FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Automated Software Process Performance Analysis and Improvement Recommendation

**Mushtaq Raza**

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MAP-i Doctoral Program in Computer Science

Supervisor: João Pascoal Faria

June 27, 2017

# Automated Software Process Performance Analysis and Improvement Recommendation

**Mushtaq Raza**

MAP-i Doctoral Program in Computer Science

Dissertation submitted to the Faculty of Engineering, University of Porto
in partial fulfillment of the requirements for the degree of
**Doctor of Philosophy**

June 27, 2017

# Abstract

Software development processes can generate significant amounts of data that can be periodically analyzed to identify performance problems, determine their root causes and devise improvement actions. However, there is a lack of methods and tools for helping in that kind of analysis. Conducting the analysis manually is challenging because of the potentially large amount of data to analyze, the effort and expertise required and the lack of benchmarks for comparison.

Hence, the goal of this dissertation is to develop methods, models and tools for automating the analysis of process performance data and identifying and ranking performance problems and their root causes, reducing effort and errors and improving user satisfaction as compared to previous approaches.

The main contributions of the dissertation are a novel method for process performance analysis and improvement recommendation (the ProcessPAIR method), a support tool (the ProcessPAIR tool), and a performance model for instantiating ProcessPAIR for the Personal Software Process (the ProcessPAIR model for the PSP).

In the ProcessPAIR method, the analysis of the performance data of an individual process user is based on a performance model that is defined by an expert in the process under consideration and calibrated automatically from the performance data of many process users. To enable the automatic identification of performance problems, the process expert has to define the relevant (top-level) performance indicators (PIs); recommended performance ranges, needed for classifying and ranking the values of PIs according to three semaphors (red, yellow or green), are derived from calibration data sets. To enable the automatic identification of potential root causes, the process expert has to indicate hierarchical cause-effect relationships between the top-level and more detailed performance indicators. To rank the identified root causes, a novel ranking method is proposed based on the computation of a ranking coefficient that represents a cost-benefit estimate of improvement efforts; the needed data comes from the training data set (statistical distribution of individual PIs and regression models between related PIs).

The support ProcessPAIR tool is able to automatically identify and rank performance problems and their potential root causes, so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused. Performance models for a process under consideration are defined as tool extension and calibrated automatically based on the performance data of many process users. The results of the analysis of the performance data of an individual process user are shown in multiple views.

A performance model was developed to enable the application of ProcessPAIR for the Personal Software Process (PSP). The model addresses time estimation accuracy, quality, and productivity as top-level PIs. A PSP data set from the Carnegie Mellon Software Engineering Institute, referring to more than 30,000 projects, was used to validate and calibrate the model.

Two experiments were conducted to validate the approach. In a postmortem experiment in which we compared the results of manual performance analysis (by 20 students from Instituto Tecnológico de Monterrey in Mexico) and automatic performance analysis, ProcessPAIR was able to accurately identify performance problems in 96% of the cases (292 out of 302 cases), and root causes or intermmediate causes in 74% of the cases (the other 26% cases were not conclusive). A controlled experiment involving 61 software engineering master students also from Instituto Tecnologico de Monterrey in Mexico, half of whom used ProcessPAIR in a PSP performance analysis assignment, showed significant benefits in terms of students' satisfaction (average score of 4.78 in a 1-5 scale for ProcessPAIR users, against 3.81 for non-ProcessPAIR users), quality of the analysis outcomes (average grades achieved of 88.1 in a 0-100 scale for ProcessPAIR users, against 82.5 for non-ProcessPAIR users), and time required to do the analysis (average of 252 minutes for ProcessPAIR users, against 262 minutes for non-ProcessPAIR users, but with much room for improvement).

# Resumo

Os processos de desenvolvimento de software podem gerar quantidades significativas de dados que podem ser periodicamente analisados para identificar problemas de desempenho, determinar as suas causas raiz e planear ações de melhoria. No entanto, há uma falta de métodos e ferramentas para ajudar nesse tipo de análise. Realizar a análise manualmente é um desafio por causa da quantidade potencialmente grande de dados para analisar, do esforço e conhecimentos necessários e da falta de *benchmarks* para comparação.

Assim, o objetivo desta dissertação é desenvolver métodos, modelos e ferramentas para automatizar a análise de dados de desempenho de processos de desenvolvimento de software e identificar e priorizar os problemas de desempenho e suas causas, reduzindo o esforço e erros e melhorando a satisfação dos utilizadores em comparação com abordagens anteriores.

As principais contribuições da dissertação são um novo método para a análise de desempenho e recomendação de melhorias em processos de desenvolvimento de software (o método ProcessPAIR), uma ferramenta de suporte (a ferramenta ProcessPAIR) e um modelo de desempenho para instanciar ProcessPAIR para o Personal Software Process (o modelo ProcessPAIR para PSP).

No método ProcessPAIR, a análise dos dados de desempenho de um utilizador individual de um processo de desenvolvimento de software é baseada num modelo de desempenho que é definido por um especialista no processo em consideração e calibrado automaticamente a partir dos dados de desempenho de muitos utilizadores do processo. Para permitir a identificação automática de problemas de desempenho, o especialista no processo tem de definir os indicadores de desempenho relevantes (de nível de topo); os intervalos de desempenho necessários para classificar os valores de cada indicador de acordo com três semáforos (vermelho, amarelo ou verde) são derivados de conjuntos de dados de calibração. Para permitir a identificação automática de possíveis causas raiz, o perito no processo deve indicar relações de causa-efeito hierárquicas entre os indicadores de desempenho de nível superior e mais detalhados. Para classificar as causas raiz identificadas, um novo método de priorização é proposto com base no cálculo de um coeficiente que representa uma estimativa de custo-benefício dos esforços de melhoria; os dados necessários provêm do conjunto de dados de calibração (distribuição estatística de indicadores individuais e modelos de regressão entre indicadores relacionados).

A ferramenta ProcessPAIR de suporte é capaz de identificar e priorizar automaticamente os problemas de desempenho e as possíveis causas raiz, de modo que a análise manual subsequente para a identificação de causas mais profundas e ações de melhoria pode ser devidamente focada. Modelos de desempenho para um processo em consideração são definidos como extensões da ferramenta e calibrados automaticamente com base nos dados de desempenho de muitos utilizadores do processo. Os resultados da análise dos dados de desempenho de um utilizador individual são apresentados em múltiplas vistas.

Um modelo de desempenho foi desenvolvido para permitir a aplicação de ProcessPAIR para o Personal Software Process (PSP). O modelo aborda a precisão das estimativas de tempo, qualidade e produtividade como indicadores de nível de topo. Um conjunto de dados de PSP do Carnegie

Mellon Software Engineering Institute, referindo-se a mais de 30.000 projetos, foi usado para validar e calibrar o modelo.

Duas experiências foram conduzidas para validar a abordagem. Numa experiência *post-mortem*, em que foram comparados os resultados da análise de desempenho manual (por 20 estudantes do Instituto Tecnológico de Monterrey no México) e da análise automática, a ferramentas ProcessPAIR conseguiu identificar com precisão os problemas de desempenho em 96% dos casos (292 de 302 casos) e causas raiz ou causas intermediárias em 74% dos casos (os outros 26% casos não foram conclusivos). Uma experiência controlada envolvendo 61 estudantes de mestrado em engenharia de software do Instituto Tecnológico de Monterrey no México, metade dos quais utilizou o ProcessPAIR num trabalho de análise de desempenho PSP, mostrou benefícios significativos em termos de satisfação dos alunos (valor médio de 4,78 numa escala de 1-5 para utilizadores de ProcessPAIR e 3,81 para os outros utilizadores ), qualidade dos resultados da análise (média de 88,1 numa escala de 0-100 para os utilizadores de ProcessPAIR, contra 82,5 para outros utilizadores) e tempo necessário para fazer a análise (média de 252 minutos para utilizadores de ProcessPAIR, contra 262 minutos para outros utilizadores, mas com muito espaço para melhoria).

# Acknowledgments

For me this thesis is not only a representation of my academic work that I have performed at the Department of Informatics Engineering but also a representation of five years spent away from home in Porto. Producing this thesis wouldn't have been possible without the help, support, encouragement and love of many people both in professional and personal spheres.

First and foremost, I want to thank my supervisor and mentor Professor João Pascoal Faria for believing in me, and for providing continuous guidance and support. His scholarly advice, and scientific approach has been an inspiration and beacon of light for me throughout my work. I have learnt a lot from the excellent example he has provided as a software engineer, manager, and professor.

In addition, I want to thank Professor Gabriel David, who introduced me to the Department of Informatics Engineering, FEUP and later helped me in selecting my supervisor that was the first successful step in my PhD journey. I also want to thank Professor Raul Moreira Vidal for being a constant support. I am deeply indebted to Mrs. Marisa Silva for her support in administrative tasks, which made my work a lot easier.

I also want to thank all the technical and administrative staff members of Faculty of Engineering, University of Porto, for their kind help and co-operation throughout my study period, and for making me feel at home.

I would like to acknowledge the funding and data sources that made my Ph.D. work possible. SEI - Software Engineering Institute and Tecnológico de Monterrey for giving access to the data for performing this study. FCT - Fundação para a Ciência e a Tecnologia for supporting this study under research grant SFRH/BD/85174/2012 and as part of project UID/EEA/50014/2013. Abdul Wali Khan University Mardan for their partial initial grant. Additionally, this research has been partially supported by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), in the scope of the BEST CASE program.

I would like to thank my family for their unconditional love and support over the past five years. To my mother and father, for all they have given me and for making me the person I am today. To my mother-in-law and father-in-law for their support. To my brothers and sister, specifically, to Hassan Raza for keeping me informed about matters back home in Kokarai, Swat Pakistan. To my uncle, Sanaullah, whose words of encouragement and motivation have kept me going.

I would also like to thank my friends who were always by my side, for their encouragement and motivation. To Muhammad Ali Khan, Bruno Lima, Arsalan, Muhammad Asif, Muhammad Ajmal, Bilal, Saad Sultan, David, André Pilastri, Ali Awan, Niaz Ali Khan, Aftab Rashid, Alam, Saqlain, Arif, Farhan, Muhammad Farooq, Zahid Iqbal, Asad, and Kashif Mushtaq for making this academic adventure more enjoyable. I genuinely hope your friendship will accompany me much further in the future.

Last but not the least, I want to thank my wife Palwasha for her love, support and for making me feel at home thousands of miles away from my family, and for taking care of our son, Muwahid while I worked long hours.

Mushtaq Raza

*“You should be glad that bridge fell down.*
*I was planning to build thirteen more to that same design”*

Isambard Kingdom Brunel

# Contents

# List of Figures

# List of Tables

# Abbreviations and Symbols

| | |
|---|---|
| A/FR | Appraisal to Failure Ratio |
| CAR | Causal Analysis and Resolution |
| CMMI | Capability Maturity Model Integration |
| CMU | Carnegie Mellon University |
| COQ | Cost of Quality |
| CR | Code Review |
| DDC | Defect Density in Compile |
| DDUT | Defect Density in Unit Test |
| DI | Defects Injected |
| DLDR | Detailed-Level Design Review |
| DRL | Defect Removal Leverage |
| DRR | Defect Removal Rate |
| EO | Extraneous Objects |
| LOC | Line of Code |
| LPI | Lower Prediction Interval |
| MA | Measurement and Analysis |
| MO | Missing Objects |
| OEA | Objects Estimation Accuracy |
| OPM | Organization Process Management |
| OPP | Organization Process Performance |
| PEA | Productivity Estimation Accuracy |
| PI | Performance Indicator |
| PIP | Process Improvement Proposal |
| PM | Performance Model |
| PM | Postmortem |
| PPB | Process Performance Baseline |
| PPM | Process Performance Model |
| PPS | Project Plan Summary |
| PQI | Process Quality Index |
| PROBE | Proxy-based Estimating |
| PSP | Personal Software Process |
| PY | Process Yield |
| SEA | Size Estimation Accuracy |
| SEI | Software Engineering Institute |
| TEA | Time Estimation Accuracy |
| TSP | Team Software Process |
| UPI | Upper Predication Interval |
| UT | Unit Test |
| XML | Extensible Markup Language |

# Chapter 1

# Introduction

## 1.1 Motivation

According to Bohem (Boehm, 2011), the top two software engineering challenges are the increasing emphasis on rapid development and adaptability, and the increasing software criticality and need for assurance.

The need to ensure the quality of software products in a cost effective way drives companies and organizations to seek to improve their software development process, as it is becoming more and more accepted in industrial production in general and in the software industry in particular that the quality of the process directly affects the quality of the product(Kenett and Baker, 1999) “There is a very strong link between software quality and the processes used to develop it.” (Kenett and Baker, 1999).

The Team Software Process (TSP) (Humphrey, 2005) and the accompanying Personal Software Process (PSP) (Humphrey and Over, 2010) are examples of processes that can help individuals and teams improve their performance and produce virtually defect free software on time and budget (Rombach et al., 2008), addressing those challenges. One of the pillars of the TSP/PSP is its measurement framework; based on four simple measures—effort, schedule, size, and defects—it supports several quantitative methods for project and quality management and process improvement (Humphrey, 2005).

“Measurements are key. If you cannot measure it, you cannot control it. If you cannot control it, you cannot manage it. If you cannot manage it, you cannot improve it.”(Harrington et al., 1991)

Software process improvement and measurement go hand in hand: measures are the only way to prove improvements in a process.

Software processes that make intensive use of metrics and quantitative methods, such as the TSP/PSP, can generate large amounts of data that can be periodically analyzed to identify performance problems, determine their root causes, and devise improvement actions (Daniel Burton, 2006).

The manual analysis of performance data for determining root causes of performance problems and devising improvement actions is challenging because of the potentially large amount of data to analyze (Daniel Burton, 2006), the effort and expert knowledge required, and the lack of benchmarks for comparison.

Although several tools exist to automate data collection and produce performance charts and reports for manual analysis of TSP/PSP data (Nasir and Yusof, 2005), practically, no tool support exists for automating the performance analysis. There are also some studies that show cause–effect relationships among performance indicators (PIs) (Kemerer and Paulk, 2009) (Shen et al., 2011), but no automated root cause analysis is proposed.

## 1.2 Research goals

In this research work, we tackle the challenge of automating the analysis of performance data produced in the context of software development processes for determining performance problems and their root causes and devising improvement actions.

Our thesis statement is that, *by taking advantage of performance models derived from the performance data of many process users, it is possible to automatically analyze the performance data of individual developers and identify and rank performance problems and their root causes, reducing manual effort and errors in performance analysis, and improving user satisfaction.*

The main goal is to develop methods and tools to answer the following research questions (derived from the thesis statement):

*RQ1. Is it possible to automatically identify performance problems of individual developers, by taking advantage of performance models derived from the performance data of many process users?*

*RQ2. Is it possible to automatically identify the root causes of the identified performance problems, by taking advantage of performance models derived from the performance data of many process users?*

*RQ3. Is it possible to automatically rank the identified performance problems, by taking advantage of performance models derived from the performance data of many process users?*

*RQ4. Is it possible to automatically rank the identified root causes, by taking advantage of performance models derived from the performance data of many process users?*

*RQ5. By automating the performance analysis as described in RQ1 to RQ4, is it possible to reduce effort and errors and improve user satisfaction as compared to manual analysis?*

To achieve the main goal and prove the thesis, the following tasks are considered:

- to construct and validate one or more performance models (defining performance indicators, cause-effect relationships between them, and recommended ranges or thresholds) that can be used to automatically identify performance problems and their root causes in software development organizations using medium to high-maturity processes such as, but not limited to, the PSP/TSP; we take advantage of a partnership with the Software Engineering Institute to have access to PSP/TSP data to calibrate the performance model;

- to develop and validate algorithms for automatically evaluating and ranking (prioritizing) performance problems and their root causes;

- to design and develop a tool for automating the performance analysis and improvement recommendation, flexible enough to accept multiple performance models and data sources;

- to conduct experiments in real world contexts, to validate the developed models and tool and show the benefits of the approach.

## 1.3 Research contributions

The main contributions of the research work are:

- the ProcessPAIR method for automated performance analysis and improvement recommendation in software development (described in chapter 3);

- the ProcessPAIR support tool (described in chapter 5 and publicly available in `http://blogs.fe.up.pt/processpair/`);

- the performance models for PSP (described in chapter 4);

- the experimental results of both tool and model (described in chapter 6).

ProcessPAIR is a novel tool for automated performance analysis and improvement recommendation in software development. Based on performance models defined by process experts and calibrated automatically from the performance data of many developers, it automatically identifies and ranks potential performance problems and their root causes of individual developers, so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused.

The PSP (Humphrey, 2005) is a software process, with a stepwise training strategy, designed by the Software Engineering Institute (SEI) to help engineers improve their performance and produce defect-free software (Rong et al., 2016). The availability of a large PSP data set for model calibration purposes, and the lack of adequate tools for helping PSP students analyzing their performance data in the PSP Final Report assignment (see Appendix A), were the main drivers for applying ProcessPAIR for the PSP.

A postmortem experiment in which we compared the results of manual and automated (with ProcessPAIR) performance analysis (Raza et al., 2016), showed that ProcessPAIR is able to accurately identify performance problems and their root causes of PSP developers.

A controlled experiment showed that software engineering students were able to successfully analyze their personal performance data with the help of ProcessPAIR. The results show significant benefits in terms of students' satisfaction (average rating of 4.78 in a 1-5 scale for ProcessPAIR users, against 3.81 for non-ProcessPAIR users), quality of the analysis outcomes (average grades given by instructor of 88.1 in a 0-100 scale for ProcessPAIR users, against 82.5 for non-ProcessPAIR users), and time required to do the analysis (average of 252 minutes for ProcessPAIR users, against 262 minutes for non-ProcessPAIR users, but with significant room for improvement).

Following is the list of articles published from this research work:

- RAZA, Mushtaq; FARIA, João Pascoal. ProcessPAIR: a tool for automated performance analysis and improvement recommendation in software development. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016. p. 798-803.

  This article is about the ProcessPAIR, a novel tool designed to help developers analyze their performance data with less effort, by automatically identifying and ranking performance problems and their potential root causes, so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused. Chapter 5 is based on this article.

- RAZA, Mushtaq; FARIA, João Pascoal; SALAZAR, Rafael. Empirical Evaluation of the ProcessPAIR Tool for Automated Performance Analysis. In The 28th International Conference on Software Engineering and Knowledge Engineering, June 30-July 3, 2016.

  This article is about the results of a postmortem experiment conducted in the context of PSP training, to show that ProcessPAIR is able to accurately identify and rank performance problems and their potential root causes of individual developers so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused. Part of chapter 6 is based on this article. One of the co-authors is Rafael Salazar, director of the Mexican TSP Initiative at Tecnológico de Monterrey in Mexico.

- RAZA, Mushtaq.; FARIA, João Pascoal. A model for analyzing performance problems and root causes in the personal software process. Journal of Software: Evolution and Process, 2015.

  This article (extended version of ICSSP 14 article) is about the PSP performance model, addressing time estimation accuracy, quality, and productivity, to enable the automated (tool based) analysis of performance data produced by PSP developers, namely, identify and rank performance problems and their root causes. A PSP data set referring to more than 30 000 projects was used to validate and calibrate the model. Parts of chapter 3 and 4 are based on this article.

- RAZA, Mushtaq; FARIA, João Pascoal. A Benchmark-based Approach for Ranking Root Causes of Performance Problems in Software Development. In: International Conference on

Product-Focused Software Process Improvement. Springer International Publishing, 2014. p. 314-317.

This article presents an approach for automatically ranking the potential root causes identified during performance analysis of individual process users, based on a cost-benefit estimate that takes into account historical data. The approach was applied for the PSP, taking advantage of a large data set from SEI. Part of chapter 3 of the thesis document is based on this article.

- RAZA, Mushtaq; FARIA, João Pascoal. A model for analyzing estimation, productivity, and quality performance in the personal software process. In: Proceedings of the 2014 International Conference on Software and System Process. ACM, 2014. p. 10-19.

  This article presents a comprehensive performance model, addressing time estimation accuracy, quality and productivity, to enable the automated analysis of performance data produced in the context of the PSP, namely, identify performance problems and their root causes, and subsequently recommend improvement actions. Part of chapter 3 and 4 is based on this article.

- RAZA, Mushtaq; FARIA, João Pascoal. Factors Affecting Personal Software Development Productivity: A Case Study with PSP Data. IASTED SE, 2014.

  This article is about the analysis of factors (personal and non-personal) that affect the productivity of software developers and may cause productivity variations among individuals and projects.

- RAZA, Mushtaq; FARIA, João Pascoal; HENRIQUES, Pedro, NICHOLS, William. Factors affecting productivity performance in PSP training. In: Annual Software Engineering Institute (SEI) TSP Symposium. 2013. p. 35-45.

  This article is an extended version of IASTED SE 2014 article. In this article, we analyzed the personal and non-personal factors that affect productivity performance. Regarding non-personal factors, we found both process changes and program complexity to be important factors explaining productivity variations. Regarding personal factors, we found significant variations among individuals that can be partially explained by personal experience and programming language used. This article is co-authored by William Nichols, senior member of the technical staff and PSP instructor and TSP coach with the Team Software Process Program at Software Engineering Institute, CMU, and Pedro Henriques, CEO of Strongstep- Innovation in software quality.

- DUARTE, César Barbosa; FARIA, João Pascoal; RAZA, Mushtaq. PSP PAIR: automated personal software process performance analysis and improvement recommendation. In: Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the. IEEE, 2012. p. 131-136.

This article is about initial work on performance model (limited to the analysis of the time estimation performance of PSP developers) and tool prototype to automate the analysis of performance data produced in the context of the PSP, namely, identify performance problems and their root causes, and recommend improvement actions.

- DUARTE, César Barbosa; FARIA, João Pascoal; RAZA, Mushtaq; HENRIQUES, Pedro. Model and Tool for Analyzing Time Estimation Performance in PSP. In: Annual Software Engineering Institute (SEI) TSP Symposium. 2012. p. 21-40.

  This article (extended version of QUATIC 12) presents a performance model and a tool prototype (limited to the analysis of the time estimation performance of PSP developers) to automate the analysis of performance data produced in the context of the PSP , namely, identify performance problems and their root causes, and recommend improvement actions.

- RAZA, Mushtaq; FARIA, João Pascoal; SALAZAR, Rafael. Helping Software Engineering Students Analyzing their Performance Data: Tool Support in an Educational Environment. In: International Conference on Software Engineering (ICSE). May 20-28, 2017.

  This article is about the results of a controlled experiment involving 61 software engineering master students, half of whom used ProcessPAIR in a PSP performance analysis assignment. Part of chapter 6 is based on this article. One of the co-authors is Rafael Salazar, director of the Mexican TSP Initiative at Tecnológico de Monterrey in Mexico.

- RAZA, Mushtaq; FARIA, João Pascoal; AMARO, Luis; HENRIQUES, Pedro Castro. WebProcessPAIR: Recommendation System for Software Process Improvement. In: International Conference on Software and System Processes (ICSSP). July 5-7, 2017.

  This article is about WebProcessPAIR, which extends ProcessPAIR with the ability to recommend improvement actions to address the root causes identified, based on a crowdsourcing approach.

## 1.4 Document structure

The rest of the thesis document is organized as follows.

Chapter 2 describes state of the art analysis and is organized as follows. Section 2.1 describes performance measurement and analysis in CMMI, covering CMMI practices related to process performance analysis. Section 2.2 describes performance measurement and analysis in PSP, covering process structure, measurement framework, final report assignment, and tool support for PSP. Section 2.3 presents performance measurement and analysis in other process. Section 2.4 presents performance problem identification techniques, covering control charts and benchmark based software evaluation. Section 2.5 explains root cause analysis techniques, covering fishbone diagrams, defect causal analysis, process performance and regression models, and sensitivity analysis.

Chapter 3 describes the proposed performance analysis method and is organized as follows. Section 3.1 introduces the overall approach, covering problem and root causes identification and ranking. Section 3.2 describes performance model definition. Section 3.3 describes the performance model calibration, covering approximate cumulative distribution functions, performance ranges, regression models and sensitivity coefficients, and data set filtering. Section 3.4 presents model based performance analysis.

Chapter 4 covers the performance model for PSP and is organized as follows. Section 4.1 introduces model definition for PSP and covers performance indicators and dependencies for predictability, quality, and productivity. Section 4.2 describes model validation and calibration, covering available data set, model validation, and model calibration. In section 4.3, support data for the ranking method is presented which covers sensitivity and percentile coefficients together with ranking example.

Chapter 5 describes the ProcessPAIR tool implementation and is organized as follows. Section 5.1 presents the architecture of the tool. Sections 5.2 and 5.3 explain the model calibration, and performance analysis user interfaces. Section 5.4 presents analysis views: report, table, indicator and cause-effect. Finally, section 5.5 presents the tool extension API.

Chapter 6 describes the postmortem and the controlled experiments. Regarding the postmortem experiment, section 6.1.1 presents the research questions to be answered in the experiment. Section 6.1.2 presents the input data collected for the analysis. Section 6.1.3 and 6.1.4 present the data analysis procedures and the results obtained. Section 6.1.5 and 6.1.6 present the overall statistics and discussion. Finally, section 6.1.7 presents limitations and threats to validity. Regarding the controlled experiment section 6.2.1 presents the context of the experiment, section 6.2.2 presents the ProcessPAIR tuning for this experiment, section 6.2.3 presents the experiment design, section 6.2.4 and 6.2.5 present the experimental results and discussion, and section 6.2.6 presents threats to validity.

Chapter 7 presents the conclusions of this research work.

Appendix A presents a tutorial on the PSP Final Report with ProcessPAIR.

# Chapter 2

# State of the art

Our approach draws a strong inspiration from process performance models (PPM) and performance measurement and analysis in the context of CMMI and TSP/PSP.

A PPM in the context of CMMI is a description of the relationship among attributes of a process or sub-process and its outcomes, developed from historical performance data, and calibrated using collected process and product measures (Chrissis et al., 2011). The purpose of using measurement and analysis in the context of CMMI and TSP/PSP is to measure project progress, product size or process performance in support of making decisions and taking corrective action.

This chapter is organized as follows. Section 2.1 describes performance measurement and analysis in CMMI, covering CMMI practices related to process performance analysis. Section 2.2 describes performance measurement and analysis in PSP, covering process structure, measurement framework, final report assignment, and tool support for PSP. Section 2.3 presents performance measurement and analysis in other processes. Section 2.4 presents performance problem identification techniques, covering control charts and benchmark based software evaluation. Section 2.5 explains root cause analysis techniques, covering fishbone diagrams, defect causal analysis, process performance and regression models, and sensitivity analysis.

## 2.1 Performance measurement and analysis in CMMI

Since the CMMI is a recognized model of software engineering best practices, it is relevant to look at the best practices for performance analysis in software engineering embodied in the CMMI.

### 2.1.1 Overview

CMMI (Chrissis et al., 2011) stands for Capability Maturity Model Integration and was developed by the Software Engineering Institute (SEI). It is a process improvement maturity model.

All CMMI models are produced from the CMMI Framework (a collection of all model components, training material components, and appraisal components), containing 16 core process areas. These process areas cover basic concepts that are fundamental to process improvement in any area of interest. Also these areas are shared among three groups: CMMI for Development (CMMI-DEV), CMMI for Services (CMMI-SVC), and CMMI for Acquisition (CMMI-ACQ).

The CMMI-DEV model (Chrissis et al., 2011) provides guidance for applying CMMI best practices in a development organization. These best practices focus on activities for developing quality services and products to meet the needs of end users and customers. The CMMI-DEV identifies 22 process areas given in Figure 2.1.

Each process area is described by a set of specific goals and practices, which describe activities that implement the process area. Generic goals and practices apply to all process areas, and describe activities that institutionalize the process areas (see Figure 2.2).

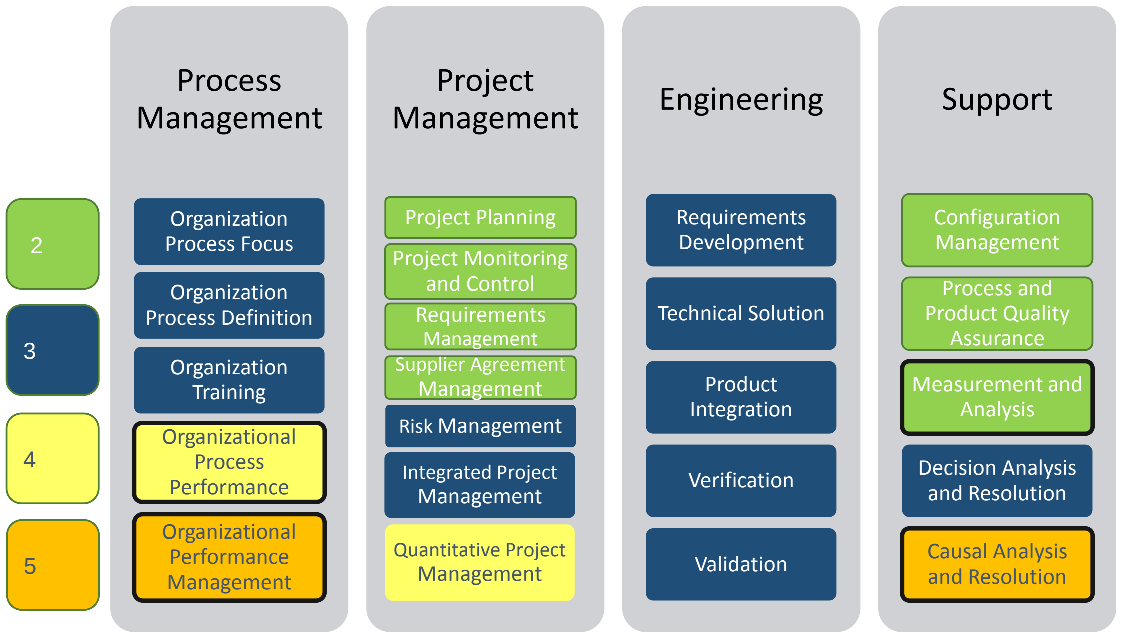

Process Management
Project Management
Engineering
Support
2
3
4
5
Organization Process Focus
Organization Process Definition
Organization Training
Organizational Process Performance
Organizational Performance Management
Project Planning
Project Monitoring and Control
Requirements Management
Supplier Agreement Management
Risk Management
Integrated Project Management
Quantitative Project Management
Requirements Development
Technical Solution
Product Integration
Verification
Validation
Configuration Management
Process and Product Quality Assurance
Measurement and Analysis
Decision Analysis and Resolution
Causal Analysis and Resolution

Figure 2.1: CMMI process areas organized by categories and maturity levels.

### 2.1.2 CMMI practices related to process performance analysis

Table 2.1 shows the specific practices and their position in the CMMI model in the scope of this research work.

This research work is first related to OPM SP 1.2, OPM SP 1.3, and CAR SP 1.2, because the goal is to analyze process performance data (OPM SP 1.2) of a given subject (individual team or organization) for the identification of performance problems (OPM SP 1.3) and root causes (CAR SP 1.2).

In second place, it is related to OPP SP 1.3, OPP SP 1.4 and OPP SP 1.5, because, to achieve the stated goals, we have to define the relevant performance measures (OPP SP 1.3), and performance ranges taking advantage of historical data from many subjects (OPP SP 1.4), and also establish relevant cause-effect relationships from that data (OPP SP 1.5).

