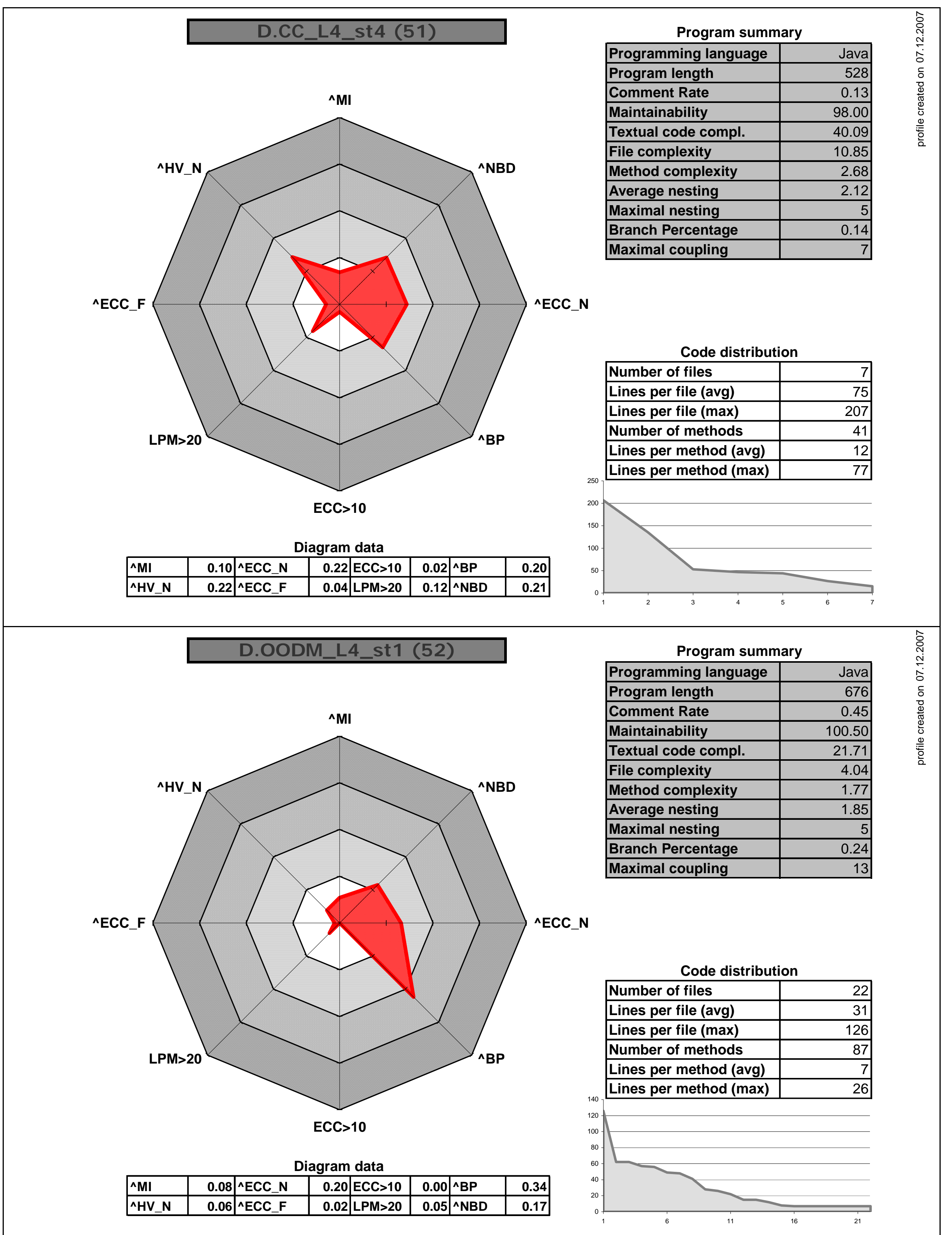
Code distribution

Number of files	2
Lines per file (avg)	182
Lines per file (max)	245
Number of methods	17
Lines per method (avg)	20
Lines per method (max)	53

Diagram data

^MI	0.23	^ECC_N	0.38	ECC>10	0.18	^BP	0.57
^HV_N	0.23	^ECC_F	0.18	LPM>20	0.29	^NBD	0.17


D.CC_L4_st2 (49)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.24 ^ECC_N 0.21 ECC>10 0.04 ^BP 0.24
^HV_N 0.20 ^ECC_F 0.09 LPM>20 0.35 ^NBD 0.07
Program summary
Programming language C
Program length 475
Comment Rate 0.05
Maintainability 81.66
Textual code compl. 38.37
File complexity 22.33
Method complexity 3.42
Average nesting 1.16
Maximal nesting 3
Branch Percentage 0.17
Maximal coupling 0
Code distribution
Number of files 3
Lines per file (avg) 158
Lines per file (max) 299
Number of methods 26
Lines per method (avg) 16
Lines per method (max) 46
profile created on 07.12.2007
D.CC_L4_st3 (50)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.27 ^ECC_N 0.28 ECC>10 0.06 ^BP 0.25
^HV_N 0.36 ^ECC_F 0.27 LPM>20 0.33 ^NBD 0.34
Program summary
Programming language Java
Program length 365
Comment Rate 0.00
Maintainability 79.00
Textual code compl. 56.45
File complexity 67.00
Method complexity 4.67
Average nesting 2.97
Maximal nesting 7
Branch Percentage 0.18
Maximal coupling 3
Code distribution
Number of files 1
Lines per file (avg) 365
Lines per file (max) 365
Number of methods 18
Lines per method (avg) 19
Lines per method (max) 117
profile created on 07.12.2007



Program summary

Programming language	Java
Program length	528
Comment Rate	0.13
Maintainability	98.00
Textual code compl.	40.09
File complexity	10.85
Method complexity	2.68
Average nesting	2.12
Maximal nesting	5
Branch Percentage	0.14
Maximal coupling	7

Code distribution

Number of files	7
Lines per file (avg)	75
Lines per file (max)	207
Number of methods	41
Lines per method (avg)	12
Lines per method (max)	77

Diagram data

^MI	0.10	^ECC_N	0.22	ECC>10	0.02	^BP	0.20
^HV_N	0.22	^ECC_F	0.04	LPM>20	0.12	^NBD	0.21

Program summary

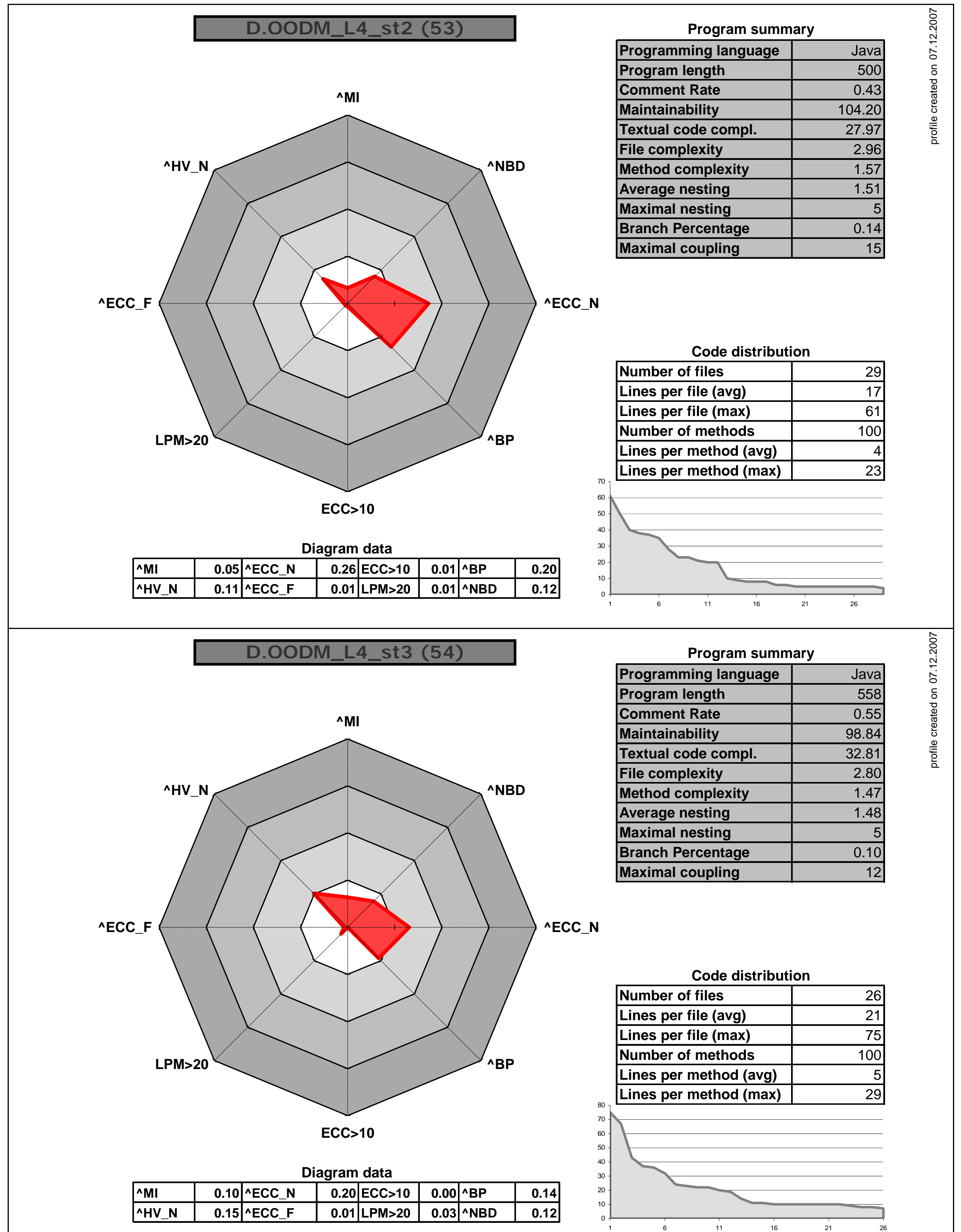
Programming language	Java
Program length	676
Comment Rate	0.45
Maintainability	100.50
Textual code compl.	21.71
File complexity	4.04
Method complexity	1.77
Average nesting	1.85
Maximal nesting	5
Branch Percentage	0.24
Maximal coupling	13