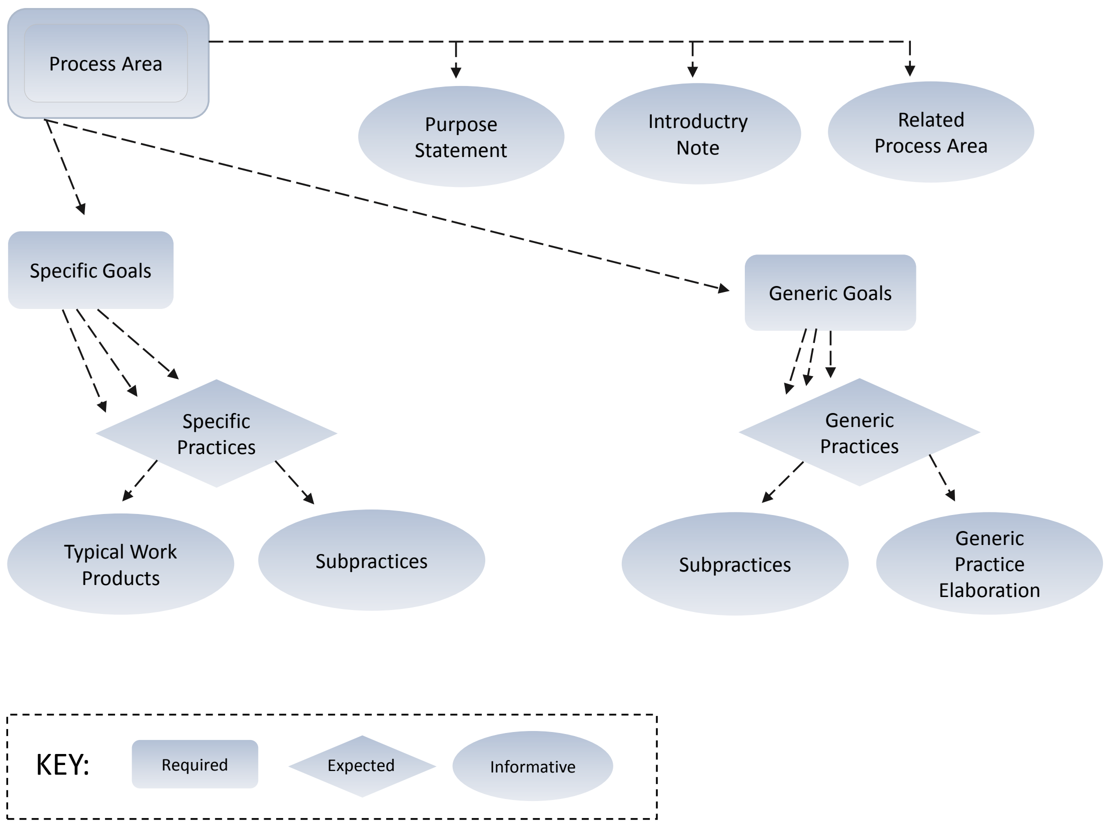


Figure 2.2: CMMI process area decomposition.

MA SP 1.2 and MA SP 2.2 are support practices needed to achieve the aforementioned practices, so this research work is also indirectly related with them.

The purpose of using Measurement and Analysis (MA) (Chrissis et al., 2011) is to measure project progress, product size or process performance in support of making decisions and taking corrective action. Basically, MA comprises the specification of measures, mechanisms for the collection of data, analysis techniques for the collected data, data storage, reporting and feedback. The goal of analysis is to provide objective results that can be used for decision making and taking appropriate corrective action. The practices performed in MA are given in Table 2.1.

The purpose of Organizational Process Performance (OPP) (Chrissis et al., 2011) is to establish and maintain quantitative understanding of the performance of selected processes in the organization's set of standard processes in support of achieving quality and process performance objectives, and to provide process performance data, baseline, and model to quantitatively manage the organization's projects. The practices performed in OPP are given in Table 2.1.

The OPP includes specific practices (see Table 2.1) to establish Process Performance Baselines (PPBs). PPBs are important for organizations because when the organization has enough measures, data, and analytical techniques for critical process, product, and service characteristics, it can be able to determine whether the processes are behaving consistently or not. It can identify:

- processes in which the performance is within natural bounds;

- processes that show unusual behavior;
- processes or aspects of processes that can be improved in the organization.

The Organizational Performance Management (OPM) (Chrissis et al., 2011) process area enables the organization to manage organizational performance by iteratively analyzing aggregated project data, identifying gaps in performance against the business objectives, and selecting and deploying improvements to close the gaps. Process performance baselines and process performance models, developed using Organizational Process Performance processes, are used as part of the analysis. The practices performed in OPM are given in Table 2.1.

The purpose of Causal Analysis and Resolution (CAR) (Chrissis et al., 2011) is to identify causes of selected outcomes and take action to improve process performance. Causal analysis and resolution improves quality and productivity by preventing the introduction of defects or problems and by identifying and appropriately incorporating the causes of superior process performance. The practices performed in CAR are given in Table 2.1.

The CMMI is a model of best practices that suggest practices to be implemented (the *what*), but does not describe corresponding operational procedures (the *how*). In the next section, we analyze a process that provides some of those operational procedures.

## 2.2 Performance measurement and analysis in PSP

The Personal Software Process (PSP), developed by Watt S. Humphrey and the Software Engineering Institute (SEI), Carnegie Mellon University, is a framework for improving the overall quality of software development processes for individuals (Humphrey, 2005).

The PSP heavily relies on the collection and analysis of personal data in order to demonstrate to software engineers' the way they develop software, weaknesses and strengths.

The main measurements which are basically collected in the PSP include the size of the software, development time, and number of defects. Next sections will explain in detail all these measurements (base and derived) and their collection process.

### 2.2.1 Overview

The PSP is a self-improvement process that helps developers in improving, controlling and managing their way of working. It is an organized process framework of procedures, guidelines and forms for developing software.

Using PSP in an appropriate way leads to the data that developers need for making and meeting commitments; furthermore, it makes the routine elements of developers' job more efficient and predictable.

The core purpose of the PSP is to guide developers in defining their own processes, planning and tracking their own work and managing the quality of the products they produce. It also helps developers to manage their work, assess their talents, and build their skills.

The PSP makes available for software engineers:

Table 2.1: CMMI practices in the scope of this work (highlighted) and their positioning in the CMMI model.

| | | |
|---|---|---|
| **MA: Measurement and Analysis (ML 2)** | | |
| | SG 1 Align Measurement and Analysis Activities | |
| | | SP 1.1 Establish Measurement Objectives |
| | | **SP 1.2 Specify Measures** |
| | | SP 1.3 Specify Data Collection and Storage Procedures |
| | | SP I.4 Specify Analysis Procedures |
| | SG 2 Provide Measurement Results | |
| | | SP 2.1 Obtain Measurement Data |
| | | **SP 2.2 Analyze Measurement Data** |
| | | SP 2.3 Store Data and Results |
| | | SP 2.4 Communicate Results |
| **OPP: Organizational Process Performance (ML4)** | | |
| | SG 1 Establish Performance Baselines and Models | |
| | | SP 1.1 Establish Quality and Process Performance Objectives |
| | | SP 1.2 Select Processes |
| | | **SP 1.3 Establish Process Performance Measures** |
| | | **SP 1.4 Analyze Process Performance and Establish Process Performance Baselines** |
| | | **SP 1.5 Establish Process Performance Models** |
| **OPM: Organizational Performance Management (ML5)** | | |
| | SG 1 Manage Business Performance | |
| | | SP 1.1 Maintain Business Objectives |
| | | **SP 1.2 Analyze Process Performance Data** |
| | | **SP 1.3 Identify Potential Areas for Improvement** |
| | SG 2 Select Improvements | |
| | SG 3 Deploy Improvements | |
| **CAR: Causal Analysis and Resolution (ML5)** | | |
| | SG 1 Determine Causes of Selected Outcomes | |
| | | SP 1.1 Select Outcomes for Analysis |
| | | **SP 1.2 Analyze Causes** |
| | SG 2 Address Causes of Selected Outcomes | |

Note: SG: Specific Goal, SP: Specific practice, ML: Maturity Level

- the data and analysis techniques they need for determining the technologies and methods that work best for them;
- a framework for learning why they make errors and what is the best way of finding, fixing, and preventing them.

This way, software engineers can easily determine the quality of their reviews, the defect types they usually miss, and the quality approaches that are most effective for them (Humphrey, 2005).

The recommended approach for learning the PSP is by doing a sequence of small projects guided by a PSP instructor, using increasingly refined and more complete processes (see Figure 2.3).

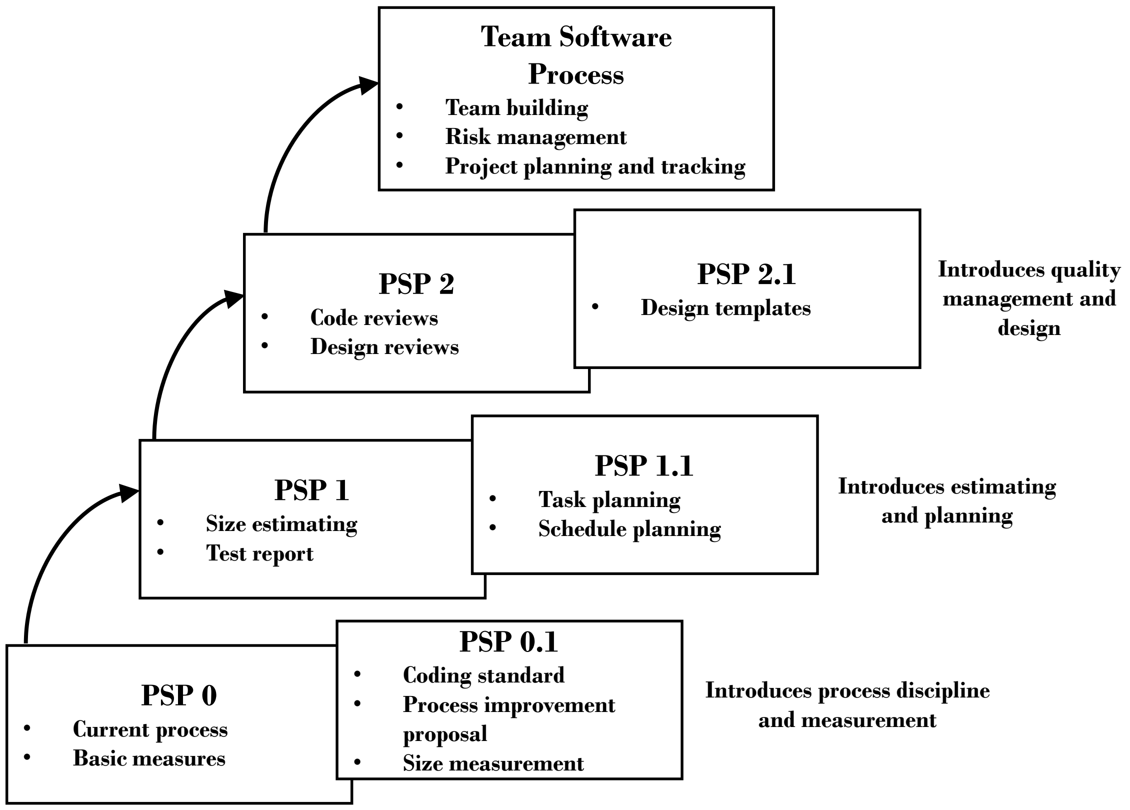


Figure 2.3: PSP process evolution (Humphrey, 2005).

*PSP0 - The Baseline Personal Process (establish a measured performance baseline)*

The first step in the PSP is to establish a baseline that includes some measurements (the time spent per phase and the defects found per phase) and a reporting format (Humphrey, 1997) (Humphrey, 2005). The time spent per phase is a simple record of the clock time spent in each part of the PSP process and a defect is counted every time one changes a program to fix a problem.

The objective of the baseline process (PSP0) is to provide engineers a framework that can help them in gathering data and writing their first PSP program. The gathered data (time and defects) helps developers in planning and managing their projects along the PSP training and also shows developers where more time is spend and where most defects are injected and fixed, which ultimately provides a consistent base for measuring progress and defining a foundation on which to improve.

After the first programming assignment, PSP0 is upgraded to PSP0.1 which introduces coding standards, size estimation, and process improvement proposals (PIP).

The process improvement proposal is an approach to report and record process problems encountered, and suggest improvements. It is additionally a part of the process improvement process. The main goal of PSP0.1 is to give engineers a comprehension of size estimation principles.

*PSP1 - The Personal Planning Process (make size, resource and schedule plans)*

PSP1, which adds planning steps to PSP0, introduces size and resource estimation and a test report. In PSP1.1, schedule planning and status tracking are introduced. The goals of PSP1 and PSP1.1 are to train developers for understanding the significance of the relationship between the size and time of the programs they develop, to know the way of making commitments that they can meet, to have an organized plan for doing their tasks and a framework for tracking their status (Humphrey, 1997) (Humphrey, 2005).

*PSP2 - Personal Quality Management (practice defect management and yield management)*

The goal of PSP2 is to improve the developers' capability to produce high quality programs. This goal can be achieved by presenting to the developers technique for better defect detection and prevention. This process will reduce the number of defects in compile and test phases. It is also observed that in most cases defects are simple typos, oversights, or dumb mistakes. So this way of measuring and tracking defects will teach developers how to deal with defects objectively to reduce rapidly the number of defects they usually inject.

PSP2 usually reduces the defects production rate by adding design and code reviews to PSP1. These added reviews help developers to identify defects earlier in their process and reduce the extra cost for identifying them in later stages. For developers, it is important to know the number, cost and main causes of the defects they produced.

In PSP 2.1 the design process is addressed. The main objective of PSP2.1 is to address the criteria for design completion and not to teach the developers how to design. After finishing design, PSP2.1 demonstrates different design verification methods and design completeness criteria. In this case, completeness criteria is established for design only, but similar criteria can be established for other process phases (test development, documentation development and requirement specification) (Humphrey, 1997) (Humphrey, 2005).

*PSP3 - The Cyclic Personal Process (scale up PSP methods to larger projects)*

The PSP up to this point has focused only on a linear process for building small programs. For scaling PSP2 up to larger projects, the approach is to subdivide larger programs into PSP2-sized pieces. These smaller programs are then separately developed and integrated into the larger total program. PSP3 is a cyclic development process that follows the principles of Boehm's spiral model [Boehm 1988]. The first PSP3 build provides a base that is enhanced with each successive cycle. In each iteration, a complete PSP2- like process is used, including design, code, compile, and test. The test step, however, is typically a combination of unit test and integration. As long as each increment is of high quality, this testing strategy is practical. The PSP3 process is suitable for programs of up to several thousand lines of code (KLOC). (Humphrey, 1997) (Humphrey, 2005).

The PSP3 objective is to introduce the engineers to the principles of process scaling. They find that by assuring the quality of each successive development cycle, they can concentrate on verifying the performance of the latest increment without interference from previously-introduced defects. If a prior increment has many defects, however, testing will be much more complex and the scale-up benefits will largely be lost. With high-quality increments, developer productivity often increases with program size.

*TSP- The Team Software Process*

Developers can use PSP3 to build programs of several KLOC. The PSP provides a disciplined foundation the developers can use to define an effective team process. While the PSP introduces the concepts for relating individual PSPs to the team, it does not define team processes or provide team exercises (Humphrey, 1997) (Humphrey, 2005). That is addressed by the Team Software Process (TSP).

### 2.2.2 Process structure

The PSP process flow is show in Figure 2.4. The first step of the process is planning, having as input the requirements. There is a script to guide the work and a plan summary to record the data in this first step. The other steps also have scripts to guide the work and logs to record the data. In the postmortem phase (PM), the data from the logs is summarized, the program size is measured and all of this is then recorded into the plan summary form. The output of this phase is the finished product and the completed plan summary form. It should be noted that (contrarily to the more complex TSP) the PSP is designed for guiding the development of small programs (or components of large programs) by individual engineers, which is why it neither includes requirements gathering nor high-level (architectural) design.

As illustrated in Figure 2.3 the PSP has six process versions that are labeled from PSP0 to PSP2.1. They all have a similar collection of scripts, forms, logs and standards. The scripts instruct the software engineers, step by step, through each part of the process. To record data there are templates (for the logs and forms) and the standards that help guide the work. The elements described above are shown in Figure 2.5.

### 2.2.3 Measurement framework

In the PSP course, developers gather data regarding the size of the products they produce, the time that they spend on each process phase (e.g., plan, design, design review etc.), and the quality of these products, to monitor their work and to help them make better plans (i.e., improve predictability) and produce high-quality products (i.e., improve product quality) in a cost-effective way (i.e., improve productivity).

Base data is gathered in three different artifacts:

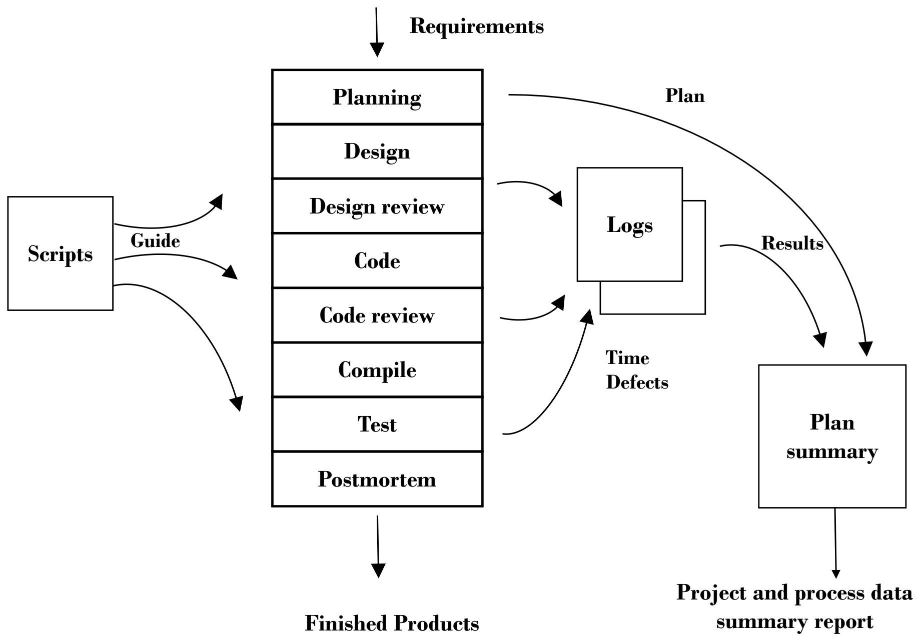


Figure 2.4: PSP process flow (Humphrey, 2000).

- Time log - used for recording actual time spent in each process phase (possibly with multiple entries for the same phase, with different start and stop times);
- Defect log – used for recording defects, with description, defect type (according to a defect standard), phase in which the defect was injected, phase in which the defect was removed, and fix time;
- PROBE estimating template and plan summary – use for estimating size, time, number of defects (in the beginning of the project) and for recording actual size (at the end of the project).

From this data, several measures are defined as indicated in Table 2.2.

### 2.2.4 Final report assignment

At the end of the PSP training, students are usually asked to perform the “PSP Final Report” assignment (or “PSP Performance Analysis Report”) for better understanding their current software development performance and highest-priority areas for improvement. In the assignment, students are asked to analyze their personal performance data collected throughout several projects they developed and document their findings and improvement proposals in a report. To guide their analysis, students are asked to address several questions as indicated in Table 2.3.

Table 2.2: PSP base and derived measures.

| **Measure** | **Definition** |
|---|---|
| Planned Time in Phase | The estimated time to be spent in a phase for a project |
| Actual Time in Phase | The sum of time spent (delta times in time log) for a phase of a project |
| Total Time | The sum of planned or actual time for all phases of a project |
| Time in Phase to Date | The sum of Actual Time in Phase for all completed projects |
| Total Time to Date | The sum of Time to Date for all phases of all projects |
| Time in Phase to Date% | 100 $*$ Time in Phase To Date for a phase divided by Total Time in Phase To Date |
| Compile Time | The time from the start of the first compile until the first clean compile |
| Test Time | The time from the start of the initial test until test completion |
| Defect | Any element of a program design or implementation that must be changed to correct the program |
| Defect type | Each defect is classified according to a defect type standard. It includes 10 defect types in a simple classification scheme designed to support defect analysis. |
| Fix Time | The time to find and fix a defect |
| LOC (Size) | A logical line of code as defined in the engineers counting and coding standard |
| LOC Type | There are 7 LOC types, Base, Deleted, Modified, Added, Reused, Added and Modified, Total LOC and Total New Reused |
| LOC / Hour (Productivity) | Total added and modified LOC developed divided by the total development hours |
| Estimating Accuracy | The degree to which the estimate matches the result. Calculated for time and size %Error = 100$*$(Actual-Estimate)/Estimate |
| Test Defects/KLOC | The defects removed in the test phase per added and modified KLOC. 1000$*$(Defects removed in Test)/(Actual Added and Modified LOC) |
| Compile Defects/KLOC | The defects removed in compile per added and modified KLOC. 1000$*$(Defects removed in Compile)/(Actual Added and Modified LOC) |
| Total Defects/KLOC | The total defects removed per added and modified KLOC. 1000$*$(Total Defects removed)/(Actual Added and Modified LOC) |
| Yield | The percent of defects injected before the first compile that are removed before the first compile. 100$*$(defects found before the first compile)/(defects injected before the first compile) |
| Appraisal Time | Time spent in design and code reviews |
| Failure Time | Time spent in compile and test |
| Appraisal Cost of Quality (COQ) | 100$*$(design review time + code review time)/(total development time) |
| Failure COQ | 100$*$(compile time + test time)/(total development time) |
| Total COQ | Total Cost of Quality = Appraisal COQ + Failure COQ |
| COQ Appraisal/Failure Ratio (A/FR) | A/FR = Appraisal Time/Failure Time |
| Review Rate | Lines of code reviewed per hour. 60 $*$ (Added and Modified LOC)/(review minutes) |

Table 2.3: Example of questions to be addressed in the PSP Final Report assignment based on AssignmentKit (2006).

**Analysis of size estimating accuracy**

- What are the average, maximum, and minimum actual sizes of your programs in LOC to date?
- Excluding assignment 1, what percentage over or under the actual size was the estimated size for each program (for example, if estimated/actual is in %, 85% is 15% under, 120% is 20% over)? What are your average, maximum, and minimum values for these?
- How often was my actual program size within my 70% statistical prediction interval (when you used methods A or B)?
- Do I have a tendency to add/miss entire objects?
- Do I have a tendency to misjudge the relative size of objects?
- Based on my historical size-estimating accuracy data, what is a realistic size-estimating goal for me?
- How can I change my process to meet that goal?

**Analysis of time estimating accuracy**

- What are the average, maximum, and minimum times of your assignments to date?
- What percentage over or under the actual time was the estimated time for each program (for example, if estimated/actual is in %, 85% is 15% under, 120% is 20% over)?
- What are your average, maximum, and minimum values for these?
- How often was my actual development time within my 70% statistical prediction interval (when you used methods A or B)?
- What are the average, maximum, and minimum values for productivity per program to date in LOC/hr.?
- Is my productivity stable? Why or why not?
- How can I stabilize my productivity?
- How much are my time estimates affected by the accuracy of my size estimates?
- Based on my historical time-estimating accuracy data, what is a realistic time-estimating goal for me?
- How can I change my process to meet that goal?

**Defect and yield analysis**

- Which defect type accounts for the most time spent in compile?
- In test? In which phase was each type of defect injected most often?
- What type of defects do I inject during design and coding?
- What trends are apparent in defects per size unit (e.g., KLOC) found in reviews, compile, and test? What trends are apparent in total defects per size unit?
- How do my defect removal rates (defects removed/hour) compare for design review, code review, compile, and test?
- What are my review rates (size reviewed/hour) for design review and code review?
- What are my defect-removal leverages for design review, code review, and compile versus unit test?
- Is there any relationship between yield and review rate (size reviewed/hour) for design and code reviews?
- Is there a relationship between yield and A/FR?

**Quality analysis**

- How much did the quality of the programs entering unit test change? Why?
- Am I finding my defects in design and code reviews? Why or why not?
- Based on my historical data, what are some realistic quality goals for me?
- How can I change my process to meet those goals?

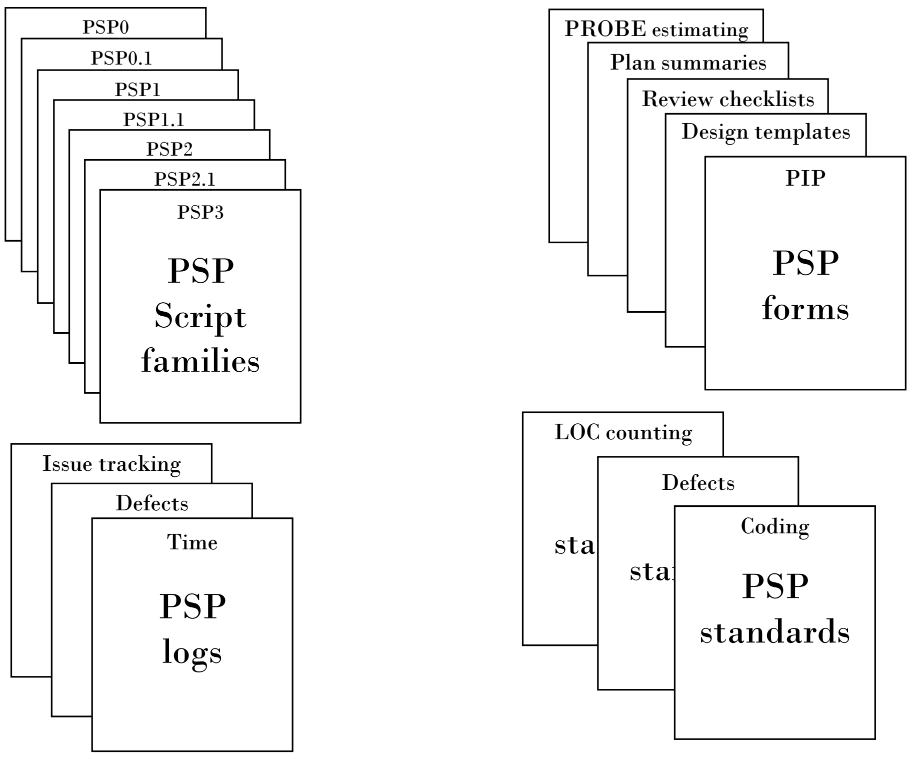


Figure 2.5: PSP process elements (Humphrey, 2000).

### 2.2.5 Tool support for PSP

A variety of tools have been developed over the years to tackle the problem of manual data gathering by providing semi or fully automated data collection solutions. Several of these tools can be downloaded and used for free. Some of these tools are reviewed below (also see Table 2.4 for the comparison of these tools).

**PSP Student Workbook** (PSPWorkBook), is the official tool for PSP courses developed by the SEI and is based on Microsoft Access. This tool can be used on Microsoft Windows or emulated Microsoft Windows operating systems.

It provides good support for the learning of this methodology as it has a lot of supporting materials, such as course materials, scripts, forms, and templates, etc. It provides a lot of so-called analysis tools to help the student in the analysis of their personal performance data (see examples in Figure 2.6). This tool basically transforms the data into graphs of the known performance indicators (that require some calculations), such as Time Estimation Error, Size Estimation Error and Productivity. The different forms available for users are described below:

- Project Plan Summary: is basically used for representing the overall information of the development such as estimated development time, estimated program size, productivity, defect density and cost-performance index.
- Time Recording Log: records details of the time spent in each development phase like Plan, Design, Design Review, Code, Code Review, Compile, Unit Test and Postmortem.

- Defect Recording Log: stores information of each defect such as its injection and removal phases, type, and description.

- Test Report: summarizes the actual and expected test results as well as testing environments.

- Process Improvement Proposal: focuses on the proposals for process improvement based on the user's self-evaluation.

In addition to the lack of support for problem identification and their root cause analysis, there are several other limitations of PSP Student Workbook. E.g., several important PSP elements such as counting standard, coding standard, design review checklist, design templates, and code review checklist are not supported by this tool. The users need to separately create and store these documents which is really an overhead.

Furthermore, PSP Student Workbook might not be suitable for professional use since the Access database can inevitably become unnecessarily large and this could cause serious performance problems for the development environments.

**Process Dashboard** (Dashboard), is an open source tool, originally developed by the United States Air Force in 1998. It supports all the standard functionalities which can be found in the PSP Student Workbook. This tool supports the PROxy-Based Estimating method (PROBE), which is the main feature of the PSP framework. It is developed in Java and can be run on Windows, Linux, Unix, Macintosh, etc. Like other tools, Process Dashboard displays graphs and reports based on the development data of developers (see examples in Figure 2.7).

**Hackystat** (Hackystat) is an example of an extensible sensor-base tool for automatic data collection during software development (such as failed tests, code changes, etc.). Hackystat provides a range of sensors to collect data from tools such as Eclipse, Ant, Checkstyle, Clover, FindBugs, JUnit, and Visual Studio, and APIs for the creation of customized sensors. It also provides simple forms and APIs for data access. One limitation of Hackystat is that it only collects data during development (and not during planning, design, and postmortem); another limitation is the difficulty to extract higher level information (e.g., defects) from the observed events (e.g., failed test cases).

**Jasmine** (Shin et al., 2007), is a sensor-based automated data collection tool which helps developers in applying the PSP. In addition to sensor-based data collection like Hackystat, Jasmine also provides support for activities such as planning, estimation, and tracking. Furthermore, it also includes EPG (Electronic Process Guide) and ER (Experience Repository) to easily navigate the PSP elements, and to share and sort additional process related information respectively. Jasmine provides analysis of the performance data in the form of tables, graphs and charts.

**PSP.NET** (Nasir and Yusof, 2005), is a web-based application comprising two layers: application and database. It is basically a PSP supporting tool which automatically collects the primary data using electronic forms to reduce the time spend and overhead during recording data in PSP courses. It provides support in PSP components such as templates, checklist, and log form. PSP.NET can also print (according to the PSP level) the PSP forms after publishing in HTML format. Furthermore, PSP.NET provides help in producing project plan summaries for analysis.

**PSPA** (Sison, 2005), is a system of Web clients written in .Net languages. The main feature of this tool is the automatic recording of compile defects from the programs written in C and JAVA languages. The tool is mainly supported by Eclipse and is not a good option for programmer using other integrated development environments. The tool also provides: automatic support for lines-of-code (LOC) counting; daily schedule tracking; size and time estimation; preparing PSP reports and graphs; editing PSP logs.

**WBPS** (Thisuk and Ramingwong, 2014), is implemented in PHP and MySQL, and provides standard PSP functions which can be found in the official PSP Student Workbook such as time logging, defect logging and process improvement proposal. In addition to the standard functions, WBPS also offers several other features, such as management of design documents and standards, integrated screens, review assistant and instructor assistant. It also proposes a multi-language interface and online accessibility.

The tools discussed above mostly collect the data in automatic or semi-automatic way and present them in the form of graphs, charts and summaries for further analysis but in practice there is no automatic or semi-automatic support for the collected performance data to be analyzed for problem and root causes identification and ranking.

In Figure 2.6 and Figure 2.7 are given examples of how PSP Student Workbook and Process Dashboard tools provide support in the form of graphs, charts and reports for further manual analysis.

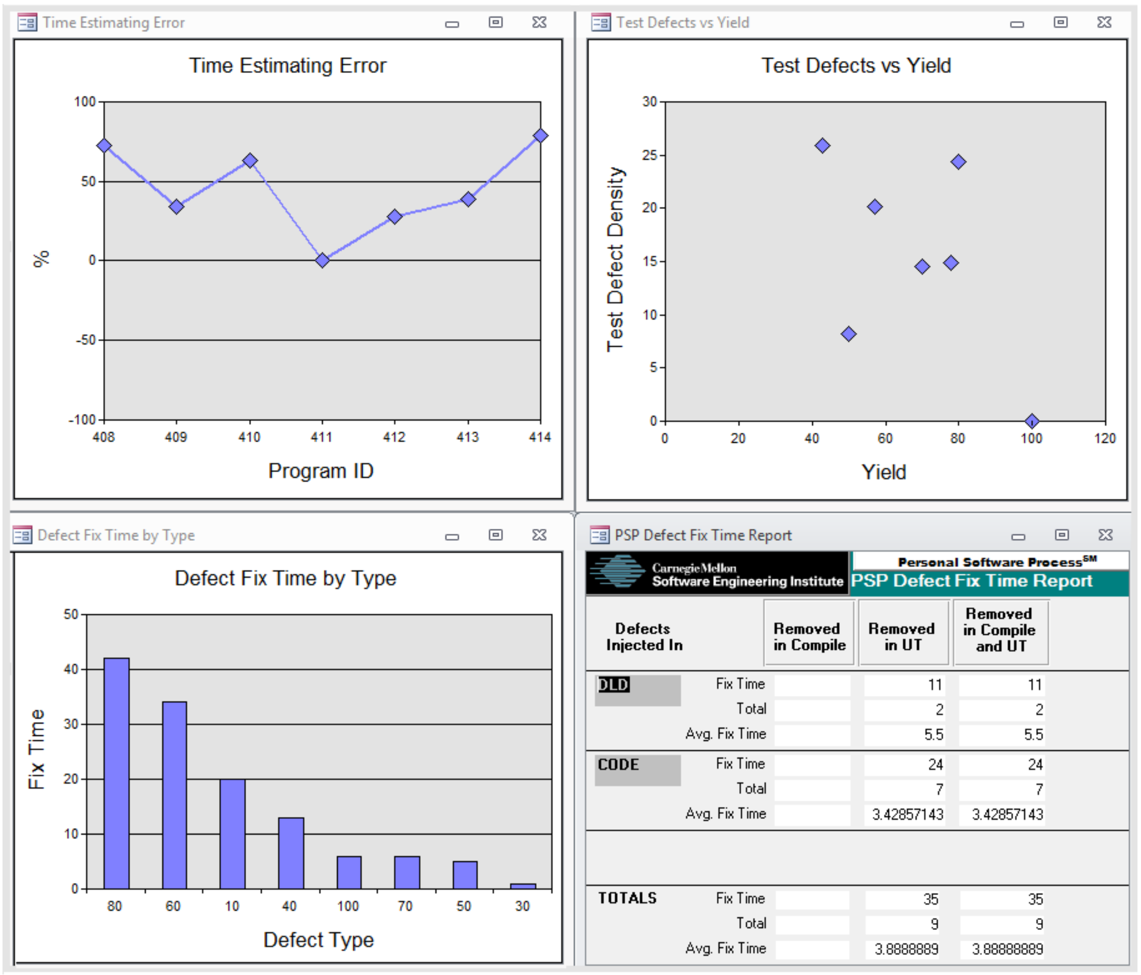
By doing this analysis automatically, developers can save time for further in-depth analysis of their performance data.