Code distribution

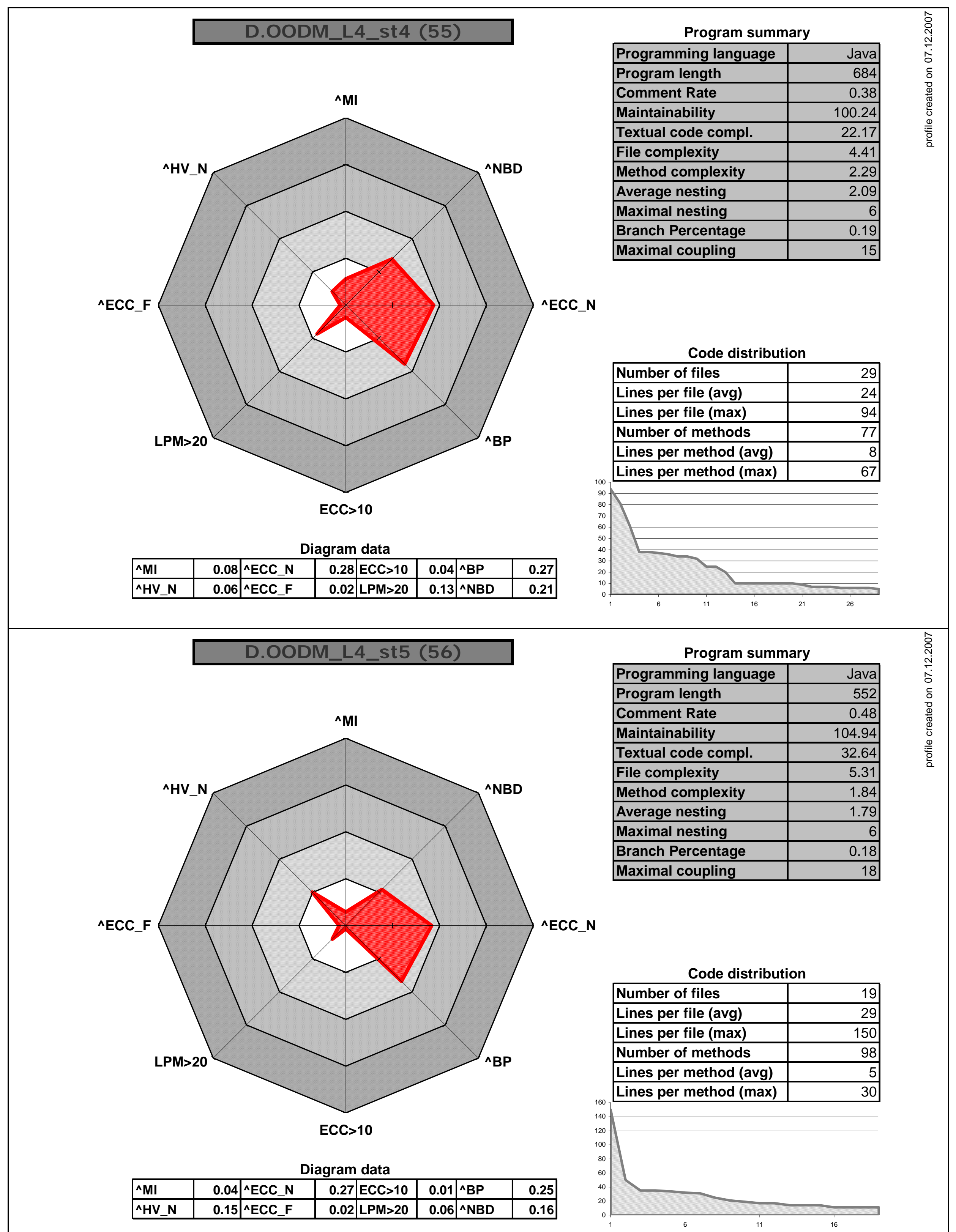
Number of files	22
Lines per file (avg)	31
Lines per file (max)	126
Number of methods	87
Lines per method (avg)	7
Lines per method (max)	26