## 2.3 Performance measurement and analysis in other processes

Agile methods are the dominant paradigm in software engineering industry and education. An example of metrics collection and analysis in agile project courses is described in (Alperowitz et al., 2016). Some metrics are introduced to measure the success of three key workflows (merge management, continuous integration and continuous delivery), giving instructors and project leaders a quick overview of the project status and problems which they can react upon. The main differences with respect to this research work are: in this research work the goal is to retrospectively analyze completed projects; we try to identify and rank potential causes; for problem identification, we use thresholds derived from large data sets. Nevertheless, the metrics introduced by the authors could be adapted and explored in our approach. In principle, performance measures can be defined for assessing the adherence to agile practices, such as the twelve XP practices (Beck, 2000). One of the challenges is the automatic collection of relevant process measures (e.g., to assess the adherence to “sustainable pace”); the other challenge is the availability of large data sets for model calibration.

Table 2.4: PSP tool support.

| Characteristics / Tool | **Source** | **Technology** | **Process** | **Data collection** | **Data analysis** |
|---|---|---|---|---|---|
| **PSP Student Workbook** | Software Engineering Institute (SEI) | Microsoft Access | PSP | Manual | Standard PSP charts and reports showing historical trends |
| **Process Dashboard** | US Air Force Tuma Solutions (1998-present) | Java | PSP / TSP | Manual | Standard PSP charts and reports showing historical trends |
| **Hackystat** | University of Hawaii (2001-present) | Java | Any (with plug-ins) | Automatic (with sensors) | Summaries and graphs |
| **PSP.NET** | University of Malaya, Malaysia 2005 | Web | PSP | Manual | No information |
| **PSPA** | University of Manila, Philippines, 2005 | Desktop (Microsoft. NET) | PSP | Automated (with IDE plugins) | No information |
| **Jasmine** | Information and Communications University (now merged into KAIST), South Korea, 2007 | Web | PSP | Automated (with sensors) | No information |
| **WBPS** | Chiang Mai University, Thailand, 2014 | Web (PHP, MySql) | PSP | Manual (automatic LOC counting) | No information |



PSP Defect Fix Time Report

| Defects Injected In | | Removed in Compile | Removed in UT | Removed in Compile and UT |
|---|---|---|---|---|
| DLD | Fix Time | | 11 | 11 |
| | Total | | 2 | 2 |
| | Avg. Fix Time | | 5.5 | 5.5 |
| CODE | Fix Time | | 24 | 24 |
| | Total | | 7 | 7 |
| | Avg. Fix Time | | 3.42857143 | 3.42857143 |
| TOTALS | Fix Time | | 35 | 35 |
| | Total | | 9 | 9 |
| | Avg. Fix Time | | 3.8888889 | 3.88888889 |

Figure 2.6: Examples of data analysis charts and reports from PSP student workbook.

## 2.4 Performance problem identification techniques

In this section, specific techniques are analyzed for the identification, visualization and rating of performance problems in relation to specific performance indicators under analysis. In general, to decide if the value of a particular indicator is problematic and how good/bad it is, one needs some thresholds or reference values. Ideally, those thresholds should be derived from historical data in an automatic way, and not arbitrary values defined manually based on expert opinion. The techniques described next differ in the way such thresholds are derived and visualized.

### 2.4.1 Control charts

In Statistical Process Control (SPC), *run charts* are used to graphically represent the behavior over time of variables that characterize process performance (i.e., process performance indicators) and help assessing process stability and capability. *Stability* has to do with the level of variability in the variable under analysis. *Capability* has to do with the ability to meet desired performance levels (Navidi, 2008).

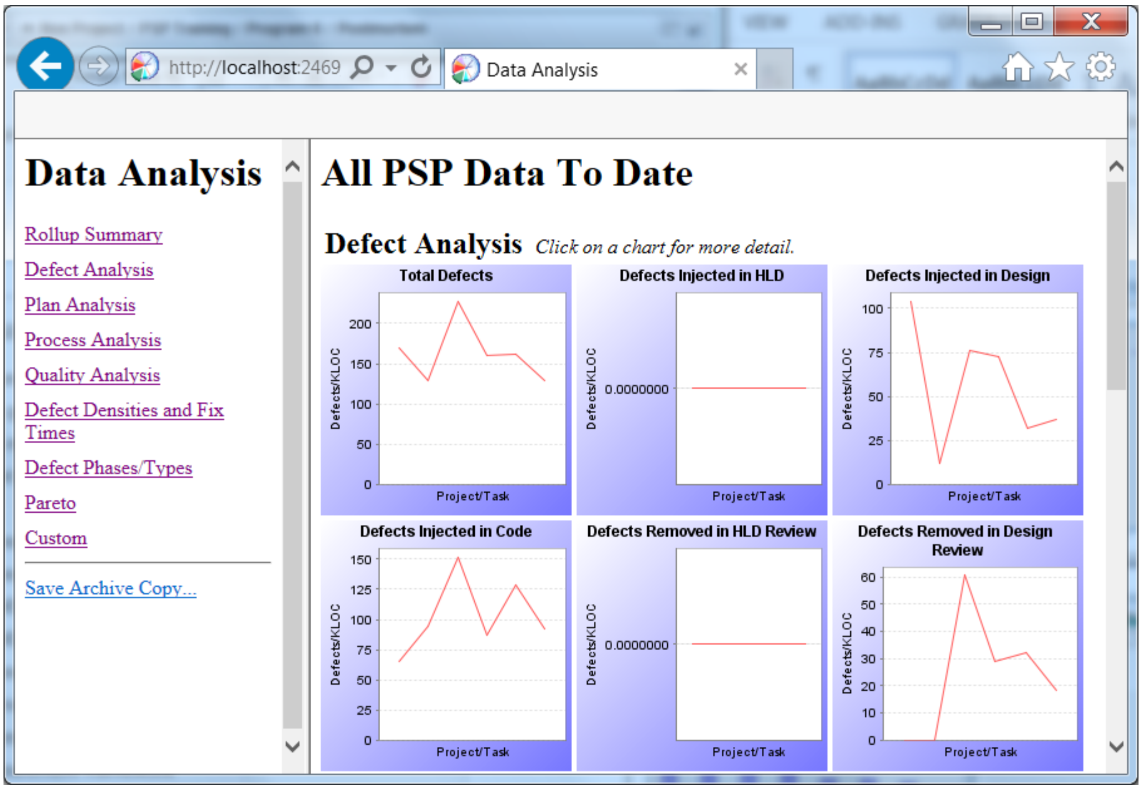


Figure 2.7: Examples of data analysis charts from Process Dashboard tool.

To assess if a process is under control (or stable), it uses control charts, which are basically *run charts* with superimposed control limits – upper control limits (UCL) and lower control limits (LCL) (see example in Figure 2.8).

Control limits are derived based on the standard deviation ($\sigma$) computed from historical process data. The area between each control limit and the centerline is divided into zones. The closest zone to the centerline is referred to as Zone A (1-sigma zone), the next is referred to as Zone B (2-sigma zone), and the third one is referred to as the control limit Zone C (3-sigma zone) (see Figure 2.9). The standard UCL and LCL are usually chosen as the 3-sigma limits.

When used to monitor the process, control charts can uncover inconsistencies and unnatural variations.

For assessing process capability, it uses specification limits (USL and LSL) derived from requirements or some external reference. Usually, a process is considered capable if the control limits (derived from the process performance data) are within the specification limits. Examples of control charts with both control limits and specification limits, for different stability and capability status are shown in Figure 2.10.

The main focus of our work is on checking capability. A technique that can be adapted for automatically deriving specification limits from existing data for process capability analysis is described in the next section.

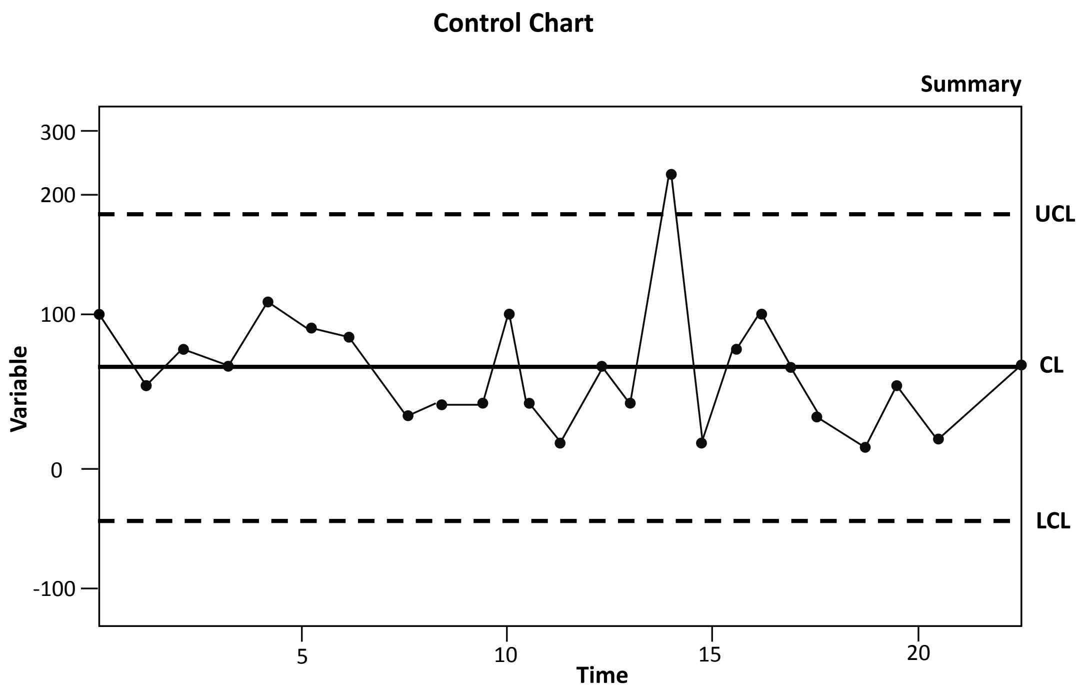


Figure 2.8: Illustration of a control chart.

### 2.4.2 Benchmark based software evaluation

In our approach, in order to enable the automated identification of performance problems, after deciding on the relevant PIs, one has to decide on the relevant ranges. Our approach for defining such ranges draws inspiration from the benchmark-based approach developed by researchers of the Software Improvement Group (Alves, 2012) (Alves et al., 2010) to rate the maintainability of software products, with adaptations for process evaluation instead of product evaluation.

In (Alves, 2012) (Alves et al., 2010), the authors claim that the effective use of software metrics is hindered by the lack of meaningful thresholds. They also note that thresholds have been proposed for a few metrics only, mostly based on expert opinion and a small number of observations, or systematically derived based on unjustified assumptions about the statistical properties of the metrics (such as normality). Consequently, they propose a method to empirically derive in a systematic way metric thresholds from measurement data (benchmarks), in order to determine risk profiles and maintainability ratings for products under analysis. They propose a discrete rating schema (from 1 to 5 stars), based on thresholds that correspond to the 20%, 40%, 60% and 80% quantiles. This idea can be applied in our work, but a simpler rating scheme with three levels only (and thresholds for the 33% and 66% quantiles), corresponding to the green, yellow and red semaphores, seems sufficient and intuitive.

In our case, benchmarks may be derived from the performance data of a large community of process users.

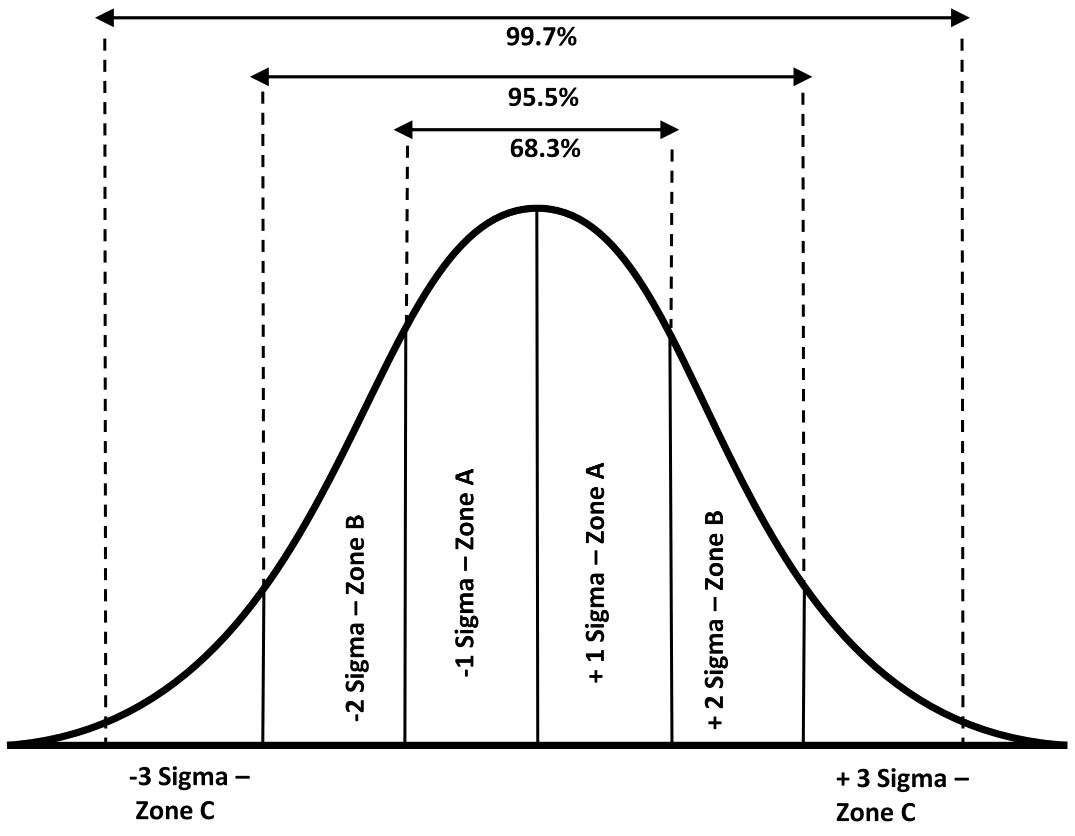


Figure 2.9: Illustration of deriving control limits and zones from the statistical distribution of historical data, assuming normal distribution.

## 2.5 Root cause analysis techniques

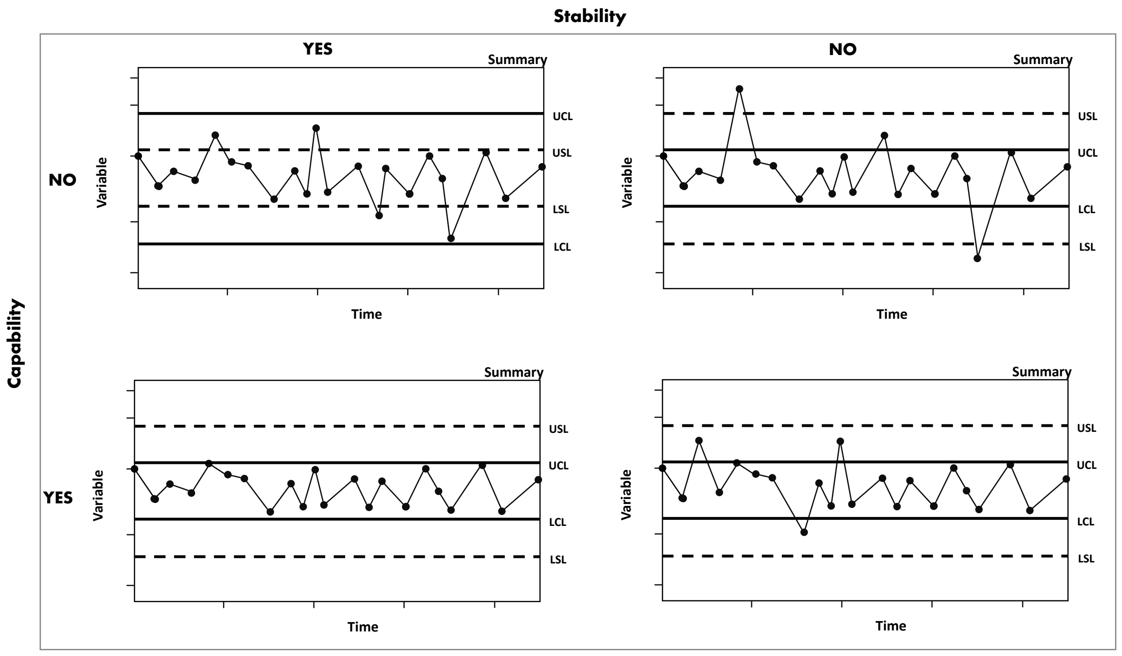
After identifying performance problems, it is important to find their root causes, so that improvement actions can subsequently be defined to address the relevant causes.

In this section, several techniques that can be used for identifying root causes are presented.

### 2.5.1 Fishbone diagrams

The Cause-and-Effect or Fishbone diagram (see Figure 2.11) is a classic technique for helping in manual causal analysis and for visualizing relationships between causes and effects. It combines the brainstorming and Mind Map to push one to consider all the possible causes of a problem. It is also known as Ishikawa diagrams because Kaoru Ishikawa developed them in 1943 (Surhone et al., 2010).

The Ishikawa diagram organizes and displays the relationships between different causes for the effect that is being examined. This diagram also helps organize the brainstorming process. The major categories of causes are put on major branches connecting to the backbone, and various sub-causes are attached to the branches. A tree-like structure results, showing the many facets of



**Legend**: UCL: Upper control limit; LCL: Lower control limit; USL: Upper specification limit; LSL: Lower specification limit.

Figure 2.10: Process stability versus capability.

the problem. The method for using this chart is to put the problem to be solved at the head, then fill in the major branches.

### 2.5.2 Defect causal analysis

An example of applying Fishbone techniques in software engineering is Defect Causal Analysis (DCA). Many process improvement approaches (e.g., Six Sigma (Kwak and Anbari, 2006) or FMEA (Campos, 2012)) described in literature and practiced in industry include causal analysis activities for determining the causes of defects and other problems (Kalinowski et al., 2012). However, most of the techniques are essentially manual. Defect Causal Analysis (Card, 1993) is one of the prominent methods for analyzing defects and identifying root causes for improvement in software engineering. Furthermore, the learning capability of DCA from defects enable improvement of processes and products, which is a significant benefit in the context of continuous improvement strategies (Card, 2005).

The DCA process involves 6 steps to be performed in DCA workshops (Card, 1993): (1) select problem sample; (2) classify selected defects (e.g., using ODC); (3) identify systematic errors (e.g., with Pareto charts); (4) determine main causes (e.g., using Fishbone or cause-effect diagrams); (5) develop action proposals; (6) document meeting results.

The main problem of DCA is that it is basically a manual process, and our goal is to automate, at least partially, the root causes identification of performance problems. The idea is to automatically drill down from performance problems to causes up to the level permitted by the

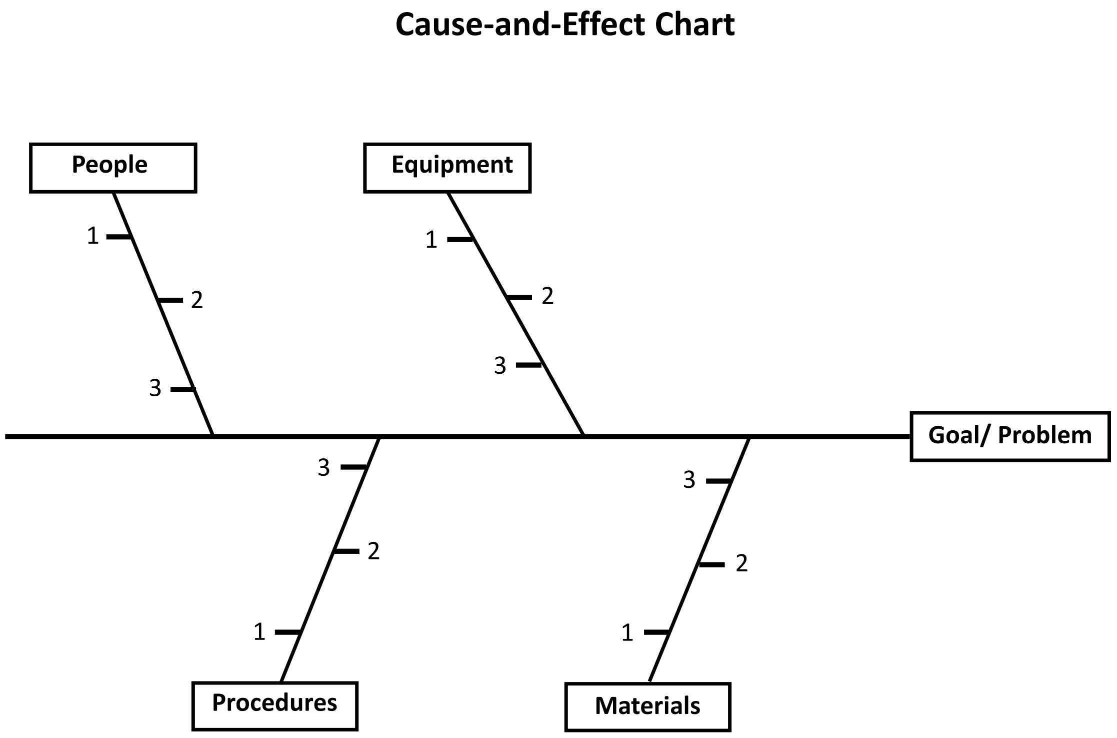


Figure 2.11: Fishbone diagram.

data available. Manual analysis may still be required for identifying deeper causes not apparent in the available data. Next we investigate a technique that can help in automatic causal analysis.

### 2.5.3 Process performance models

In order to be able to automatically identify and rank (prioritize) the causes of performance problems, we need to have some quantitative relationship between factors (causes) and outcomes. For that purpose, the CMMI suggests the usage of process performance models (PPM).

In the context of the CMMI process improvement framework, a PPM is a description of the relationship among attributes of a process or sub-process and its outcomes, developed from historical process performance data and calibrated using collected process and product measures (Chrissis et al., 2011).

In the case of continuous variables, a PPM often takes the form of a regression equation, relating controllable or uncontrollable factors (x) with outcomes (y), together with an indicator of the degree of variability in the model, such as the R2 statistic. In the case of discrete variables, PPMs may be based on Bayesian networks (Zubrow et al., 2009).

PPMs are useful tools for project management and process management and improvement. In the latter case, PPMs help organizations identify and leverage important relationships among process factors and outcomes, and estimate the effects of alternative process changes. The creation of a PPM usually involves the following steps, among others: (1) decide what outcomes to analyze; (2) hypothesize factors to investigate; (3) select the modeling techniques to use; (4) obtain relevant data; (5) fit the model to the data and evaluate the degree of fitness according to statistical and business criteria (Zubrow et al., 2009).

Examples of PPMs that can be constructed and applied in the context of the TSP/PSP, together with examples of outcomes and factors to consider for a few sub-processes, are described in (Tamura, 2009). An example of a PPM created by a TSP team for establishing a target code review rate (number of lines of code reviewed per hour), based on the predicted impact on the code review yield (percentage of defects found in code reviews), is as follows (Tamura, 2009):

- Regression equation: $CodeReviewYield = 146 - 0.364 \times CodeReviewRate$
- $R^2 = 94.1\%$, $p - value = 0.000$

Examples of possible factors to consider for analyzing the code review yield are (Tamura, 2009):

- Outcome (y): code review yield;
- Factors (x) currently collected by TSP: requirements inspections rate, high-level design inspections rate, detailed design review rate, detailed design inspection rate, code review rate, code review/coding time;
- Other factors (x) that could be collected: code complexity, encapsulation, programming language & tools, code review checklist, coding skills and experiences with the programming languages and tools used, code review skills and experiences, quality of reused source code.

PPMs can be applied in our work with adaptations, as will be explained in more detailed in chapter 3. All the steps discussed above for the creation of PPMs will be applied, with some adaptations and choices: in step 2, we follow a hierarchical approach (factors that can in turn be affected by other factors, like in Fishbone diagrams); in step 3, we use a regression model in case of statistical relationship between factors and outcomes, and an exact formula when there is an algebraic relationship. The type of regression models applicable for our work are discussed in section 2.5.4.

As mentioned before, PPMs can be used to estimate the effects of alternative process changes. In our case, PPMs can be used to rank the factors, according to the effect of changes in single factors on the outcome under analysis. Since our goal is just to rank the factors, sensitivity analysis techniques can be used. Sensitivity analysis techniques are discussed in section 2.5.5. The exact approach will be explained in chapter 3.

### 2.5.4 Regression models

Regression analysis is a statistical process that attempts to determine the strength of the relationship between a dependent variable (e.g., Process Yield) and independent variables (e.g., Design

Review Yield, Code Review Yield, etc.). It may also be referred to as the study of how a quantitative response variable (or dependent variable) varies when an explanatory variable (or independent variable) changes.

In general, regression analysis attempts to find a function that fits a series of observations with minimal error. It should also be noted that regression does not necessarily suggest a causal relationship between response variable and explanatory variables but a significant association.

Analysis that assumes a single independent variable to predict the value of a dependent variable is known as simple regression; analysis that assumes two or more independent variables for the prediction of the value of a dependent variable is known as multiple regression (Chatterjee and Hadi, 2015).

Regression modeling can be performed in different ways. Traditionally, regression techniques are categorized as linear or nonlinear. Linear regression tries to model the relationship between dependent and independent variable(s) by fitting a linear equation to observed data. For random noise that cannot be explained by the linear relationship, a variable known as model error is introduced. Linear regression models assume the form of the following expression:

$$y = b_0 + b_1x_1 + b_2x_2 + ... + b_ix_i + e \tag{2.1}$$

Linear regression models attempt to determine these parameters by minimizing the function in regards to an expression that compares observed with predicted values (the least squares method is a typical approach for this). Generally, when using linear regression models, it is simple to interpret the relationship between dependent variable and predictors and analyze the correlation among predictors (Kuhn and Johnson, 2013).

Nonlinear regression models (Seber and Wild, 1989) are expressed by functions that are not linear in the parameters. These models tend to vary in approach, and comprise many different techniques. Common models of this type include neural networks, Support Vector Machines (SVM), k-Nearest Neighbors (k- NN), Multivariate Adaptive Regression Splines (MARS) and tree-based models (Aggarwal, 2014).

### 2.5.5 Sensitivity analysis

When an outcome ($y$) is affected by multiple factors ($x_1...,x_n$, with $n > 0$), sensitivity coefficients may be used to measure the importance of those factors. The sensitivity coefficient is basically the ratio of the change in output to the change in input while keeping all other parameters constant.

In our model, we will expect two types of relationships between independent variables and dependent variables.

In case there is an explicit algebraic equation that describes the relationship between the independent variables and the dependent variable, the sensitivity coefficient, $\phi_i$, for a particular independent variable $X_i$ can be calculated from the partial derivative of the dependent variable with respect to the independent variable (Hamby, 1994), i.e.

$$\phi_{X_i} \rightarrow_{Y} = \frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right) \tag{2.2}$$

where the quotient, $X_i/Y$, is introduced to normalize the coefficient by removing the affects of units. Inherent to this calculation are the assumptions that the higher ordered partials are negligible and there is no correlation between input parameters.

In case there is no explicit algebraic equation that describes the relationship between the independent and the dependent variables, then a regression formula can be calibrated based on the historical data form many process users for sensitivity coefficient (Navidi, 2011).

# Chapter 3

# Proposed performance analysis method

This chapter presents the ProcessPAIR method for automated performance analysis and improvement recommendation. Section 3.1 introduces the overall approach covering performance problems and their root causes identification, and ranking. Section 3.2 describes performance model definition. Section 3.3 describes the performance model calibration covering cumulative distribution functions, performance ranges, regression models and sensitivity coefficients, and data set filtering. Section 3.4 presents model based performance analysis. The method is applicable to any software development process or business process.

## 3.1 Overall approach

Our approach involves three main steps (see Figure 3.1):

1. Define: Process experts define the structure of a performance model (PM) suited for the development process under consideration. In our approach, a PM comprises a set of top-level and child performance indicators (PIs), organized hierarchically by cause-effect relationships.

2. Calibrate: The PM is automatically calibrated by ProcessPAIR based on the performance data of many process users. The statistical distribution of each PI and statistical relations between PIs are computed from the calibration data set.

3. Analyze: Once a PM is defined and calibrated, the performance data of individual developers can be automatically analyzed by ProcessPAIR, to:

   (a) identify performance problems (in top-levels PIs),

   (b) identify potential root causes (related with child PIs), and

   (c) rank the performance problems and their potential root causes.

The 3 steps of our approach are generically interrelated as described in the next sub-sections.

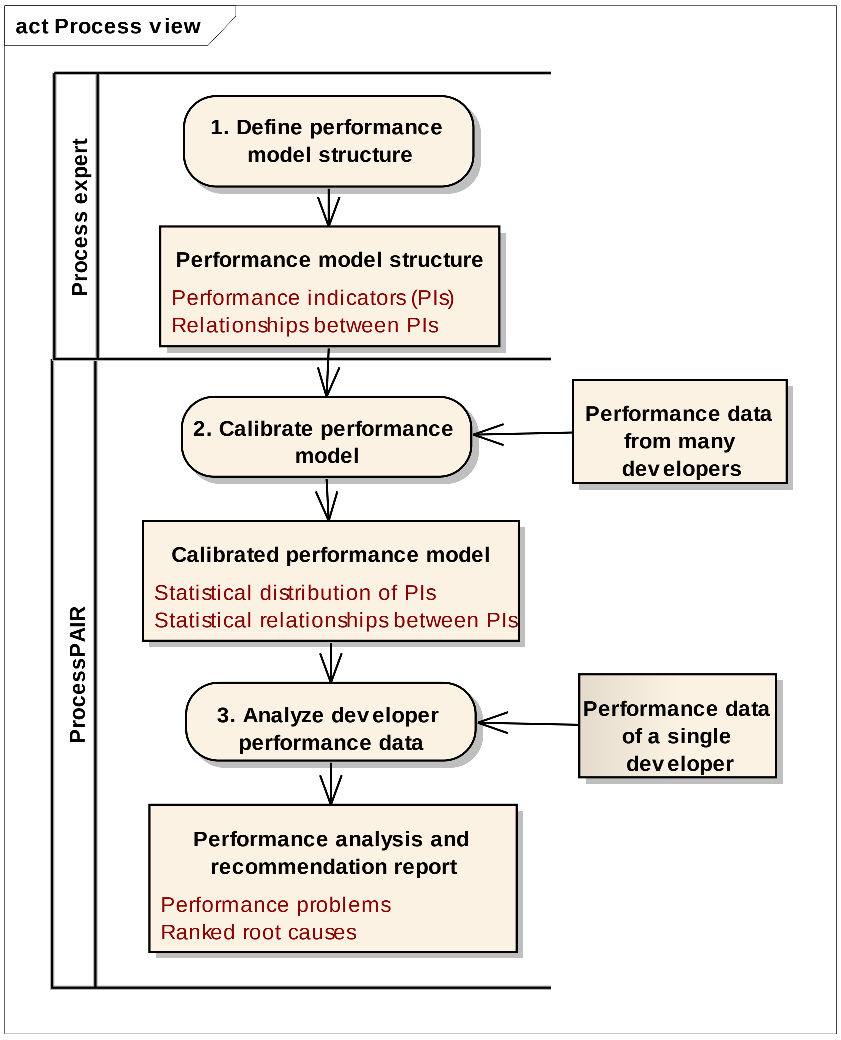

act Process view
Process expert
ProcessPAIR
1. Define performance model structure
Performance model structure
Performance indicators (PIs)
Relationships between PIs
2. Calibrate performance model
Performance data from many developers
Calibrated performance model
Statistical distribution of PIs
Statistical relationships between PIs
3. Analyze developer performance data
Performance data of a single developer
Performance analysis and recommendation report
Performance problems
Ranked root causes

Figure 3.1: UML activity diagram (UML2.5, 2015) depicting the main activities and artifacts in the ProcessPAIR approach.

### 3.1.1 Problem identification approach

In order to enable the automated identification of performance problems in step 3, one has to first decide on the relevant (top-level) PIs and recommended performance ranges. The relevant top-level PIs are identified by the process expert in step 1. The recommended ranges of each PI are determined automatically in step 2, based on its statistical distribution in the calibration data set, according to the following criteria: the 1/3 values closest to the optimal value correspond to "good" performance (no performance problem); the 1/3 values farthest from the optimal value correspond to "bad" performance (clear performance problem); the 1/3 values in between correspond to intermediate performance (potential performance problem).

The optimal value of each PI is defined by the process expert in step 1. In most cases, the optimal value is implicit in the definition of the PI (e.g, 0 is the optimal value for defect density).

### 3.1.2 Root cause identification approach

In order to enable the automated identification of root causes of performance problems in step 3, one has to first decide on the relevant cause-effect relationships. In our approach, lower-level PIs that may affect, directly or indirectly, top-level PIs according to a cause-effect relationship are specified by the process expert in step 1. Then, in step 3, it is possible to recursively drill down from problematic top-level PIs (with a clear or potential performance problems) to problematic lower-level PIs (with a clear or potential performance problems). PIs that can not be further drilled down indicate the potential root causes.

### 3.1.3 Ranking approach

When multiple potential root causes (problematic child PIs) are identified for a performance problem in a top-level PI, it is important to rank them (step 3).

Let $X_1,...,X_n$ be a set of lower-level PIs that affect the value of a higher-level PI $Y$. Let us assume that such relationship can be described by a function $Y = f(X_1,...,X_n)$, representing an exact formula for deriving $Y$ or a regression formula derived from the calibration data set. We rank the factors $X_i$ according to the values of a *ranking coefficient* $\rho_i$ that represents a cost-benefit estimate of improving each factor $X_i$ whilst keeping the other factors unchanged.

The benefit on $Y$ of a change in the value of a factor $X_i$ can be expressed by the resulting relative variation in the value of $Y$, i.e., $\Delta Y/Y$.

As for the cost of changing the value of a factor $X_i$, intuitively, the closest the value is to the optimal value, in terms of percentiles, the more difficult (and less important) it is to improve it. Let us denote by $P_i(X_i) = F_i(X_i) - F_i(o_i)$ the *percentile distance* of $X_i$ to the optimal value, where $F_i$ represents the approximate cumulative distribution function of $X_i$, and $o_i$ represents the optimal value of $X_i$. Our base heuristic is that equal relative variations in the $P_i's$ have similar costs. So, we take as cost estimate the relative variation $\Delta P_i/P_i$.

We approximate the cost-benefit ratio using partial derivatives (for small variations) to derive a ranking coefficient $(\rho_i)$:

$$\frac{\frac{\Delta Y}{Y}}{\frac{\Delta P_i}{P_i}} = \left[\frac{\Delta Y}{\Delta X_i}\left(\frac{X_i}{Y}\right)\right] \times \left[\frac{\Delta X_i}{\Delta P_i}\left(\frac{P_i}{X_i}\right)\right] \approx \left[\frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right)\right] \times \left[\frac{1}{\frac{\partial P_i}{\partial X_i}}\left(\frac{P_i}{X_i}\right)\right] = \sigma_{X_i \to Y} \times \pi_{P_i \to X_i} = \rho_i$$

The first coefficient, $\sigma_{X_i \to Y} = \frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right)$, is a sensitivity coefficient (Saltelli et al., 2008) that computes the impact of small variations in the value of a factor $X_i$ on the value of $Y$, whilst keeping all the other factors unchanged. The second coefficient, $\pi_{P_i \to X_i} = \frac{1}{\frac{\partial P_i}{\partial X_i}}\left(\frac{P_i}{X_i}\right) = \frac{F_i(X_i)-F_i(o_i)}{X_i F_i'(X_i)}$, which we call a *percentile coefficient*, computes the impact of small variations in the current percentile distance $(P_i)$ of $X_i$ to the optimal value $(o_i)$ on the value of $X_i$. We denote by $F_i'(X_i)$ the first derivative of $F_i(X_i)$, representing the probability density function.

The optimal value of each PI is provided by the process expert in step 1. The approximate statistical distribution of each PI is automatically computed from the calibration data set in step 2. Regarding the sensitivity coefficients, in case parent and child PIs have an exact relationship, the sensitivity coefficient is provided by the process expert in step 1; in case parent and child PIs are statistically related, a regression model and a corresponding sensitivity coefficient are computed automatically in step 2 from the calibration data set.

The next sections give further details about the three steps of our approach.

## 3.2 Performance model definition

A performance model for a development process under consideration is defined by means of the following information (see Figure 3.2):

1. Set of relevant base measures generated by the development process under consideration, at the project level, with their name (short name and long name), description, scale (minimum value, maximum value and precision digits), and measurement units.
2. Set of relevant top-level PIs, described by the same attributes as the base measures, plus the optimal value (usually implied by the definition of each PI) and formula for computation from base measures.
3. Child PIs that affect directly or indirectly (by a cause-effect relationship) the top-level PIs according to a formula or statistical evidence.
4. Sensitivity coefficients (Saltelli et al., 2008) $\sigma_{X_i \to Y} = \frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right), i = 1...n$ for each PI $Y$ that is affected by child PIs $X_1, ..., X_n$, according to an exact formula $Y = f(X_1, ..., X_n)$.

Concrete examples can be consulted in chapter 4.

## 3.3 Performance model calibration

The PM is automatically calibrated by ProcessPAIR from training data sets (containing values of base measures for many users and projects), generating the following data (also visible in Figure 3.2):

- approximate statistical distribution of each PI, represented by a cumulative distribution function;
- recommended performance ranges for each PI;
- regression models and sensitivity coefficients between related PIs, but not by an exact formula.

### 3.3.1 Approximate cumulative distribution functions

The approximate cumulative distribution function of each PI could be obtained by computing a theoretical distribution that best fits the training data, or by linear interpolation between a few

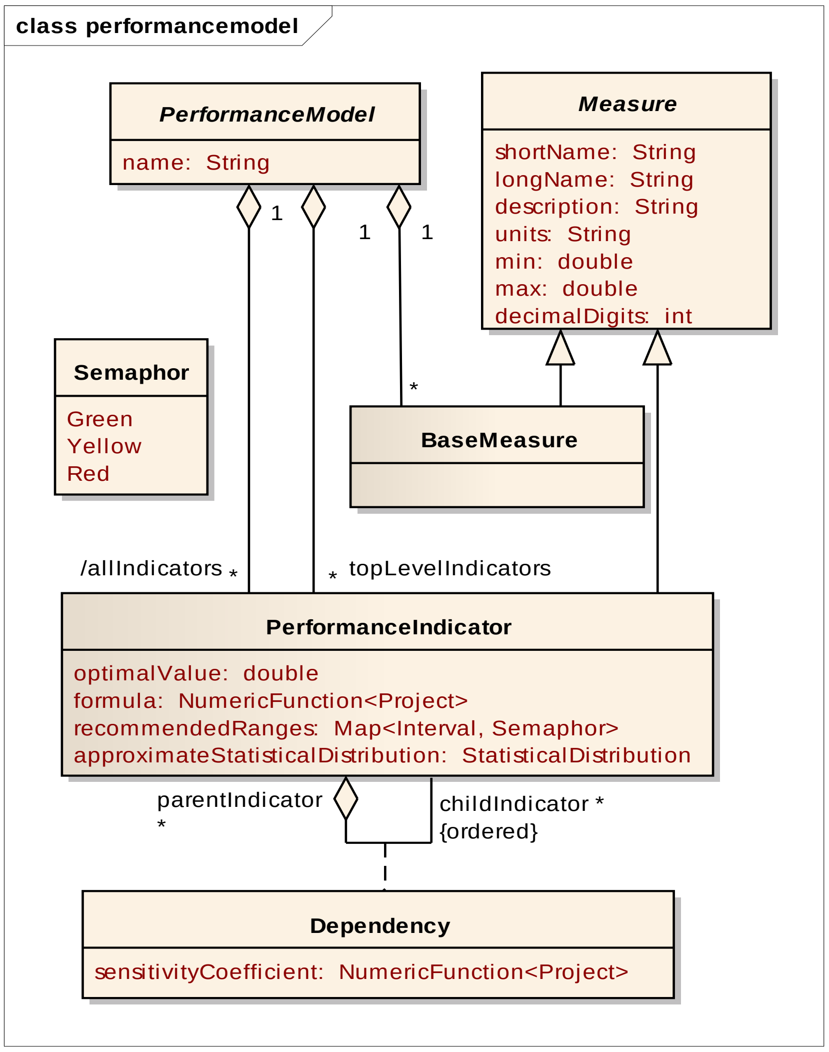


Figure 3.2: UML class diagram depicting the main concepts involved in model definition and calibration.

percentiles computed from the training data. Since different PIs may follow different types of continuous distributions or may even follow a hybrid continuous-discrete distribution, with non-zero probability at the ends of the scale, we opted for the second method.

To balance fit, smoothing and storage space, we sample the cumulative frequency distribution of the training data for the following relative frequencies: 0%, 1%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 95%, 99%, 100%, maximum relative frequency for which the observed value equals the minimum of the scale (if any observed), and minimum relative frequency for which the observed value equals the maximum of the scale (if any observed). The approximate cumulative distribution function of the PI under consideration is a piecewise linear function that connects the sampling points, as illustrated in Figure 3.3. The only calibration information that needs to be stored are the sampling points (illustrated by red circles in Figure 3.3).

Examples of concrete distributions can be consulted in chapter 4.

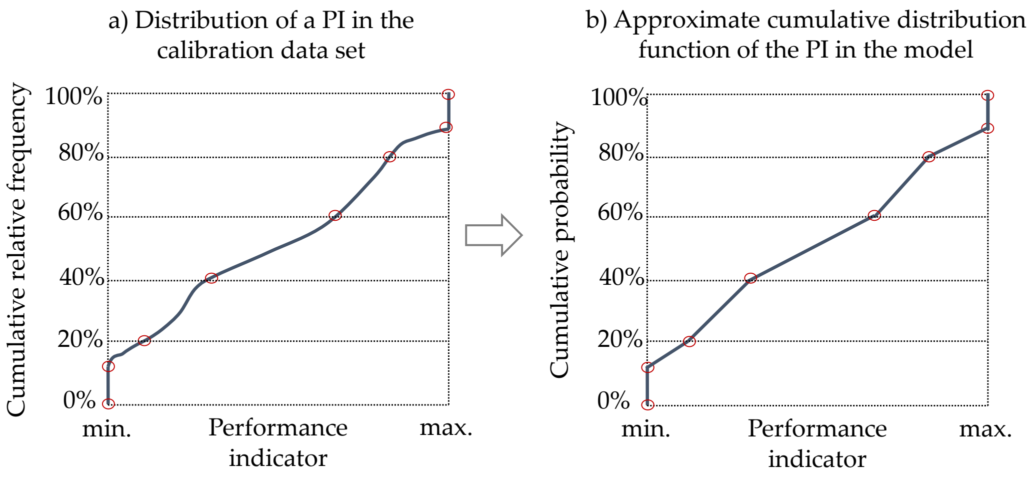


Figure 3.3: Illustration of the procedure for obtaining an approximate cumulative distribution function of a PI (example with hybrid continuous-discrete distribution, and sampling at 20% intervals).

### 3.3.2 Performance ranges

As previously explained, performance ranges are needed for classifying values of each PI of a subject under analysis into three semaphores:

- green - no performance problem;
- yellow - a possible performance problem;
- red - a clear performance problem.

Such ranges are determined automatically from the approximate cumulative distribution function computed in the previous step, so that there is an even distribution of training data points by the colors (1/3 of data points per color). In case the optimal value is located in one of the extremes of the scale, the 'green' range is also located in the same extreme of the scale, the 'red' range in the other extreme, and the 'yellow' range in the middle. In that case, thresholds are computed from the cumulative distribution function based on terciles and subsequently rounded to the number of precision digits specified for the PI under consideration. In case the optimal value is located in the middle of the scale, we split the intervals to the left and the right of the optimal value based on terciles, in order to derive the performance ranges, as illustrated in Figure 3.4.

### 3.3.3 Regression models and sensitivity coefficients

Sensitivity coefficients between PIs not related by an exact formula are computed by first determining a regression model from the calibration data set and subsequently computing the corresponding sensitivity coefficients.

Assume that in the first step of our method the process expert indicated that a performance indicator $Y$ is affected by $X_1,...,X_n (n \geq 1)$, but $Y$ is not determined by those factors according

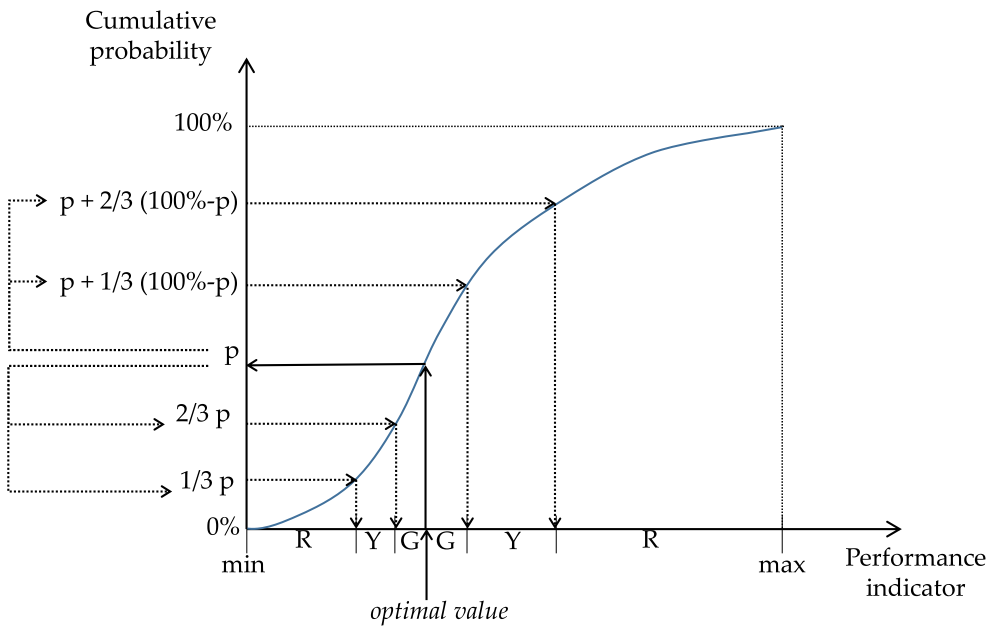


Figure 3.4: Illustration of the procedure for determining the green (G), yellow (Y) and red (R) ranges from the cumulative distribution function of a PI, in case the optimal value lays in the middle of the scale.

to an exact formula. An example in the PSP is the *Review Yield* (percentage of defects found in reviews) which, according to (Kemerer and Paulk, 2009), is affected by the *Review Rate* (size units reviewed per time units).

In such case, we first compute a regression model $\hat{Y} = f(X_1, \ldots, X_n)$ from the calibration data set for predicting the value of $Y$ from the values of $(X_1, ..., X_n)$, and subsequently compute the sensitivity coefficients from that model as

$$\sigma_{X_i \to Y} = \frac{\partial \hat{Y}}{\partial X_i} \frac{X_i}{Y}, i = 1, ..., n. \tag{3.1}$$

Because the relationship between the PIs involved may be non-linear, instead of computing a global linear model (derived by simple or multiple regression, depending on the value of $n$), we have the option to compute a piecewise linear model organized as a regression tree (Breiman, 1984). The type of model to use (global or piecewise linear model) is selected by the process expert in the first step of our method. A regression tree (Breiman, 1984) is a binary decision tree; at each non-leaf node, the value of one of the factors *($X_i$)* is compared to a so called split value in order to decide the sub-tree to follow. Hence, each leaf will correspond to a subset (n-dimensional cube or cell) of the space of possible values of $X_1, ..., X_n$. Since we are interested in computing a linear piecewise model, each leaf node $k$ has an associated linear model $\hat{Y} = \beta_{k,0} + \beta_{k,1} X_1 + ... + \beta_{k,n} X_n$, computed from the points in the calibration data set that fall in that cube, by simple *(n=1)* or multiple *(n>1)* linear regression.

Special provisions are taken for handling variables with a constant value: if a variable $\mathrm{X_i}$ is constant, then $\beta_{k,i} = 0$; if $Y$ is constant, then $\beta_{k,1} = ... = \beta_{k,n} = 0$.

In the recursive tree construction process, we use the following split criterion: as suggested in (Shalizi, 2009), we select the split variable ($\mathrm{X_i}$) that provides the highest decrease of the tree SSE (sum of squared errors); for each tentative split variable $\mathrm{X_i}$, we do a split as closest as possible to a binary split that guarantees disjoint $\mathrm{X_i}$ values on both segments (this is important to handle repeated values). We use the following stop criterion: a cell size cannot contain less than a specified minimum number of training data points (100 for inner cells, and a number between 100 and 25 for border cells, depending on the number of borders).

An alternative to regression trees are multivariate adaptive regression splines (MARS) (Friedman, 1991), which also handle non-linearities and interactions between factors, and generate more compact models than regression trees. In the experiments conducted, we obtained lower mean absolute errors (MAE) and root mean squared errors (RMSE) with regression trees as compared to MARS, particularly in cases where the dependent variable $Y$ is strongly determined by the factors $X_1,...,X_n$ (with a coefficient of determination $R^2$ close to 1) in a non-linear manner. So, we use the described regression trees, in spite of the higher storage space required, but can also support MARS in the future.

In each leaf node $k$ we only have to store the coefficients $\beta_{k,0}, \beta_{k,1}, ..., \beta_{k,n}$ of the model. In each non-leaf node, we have to store the index and value of the split variable. During performance analysis (step 3), once determined the cell $k$ corresponding to a data point $(X_1,...,X_n,Y)$ under consideration, the sensitivity coefficients are simply computed at runtime as:

$$\sigma_{X_i \to Y} = \frac{\partial \hat{Y}}{X_i}\left(\frac{X_i}{Y}\right) = \beta_{k,i}\frac{X_i}{Y}(i = 1,...,n). \tag{3.2}$$

See example of excerpt of concrete regression tree in Figure 4.3 in chapter 4.

### 3.3.4 Data set filtering

Instead of using the full data set for calibration, it may make sense to filter the data points to be used for calibration. One possibility is to restrict (or filter) the data points to the ones most similar to a given user profile. This requires that each data point in the calibration data set, besides values for the base measures, also contains values for the variables that can be used for filtering. Those variables should be specified by the process expert together with the PM.

Similarity is computed with the Gower similarity co-efficient (Gower, 1971), because it provides a measure of proximity adequate for mixed data types (combining numeric and categorical data).

Let $n$ be the number of variables under consideration (the ones constrained in the supplied profile), and let $p$ and $t$ be vectors that contain the values of those variables in the supplied profile and in a data point in the training data set, respectively.

In the case of a categorical variable $k$ (e.g., programming language), the similarity $s_k$ between $t$ and $p$ regarding $k(1 \leq k \leq n)$, is a number between 0 (least similar) and 1 (most similar) computed as (Gower, 1971):

$$s_k = \begin{cases} 1 & \text{if } t_k = p_k, \\ 0 & \text{otherwise .} \end{cases} \tag{3.3}$$

In the case of a numerical variable with a finite range (e.g., programming experience in years), the similarity is computed as (Gower, 1971):

$$s_k = 1 - \frac{|t_k - p_k|}{r_k} \tag{3.4}$$

where $\mathrm{r_k}$ is the range of values for the $k$th variable (distance between minimum and maximum allowed or observed values).

However, many numerical variables of interest have positive values spread along a wide range, with many comparatively small values and a few comparatively large values. E.g., in the training data set that we used for calibrating the PSP model (see chapter 4), 1,604 developers (out of 3,112) indicated their programming experience in KLOC, with a median of 10 KLOC and a maximum above 10,000 KLOC. Using the above formula, the similarity would be close to 0 for most of the pairs of data points. In such cases, the process expert may specify a monotonic scale transformation function to be applied prior to using the similarity formula. E.g., the function $1 - e^{\frac{-x}{\tau}}$, where $x$ is the variable of interest and $\tau$ is a constant, transforms the $[0, +\infty[$ interval to the [0,1[ interval.

Another possibility (that can be selected by the process expert) is to compute the similarity based on the ranking of $x$ in the training data set (i.e., the relative position of $x$ in the sorted multiset of values in the training dataset).

By default, all the variables are given equal weights, so the similarity $s$ between $t$ and $p$ (considering all the variables) becomes a simple average:

$$s = \frac{1}{n} \sum_{k=1}^{n} s_k \tag{3.5}$$

Regarding the number of most similar data points to select, we use the following criteria: for statistical significance, at least 50 data points are selected; additionally, all data points with a similarity to the given profile greater or equal than 0.9 are selected. These numbers can be configured. Data points with undefined values for any of the variables under consideration are not selected.

## 3.4 Model-based performance analysis

The base performance data of a subject under analysis (developer, team or company) that needs to be uploaded by ProcessPAIR, consists of the values of the base measures defined in the selected performance model for a sequence of projects (see *Subject*, *Project*, and *ProjectBaseMeasure* in Figure 3.5).

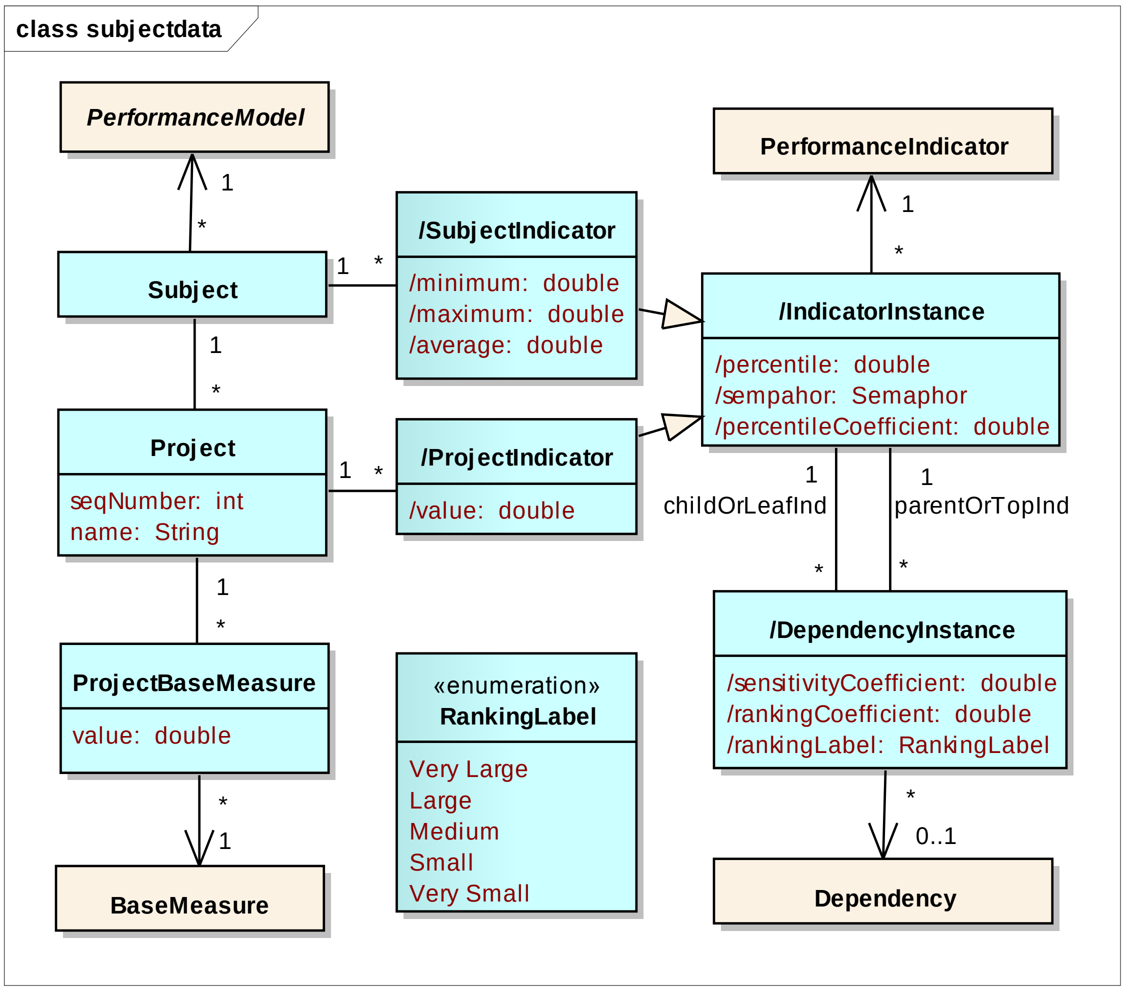


Figure 3.5: UML class diagram depicting the main concepts involved in the analysis of subject data.

### 3.4.1 Project level information

Based on the values of the base measures, ProcessPAIR computes the following data for each PI and project (see *ProjectIndicator* and *IndicatorInstance* in Figure 3.5):

- *value* – computed from the base measures and PI's formula; this value may be undefined in some cases;
- *normalized percentile* – computed from the previous value and the statistical distribution of the PI in the PM, normalized so that 100% corresponds to the optimal value and 0% corresponds to extreme values to the left or to the right of the optimal value. Formally, denoting by $F_i$, $o_i$ and $x$ the approximate cumulative distribution function, optimal value and

actual value of the PI under consideration, respectively, and by $N_i$ the normalized percentile, we have:

$$N_i = \begin{cases} 1 & \text{if } x = o_i, \\ \frac{F_i(x)}{F_i(o_i)} & \text{if } x < o_i, \\ 1 - \frac{F_i(x)-F_i(o_i)}{1-F_i(o_i)} & \text{if } x > o_i. \end{cases} \quad (3.6)$$

- *semaphore* – determined by comparing the PI value with the calibrated performance ranges, or, equivalently, computed from the normalized percentile as follows: green for the 66.7%-100% range, yellow for the 33.3%-66.7% range, and red for the 0%-33.3% range;
- *percentile coefficient* – as previously explained, the percentile coefficient of a PI $X_i$ with value $x$ is computed as $\pi_{P_i \to X_i} = \frac{F_i(x)-F_i(o_i)}{xF_i^{'}}$, where $o_i$ is the optimal value of $X_i$, $F_i$ is the approximate cumulative distribution function of $X_i$, and $F_i^{'}$ is the first derivative of $F_i$. The percentile coefficient is needed for computing the ranking coefficient, but is hidden from the normal user.

For each dependency defined in the PM and project, the following information is computed (see *DependencyInstance* in Figure 3.5):

- *sensitivity coefficient* computed from the project data and the sensitivity formula defined in the PM. The sensitivity coefficient is needed for computing the ranking coefficient, but is hidden from the normal user;
- *ranking coefficient* - computed as the product of the previous sensitivity coefficient and the percentile coefficient of the child PI;
- *ranking label* – a discretization of the ranking coefficient in terms of T-shirt sizes, for user presentation purposes, according to the mapping $\{]-\infty, 0.01[\rightarrow \mathit{VerySmall}, [0.01, 0.1[\rightarrow \mathit{Small}, [0.1, 1[\rightarrow \mathit{Medium}, [1, 10[\rightarrow \mathit{Large}, [10, +\infty[\rightarrow \mathit{VeryLarge}\}$. Such thresholds may be interpreted as follows: considering an improvement by 10% in the child PI (reduction of the percentile distance to the optimal value), the estimated improvement in the parent PI will be $\leq$0.1% (very small), 0.1%-1% (small), 1% - 10% (medium), 10%-100% (large), >=100% (very large).

The following information is also computed between child indicators (at any level) and top-level indicators:

- *composite sensitivity coefficient* – recursively computed from the (elementary) sensitivity coefficients, as explained in section 3.4.3;
- *composite ranking coefficient* – computed as the product of the previous sensitivity coefficient and the percentile coefficient of the child PI;
- *composite ranking label* – computed in a way similar to the ranking label for directly PIs.

### 3.4.2 Subject level information

Summary information for each PI is computed at the subject level (see *SubjectIndicator* and *IndicatorInstance* in Figure 3.5):

- *minimum, maximum, average* – simple statistics calculated from the values computed at the project level;

- *aggregated normalized percentile* – weighted average of the normalized percentiles computed at the project level, using an exponentially decaying weight for older projects with a configurable "memory retention factor". Formally, denoting by $x_1, ..., x_m$ and $N_{i,1}, ..., N_{i,m}$ the values and corresponding normalized percentiles of the PI under consideration $(X_i)$ for a sequence of projects $1, ..., m$, by $f$ the memory retention factor $(0 < f < 1, e.g., 0.85)$, by $defined(x_j)$ a predicate that indicates if $x_i$ is defined for project $j$, and by $N_{i,*}$ the aggregated normalized percentile, we have:

$$N_{i,*} = \frac{\sum_{j \in \{1,...,m\} \wedge defined(x_j)} f^{\text{m-j}} N_{i,j}}{\sum_{j \in \{1,...,m\} \wedge defined(x_j)} f^{\text{m-j}}} \tag{3.7}$$

- *aggregated semaphore* – computed from the aggregated normalized percentile, in a similarly way to what is done at the project level.

The following aggregated information is computed at the subject level, for direct dependencies:

- *aggregated ranking coefficient* - weighted average of the ranking coefficients computed at the project level, using an exponentially decaying weight for older projects with a configurable factor, in a way similar to the aggregated normalized percentile;

- *aggregated ranking label* – computed in a way similar to the ranking label at the project level.

The following aggregated information is also computed at the subject level, for indirect dependencies:

- *aggregated composite ranking coefficient* - weighted average of the composite ranking coefficients computed at the project level, using an exponentially decaying weight for older projects with a configurable factor, in a way similar to the aggregated normalized percentile;

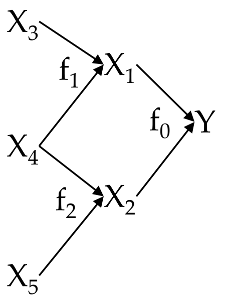
- *aggregated composite ranking label* – computed in a way similar to the ranking label at the project level.

### 3.4.3 Ranking calculations

#### 3.4.3.1 Compositive sensitivity coefficients

In the general case, the root causes to be ranked $X_1, \ldots, X_n$ may be indirect children of the problematic top-level PI under consideration *(Y)*. In this case, the function $f$ that relates the involved PIs may be expressed as a composite function, based on the elementary functions that relate each PI with its direct children. As a consequence, the sensitivity coefficients may be computed using the laws of partial derivatives of composite and multivariate functions. For example, regarding the PIs illustrated in Figure 3.6, we can compute a sensitivity coefficient between indirectly related PIs as follows:

$$\begin{aligned}
\sigma_{X_4} \rightarrow_Y &= \frac{\partial Y}{\partial X_4}\left(\frac{X_4}{Y}\right) \\
&= \frac{\partial Y}{\partial X_1}\frac{\partial X_1}{\partial X_4}\left(\frac{X_4}{Y}\right) + \frac{\partial Y}{\partial X_2}\frac{\partial X_2}{\partial X_4}\left(\frac{X_4}{Y}\right) \\
&= \frac{\partial Y}{\partial X_1}\frac{X_1}{Y}\frac{\partial X_1}{\partial X_4}\frac{X_4}{X_1} + \frac{\partial Y}{\partial X_2}\frac{X_2}{Y}\frac{\partial X_2}{\partial X_4}\frac{X_4}{X_2} \\
&= \sigma_{X_1} \rightarrow_Y \sigma_{X_4} \rightarrow_{X_1} + \sigma_{X_2} \rightarrow_Y \sigma_{X_4} \rightarrow_{X_2}
\end{aligned} \tag{3.8}$$



$$\sigma_{X_3\rightarrow Y} = \sigma_{X_1\rightarrow Y}\sigma_{X_3\rightarrow X_1}$$

$$\sigma_{X_4\rightarrow Y} = \sigma_{X_1\rightarrow Y}\sigma_{X_4\rightarrow X_1} + \sigma_{X_2\rightarrow Y}\sigma_{X_4\rightarrow X_2}$$

$$\sigma_{X_5\rightarrow Y} = \sigma_{X_2\rightarrow Y}\sigma_{X_5\rightarrow X_2}$$

Figure 3.6: Computation of sensitivity coefficients between indirectly related PIs.