Diagram data

^MI	0.08	^ECC_N	0.20	ECC>10	0.00	^BP	0.34
^HV_N	0.06	^ECC_F	0.02	LPM>20	0.05	^NBD	0.17



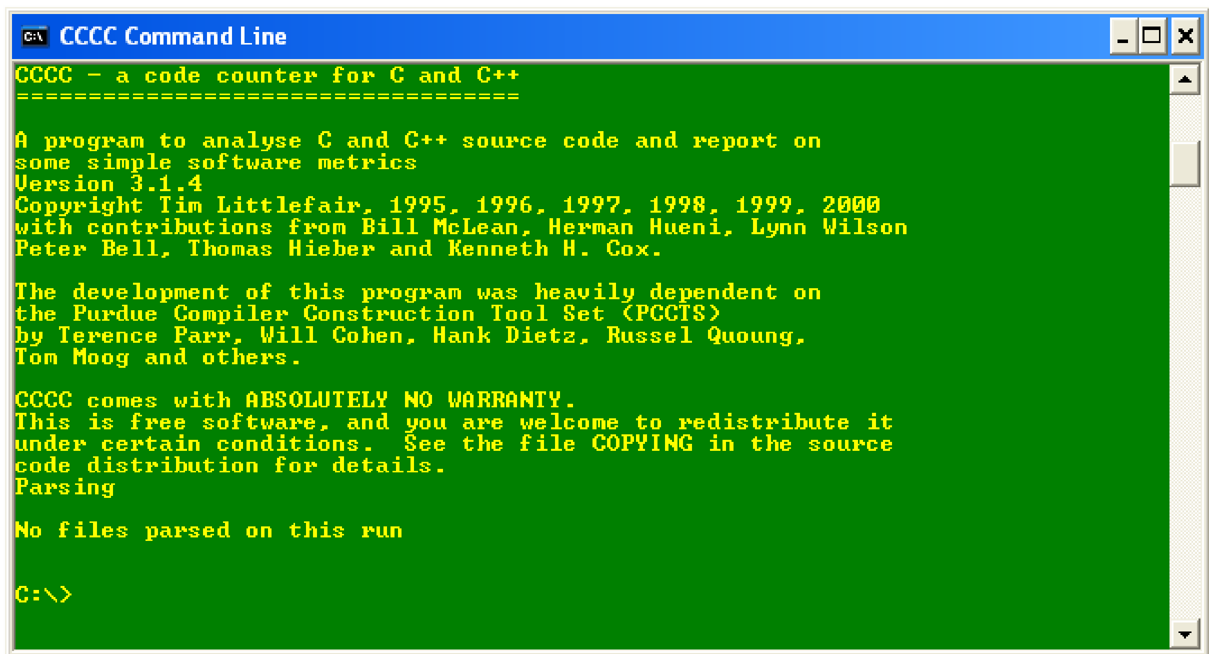
D.OODM_L4_st2 (53)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.05 ^ECC_N 0.26 ECC>10 0.01 ^BP 0.20
^HV_N 0.11 ^ECC_F 0.01 LPM>20 0.01 ^NBD 0.12
Program summary
Programming language Java
Program length 500
Comment Rate 0.43
Maintainability 104.20
Textual code compl. 27.97
File complexity 2.96
Method complexity 1.57
Average nesting 1.51
Maximal nesting 5
Branch Percentage 0.14
Maximal coupling 15
Code distribution
Number of files 29
Lines per file (avg) 17
Lines per file (max) 61
Number of methods 100
Lines per method (avg) 4
Lines per method (max) 23
profile created on 07.12.2007
D.OODM_L4_st3 (54)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.10 ^ECC_N 0.20 ECC>10 0.00 ^BP 0.14
^HV_N 0.15 ^ECC_F 0.01 LPM>20 0.03 ^NBD 0.12
Program summary
Programming language Java
Program length 558
Comment Rate 0.55
Maintainability 98.84
Textual code compl. 32.81
File complexity 2.80
Method complexity 1.47
Average nesting 1.48
Maximal nesting 5
Branch Percentage 0.10
Maximal coupling 12
Code distribution
Number of files 26
Lines per file (avg) 21
Lines per file (max) 75
Number of methods 100
Lines per method (avg) 5
Lines per method (max) 29
profile created on 07.12.2007