In general, denoting by $child(X_j)$ the direct child of an indicator $X_j$, the (composite) sensitivity coefficient $\sigma'$ between any two PIs (directly related, indirectly related or unrelated) can be computed recursively from the elementary sensitivity coefficients as follows:

$$\sigma'_{X_i\rightarrow X_j} = \begin{cases} \sigma_{X_i\rightarrow X_j} & \text{if } X_i \in child(X_j), \\ 1 & \text{if } i = j, \\ \sum_{k\in child(X_j)} \sigma_{X_i\rightarrow X_k}\sigma_{X_k\rightarrow X_j} & \text{otherwise .} \end{cases} \tag{3.9}$$

#### 3.4.3.2 Corner cases in ranking calculations

The value of a percentile coefficient ($\pi_{P_i \to X_i}$), sensitivity coefficient ($\sigma_{X_i \to Y}$) or ranking coefficient ($\rho_i$) may be indeterminate for corner values (C) of the variable involved ($X_i$), because of non-linearities (making the derivative undefined) or operations such as $0/0$ or $0 \times \infty$. In order to try to remove the indeterminacy, we take the limit, as follows:

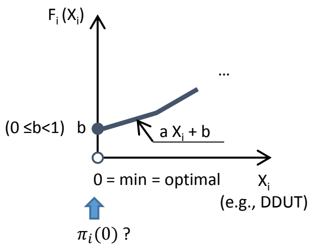
$$\pi_{P_i \to X_i}(C) = \lim_{X_i \to C} \frac{\partial X_i}{\partial P_i}\left(\frac{P_i}{X_i}\right)$$

$$\sigma_{X_i \to Y}(C) = \lim_{X_i \to C} \frac{\partial Y}{\partial X_i}\left(\frac{X_i}{Y}\right)$$

$$\rho_i(C) = \lim_{X_i \to C}\left[\sigma_{X_i \to Y}(x) \times \pi_{P_i \to X_i}(x)\right]$$

In most cases, this is sufficient to remove the indeterminacy. Some representative cases are illustrated in Figure 3.7 and Figure 3.8.

**i)**



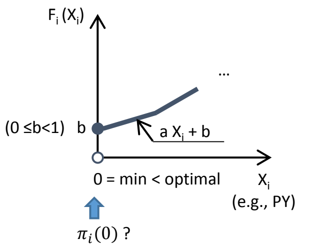
*Not using limits:*

$$\pi_i(0) = \left.\frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)}\right|_{X_i=0} = \frac{b-b}{0 \times F_i'(0)} = \begin{cases} \frac{0}{0 \times undefined}, & if\ b > 0 \\ \frac{0}{0 \times a}, & if\ b = 0 \end{cases} = undefined$$

*Using limits:*

$$\pi_i(0) = \lim_{X_i \to c} \left.\frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)}\right|_{c=0} = \lim_{X_i \to 0+} \frac{(aX_i + b) - b}{X_i\, a} = 1$$

**ii)**



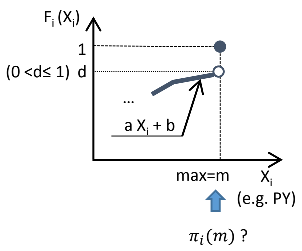
*Not using limits:*

$$\pi_i(0) = \left.\frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)}\right|_{X_i=0} = \frac{b - F_i(o_i)}{0 \times F_i'(0)} = \begin{cases} \frac{b - F_i(o_i)}{0 \times undefined}, & if\ b > 0 \\ \frac{-F_i(o_i)}{0 \times a}, & if\ b = 0 \end{cases} = \begin{cases} undefined, & if\ b > 0 \\ \frac{-F_i(o_i)}{0 \times a}, & if\ b = 0 \end{cases}$$

*Using limits:*

$$\pi_i(0) = \lim_{X_i \to c} \left.\frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)}\right|_{c=0} = \lim_{X_i \to 0+} \frac{(aX_i + b) - F_i(o_i)}{X_i\, a} = -\infty$$

**iii)**



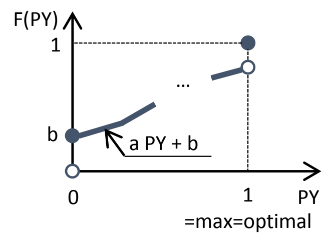
*Not using limits:*

$$\pi_i(m) = \left.\frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)}\right|_{X_i=m} = \frac{1 - F_i(o_i)}{m \times F_i'(m)} = \begin{cases} \frac{1 - F_i(o_i)}{m \times undefined}, & if\ d < 1 \\ \frac{1 - F_i(o_i)}{m \times a}, & if\ d = 1 \end{cases} = \begin{cases} undefined, & if\ d < 1 \\ \frac{1 - F_i(o_i)}{m \times a}, & if\ d = 1 \end{cases}$$

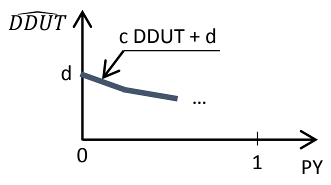
*Using limits:*

$$\pi_i(m) = \lim_{X_i \to c} \left.\frac{F_i(X_i) - F_i(o_i)}{X_i F_i'(X_i)}\right|_{c=m} = \lim_{X_i \to m-} \frac{(aX_i + b) - F_i(o_i)}{X_i\, a} = \frac{d - F_i(o_i)}{m \times a}\ (defined)$$

Figure 3.7: Examples of solving indeterminacy in the calculation of the percentile coefficient by using limits.



$$\pi_{PY}(0) = \lim_{PY\to 0} \frac{F(PY)-F(1)}{PY\,F'(PY)} = \lim_{PY\to 0+} \frac{(a\,PY+b)-1}{PY\,a} = \lim_{PY\to 0+}\frac{b-1}{PY\,a} = -\infty$$



$$\sigma_{PY\to DDUT}(0) = \lim_{PY\to 0}\frac{\partial\,\widehat{DDUT}}{\partial\,PY}\,\frac{PY}{DDUT} = \lim_{PY\to 0}\frac{c\,PY}{DDUT} = 0\ (if\ DDUT \neq 0)$$

*Not using limits:* $\rho_{PY\to DDUT}(0) = \pi_{PY}(0)\times\sigma_{PY\to DDUT}(0) = -\infty\times 0 = undefined$

*Using limits:* $\rho_{PY\to DDUT}(0) = \lim_{PY\to 0}\pi_{PY}(PY)\times\sigma_{PY\to DDUT}(PY) = \lim_{PY\to 0}\frac{b-1}{PY\,a}\frac{c\,PY}{DDUT} = \frac{(b-1)c}{a\,DDUT}\,(defined)$

Figure 3.8: Example of solving indeterminacy in the calculation of the ranking coefficient by using limits.

#### 3.4.3.3 Selection and sorting rules

In this section, we explain how the relevant root causes of an identified performance problem are selected based on the ranking coefficients and semaphores.

Assume that a problematic top-level PI $Y$ was identified (with a red or yellow semaphore) for a specific project or overall for a subject under analysis, as illustrated in Figure 3.9. Also assume that $Y$ has the direct and indirect child PIs indicated in Figure 3.9, with their semaphores and ranking coefficients already calculated.

The selection of the relevant root causes is performed in three steps.

In step 1 (cut non major issues), we cut the child PIs (and their direct or indirect children) that (i) have a green semaphore or (ii) a ranking coefficient to parent less than a defined threshold (e.g., 0.1). A node in the tree with multiple parents is cut only if it (i) has a green semaphore, or (ii) all the ranking coefficients to the parents are less than the defined threshold, or (iii) all the parents have been cut. E.g., In Figure 3.9, $X_7$ is not cut because only one of its two parents was cut.

In step 2 (cut non relevant leafs), we cut the leafs that have a composite ranking coefficient to root less than the defined threshold (e.g., 0.1). This procedure is repeated (bottom-up) until there are no leafs with that condition.

Finally, in step 3 (show only leaf causes), we select the leafs resulting from the previous steps as the relevant root causes to be presented to the user, sorted according to the ranking coefficient to root.

The reason for using the 0.1 threshold is based on the following intuition. In case of a ranking coefficient of 0.1, if we reduce by 10% the percentile distance to the optimal value of a factor $X_i$ , then the relative variation in the value of $Y$ will be just 1%, which is almost insignificant.

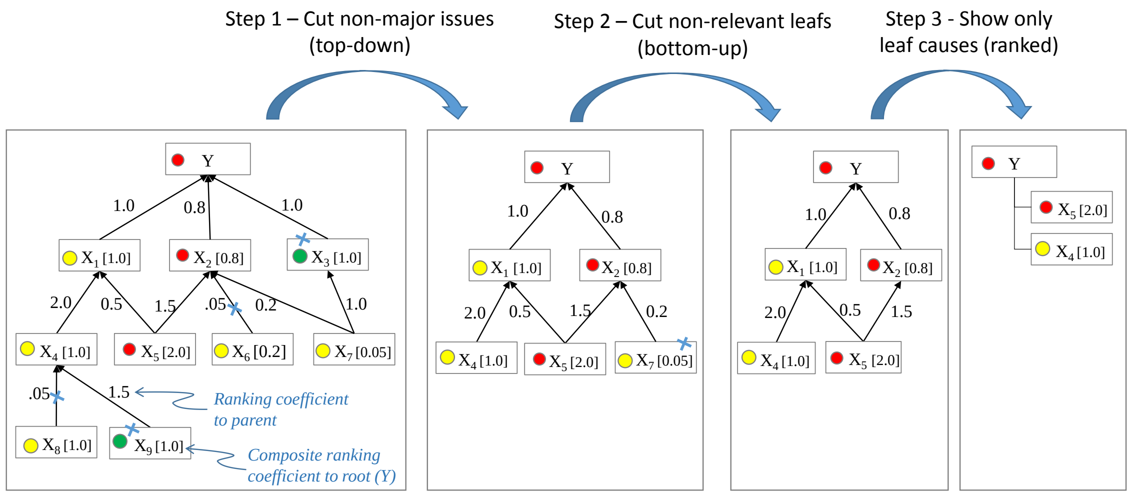

Step 1 – Cut non-major issues (top-down)
Step 2 – Cut non-relevant leafs (bottom-up)
Step 3 - Show only leaf causes (ranked)
Y
X1 [1.0]
X2 [0.8]
X3 [1.0]
X4 [1.0]
X5 [2.0]
X6 [0.2]
X7 [0.05]
X8 [1.0]
X9 [1.0]
Ranking coefficient to parent
Composite ranking coefficient to root (Y)

Figure 3.9: Example of selection and sorting rules

# Chapter 4

# Performance model for the PSP

High-maturity software development processes, making intensive use of metrics and quantitative methods, such as the Team Software Process (TSP) and the accompanying Personal Software Process (PSP), can generate a significant amount of data that can be periodically analyzed to identify performance problems, determine their root causes and devise improvement actions. However, there is a lack of tool support for automating the data analysis and the recommendation of improvement actions, and hence diminish the manual effort and expert knowledge required.

In this chapter we are going to present a comprehensive performance model (the ProcessPAIR model for the PSP), addressing time estimation accuracy, quality and productivity, to enable the automated analysis (see automated analysis method presented in chapter 3) of performance data produced in the context of the PSP, namely, identify performance problems and their root causes, and subsequently recommend improvement actions. Performance ranges and dependencies in the model were calibrated and validated, respectively, based on a large PSP data set referring to more than 30,000 finished projects.

This chapter is organized as follows. Section 4.1 introduces model definition for PSP and covers performance indicators and dependencies for predictability, quality, and productivity. Section 4.2 describes model validation and calibration. In section 4.3, support data for the ranking method is presented which covers sensitivity and percentile coefficients together with ranking example.

## 4.1 Model definition

A performance model comprises a set of performance indicators (top level and nested) and dependencies, with their respective attributes.

### 4.1.1 Performance indicators and dependencies

This section presents a Performance Model (PM) specifically conceived for the PSP. It will summarize (in Figure 4.1 and Tables 4.4 and 4.5) the PIs and dependencies of the PM that we conceived, based on literature review and PSP specifications, for analyzing performance problems and their root causes in the context of the PSP.

The three top-level PIs refer to the major performance aspects usually analyzed: predictability, quality and productivity.

### 4.1.2 Predictability

The major predictability PI in the PSP is the *Time Estimation Accuracy*, which we measure by the ratio between actuals and estimates, to simplify ranking calculations. Because in the PSP's PROBE estimation method (Humphrey, 2005), a time (effort) estimate is obtained based on a size estimate of the deliverable (in added or modified size units) and a productivity estimate (in size per time units), we indicate in Figure 4.1 that the *Time Estimation Accuracy* is affected by the *Size Estimation Accuracy* and the *Productivity Estimation Accuracy* (the exact formula for this dependency is shown in Table 4.5).

Because in the PROBE method productivity estimates are based on historical productivity (Humphrey, 2005), we indicate in Figure 4.1 that the *Productivity Estimation Accuracy* depends on the *Productivity Stability* (refer to exact definition in Table 4.4).

Since in the PSP time is recorded per process phase (Humphrey, 2005), when a productivity stability problem is encountered one can analyze the productivity stability per phase, in order to determine the problematic phase(s). Hence, we indicate in Figure 4.1 a set of PIs for the productivity stability per phase, which together affect the overall productivity stability (the exact formula for this dependency is shown in Table 4.5). Since the scope of the PSP is the development of small programs or components of larger programs, the Requirements, High Level Design, and System Testing phases are absent (they can be found in the more complete TSP (Humphrey and Over, 2010)). In some programming environments the Compile phase may be absent.

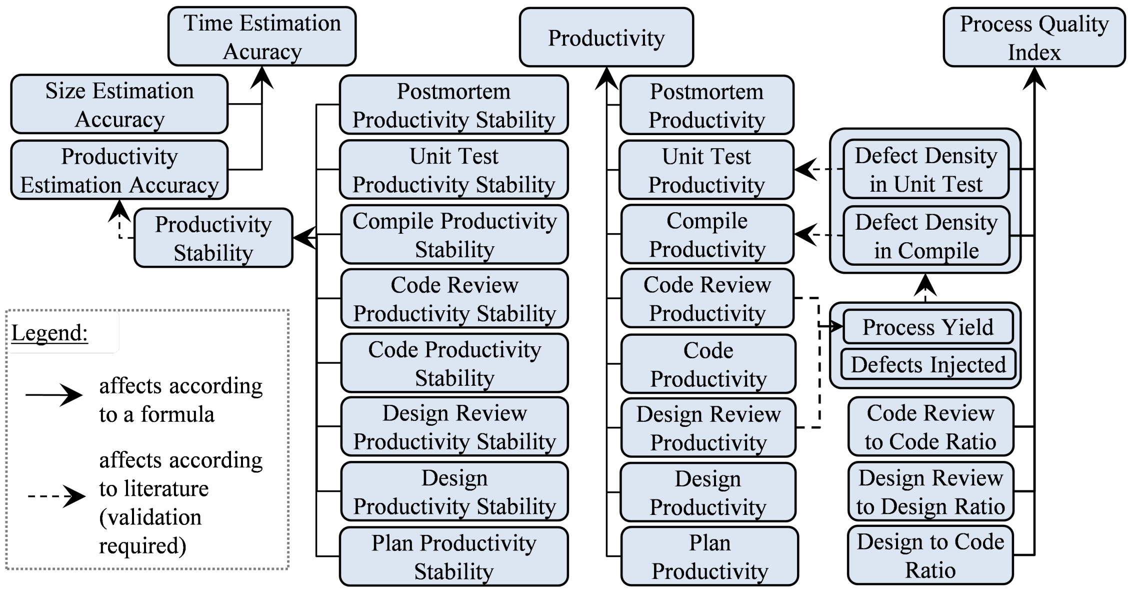

Time Estimation Acuracy
Productivity
Process Quality Index
Size Estimation Accuracy
Productivity Estimation Accuracy
Productivity Stability
Postmortem Productivity Stability
Unit Test Productivity Stability
Compile Productivity Stability
Code Review Productivity Stability
Code Productivity Stability
Design Review Productivity Stability
Design Productivity Stability
Plan Productivity Stability
Postmortem Productivity
Unit Test Productivity
Compile Productivity
Code Review Productivity
Code Productivity
Design Review Productivity
Design Productivity
Plan Productivity
Defect Density in Unit Test
Defect Density in Compile
Process Yield
Defects Injected
Code Review to Code Ratio
Design Review to Design Ratio
Design to Code Ratio
Legend:
affects according to a formula
affects according to literature (validation required)

Figure 4.1: Performance indicators and dependencies.

### 4.1.3 Quality

Product quality is usually measured by post-delivery defect density (Jones, 2000). However, because the scope of the PSP is the development of small programs or components of large programs, post-delivery defects are seldom known.

In the PSP, an aggregated quality measure is proposed—the Process Quality Index (PQI)—that constitutes an effective predictor of post-delivery defect density (Humphrey, 2005) (Daughtrey, 2002). Hence, we use the PQI as the top-level quality indicator to analyze.

The PQI is computed based on five components: the ratio of design time to coding time (indicator of design quality), the ratio of design review time to design time (indicator of design review quality), the ratio of code review time to coding time (indicator of code review quality), the ratio of compile defects to a size measure (indicator of code quality), and the ratio of unit test defects to a size measure (indicator of program quality). The components are normalized to [0, 1] such that 0 represents poor practice and 1 represents desired practice (refer to exact formula in Table 4.5). Hence, in Figure 4.1 we indicate those components as factors that affect the PQI according to a formula.

In turn, both the *Defect Density in Compile* and *Unit Test* are affected by the total density of *Defects Injected* (and found) and the percentage of defects removed before compile and test (called *Process Yield* in the PSP). In fact, high defect densities in compile and test may be caused by a large number of defects injected (because of poor defect prevention) or a large number of defects escaped from previous defect filters (because of poor design and code reviews).

According to Humphrey and Tamura (Humphrey, 2005) (Tamura, 2009), the time spent in reviewing a work product in relation to its size is a leading indicator of the review yield (percentage of defects found) and consequently of the process yield. In a published study (Kemerer and Paulk, 2009), the recommended review rate of 200 lines of code (LOC) per hour or less was found to be an effective rate, identifying nearly two-thirds of the defects in design reviews and more than half in code reviews. Hence, we indicate in Figure 4.1 that the *Process Yield* is affected by the *Design Review Productivity* and the *Code Review Productivity*.

### 4.1.4 Productivity

In general, in the PSP, productivity may be measured in any size units per time units. Any size measure can be used (function points, LOC, etc.) as long as it correlates with effort (in order to enable effort estimation based on size estimation) and can be objectively measured (to automate size measurement and compare actuals and estimates) (Humphrey, 2005). In this study, we use LOC per hour, in spite of its well-known limitations (Wagner et al., 1980) (Jones, 2009), because LOC is the size measure available in the data set we used.

Because in the PSP time is recorded per process phase, when a productivity problem is encountered one can analyze the productivity per phase, in order to determine the problematic phase(s). Hence, we indicate in Figure 4.1 a set of PIs for the productivity per phase, which together affect the overall productivity (the formula for this dependency is shown in Table 4.5).

In turn, the time spent in the compile and test phases (which in the PSP include defect fixing) may be affected by the number of defects to fix, so we indicate in Figure 4.1 that the *Compile Productivity* and the *Unit Test Productivity* may be affected by the *Defect Density in Compile* and *Defect Density in Unit Test*, respectively.

## 4.2 Model validation and calibration

### 4.2.1 Data set

To validate and calibrate the performance model, we used a large PSP data set from the Software Engineering Institute (SEI) referring to 31,140 projects concluded by 3114 engineers during 295 classes of the classic PSP for Engineers I/II training courses running between 1994 and 2005. In this training course, targeting professional developers, each developer develops 10 small projects.

Associated with the PSP Performance Model (see section 4.1) we defined the following data quality checks on the values of base measures that can be applied for any PSP calibration data set:

- out of range values of base measures (namely, negative values);
- inconsistency between defects fixed in Compile and time in Compile: defects (>0) found and fixed in 0 time, or more than 1 hour per defect found and fixed;
- inconsistency between defects fixed in Unit Test and time in Unit Test: defects (>0) found and fixed in 0 time, or more than 2 hours per defect found and fixed;
- inconsistency between actual time and sum of actual time per phase: difference greater than 5 minutes (to accommodate round-off errors);
- inconsistency between total defects and sum of defects injected per phase;
- inconsistency between total defects and sum of defects removed per phase;
- inconsistency between defects injected and removed up to a phase.

In the data set used, we found 250 inconsistencies in 242 data points (project submissions) that can be consulted in Table 4.1.

### 4.2.2 Model validation

The performance model of Figure 4.1 indicates several 'affects according to literature' relationships between pairs of PIs, suggested from literature. In order to validate each relationship, using the PSP data set previously described, we computed the Pearson's linear correlation coefficient ($r_{\text{pearson}}$) and tested the null hypothesis $H_0 : r = 0$ against the alternative hypothesis $H_1 : r > 0$ or $H_1 : r < 0$ , depending on the sign of the expected correlation. Because the PIs under analysis may have non-linear relationships that are not adequately captured by the Pearson's linear correlation coefficient, we also computed the Spearman's (Navidi, 2008) rank correlation coefficient

Table 4.1: Inconsistencies in the PSP data set.

| **Inconsistency** | **Frequency** |
|---|---|
| Inconsistency between defects fixed in Compile and time in Compile: more than 1 hour per defect found and fixed | 34 |
| Inconsistency between defects fixed in Unit Test and time in Unit Test: more than 2 hours per defect found and fixed | 101 |
| Inconsistency between actual time and sum of actual time per phase: difference greater than 5 minutes (to accommodate round-off errors) | 115 |
| **Total** | 250 |

($r_{\text{spearman}}$). The Spearman's test checks if increasing values of $X$ are monotonically associated with increasing ($r > 0$) or decreasing ($r < 0$) values of $Y$, independently of the form of the relationship. The results are presented in Table 4.2. In all cases, it was observed a statistically significant correlation between the PIs under analysis, regarding both the Person and Spearman correlation, so the null hypothesis was rejected. We tested other dependencies, but only present in this thesis dependencies that exhibited a statistically significant correlation (Mushtaq and Faria, 2014).

### 4.2.3 Model calibration

Regarding model calibration, we derived automatically from the PSP data set a set of thresholds and ranges (Table 4.4) for classifying values of each PI into three categories: green—no performance problem; yellow—a possible performance problem; red—a clear performance problem.

The ranges are derived based on the optimal value defined for each PI and the actual distribution in the PSP data set.

In most cases, the optimal value follows directly from the semantics of the PI, being it the maximum of the scale (e.g., 1 for the PQI), the minimum (e.g., 0 for the Defect Density in Unit Test), or a special value in between (e.g., 1 for *Estimation Accuracy* and *Productivity Stability*).

In other cases, in order to balance conflicting aspects such as quality and productivity with an ultimate economic impact, we selected a recommended value from literature (we could not calibrate those values from the PSP data set, because some economic impacts occur later in the process). Regarding the *Code Review Productivity* (also called *Review Rate*), if reviews are performed too fast then the quality of the reviews may suffer; if reviews are performed too slowly, then the productivity is negatively affected. In this case, we selected the recommended value of 200 LOC per hour (Humphrey, 2005) (Tamura, 2009). A similar reasoning was followed for the *Design Review Productivity*. As for the *Design to Code Ratio*, *Design Review to Design Ratio*, and *Code Review to Code Ratio* (components of the PQI) the optimal values selected correspond to the desired values indicated in the definition of the PQI (Humphrey, 2005) (Daughtrey, 2002).

After defining the optimal values, the thresholds for the 'green', 'red', and 'yellow' ranges were calibrated automatically by ProcessPAIR tool based on the PSP data set, following the rules explained in chapter 3. The results can be consulted in Table 4.4.

Table 4.2: Results of the correlation tests.

| | | Correlation tests ($H_0 : r = 0$) | | | | | |
|---|---|---|---|---|---|---|---|
| Affected Indicator *(Y)* | Affecting Indicator *(X)* | $H_1$ | $n^a$ | $r_{pearson}$ | $r_{spearman}$ | $p^b$ | $Reject H_0?^c$ |
| Defect Density in Unit Test | Process Yield | r<0 | 9612 | -0.27 | -0.40 | <2e-16 | Yes |
| Defect Density in Unit Test | Defects Injected | r>0 | 27648 | 0.72 | 0.65 | <2e-16 | Yes |
| Defect Density in Compile | Process Yield | r<0 | 9612 | -0.24 | -0.39 | <2e-16 | Yes |
| Defect Density in Compile | Defects Injected | r>0 | 27648 | 0.74 | 0.69 | <2e-16 | Yes |
| Process Yield | Design Review Productivity | r<0 | 9371 | -0.17 | -0.23 | <2e-16 | Yes |
| Process Yield | Code Review Productivity | r<0 | 9548 | -0.17 | -0.25 | <2e-16 | Yes |
| Unit Test Productivity | Defect Density in Unit Test | r<0 | 27625 | -0.20 | -0.64 | <2e-16 | Yes |
| Compile Productivity | Defect Density in Compile | r<0 | 27625 | -0.34 | -0.72 | <2e-16 | Yes |
| Product. Estim. Accuracy | Productivity Stability | r>0 | 24574 | 0.46 | 0.65 | <2e-16 | Yes |

[a] denotes the number of data points with defined values for the variables under analysis.
[b] is a probability that indicates the statistical significance of the correlation coefficient in the one-tailed test.
[c] We reject the null hypothesis if $p < 0.05$, for a 5% significance level.

Regarding estimation accuracy (for time, size, productivity), the green range corresponds to an error of approximately $\pm 20\%$, which matches what is usually considered an acceptable error (Fenton and Bieman, 2014) (McConnell, 2006).

Regrading productivity, the green range corresponds to approximately 35 LOC/hour which also matches to recommendations from literature (Prechelt and Unger, 1999).

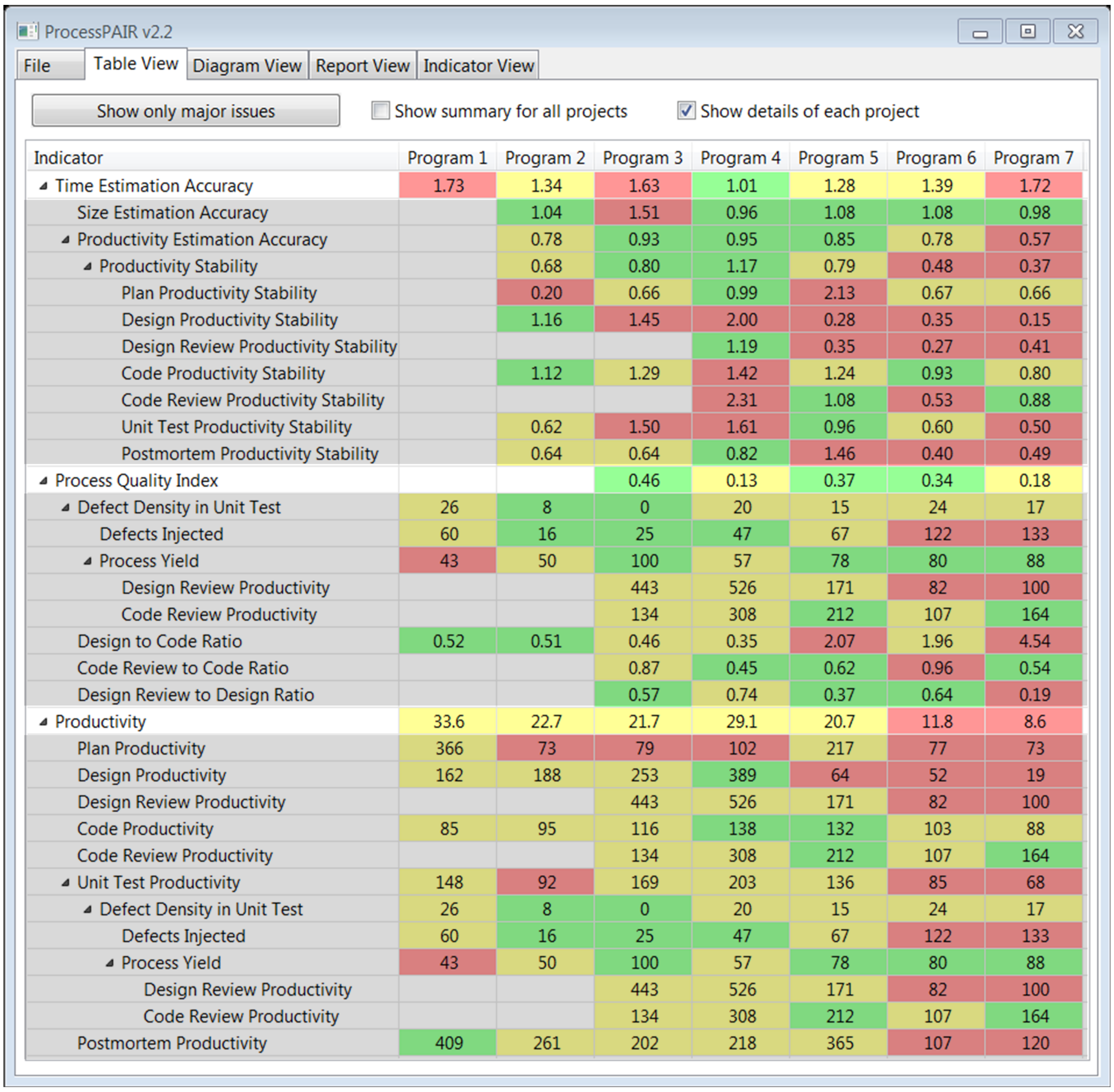
Regarding defect density in Unit Test and Compile, the green ranges correspond to <=11 and <=15 defects/KLOC, respectively, which are a bit wider than the PSP recommendations (<=5 defects/KLOC for Unit Test and <=10 defects/KLOC for Compile) (Humphrey, 2005).

## 4.3 Support data for the ranking method

### 4.3.1 Introduction

The performance model presented so far allows the automated identification of performance problems and their potential root causes for individual developers. However, when multiple potential root causes are identified for a performance problem, it does not provide enough information to prioritize or rank those root causes. For example, Figure 4.2 suggests five potential causes for the poor productivity in project 7— poor productivity in Plan, Design, Design Review, Unit Test and Postmortem phases—but does not indicate their relative importance. In chapter 3, we presented an approach to rank those factors according to a ranking coefficient that represents a cost–benefit estimate of improvement efforts.

As explained in chapter 3, for computing the ranking coefficient, we need to obtain: an approximate statistical distribution (see Figure 4.5) of each performance indicator; a regression model for PIs not related by a formula (but instead related according to statistical evidence); sensitivity coefficients for PIs related by a formula.



| Indicator | Program 1 | Program 2 | Program 3 | Program 4 | Program 5 | Program 6 | Program 7 |
|---|---|---|---|---|---|---|---|
| ◢ Time Estimation Accuracy | 1.73 | 1.34 | 1.63 | 1.01 | 1.28 | 1.39 | 1.72 |
| Size Estimation Accuracy |  | 1.04 | 1.51 | 0.96 | 1.08 | 1.08 | 0.98 |
| ◢ Productivity Estimation Accuracy |  | 0.78 | 0.93 | 0.95 | 0.85 | 0.78 | 0.57 |
| ◢ Productivity Stability |  | 0.68 | 0.80 | 1.17 | 0.79 | 0.48 | 0.37 |
| Plan Productivity Stability |  | 0.20 | 0.66 | 0.99 | 2.13 | 0.67 | 0.66 |
| Design Productivity Stability |  | 1.16 | 1.45 | 2.00 | 0.28 | 0.35 | 0.15 |
| Design Review Productivity Stability |  |  |  | 1.19 | 0.35 | 0.27 | 0.41 |
| Code Productivity Stability |  | 1.12 | 1.29 | 1.42 | 1.24 | 0.93 | 0.80 |
| Code Review Productivity Stability |  |  |  | 2.31 | 1.08 | 0.53 | 0.88 |
| Unit Test Productivity Stability |  | 0.62 | 1.50 | 1.61 | 0.96 | 0.60 | 0.50 |
| Postmortem Productivity Stability |  | 0.64 | 0.64 | 0.82 | 1.46 | 0.40 | 0.49 |
| ◢ Process Quality Index |  |  | 0.46 | 0.13 | 0.37 | 0.34 | 0.18 |
| ◢ Defect Density in Unit Test | 26 | 8 | 0 | 20 | 15 | 24 | 17 |
| Defects Injected | 60 | 16 | 25 | 47 | 67 | 122 | 133 |
| ◢ Process Yield | 43 | 50 | 100 | 57 | 78 | 80 | 88 |
| Design Review Productivity |  |  | 443 | 526 | 171 | 82 | 100 |
| Code Review Productivity |  |  | 134 | 308 | 212 | 107 | 164 |
| Design to Code Ratio | 0.52 | 0.51 | 0.46 | 0.35 | 2.07 | 1.96 | 4.54 |
| Code Review to Code Ratio |  |  | 0.87 | 0.45 | 0.62 | 0.96 | 0.54 |
| Design Review to Design Ratio |  |  | 0.57 | 0.74 | 0.37 | 0.64 | 0.19 |
| ◢ Productivity | 33.6 | 22.7 | 21.7 | 29.1 | 20.7 | 11.8 | 8.6 |
| Plan Productivity | 366 | 73 | 79 | 102 | 217 | 77 | 73 |
| Design Productivity | 162 | 188 | 253 | 389 | 64 | 52 | 19 |
| Design Review Productivity |  |  | 443 | 526 | 171 | 82 | 100 |
| Code Productivity | 85 | 95 | 116 | 138 | 132 | 103 | 88 |
| Code Review Productivity |  |  | 134 | 308 | 212 | 107 | 164 |
| ◢ Unit Test Productivity | 148 | 92 | 169 | 203 | 136 | 85 | 68 |
| ◢ Defect Density in Unit Test | 26 | 8 | 0 | 20 | 15 | 24 | 17 |
| Defects Injected | 60 | 16 | 25 | 47 | 67 | 122 | 133 |
| ◢ Process Yield | 43 | 50 | 100 | 57 | 78 | 80 | 88 |
| Design Review Productivity |  |  | 443 | 526 | 171 | 82 | 100 |
| Code Review Productivity |  |  | 134 | 308 | 212 | 107 | 164 |
| Postmortem Productivity | 409 | 261 | 202 | 218 | 365 | 107 | 120 |

Figure 4.2: Evaluation of top-level and nested (shaded) PIs in the case study for projects 1 to 7.