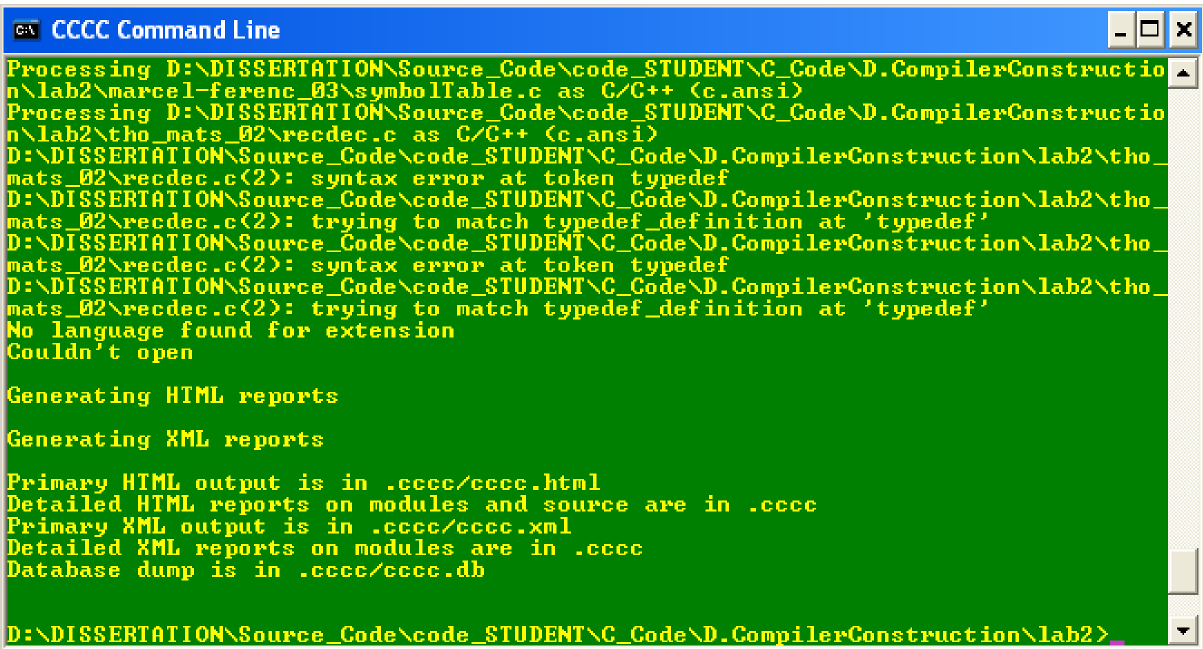


D.OODM_L4_st4 (55)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.08 ^ECC_N 0.28 ECC>10 0.04 ^BP 0.27
^HV_N 0.06 ^ECC_F 0.02 LPM>20 0.13 ^NBD 0.21
Program summary
Programming language Java
Program length 684
Comment Rate 0.38
Maintainability 100.24
Textual code compl. 22.17
File complexity 4.41
Method complexity 2.29
Average nesting 2.09
Maximal nesting 6
Branch Percentage 0.19
Maximal coupling 15
Code distribution
Number of files 29
Lines per file (avg) 24
Lines per file (max) 94
Number of methods 77
Lines per method (avg) 8
Lines per method (max) 67
profile created on 07.12.2007
D.OODM_L4_st5 (56)
^MI
^NBD
^ECC_N
^BP
ECC>10
LPM>20
^ECC_F
^HV_N
Diagram data
^MI 0.04 ^ECC_N 0.27 ECC>10 0.01 ^BP 0.25
^HV_N 0.15 ^ECC_F 0.02 LPM>20 0.06 ^NBD 0.16
Program summary
Programming language Java
Program length 552
Comment Rate 0.48
Maintainability 104.94
Textual code compl. 32.64
File complexity 5.31
Method complexity 1.84
Average nesting 1.79
Maximal nesting 6
Branch Percentage 0.18
Maximal coupling 18
Code distribution
Number of files 19
Lines per file (avg) 29
Lines per file (max) 150
Number of methods 98
Lines per method (avg) 5
Lines per method (max) 30
profile created on 07.12.2007