### 4.3.2 Support data for the sensitivity coefficient

In Table 4.5, in case of PIs not related by a formula, firstly a linear regression model from the PSP data set is computed, and subsequently the sensitivity coefficient is derived from the linear regression model. However, because in many cases the relationships are not linear, ProcessPAIR also has the option to compute a step-wise linear regression model (this is currently the default option) as explained in section 3.3.3. The obtained regression model is stored in the generated calibration file. An example of part of such a model is shown in Figure 4.3.

In case of PIs related by a formula, Table 4.5 shows the computation of the sensitivity coefficient for the dependencies identified in the Performance Model of Figure 4.1.

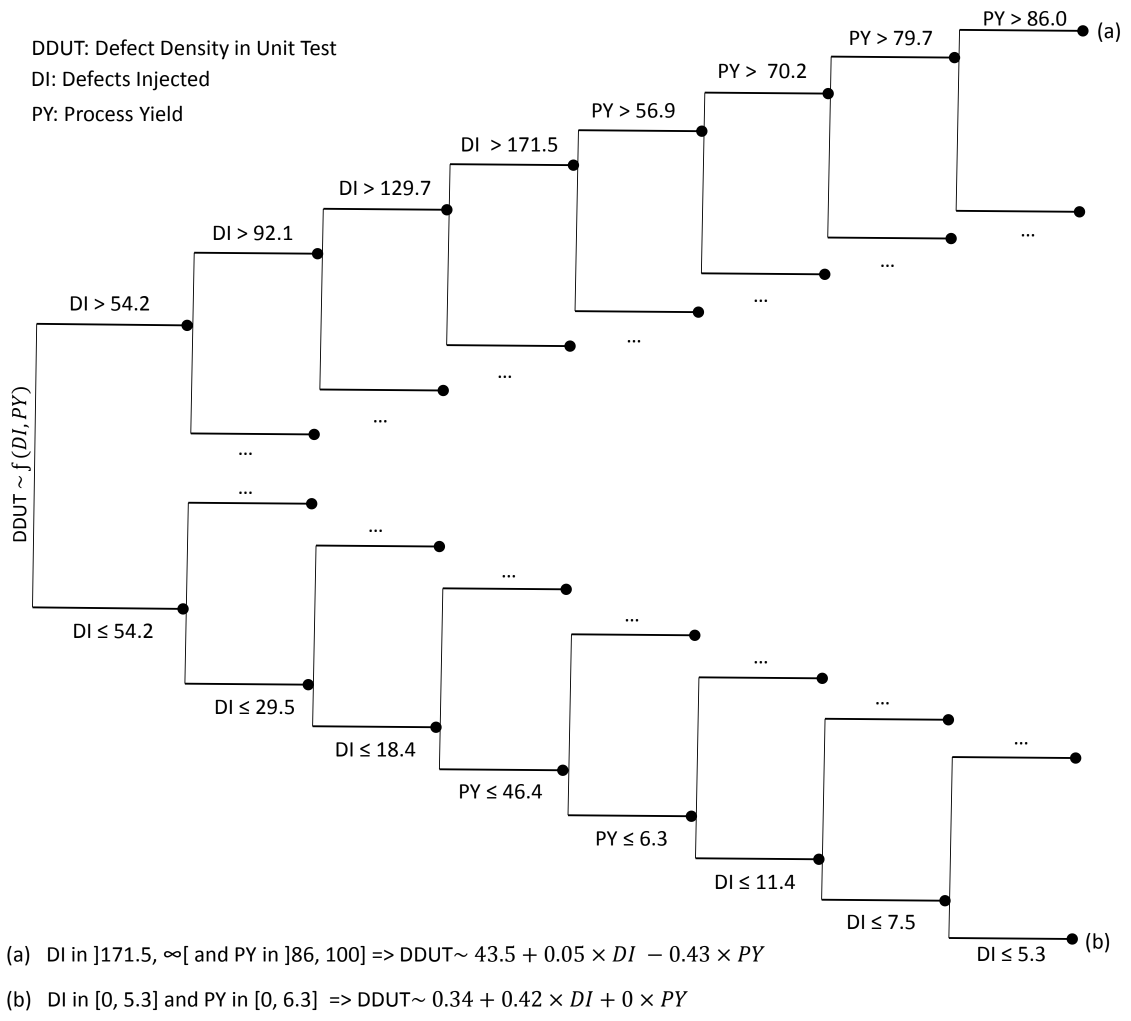


Figure 4.3: Example of generated regression tree.

For example, one of the top-level indicators of the PSP performance model is the Time Estimation Accuracy, computed from base measures as:

$$TimeEstimationAccuracy = \frac{ActualTime}{EstimatedTime}, \tag{4.1}$$

being 1 the optimal value. Since in the PSP's PROBE estimation method (Humphrey, 2005), a time estimate is obtained based on a size estimate of the deliverable (in added or modified size units) and a productivity estimate (in size per time units), we consider that the *Time Estimation Accuracy (TimeEA)* depends on the *Size Estimation Accuracy (SizeEA)* and the *Productivity Estimation Accuracy (ProdEA)*, also defined as ratios between actual and estimated values. From their definitions, we conclude that these PIs are related by the formula

$$TimeEA = \frac{SizeEA}{ProdEA}. \tag{4.2}$$

From this formula, we derive the sensitivity coefficients:

$$\sigma_{SizeEA \to TimeEA} = \frac{\partial TimeEA}{\partial SizeEA} \frac{SizeEA}{TimeEA} = \frac{1}{ProdEA} \frac{SizeEA}{TimeEA} = 1. \tag{4.3}$$

$$\sigma_{ProdEA \to TimeEA} = \frac{\partial TimeEA}{\partial ProdEA} \frac{ProdEA}{TimeEA} = -\frac{SizeEA}{ProdEA^2} \frac{ProdEA}{TimeEA} = -1. \tag{4.4}$$

This mean that *TimeEA* is equally sensitive to *SizeEA* and *ProdEA* (although in opposite directions). E.g., a 5% increase in the value of the *SizeEA*, whilst keeping *ProdEA* unchanged, leads to a 5% increase in the value of *TimeEA*. Similarly, a 5% increase in the value of the *ProdEA*, whilst keeping *SizeEA* unchanged, leads to a 5% decrease in the value of *TimeEA*.

These inter-related PIs can be represented graphically as depicted in the upper part of Figure 4.4.

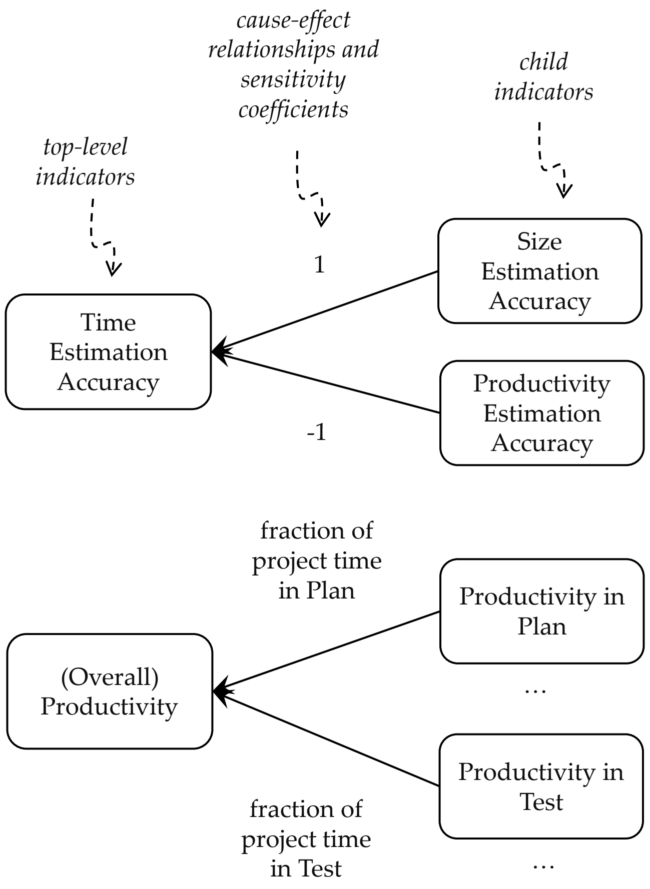


Figure 4.4: Graphical representation of part of a PM for the PSP.

In the previous example, the sensitivity coefficients are constant (see Figure 4.4). For a different example, let's analyse another top-level PI in the PSP performance model – the *Productivity*

*(Prod)*, which is computed from base measures as:

$$Prod = \frac{ActualSize}{ActualTime}. \tag{4.5}$$

In this model, the (overall) Productivity indicator has a set of child indicators representing the productivity per phase, denoted by $Prod_k$, where $k$ may be *Plan, Design, Design Review, Code, Code Review, Compile, Test or Postmortem.* The productivity per phase is computed from base measures as:

$$Prod_k = \frac{ActualSize}{ActualTime_k}. \tag{4.6}$$

where $ActualTime_k$ is the actual time spent in phase $k$.

Since

$$ActualTime = \sum_k ActualTime_k \tag{4.7}$$

we conclude that the PIs under consideration are related by the formula

$$Prod = \frac{1}{\sum_k \frac{1}{Prod_k}}. \tag{4.8}$$

From this formula, we derive the sensitivity coefficients:

$$\sigma_{Prod_k \to Prod} = \frac{\partial Prod}{\partial Prod_k}\frac{Prod_k}{Prod} = -\frac{-\frac{1}{(Prod_k)^2}}{\left(\sum_k \frac{1}{Prod_k}\right)^2}\frac{Prod_k}{Prod} = \frac{Prod}{Prod_k} = \frac{ActualTime_k}{ActualTime} \tag{4.9}$$

which is the actual fraction of project time spent in phase k.

This means that the overall productivity is more sensitive on the productivity of the more time consuming phases. E.g., if a developer spends 50% of the project time in Test, and improves the Productivity in Test by 8% (whilst keeping the productivity in the other phases unchanged), the overall productivity will improve by 4%.

### 4.3.3 Support data for the percentile coefficient

The cumulative distribution function $F_i$ needed for calculating the percentile coefficient $\pi_{P_i \to X_i}$ can be obtained by computing a theoretical distribution that best fits the historical data, or by linear interpolation between a few percentiles computed from the historical data. Because some of the performance indicators exhibit a hybrid continuous-discrete distribution, with non-zero probability at one or both ends of the scale (refer to *Process Yield, Defect Density in Compile, and Defect Density in Unit Test* in Figure 4.5), we opted for the second method. The shapes of the cumulative distribution functions $F_i$ constructed this way from the historical data are depicted in Figure 4.5. The calculation of percentile coefficient ($\pi_{P_i \to X_i}$) with this approach is illustrated in Figure 4.6.

Table 4.3: Ranking calculations for the factors that affect the overall productivity.

| i | Variable | Value (LOC/hour) | Percentile ($F_i$) [1] | Probability Density ($F'_i(x)$) [1] | Percentile[3] Coefficient $\pi_i = \frac{F_i(x)-F_i(o_i)^2}{xF'_i(x)}$ | Sensitivity[3] Coefficient $\sigma_i = \frac{Prod}{Prod_k}$ | Ranking[3] Coefficient $\rho_i = \pi_i \times \sigma_i$ |
|---|---|---|---|---|---|---|---|
| 0 | Productivity | 8.63 | 7% | 0.00936 | 11.45 | | |
| 1 | Plan Productivity | 73.5 | 10% | 0.00223 | 5.48 | 0.117 | 0.64 (3rd) |
| 2 | Design Productivity | 19.4 | 3% | 0.00172 | 29.11 | 0.446 | 12.98 (1st) |
| 3 | Design Review Productivity | 100.0 | 7% | 0.00066 | 14.26 | 0.086 | 1.23 (2nd) |
| 4 | Code Productivity | 87.8 | 45% | 0.00693 | 0.91 | 0.098 | 0.09 (7th) |
| 5 | Code Review Productivity | 163.6 | 18% | 0.00211 | 2.39 | 0.053 | 0.13 (6th) |
| 6 | Unit Test Productivity | 67.9 | 18% | 0.00353 | 3.42 | 0.127 | 0.43 (4th) |
| 7 | Postmortem Productivity | 120.0 | 20% | 0.00220 | 3.03 | 0.072 | 0.22 (5th) |

[1] Computed by liner interpolation between a few percentiles computed from the training data set.
[2] The optimal value assumed here is $o_i = \infty$, so $F_i(o_i) = 1$
[3] Absolute values

### 4.3.4 Ranking example

Table 4.3 presents productivity values from a concrete project (out of a case study), as well as all the calculations performed to rank the factors (productivity per phase) that affect the overall productivity. Regarding the sensitivity coefficient, the phases that consume more effort (i.e., with lower productivity) - Design and Unit Test - are ranked at the top 2 positions. However, the productivity in Unit Test is significantly closer to the optimal value (in terms of percentiles) than, for example, in Design Review, so, when computing the combined ranking coefficient, the productivity in Unit Test goes down to the 4th position. In the final ranking, the top two phases which productivity should be improved (for improving the overall productivity with the best cost-benefit ratio) are the Design and Design Review phases. By contrast, in Figure 4.2 all the phases with a value in the red range (percentile below 33%) - all but the Code phase in this case - would be indicated to the user as equally important for improvement.

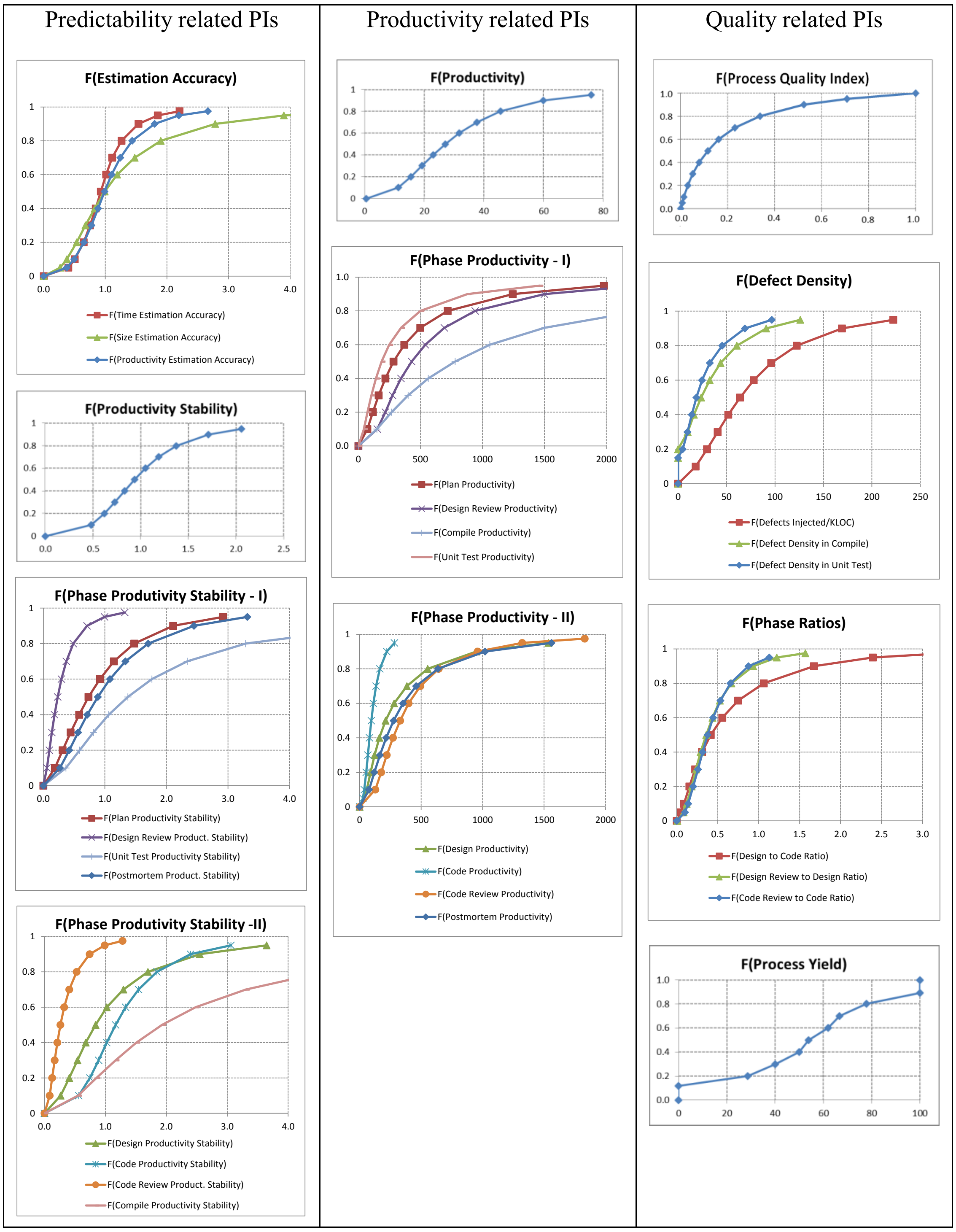
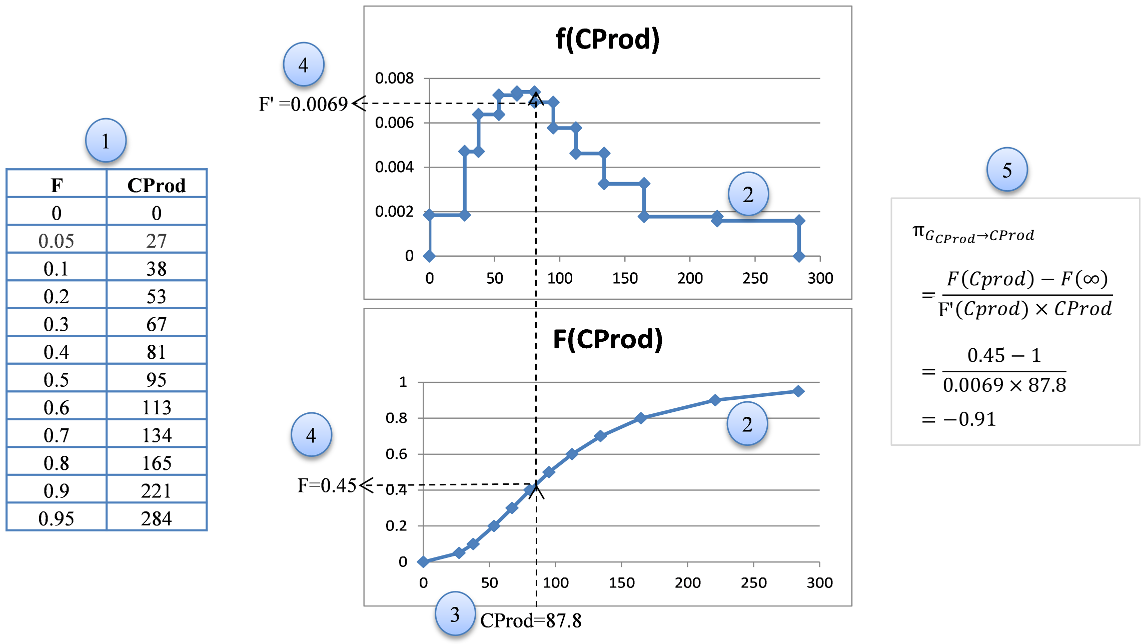


Figure 4.5: Approximate cumulative distribution functions for all PIs in our model derived from the PSP data set.



| F | CProd |
|---|---|
| 0 | 0 |
| 0.05 | 27 |
| 0.1 | 38 |
| 0.2 | 53 |
| 0.3 | 67 |
| 0.4 | 81 |
| 0.5 | 95 |
| 0.6 | 113 |
| 0.7 | 134 |
| 0.8 | 165 |
| 0.9 | 221 |
| 0.95 | 284 |

$$\pi_{G_{CProd}\to CProd}$$
$$= \frac{F(Cprod) - F(\infty)}{F'(Cprod) \times CProd}$$
$$= \frac{0.45 - 1}{0.0069 \times 87.8}$$
$$= -0.91$$

Figure 4.6: Computing the percentile coefficient for Code Productivity based on percentiles extracted from the historical data.

Table 4.4: Performance indicators and ranges (with optimal values underlined).

| Indicator | Formula | Performance Ranges | | |
|---|---|---|---|---|
| | | Green | Yellow | Red |
| Time Estimation Accuracy | $\frac{ActualTime}{EstimatedTime}$ | [0.87, 1.19] $\underline{1}$ | [0.70, 0.87[ ∪ ]1.19, 1.55] | [0, 0.70[∪]1.55, ∞[ |
| Size Estimation Accuracy | $\frac{ActualSize}{EstimatedSize}$ | [0.85, 1.22] $\underline{1}$ | [0.64, 0.85[ ∪ ]1.22, 1.63] | [0, 0.64[∪]1.63, ∞[ |
| Productivity Estimation Accuracy | $\frac{ActualProductivity}{EstimatedProductivity}$ | [0.82, 1.21] $\underline{1}$ | [0.61, 0.82[ ∪ ]1.21, 1.57] | [0, 0.61[∪]1.57,∞[ |
| Productivity Stability | $\frac{CurrentProductivity}{HistoricalProductivity_*}(_*\sum size/\sum effort)$ | [0.80, 1.19] $\underline{1}$ | [0.60, 0.80[ ∪ ]1.19 1.55] | [0, 0.60[∪]1.55,∞[ |
| Process Quality Index | (refer to definition in Table 4.5) | [0.21, $\underline{1}$] | [0.06, 0.21[ | [0, 0.06[ |
| Defect Density in Unit Test | $\frac{\#DefectsfoundandremovedinUnitTest}{ActualSize(KLOC)}$ | [$\underline{0}$, 11] | ]11, 30] | ]30, ∞[ |
| Defect Density in Compile | $\frac{\#DefectsfoundandremovedinCompile}{ActualSize(KLOC)}$ | [$\underline{0}$, 15] | ]15, 43] | ]43, ∞[ |
| Defects Injected | $\frac{\#Defectsfoundinallphases}{ActualSize(KLOC)}$ | [$\underline{0}$, 45] | ]45, 90] | ]90, ∞[ |
| Defects Density | $\frac{\#Defectsremovedinallphases}{ActualSize(KLOC)}$ | [$\underline{0}$, 45] | ]45, 90] | ]90, ∞[ |
| Process Yield | $\frac{\#DefectsremovedbeforeCompileandTest}{\#DefectsinjectedbeforeCompileandTest} \times 100$ | [65, $\underline{100}$] | [43, 65[ | [0, 43[ |
| Design Quality | $min\left(\frac{DesignTime}{CodeTime}, 1\right)$ | [0.70, $\underline{1.0}$] | [0.26, 0.70[ | [0, 0.26[ |
| Design Review Quality | $min\left(\frac{2*DesignReviewTime}{DesignTime}, 1\right)$ | [1, $\underline{1}$] | [0.51, 1[ | [0, 0.51[ |
| Code Review Quality | $min\left(\frac{2*CodeReviewTime}{CodeTime}, 1\right)$ | [1, $\underline{1}$] | [0.56, 1[ | [0, 0.56[ |
| Productivity | $\frac{ActualSize(LOC)}{ActualTime(hours)}$ | [35.7, $\underline{\infty}$[ | [20.6, 35.7[ | ]0, 20.6[ |
| Plan Productivity | $\frac{ActualSize(LOC)}{PlanActualTime(hours)}$ | [459, ∞[ | [185, 459[ | ]0, 185[ |
| Design Productivity | $\frac{ActualSize(LOC)}{DesignActualTime(hours)}$ | [353, $\underline{\infty}$[ | [137, 353[ | ]0, 137[ |
| Design Review Productivity | $\frac{ActualSize(LOC)}{DesignReviewActualTime(hours)}$ | [232, 493] $\underline{300}$ | [160, 232[ ∪ ]493, 889] | ]0, 160[∪]889, ∞[ |
| Code Productivity | $\frac{ActualSize(LOC)}{CodeActualTime(hours)}$ | [128, $\underline{\infty}$[ | [73, 128[ | ]0, 73[ |
| Code Review Productivity | $\frac{ActualSize(LOC)}{CodeReviewActualTime(hours)}$ | [163, 332] $\underline{200}$ | [119, 163[ ∪ ]332, 571] | ]0, 119[∪]571, ∞[ |
| Compile Productivity | $\frac{ActualSize(LOC)}{CompileActualTime(hours)}$ | [1370, $\underline{\infty}$[ | [467, 1370[ | ]0, 467[ |
| Unit Test Productivity | $\frac{ActualSize(LOC)}{UnitTestActualTime(hours)}$ | [315, $\underline{\infty}$[ | [120, 315[ | ]0, 120[ |
| Postmortem Productivity | $\frac{ActualSize(LOC)}{PostmortemActualTime(hours)}$ | [427, $\underline{\infty}$[ | [185, 427[ | ]0, 185[ |

[a] Open ranges are represented with the Bourbaki notation.

Table 4.5: Dependencies and sensitivity coefficients between related performance indicators.

| **Affected Indicator** *(Y)* | **Affecting Indicator**$(X_\mathrm{i})$ | **Exact Formula or Regression Formula:** $Y = f(X_1, \ldots, X_n)$ | **Sensitivity Coefficient** $\sigma_{X_i \to Y} = \frac{\delta Y}{\delta X_i}\left(\frac{X_i}{Y}\right)$ |
|---|---|---|---|
| Time Estimation Accuracy *(TimeEA)* | Size Estimation Accuracy *(SizeEA)*<br>Productivity Estimation Accuracy *(PEA)* | $TimeEA = \frac{SizeEA}{PEA}$ | 1<br>-1 |
| Productivity Estimation Accuracy *(PEA)* | Productivity Stability *(ProdS)* | $PEA \sim 0.593 + 0.455 \times ProdS$ | $0.455 \times \frac{ProdS}{PEA}$ |
| Productivity Stability *(ProdS)* | Productivity Stability in Phase $k(ProdS_k)$ | $ProdS = 1/\sum_k \frac{HistF_k^a}{ProdS_k}$ | Fraction of time in phase $k$<br>(in current project) |
| Process Quality Index *(PQI)* | Defect Density in Unit Test *(DDUT)*<br>Defect Density in Compile *(DDC)*<br>Design to Code Ratio *(D2C)*<br>Design Review to Design Ratio *(DR2D)*<br>Code Review to Code Ratio *(CR2C)* | $PQI = min\left(\frac{10}{DDUT+5}, 1\right) \times min\left(\frac{20}{DDC+10}, 1\right) \times min\left(\frac{D2C}{1}, 1\right) \times min\left(\frac{DR2D}{0.5}, 1\right) \times min\left(\frac{CR2C}{0.5}, 1\right)$ | -DDUT/(DDUT+5),<br>if DDUT>5; 0, otherwise<br>-DDC/(DDC+ 10),<br>if DDC>10; 0, otherwise<br>1, if D2C<1; 0, otherwise<br>1, if DR2D<0.5; 0, otherwise<br>1, if CR2C<0.5; 0, otherwise |
| Defect Density in Unit Test *(DDUT)* | Process Yield *(PY)*<br>Defects Injected *(DI)* | $DDUT \sim 28.5 - 0.20 \times PY$<br>$DDUT \sim 1.6 + 0.38 \times DI$ | $-0.20 \times PY/DDUT$<br>$0.38 \times DI/DDUT$ |
| Defect Density in Compile *(DDC)* | Process Yield *(PY)*<br>Defects Injected *(DI)* | $DDC \sim 28.33 - 0.22 \times PY$<br>$DDC \sim 0.19 + 0.45 \times DI$ | $-0.22 \times PY/DDC$<br>$0.45 \times DI/DDC$ |
| Process Yield *(PY)* | Design Review Productivity *(DRProd)*<br>Code Review Productivity *(CRProd)* | $PY \sim 57.59 - 0.0030 \times DRProd$<br>$-0.0048 \times CRProd$ | $-0.0030 \times DRProd/PY$<br>$-0.0048 \times CRProd/PY$ |
| Productivity *(Prod)* | Productivity in Phase $k(Prod_k)$ | $Productivity = 1/\sum_k \frac{1}{Productivity_k}$ | Fraction of time in phase $k$ |
| Unit Test Productivity *(UTProd)* | Defect Density in Unit Test *(DDUT)* | $UTProd \sim 5524.2 \times DDUT$ | $-4.2 \times DDUT/UTProd$ |
| Compile Productivity (*CompProd*) | Defect Density in Compile *(DDC)* | $CompProd \sim 230817 \times DDC$ | $-17 \times DDC/CompProd$ |

[a] $HistF_k$ = *historical fraction of time in phase k.*

# Chapter 5

# The ProcessPAIR tool implementation

This chapter presents ProcessPAIR, a novel tool designed to help developers analyze their performance data with less effort, by automatically identifying and ranking performance problems and their potential root causes, so that subsequent manual analysis for the identification of deeper causes and improvement actions can be properly focused.

The analysis is based on performance models (chapter 4) defined manually by process experts in the process under consideration, and calibrated automatically from the performance data of many developers. The chapter is organized as follows. Section 5.1 presents tool architecture. Sections 5.2 and 5.3 explain the model calibration and file selection user interfaces. Section 5.4 presents different views for analysing the performance results and finally section 5.5 presents API for defining a performance model as a tool extension.

## 5.1 Architecture

ProcessPAIR is currently implemented as a Java standalone application, comprising a core framework (with 12.8 KLOC, representing 80% of the code base), independent of the process under analysis, and an extension for the PSP (with 3.2 KLOC, representing 20% of the code base), as depicted in Figure 5.1. Other extensions may be easily implemented in the future for other processes. The core framework comprises three layers:

- a graphical user interface layer at the top (*gui* package);
- an intermediate logic layer responsible for the representation and manipulation of performance models (PMs) (*performancemodel* package) and subject data under analysis (*subjectdata* package);
- a layer with common utilities at the bottom (*statistics* package).

The PSP extension (*pspextension* package) contains the definition of PMs for the PSP and subject data loaders from the most relevant project management tools used by PSP Developers – the SEI's PSP Student Workbook and Process Dashboard (`http://www.processdash.com/`)

The following external libraries are used:

- *SWTChart* (`http://www.swtchart.org/`) – an open-source light weight chart component, based on SWT, used by the *gui* package of our tool for creating charts;
- *UCanAccess* (`http://ucanaccess.sourceforge.net/site.html`) -an open-source Java JDBC driver implementation that allows client programs to read/write Microsoft Access databases, used by the *pspextension* package of our tool to read SEI's PSP Student Workbook files;
- *JAMA* (`http://math.nist.gov/javanumerics/jama/`) – a *Java matrix* package, used by the *statistics* package of our tool for computing regression and other calculations.

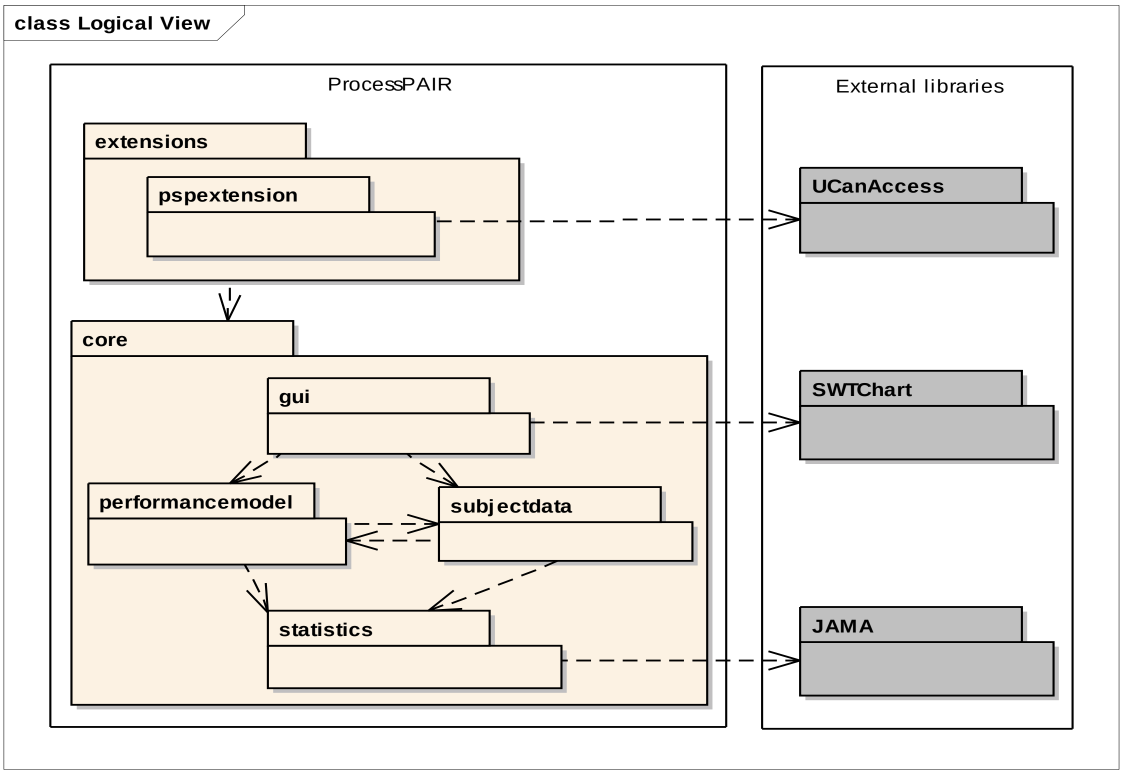
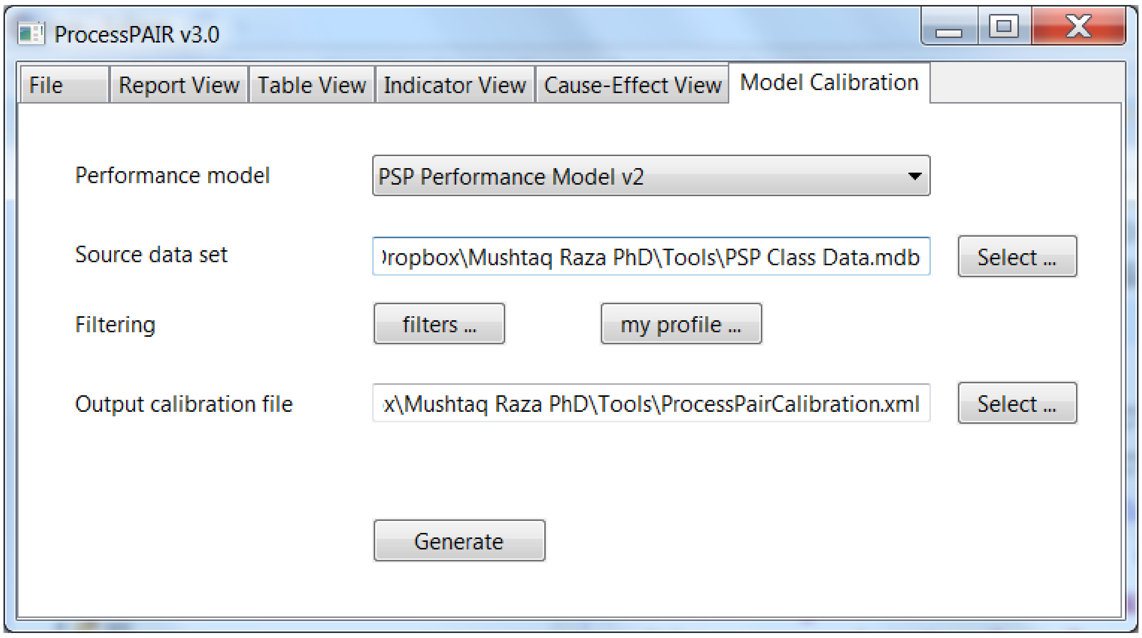


Figure 5.1: UML package diagram depicting the logical architecture of the ProcessPAIR tool.

## 5.2 Model calibration user interface

The user interface for performing the automatic calibration is shown in Figure 5.2. The user has to select the performance model to be calibrated (from the list of PMs previously defined as tool extensions), the file with the data set to be used for calibration (in a format supported by the data loaders defined together with the PM) and the XML file for saving the calibration results.

In this example, to calibrate the PSP PM, we used a large PSP data set from the Software Engineering Institute (SEI) referring to 31,140 projects concluded by 3,114 engineers during 295

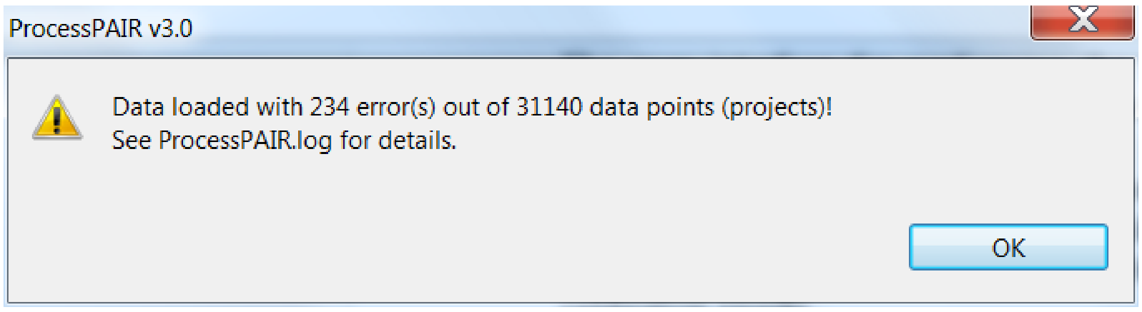


Performance model: PSP Performance Model v2
Source data set: ›ropbox\Mushtaq Raza PhD\Tools\PSP Class Data.mdb
Filtering
Output calibration file: x\Mushtaq Raza PhD\Tools\ProcessPairCalibration.xml

Figure 5.2: Model calibration window.

classes of the classic PSP for Engineers I/II training courses running between 1994 and 2005. In this training course, targeting professional developers, each engineer develops 10 small projects.

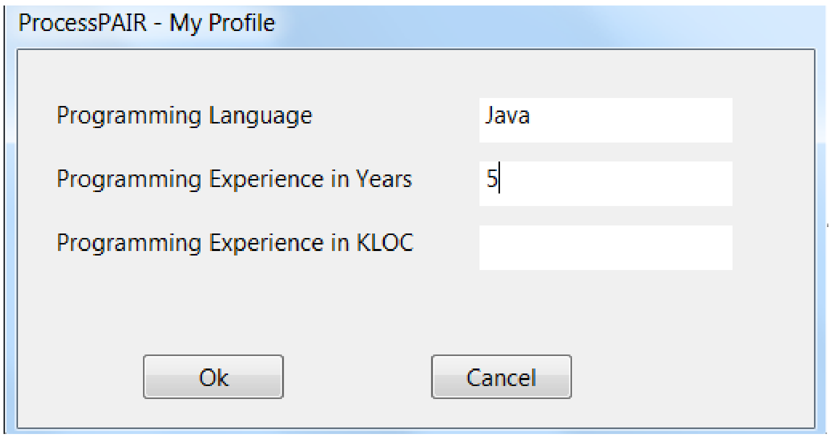
ProcessPAIR performs several data quality checks during the calibration process (according to rules defined together with the PM as explained in section 4.2.1) and presents a summary of problems encountered at the end of the calibration process, as illustrated in Figure 5.3.



Data loaded with 234 error(s) out of 31140 data points (projects)!
See ProcessPAIR.log for details.

Figure 5.3: Summary of calibration results.

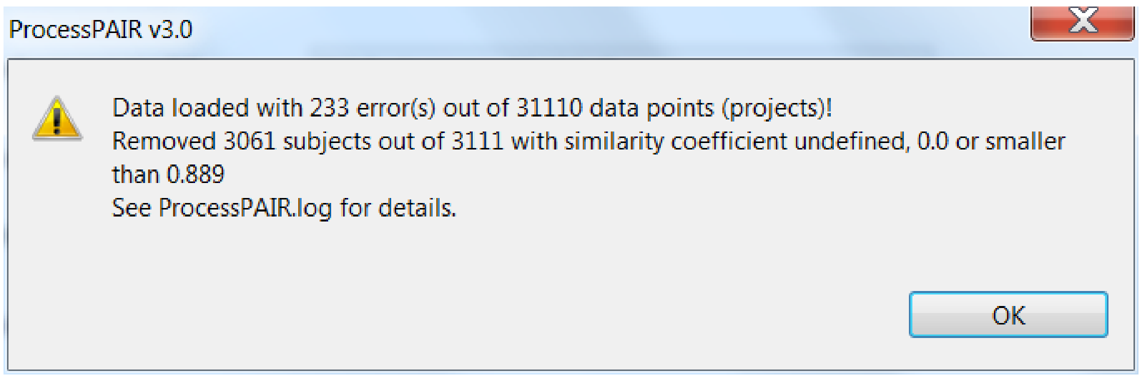
Instead of using the full data set for calibration, it is also possible to filter the data points to be used for calibration. One possibility is to restrict the data points (projects) to the ones most similar to a given user profile, as illustrated in Figure 5.4. The parameters that can be provided depend on the PM and data loader. Similarity is computed with the Gower similarity coefficient (Gower, 1971) as explained in section 3.3.4. As explained in this example (see Figure 5.5), only the 50 most similar data points were selected (minimum number required by the tool for statistical significance), with a similarity coefficient greater than 0.889.



ProcessPAIR - My Profile

| | |
|---|---|
| Programming Language | Java |
| Programming Experience in Years | 5 |
| Programming Experience in KLOC | |

Figure 5.4: Dialog for providing a user profile.



ProcessPAIR v3.0

Data loaded with 233 error(s) out of 31110 data points (projects)!
Removed 3061 subjects out of 3111 with similarity coefficient undefined, 0.0 or smaller than 0.889
See ProcessPAIR.log for details.

Figure 5.5: Calibration results with filtering.

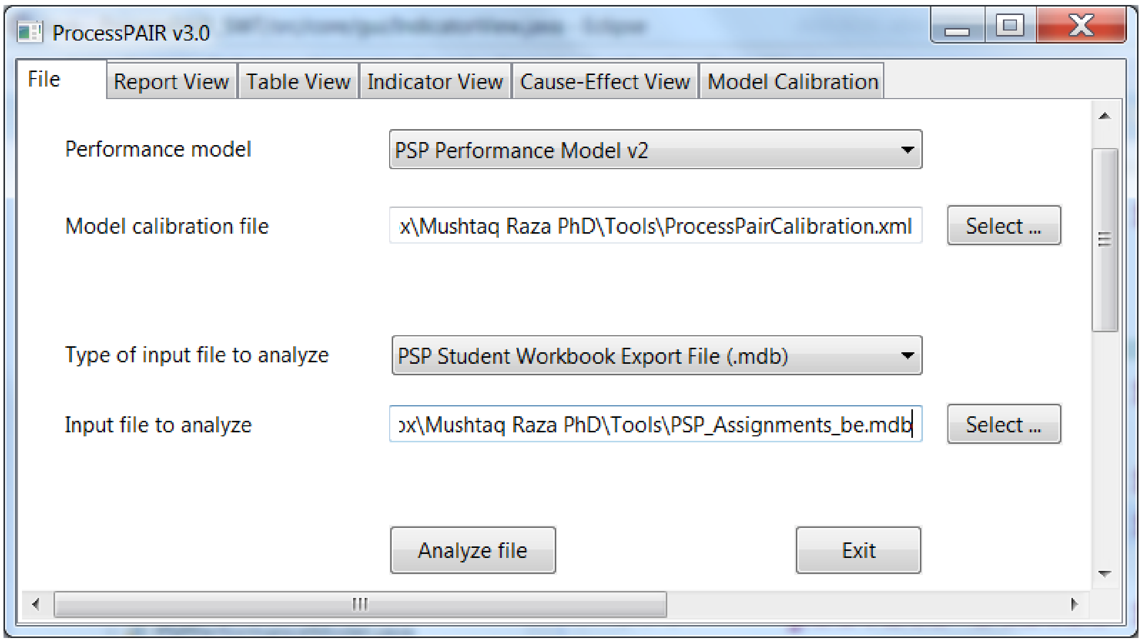
## 5.3 File selection user interface

Having defined and calibrated the performance model (PM), the performance data of individual developers can be automatically analyzed by ProcessPAIR, to identify and rank performance problems and their potential causes. As exemplified in Figure 5.6, the user has to select the performance model (from the list of PMs previously defined as tool extensions), the calibration file (generated as previously explained), the type of input file with performance data to analyze (according to the data loaders defined together with the performance model), and the file with the actual data. By pressing the "Analyze file" button, the analysis is performed and the results are presented in multiple views.

## 5.4 Analysis views

### 5.4.1 Report view

The goal of the Report view (Figure 5.7) is to indicate in a simple way, overall ("Summary") or project by project, the most relevant top-level performance problems (colored red or yellow in the



Performance model: PSP Performance Model v2

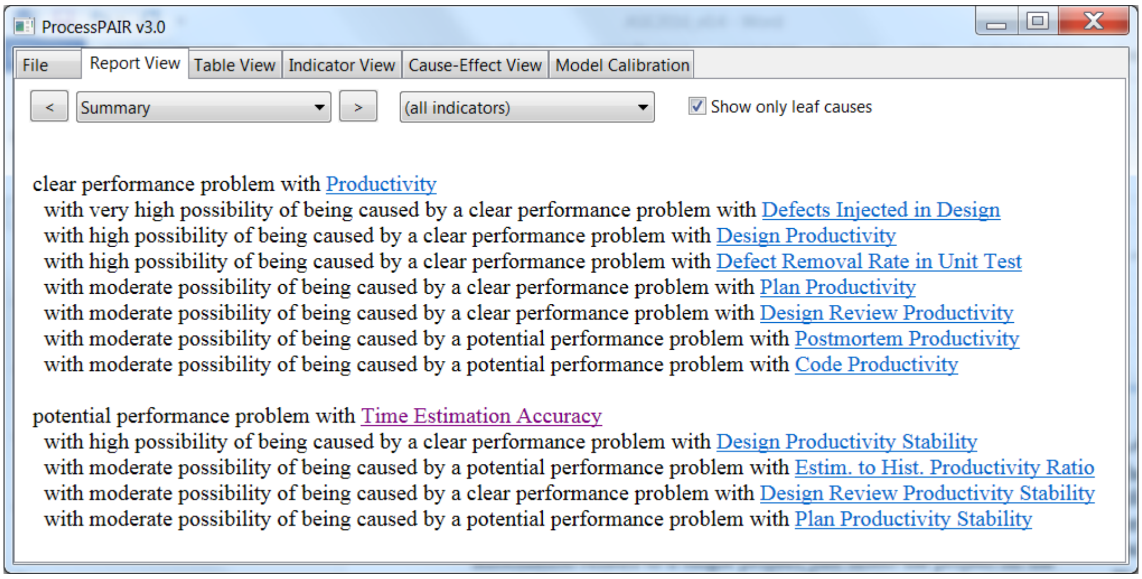
Model calibration file: x\Mushtaq Raza PhD\Tools\ProcessPairCalibration.xml

Type of input file to analyze: PSP Student Workbook Export File (.mdb)

Input file to analyze: ɔx\Mushtaq Raza PhD\Tools\PSP_Assignments_be.mdb

Figure 5.6: Entry window.

Table View) and potential root causes (leaf causes in the Cause-Effect View) properly prioritized (according to the ranking coefficients, previously explained).



clear performance problem with Productivity
  with very high possibility of being caused by a clear performance problem with Defects Injected in Design
  with high possibility of being caused by a clear performance problem with Design Productivity
  with high possibility of being caused by a clear performance problem with Defect Removal Rate in Unit Test
  with moderate possibility of being caused by a clear performance problem with Plan Productivity
  with moderate possibility of being caused by a clear performance problem with Design Review Productivity
  with moderate possibility of being caused by a potential performance problem with Postmortem Productivity
  with moderate possibility of being caused by a potential performance problem with Code Productivity

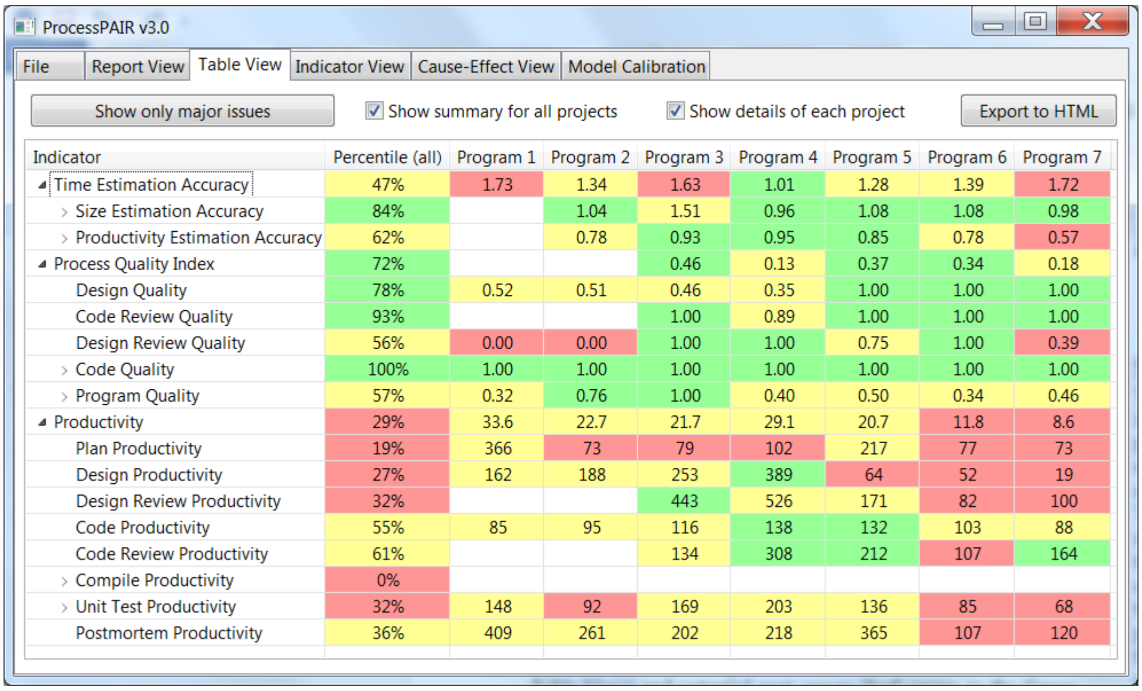
potential performance problem with Time Estimation Accuracy
  with high possibility of being caused by a clear performance problem with Design Productivity Stability
  with moderate possibility of being caused by a potential performance problem with Estim. to Hist. Productivity Ratio
  with moderate possibility of being caused by a clear performance problem with Design Review Productivity Stability
  with moderate possibility of being caused by a potential performance problem with Plan Productivity Stability

Figure 5.7: Report view example.

Intermediate causes can be consulted by unchecking the “Show only leaf causes” checkbox. Comboboxes allow selecting information for specific projects and/or PIs. The links skip to the Indicator View, for detailed information about each PI.

### 5.4.2 Table view

The Table view (Figure 5.8) shows the values of the PIs defined in the model for the projects described in the input file, as well as summarized performance information. Each cell is colored green, yellow or red, in case its value suggests no performance problem, a potential performance problem, or a clear performance problem, respectively. This way, the Table View helps in quickly identifying the performance problems. The exact ranges considered can be consulted in the "Indicator View". The "Percentile (all)" column shows an overall percentile for each PI, computed from the per project values (with higher importance for the last projects), and colored according to the percentile.



☑ Show summary for all projects ☑ Show details of each project

| Indicator | Percentile (all) | Program 1 | Program 2 | Program 3 | Program 4 | Program 5 | Program 6 | Program 7 |
|---|---|---|---|---|---|---|---|---|
| Time Estimation Accuracy | 47% | 1.73 | 1.34 | 1.63 | 1.01 | 1.28 | 1.39 | 1.72 |
| Size Estimation Accuracy | 84% | | 1.04 | 1.51 | 0.96 | 1.08 | 1.08 | 0.98 |
| Productivity Estimation Accuracy | 62% | | 0.78 | 0.93 | 0.95 | 0.85 | 0.78 | 0.57 |
| Process Quality Index | 72% | | | 0.46 | 0.13 | 0.37 | 0.34 | 0.18 |
| Design Quality | 78% | 0.52 | 0.51 | 0.46 | 0.35 | 1.00 | 1.00 | 1.00 |
| Code Review Quality | 93% | | | 1.00 | 0.89 | 1.00 | 1.00 | 1.00 |
| Design Review Quality | 56% | 0.00 | 0.00 | 1.00 | 1.00 | 0.75 | 1.00 | 0.39 |
| Code Quality | 100% | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| Program Quality | 57% | 0.32 | 0.76 | 1.00 | 0.40 | 0.50 | 0.34 | 0.46 |
| Productivity | 29% | 33.6 | 22.7 | 21.7 | 29.1 | 20.7 | 11.8 | 8.6 |
| Plan Productivity | 19% | 366 | 73 | 79 | 102 | 217 | 77 | 73 |
| Design Productivity | 27% | 162 | 188 | 253 | 389 | 64 | 52 | 19 |
| Design Review Productivity | 32% | | | 443 | 526 | 171 | 82 | 100 |
| Code Productivity | 55% | 85 | 95 | 116 | 138 | 132 | 103 | 88 |
| Code Review Productivity | 61% | | | 134 | 308 | 212 | 107 | 164 |
| Compile Productivity | 0% | | | | | | | |
| Unit Test Productivity | 32% | 148 | 92 | 169 | 203 | 136 | 85 | 68 |
| Postmortem Productivity | 36% | 409 | 261 | 202 | 218 | 365 | 107 | 120 |

Figure 5.8: Table view example (partially expanded).

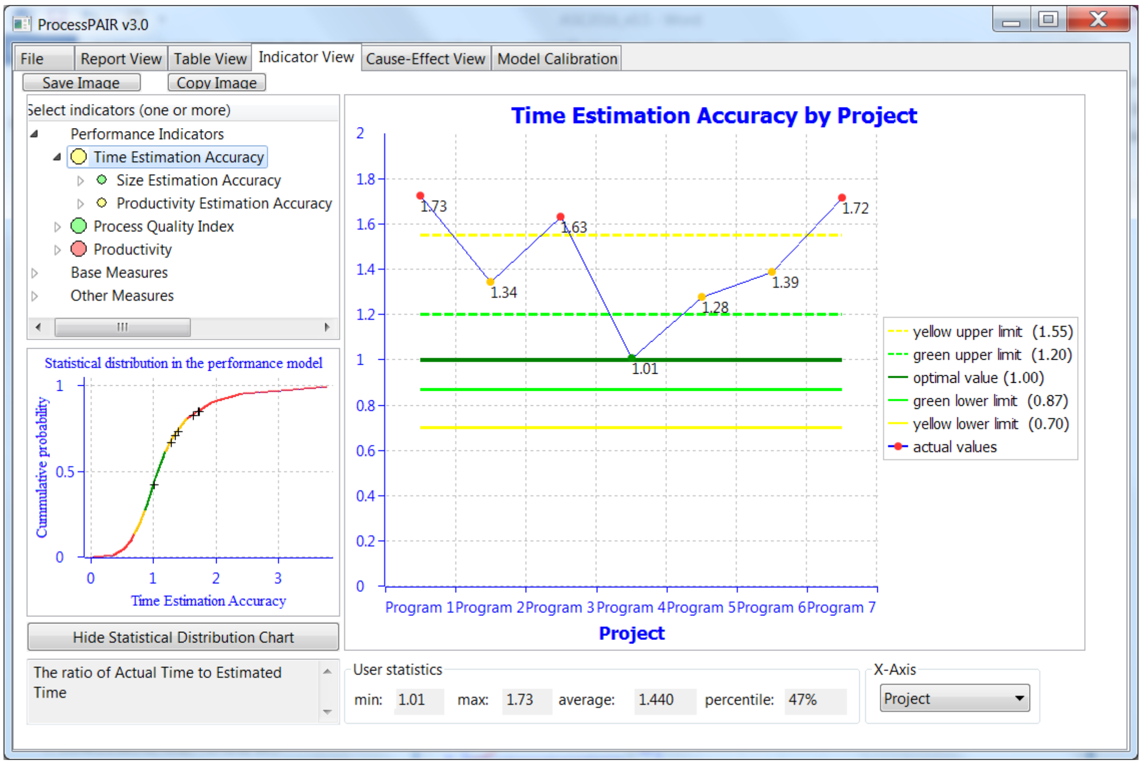
The PIs are organized hierarchically, starting from the top-level indicators (*Time Estimation Accuracy*, *Process Quality Index*, and *Productivity* in this case), and descending to lower level indicators (child indicators) that affect the higher level ones according to a formula or statistical evidence (see chapter 4). This way, by drilling down from the top-level indicators to the lower level ones, focusing on the red (or yellow) colored cells, one can easily identify potential root causes of performance problems.

### 5.4.3 Indicator view

The goal of the Indicator view (Figure 5.9) is to show the behavior of each PI throughout the projects under analysis and provide associated model definition and calibration information (description, units, optimal value, recommended performance ranges and statistical distribution).

In the bottom left, the statistical distribution of the PI in the data set used for calibrating the model is presented. The colors correspond to the performance ranges. The actual values in the file under analysis are also shown, marked with the "+" symbol, for benchmarking purposes.

The user may also select multiple PIs (see Figure 5.10) for comparative visualization in a single chart. More indicator views and features can be consulted in Appendix A.



The ratio of Actual Time to Estimated Time

User statistics

min: 1.01 max: 1.73 average: 1.440 percentile: 47%

Figure 5.9: Indicator view example.

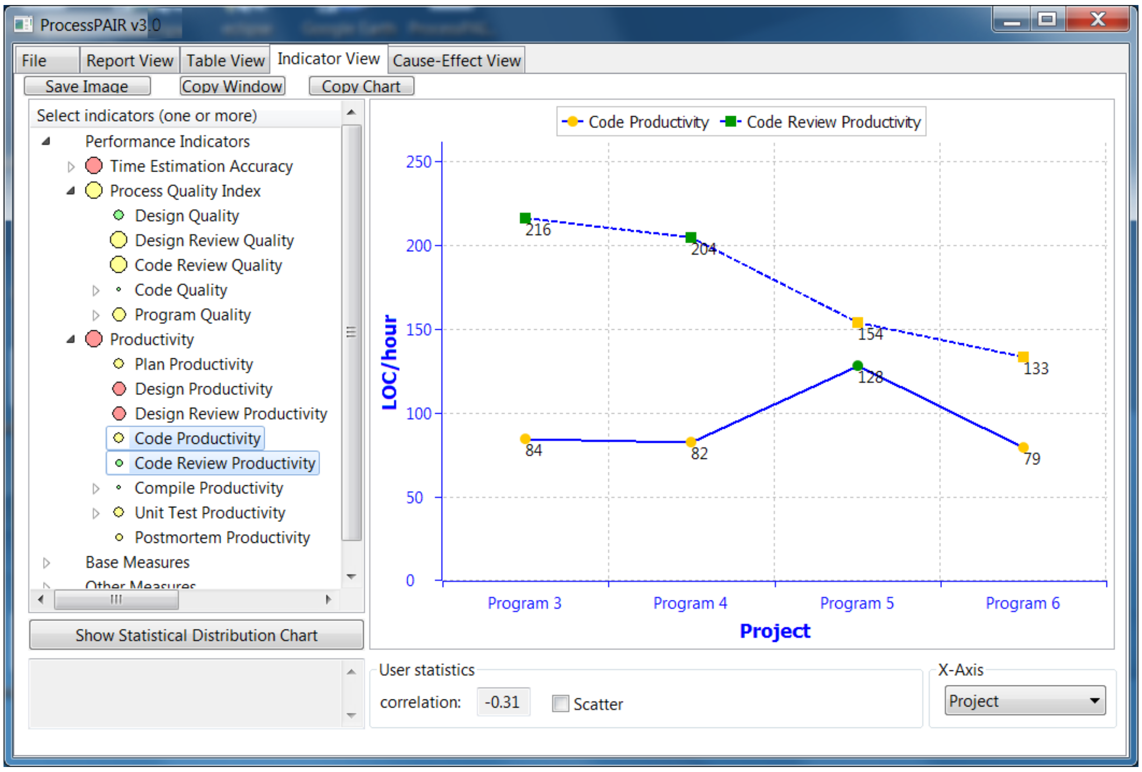
## 5.4.4 Cause-effect view

The Cause-Effect view (Figure 5.11) is an advanced view that provides essentially the same information as the report view with additional details but in a diagrammatic way.

The goal of the Cause-Effect View it to help identify and prioritize, project by project or overall, the root causes of performance problems, so that subsequent improvement actions can be properly directed. The child indicators are sorted according to the value of the ranking coefficient.

As explained previously, the ranking coefficient represents a cost-benefit estimate that relates the cost of improving the value of the child PI with the benefit on the value of the parent PI.

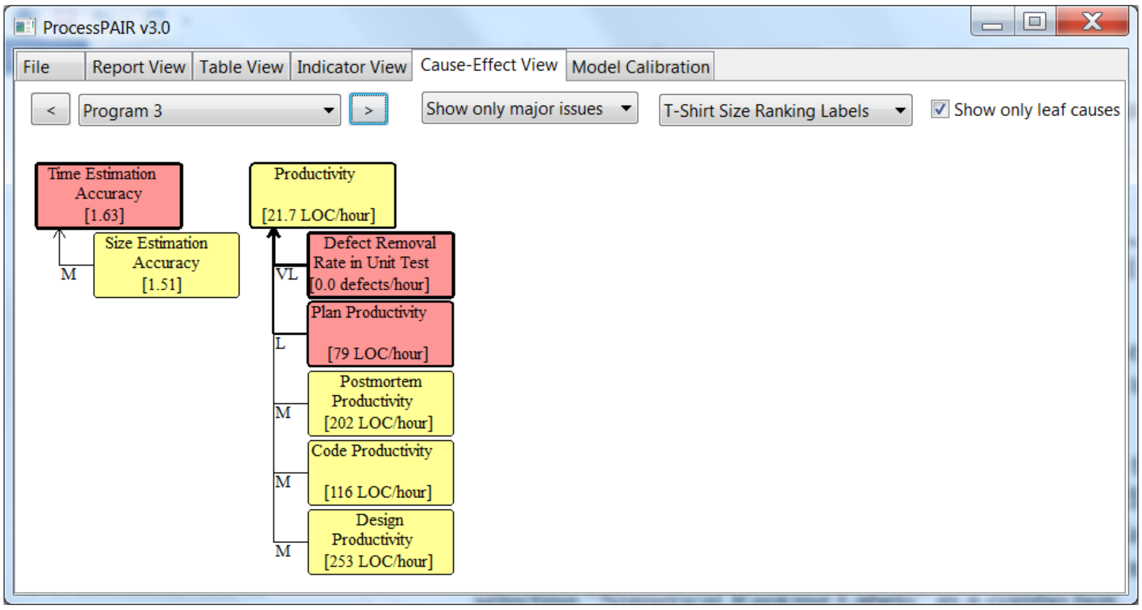
Intermediate causes may be consulted by unselecting the "Show only leaf causes" checkbox (see Figure 5.12). By default, the ranking coefficients are shown by means of T-shirt sizes (see ranking labels section 3.4.1). The numerical values of the ranking coefficients can be consulted by selecting "Numerical Ranking Labels" in a combo box (see Figure 5.13). The selection and sorting rules for problematic performance indicators are already explained in section 3.4.3.3.



correlation: -0.31

Project

Figure 5.10: Indicator view with multiple PIs selection.