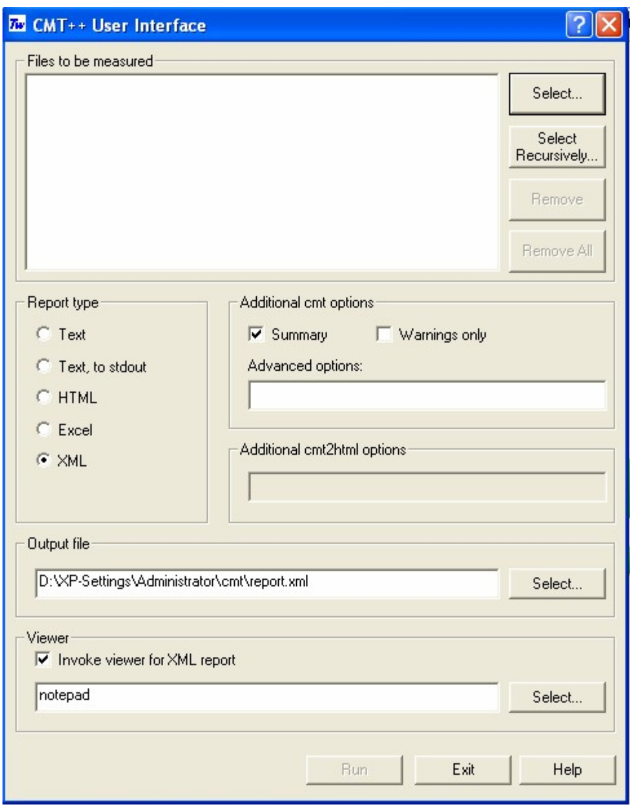
A.5 Measurement tools screenshots

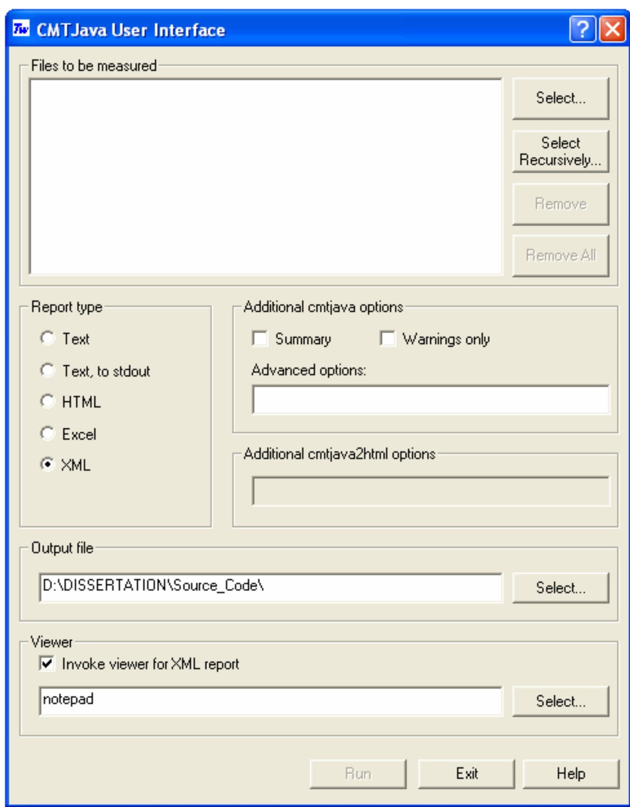
A.5.1 CCCC



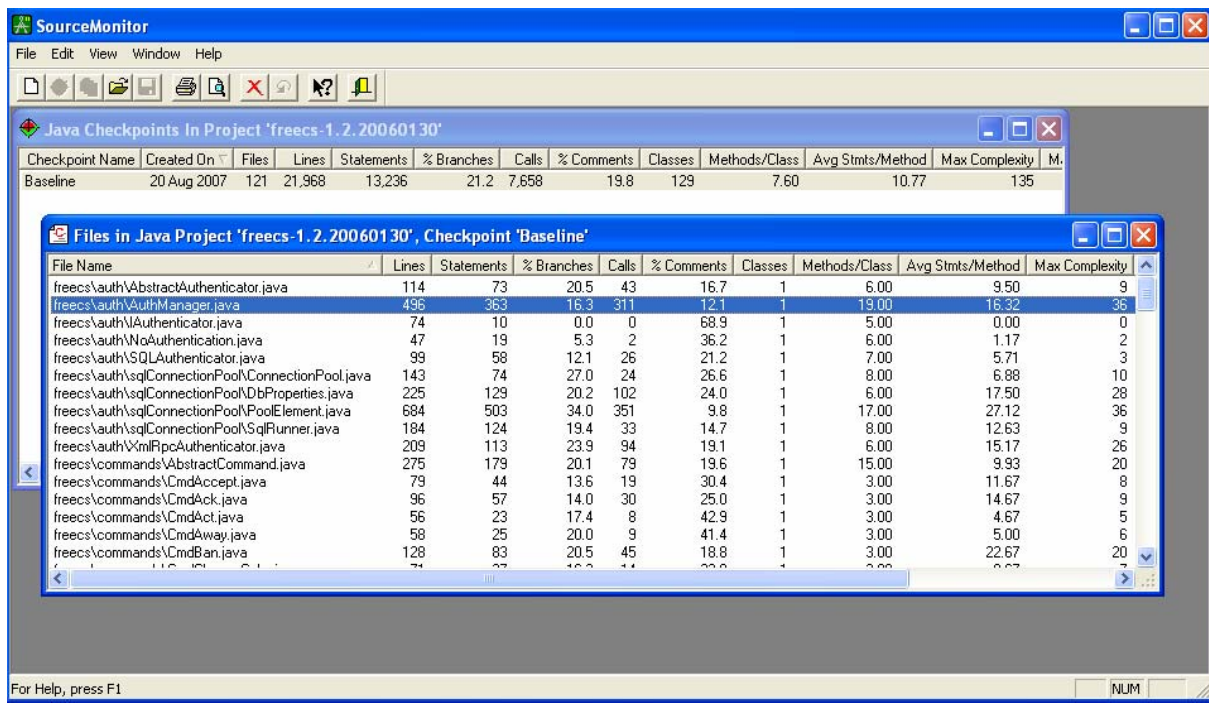


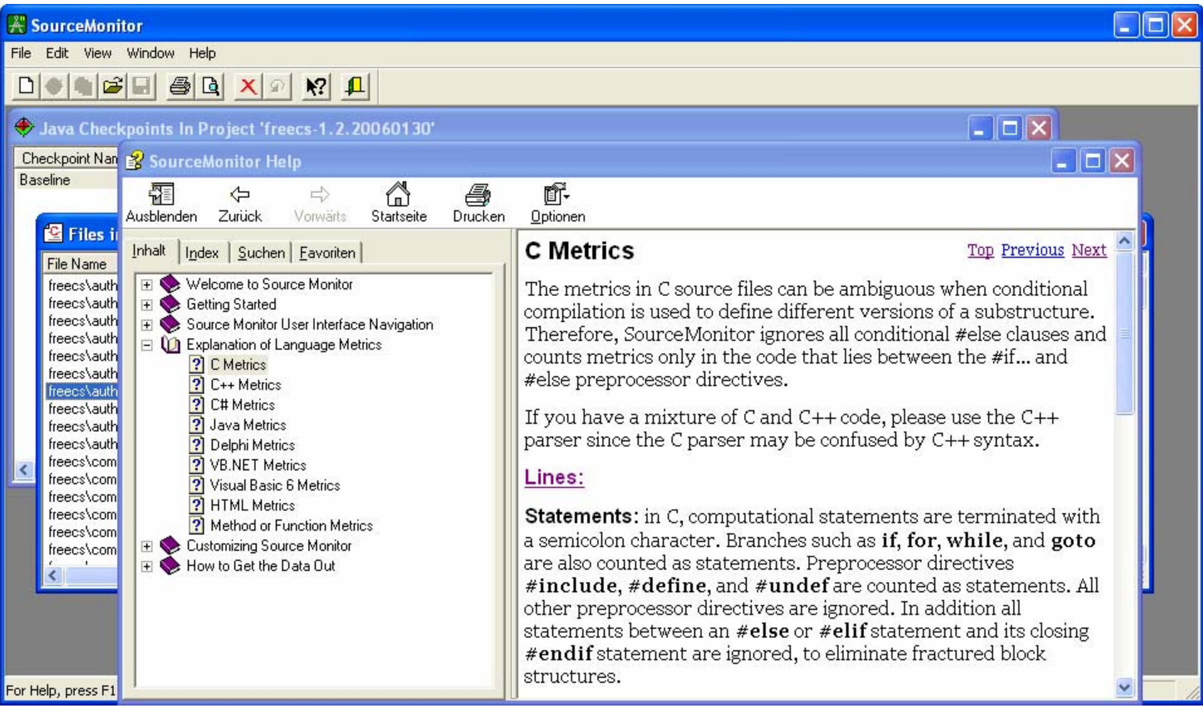
A.5.2 CMT (CMT++ / CMTJava)





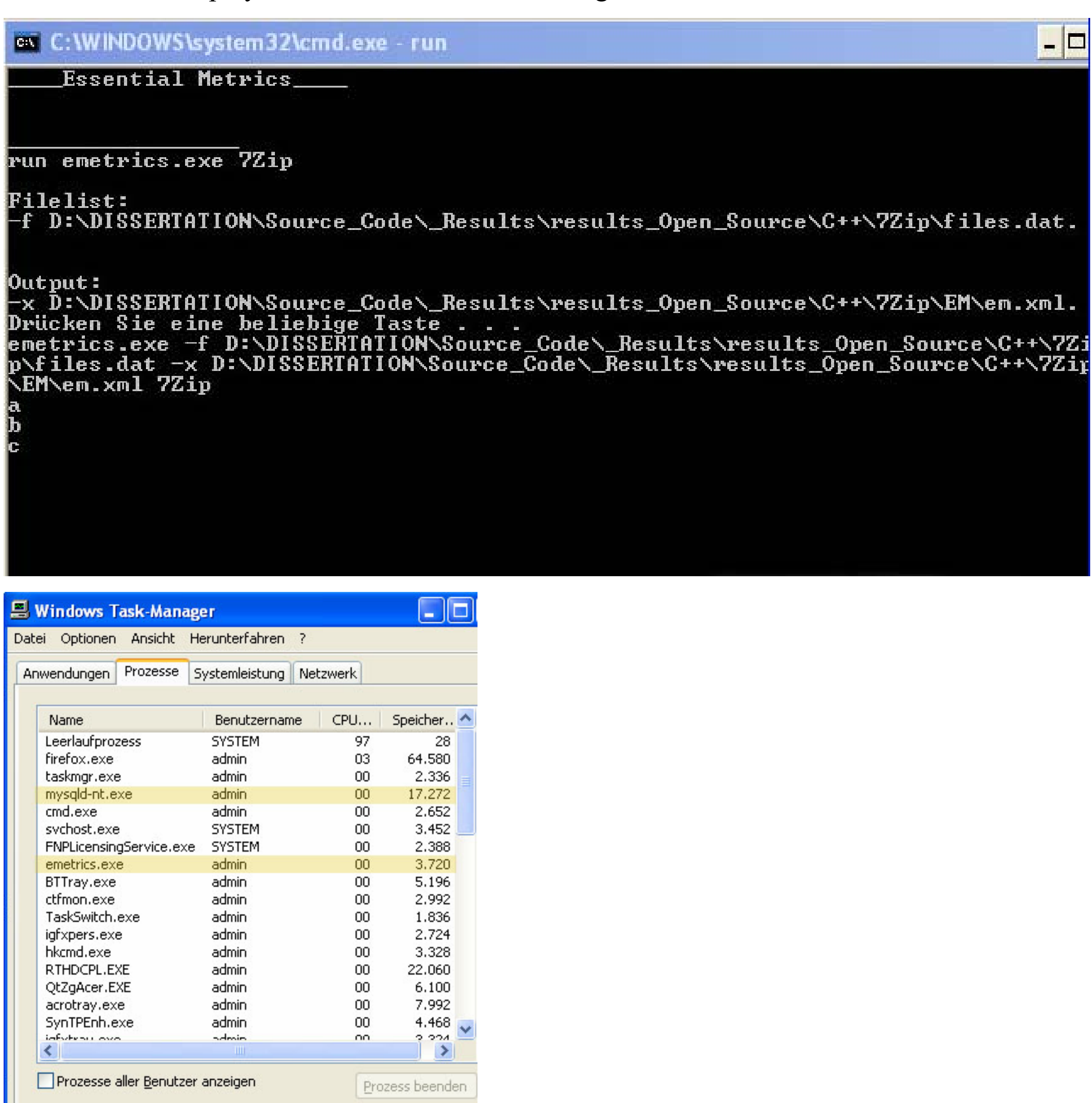
A.5.3 SourceMonitor





A.5.4 Essential Metrics

Essential Metrics displays a blank screen while running.



A.5.5 Logiscope