ProcessPAIR v3.0
File
Report View
Table View
Indicator View
Cause-Effect View
Model Calibration
<
>
Time Estimation Accuracy [1.63]
M
Size Estimation Accuracy [1.51]
Productivity [21.7 LOC/hour]
VL
Defect Removal Rate in Unit Test [0.0 defects/hour]
L
Plan Productivity [79 LOC/hour]
M
Postmortem Productivity [202 LOC/hour]
M
Code Productivity [116 LOC/hour]
M
Design Productivity [253 LOC/hour]

Program 3 | Show only major issues | T-Shirt Size Ranking Labels | ✓ Show only leaf causes

Figure 5.11: Cause-effect view showing only leaf causes.

## 5.5 Tool extension API

The following steps are involved in defining a performance model as a tool extension, using the provided API (as illustrated in Figure 5.14 for the PSP performance model):

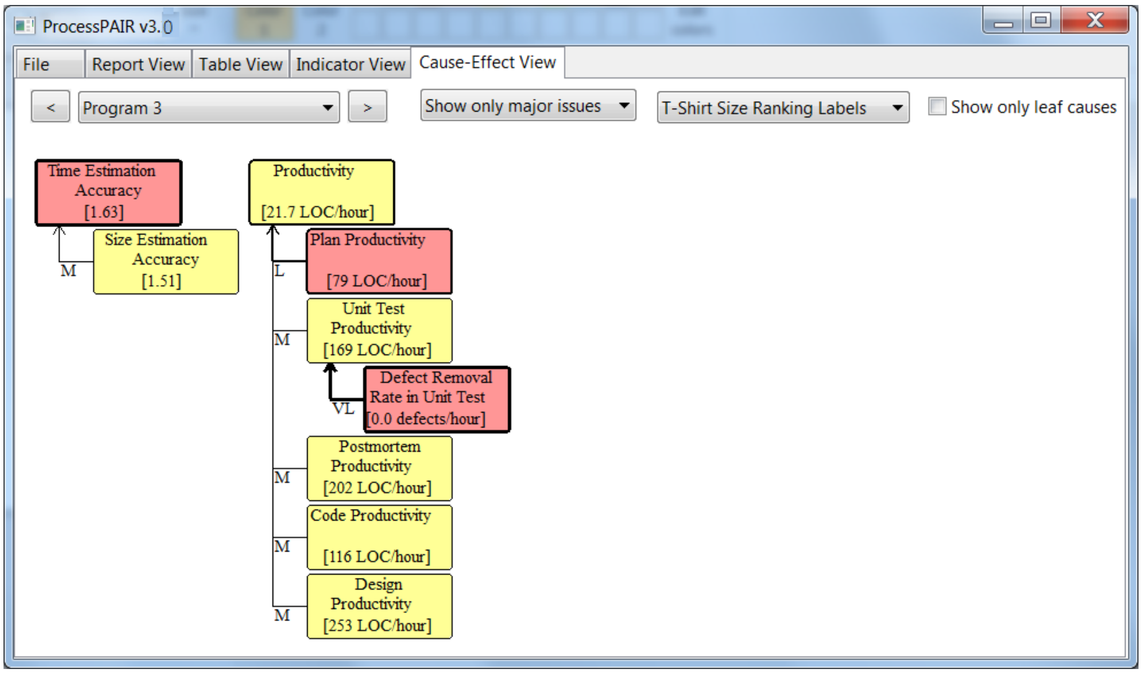

ProcessPAIR v3.0
File
Report View
Table View
Indicator View
Cause-Effect View
<
Program 3
>
Show only major issues
T-Shirt Size Ranking Labels
Show only leaf causes
Time Estimation Accuracy [1.63]
M
Size Estimation Accuracy [1.51]
Productivity [21.7 LOC/hour]
L
Plan Productivity [79 LOC/hour]
M
Unit Test Productivity [169 LOC/hour]
VL
Defect Removal Rate in Unit Test [0.0 defects/hour]
M
Postmortem Productivity [202 LOC/hour]
M
Code Productivity [116 LOC/hour]
M
Design Productivity [253 LOC/hour]

Figure 5.12: Cause-effect view showing intermediate causes.

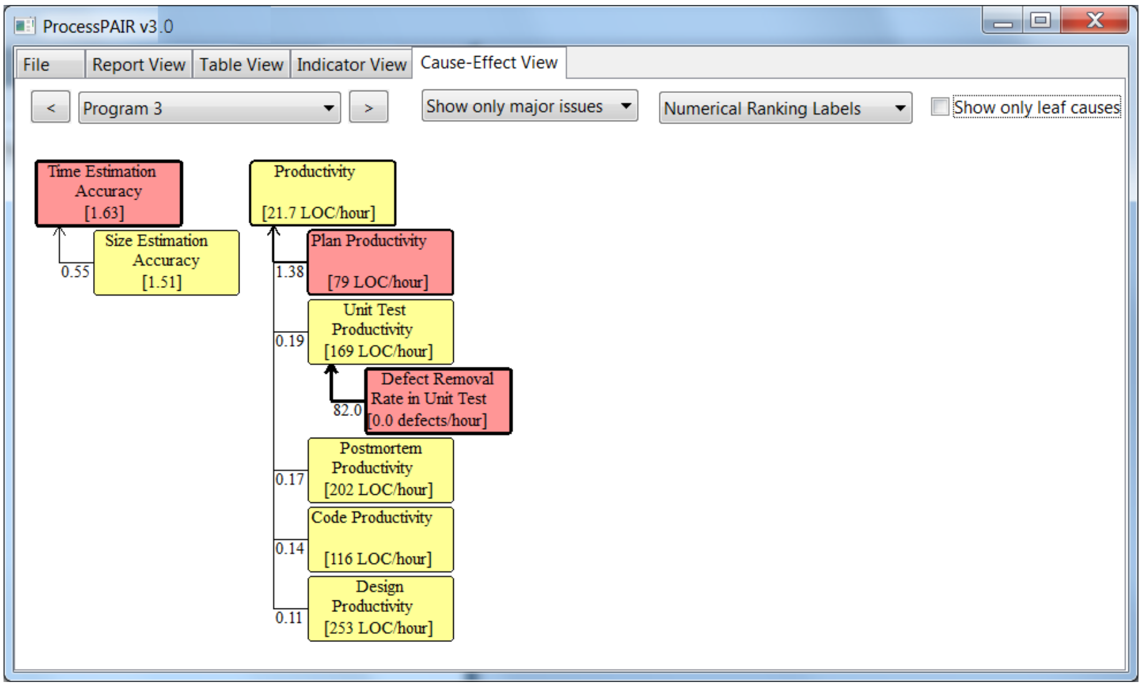

ProcessPAIR v3.0
File
Report View
Table View
Indicator View
Cause-Effect View
<
Program 3
>
Show only major issues
Numerical Ranking Labels
Show only leaf causes
Time Estimation Accuracy [1.63]
0.55
Size Estimation Accuracy [1.51]
Productivity [21.7 LOC/hour]
1.38
Plan Productivity [79 LOC/hour]
0.19
Unit Test Productivity [169 LOC/hour]
82.0
Defect Removal Rate in Unit Test [0.0 defects/hour]
0.17
Postmortem Productivity [202 LOC/hour]
0.14
Code Productivity [116 LOC/hour]
0.11
Design Productivity [253 LOC/hour]

Figure 5.13: Cause-effect view showing numerical ranking labels.

- *Define Base Measures*: Here we define the base measures (see code in Figure 5.14) for the process of our interest which can be used in the same way for any process under consideration. In this study the process of our interest is PSP because of several reason, e.g., well defined process and availability of data form SEI as explained in detail previously.

- *Define Performance Indicators*: Here we define the Performance Indicators from the base measures defined in the first step. The example in Figure 5.14 shows the code for productivity indicator defined in our model. Basically productivity in PSP is measured as size (LOC) per unit time (Hours) and therefore, the base measures (*Actual Size and Actual Time*) are defined in the first step. The same way all the base measures for defining Performance Indicators are defined in our model under the *defineBaseMeasures* method.

- *Add Top-level Indicators*: Here we add the top-level (or root) indicators in our model (see Figure 5.14). The code to add the top-level indicators (e.g., Time Estimation Accuracy, Productivity and Process Quality Index) in our PSP model is shown in the example. The same code can be used for adding more top level indicators in our current model or any other model for a new process under consideration.

- *Load Dependencies*: Here we define the dependencies based on a formula or statistical relation between the parent and child indicators. The example in Figure 5.14 shows the code for dependency relationship (based on formula) for Time Estimation Accuracy to Size and Productivity Estimation Accuracy, and dependency relationship (based on statistical relation) for Defect Density in Unit Test to Defect Injection and Process Yield. In the first case, the last parameter is the formula for the sensitivity coefficient. The same method (*loadDependencies*) can used to code more dependencies in the current model or any other model of a new process under consideration.

```
public class PSPPerformanceModel extends PerformanceModel {
      …
   @Override
   public boolean checkConsistency(Project p, PrintStream log) {
      // data quality checks
      …
   }

   @Override
   protected void defineBaseMeasures() {
      …
      addBaseMeasure("ActualSize", // short name
         "Actual Size", // long name
         "Actual number of added and modified lines of code", //description
         "LOC", // units
         0.0, // minimum
         Double.MAX_VALUE,  // maximum
         0); // decimal digits
      …
   }

   @Override
   protected void definePerformanceIndicators() {
      …
      addIndicator("Prod", "Productivity",
         "The ratio of Actual Size to Actual Time",
         "LOC/hour", 0.0, Double.MAX_VALUE, 1,
         Double.MAX_VALUE, // optimal value
         new Ratio("ActualSize", "Actual Time", 60.0)); // formula
      …
   }

   @Override
   public void addTopLevelIndicators() {
      addToplevelIndicator(getIndicator("TimeEA"));
      addToplevelIndicator(getIndicator("PQI"));
      addToplevelIndicator(getIndicator("Prod"));
   }

   @Override
   protected void loadDependencies() {
      …
      // Decomposition of TimeEA per factors (based on formula)
      addDependency("TimeEA", "SizeEA", new Constant(1.0));
      addDependency("TimeEA", "PEA",  new Constant(-1.0));
      …
      // Decomposition of DDUT per factors (based on statistical relation)
      addDependency("DDUT", "DI", RegressionType.MultipleStepwiseRegression);
      addDependency("DDUT", "PY", RegressionType.MultipleStepwiseRegression);
      …

   }
   …

}
```

Figure 5.14: Excerpt of PSP model code.

# Chapter 6

# Experimentation and validation

In this chapter we describe two experiments for evaluation of the ProcessPAIR method and tool. In the first experiment (see section 6.1) we assess the accuracy of problem and their root causes identification by comparing the results of manual analysis (as documented by the PSP students in their PSP Final Report) with automatic analysis (as produced by ProcessPAIR). The second experiment (see section 6.2) is a controlled experiment involving 61 software engineering master students, half of whom used ProcessPAIR in a PSP performance analysis assignment, to assess the benefits of ProcessPAIR in terms of efficiency, quality, and user satisfaction.

## 6.1 Postmortem experiment

In this section we present an experiment conducted in the context of PSP training, to show that the ProcessPAIR tool is able to accurately identify performance problems and their potential root causes of individual developers.

### 6.1.1 Research questions

The overall objective of the experiment is to assess whether the ProcessPAIR tool is able to accurately identify performance problems of individual developers and their potential causes, so that subsequent manual analysis for the identification of deeper causes and remedial actions can be properly focused and effort can be saved.

More specifically, the goal of the experiment is to answer the following research questions:

- *RQ1* (*problem identification*): Is it possible to automatically analyze the performance data of an individual PSP developer in order to identify performance problems, with similar results but with less effort than in manual analysis?
- *RQ2* (*causal analysis*): Is it possible to automatically analyze the performance data of an individual PSP developer in order to determine the causes of the identified performance problems, in a way consistent with manual analysis (except when manual analysis is erroneous)?

Regarding *RQ1* (problem identification), we consider that the results of automatic analysis (performed by the tool) and manual analysis (conducted by the developer) are *similar* when:

- the developer explicitly indicates *bad performance* in a PI, for a specific project or overall (summary), and the tool also points out a clear (red) or potential (yellow) performance problem in the same PI and context (specific project or summary); or
- the developer explicitly indicates *good performance* in a PI, for a specific project or overall (summary), and the tool also indicates good performance (green) in the same PI and context (specific project or summary).

We consider that the results are *dissimilar* when:

- the developer explicitly indicates bad performance in a PI in a specific project or overall (summary), but the tool indicates good performance (green) (*false negative*); or
- the developer explicitly indicates good performance in a PI in a specific project or overall (summary), but the tool indicates a clear (red) or potential (yellow) problem (*false positive*).

We consider that the results are *not comparable* whenever the developer does not explicitly mention good or bad performance. In many cases this happens with PIs that are not analyzed at all by the developer. In other cases, the PI is analyzed but the developer just points out some extreme cases or examples, without the concern for completeness.

Regarding *RQ2* (causal analysis), we restrict the comparison to performance problems in the top-level PIs considered in the manual analysis, which are identified simultaneously by the developer and the tool, for a specific project or overall (summary). For comparison purposes it is useful to think of the results of causal analysis as a tree, starting from the problematic PI and drilling down to increasingly deeper causes. We consider that the results are *consistent* when:

- there is a *match*, i.e., manual and automatic analysis point out *similar* causes, or no causes are identified in both cases; or
- there is a *different level of detail*, i.e., the tool accurately points out intermediate causes (or points out no causes at all) and the manual analysis points out deeper causes (*deeper manual analysis*), or viceversa (*deeper automatic analysis*).

We consider that the results are *inconsistent* when:

- the developer and tool point out different causes, not interrelated, because of a *tool fault* (missed important causes or pointed out extraneous causes); or
- the developer and tool point out different causes, not interrelated, because of a *developer fault* (missed important causes or pointed out extraneous causes).

### 6.1.2 Input data

To calibrate the model we used the PSP data set already mentioned in 4.2.1 from the Software Engineering Institute (SEI) referring to 31,140 projects concluded by 3,114 engineers during 295 classes of the classic PSP for Engineers I/II training courses running between 1994 and 2005. In this training course, targeting professional developers, each engineer develops 10 small projects.

The subject data under analysis is based on a data set from Instituto Tecnológico de Monterrey, in Mexico, referring to 27 subjects that developed 6 projects each using the PSP, in the scope of the "Software Quality and Testing" course in 2015. The subjects used Process Dashboard (`http://www.processdash.com/`) for collecting the standard PSP base measures during their projects. In the end of the sequence of projects, the subjects analyzed their personal performance along those projects, and documented their findings and improvement proposals in a Final Report (written in Spanish).

Because of the significant effort (on average 8h-10h per subject) required to translate, understand and extract relevant information from those reports, we restricted the case study to a random sample of 20 subjects.

For guiding the analysis and the production of their final reports, the subjects were requested by the instructor to consider 30 questions in 4 categories: analysis of size estimation, analysis of time estimation, defect and yield analysis, and quality analysis (see Table 2.3). However, some questions are not actually related with the identification of performance problems or root causes (see examples in Table 6.1), so we considered them out of the scope of this experiment.

### 6.1.3 Data analysis procedures

The author and supervisor, not involved in the PSP training in Instituto Tecnológico de Monterrey, both fluent in English and one with a good reading understanding of Spanish, translated to English and analyzed the final reports (in both English and Spanish), in order to extract relevant information for comparison with the tool-based analysis, according to the structure illustrated in the next section.

Results from tool-based analysis for each subject were effortlessly obtained by uploading the performance data stored in Process Dashboard to ProcessPAIR.

The extracted results from the final reports and from the tool for each subject were then collected into appropriate tables, as illustrated in the next section.

Subsequently, the results were classified according to the categories described in section 6.1.1, and statistics where computed.

### 6.1.4 Results

#### 6.1.4.1 Results per subject

Regarding *RQ1* (problem identification), based on the information available in the final report of each subject, we produced a table with a synthesis of cases in which the subject explicitly indicated

Table 6.1: Examples of questions that the subjects were requested to address and relevance for this case study.

| **Question** | **Relevance to this study** |
|---|---|
| What are the average, maximum, and minimum actual sizes of my programs in LOC to date? | Out of scope (size by itself is not actually a performance indicator). |
| How much are my time estimates affected by the accuracy of my size estimates? | Within scope (causal analysis). |
| How much did the quality of the programs entering unit test change? Why? | Within scope (performance problem identification). |
| Based on my historical data, what are some realistic quality goals for me? | Out of scope (improvement goals). |
| How can I change my process to meet those goals? | Out of scope (improvement actions). |

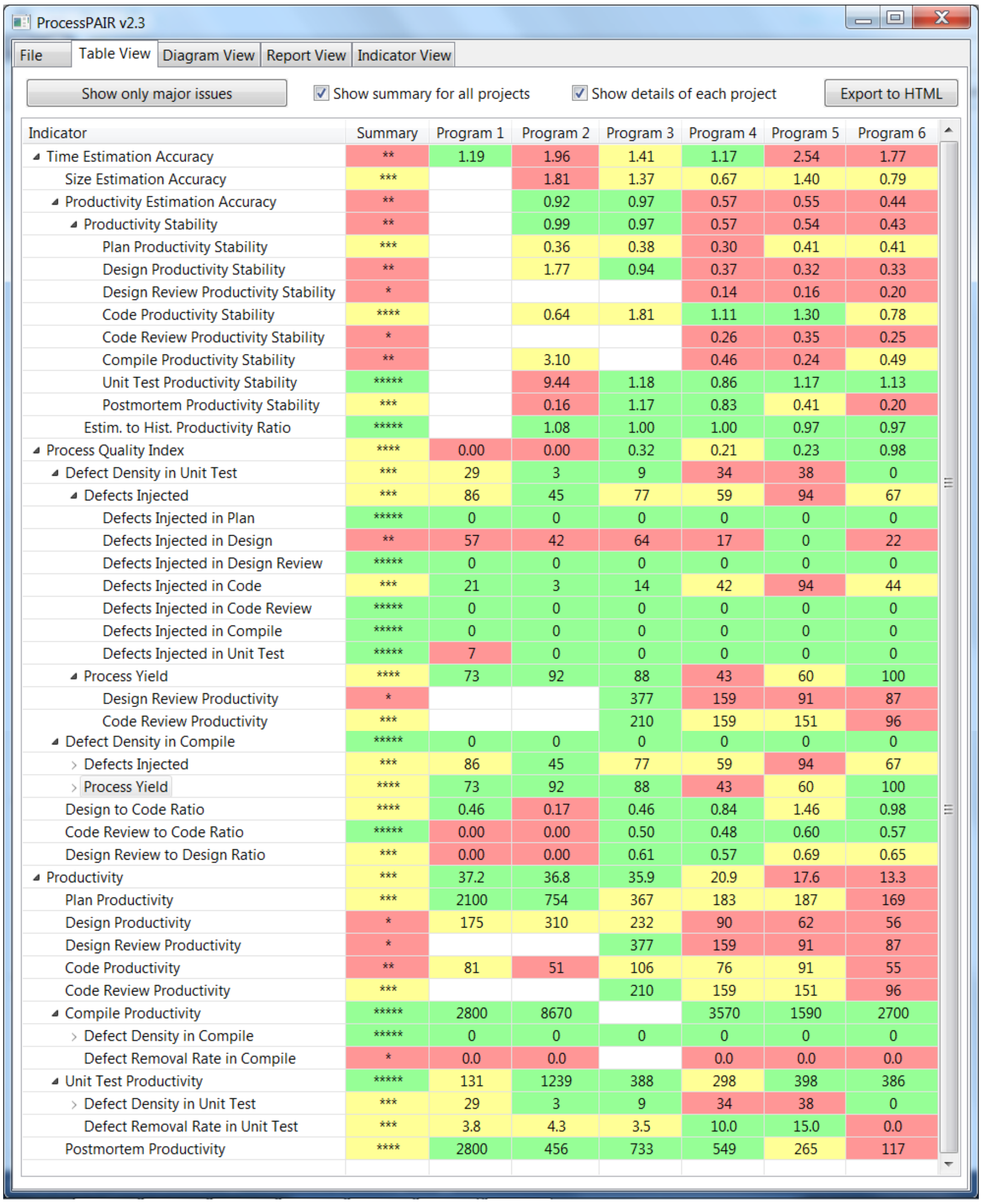
bad performance or good performance in a PI for a specific project or overall (summary). An example is shown in Table 6.2, together with the support citations extracted from the final report.

1. "With regard to program 5, the testing stage highlighted by having a high rate of removal of defects."
2. "Syntax errors were injected in the coding phase (...) errors of type function were injected in design and coding phases."
3. "Most of the defects were injected in the design phase"
4. "The programs with the lowest yield were programs 4 and 5, with values of 42.90 and 60 respectively (...). In the last program I could get a yield of 100%." In the PSP, the process yield is the percentage of defects found and fixed (usually through design and code reviews) before compile and test.
5. "The program in which I had the higher productivity was the first, with 37.2 (...) programs 5 and 6 were where more code was reused as well as also where more design and design review conducted, which explains the low productivity."

Table 6.2: Cases of bad (red-R) or good (green-G) performance explicitly indicated by one of the subjects in the case study.

| **Indicator** | **Summary** | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** |
|---|---|---|---|---|---|---|---|
| Defect Removal Rate in Unit Test (defects/hour) (1) | | | | | | G | |
| Defects Injected (defects/KLOC) (2) | R | | | | | | |
| Defects Injected in Design (3) | R | | | | | | |
| Process Yield (4) | | | | | R | R | G |
| Productivity (LOC/hour) (5) | | G | | | | R | R |
| Size Estimation Accuracy (6) | | | R | | | | G |
| Time Estimation Accuracy (7) | | G | R | | G | R | |

6. "The largest error of estimation I had was in program 2 with 80.6 error %, which is too high. (. . . ) In contrast, program 6 was where my estimate was closer to the actual result."
7. "There are two that stand out for being very large, program 2 and program 5 (. . . ) On the other hand, the closest estimates were in program 1 and 4."



| Indicator | Summary | Program 1 | Program 2 | Program 3 | Program 4 | Program 5 | Program 6 |
|---|---|---|---|---|---|---|---|
| ◢ Time Estimation Accuracy | ** | 1.19 | 1.96 | 1.41 | 1.17 | 2.54 | 1.77 |
| Size Estimation Accuracy | *** |  | 1.81 | 1.37 | 0.67 | 1.40 | 0.79 |
| ◢ Productivity Estimation Accuracy | ** |  | 0.92 | 0.97 | 0.57 | 0.55 | 0.44 |
| ◢ Productivity Stability | ** |  | 0.99 | 0.97 | 0.57 | 0.54 | 0.43 |
| Plan Productivity Stability | *** |  | 0.36 | 0.38 | 0.30 | 0.41 | 0.41 |
| Design Productivity Stability | ** |  | 1.77 | 0.94 | 0.37 | 0.32 | 0.33 |
| Design Review Productivity Stability | * |  |  |  | 0.14 | 0.16 | 0.20 |
| Code Productivity Stability | **** |  | 0.64 | 1.81 | 1.11 | 1.30 | 0.78 |
| Code Review Productivity Stability | * |  |  |  | 0.26 | 0.35 | 0.25 |
| Compile Productivity Stability | ** |  | 3.10 |  | 0.46 | 0.24 | 0.49 |
| Unit Test Productivity Stability | ***** |  | 9.44 | 1.18 | 0.86 | 1.17 | 1.13 |
| Postmortem Productivity Stability | *** |  | 0.16 | 1.17 | 0.83 | 0.41 | 0.20 |
| Estim. to Hist. Productivity Ratio | ***** |  | 1.08 | 1.00 | 1.00 | 0.97 | 0.97 |
| ◢ Process Quality Index | **** | 0.00 | 0.00 | 0.32 | 0.21 | 0.23 | 0.98 |
| ◢ Defect Density in Unit Test | *** | 29 | 3 | 9 | 34 | 38 | 0 |
| ◢ Defects Injected | *** | 86 | 45 | 77 | 59 | 94 | 67 |
| Defects Injected in Plan | ***** | 0 | 0 | 0 | 0 | 0 | 0 |
| Defects Injected in Design | ** | 57 | 42 | 64 | 17 | 0 | 22 |
| Defects Injected in Design Review | ***** | 0 | 0 | 0 | 0 | 0 | 0 |
| Defects Injected in Code | *** | 21 | 3 | 14 | 42 | 94 | 44 |
| Defects Injected in Code Review | ***** | 0 | 0 | 0 | 0 | 0 | 0 |
| Defects Injected in Compile | ***** | 0 | 0 | 0 | 0 | 0 | 0 |
| Defects Injected in Unit Test | ***** | 7 | 0 | 0 | 0 | 0 | 0 |
| ◢ Process Yield | **** | 73 | 92 | 88 | 43 | 60 | 100 |
| Design Review Productivity | * |  |  | 377 | 159 | 91 | 87 |
| Code Review Productivity | *** |  |  | 210 | 159 | 151 | 96 |
| ◢ Defect Density in Compile | ***** | 0 | 0 | 0 | 0 | 0 | 0 |
| › Defects Injected | *** | 86 | 45 | 77 | 59 | 94 | 67 |
| › Process Yield | **** | 73 | 92 | 88 | 43 | 60 | 100 |
| Design to Code Ratio | **** | 0.46 | 0.17 | 0.46 | 0.84 | 1.46 | 0.98 |
| Code Review to Code Ratio | ***** | 0.00 | 0.00 | 0.50 | 0.48 | 0.60 | 0.57 |
| Design Review to Design Ratio | *** | 0.00 | 0.00 | 0.61 | 0.57 | 0.69 | 0.65 |
| ◢ Productivity | *** | 37.2 | 36.8 | 35.9 | 20.9 | 17.6 | 13.3 |
| Plan Productivity | *** | 2100 | 754 | 367 | 183 | 187 | 169 |
| Design Productivity | * | 175 | 310 | 232 | 90 | 62 | 56 |
| Design Review Productivity | * |  |  | 377 | 159 | 91 | 87 |
| Code Productivity | ** | 81 | 51 | 106 | 76 | 91 | 55 |
| Code Review Productivity | *** |  |  | 210 | 159 | 151 | 96 |
| ◢ Compile Productivity | ***** | 2800 | 8670 |  | 3570 | 1590 | 2700 |
| › Defect Density in Compile | ***** | 0 | 0 | 0 | 0 | 0 | 0 |
| Defect Removal Rate in Compile | * | 0.0 | 0.0 |  | 0.0 | 0.0 | 0.0 |
| ◢ Unit Test Productivity | ***** | 131 | 1239 | 388 | 298 | 398 | 386 |
| › Defect Density in Unit Test | *** | 29 | 3 | 9 | 34 | 38 | 0 |
| Defect Removal Rate in Unit Test | *** | 3.8 | 4.3 | 3.5 | 10.0 | 15.0 | 0.0 |
| Postmortem Productivity | **** | 2800 | 456 | 733 | 549 | 265 | 117 |

Figure 6.1: Example of automatic evaluation of top-level and nested PIs.

From the comparison of the cells in the manual (Table 6.2) and automatic analysis (Figure

6.1), we computed the statistics for this subject and classified the cells in Table 6.3 according to the categories indicated in section 6.1.1. In this case, regarding the comparable results, we have 14 similar results and 1 false positive. The false positive has to do with a boundary situation: the size estimation accuracy in project 6 (21% error), which the tool evaluates as yellow and the subject evaluates in a positive way (see actual citation in Table 6.2).

Regarding *RQ2* (causal analysis), based on the information available in the final report of each subject and the tool results, we produced a table per subject as illustrated in Table 6.6. Each row refers to a performance problem in a top-level PI (according to the scope of the manual analysis), identified simultaneously by the developer and the tool, on a specific project, group of projects or overall (in summary). We summarize the causes indicated by the subject and by the tool, and classify their consistency according to the categories indicated in section 6.1.1. We also show the support citations extracted from the final report. The causes identified by the tool are ranked.

### 6.1.5 Overall statistics

Regarding *RQ1*, by aggregating the results of individual subjects, we obtained the overall statistics indicated in Table 6.4.

Regarding problem identification, from the 302 cases in which students explicitly characterized their performance (regarding a specific PI and a specific project or all projects), we compared the student assessment with the tool-based assessment, and got the following results:

- In 96% of the cases, the results of manual and automatic analysis matched (i.e., both the student and the tool indicated good performance or bad performance);
- In 3% of the cases, the tool indicated a clear or potential problem and the manual analysis indicated good performance (false positives);
- In 1% of the cases, the tool indicated no performance problem but the developer explicitly indicated a performance problem (false negatives).

Regarding *RQ2*, by aggregating the results of individual subjects, we obtained the overall statistics given below (also see Table 6.5):

- In 21% of the cases, the tool and the developer pointed out the same causes (or none pointed out any causes).

Table 6.3: Problem identification statistics for one of the selected subjects in the case study (1 subject x 6 projects).

| | | **Automatic Analysis** | | |
|---|---|---|---|---|
| | | **Green** | **Yellow** | **Red** |
| **Manual Analysis** | **Bad** | *0 (false negative)* | *2 (bad performance)* | *7 (bad performance)* |
| | **Good** | *5 (good performance)* | *1 (false positive)* | *0 (false positive)* |

- In 53% of the cases, the tool and the developer pointed out causes at different levels of detail, with the tool accurately pointing out intermediate causes, and the developer pointing out deeper causes. These are cases where the automatic analysis accurately points out factors to focus in subsequent manual analysis, potentially reducing the overall manual effort.

- In 26% of the cases the results are inconsistent, because of faults in the manual analysis. Such cases might be prevented if our tool was used by the subjects when conducting their performance analysis.

### 6.1.6 Discussion

#### 6.1.6.1 Answers to the research questions

Regarding *RQ1* – "Is it possible to automatically analyze the performance data of an individual PSP developer in order to identify performance problems, with similar results but with less effort than in manual analysis", Table 6.4 shows that in 292 (96%) out of the 302 cases in which the developers explicitly indicated good or bad performance, the results of manual and automatic analysis are similar, in the sense indicated in section 6.1.1. There are only eight false positives, corresponding to boundary situations, in which the tool indicated a potential performance problem (yellow) but the developer considered to exist a good performance. There are two false negatives, in which the developer considered to exist a performance problem but the tool indicated good performance (green); however, those cases refers to subjects that considered abnormally tight thresholds, so they could also be classified as an erroneous evaluation by the subject. Hence, regarding RQ1, we conclude that in this experiment the automatic analysis produces similar results (without essentially any manual effort), with a very small number of false positives (3%).

Regarding *RQ2* – "Is it possible to automatically analyze the performance data of an individual PSP developer in order to determine the causes of the identified performance problems, in a way consistent with manual analysis (except when manual analysis is erroneous)?", the results in Table 6.5 show that, in the cases in which the manual analysis was not erroneous (it was erroneous in 26% of the cases!), the tool-based analysis was able to point out the same causes as the ones found by the developers in their manual analysis (21% of the cases) or was able to point out intermediate causes in the same direction as the deeper causes identified in manual analysis (53%) of the cases.

Table 6.4: Problem identification statistics for all selected subjects in the case study (20 subjects x 6 projects).

| | | **Automatic Analysis** | | |
|---|---|---|---|---|
| | | **Green** | **Yellow** | **Red** |
| **Manual Analysis** | **Bad** | *2 (1% false negative)* | *82 (bad performance)* | *114 (bad performance)* |
| | **Good** | *96 (good performance)* | *8 (3% false positive)* | *0 (false positive)* |

Hence, regarding RQ2, we conclude that the automatic analysis was able to identify either the same causes or causes in the same direction as the manual analysis.

Overall, the benefits of the tool-based analysis are:

- it can correctly identify the performance problems, saving manual effort;
- it can correctly identify causes for the identified performance problems, so that subsequent manual analysis for searching deeper causes can be properly focused, reducing the overall manual effort needed and the errors in manual analysis.

### 6.1.7 Limitations and threats to validity

In the experiment presented, using real world data, the conclusions obtained by the model-based analysis are very close to the ones obtained by the developers in their manual analysis. This suggests that our approach can be helpful in performance analysis and process improvement, by pointing out the areas to focus in manual analysis. However, further experiments need to be conducted to quantify the effort savings that can be achieved by conducting performance analysis with the help of our tool from the beginning (that will be the goal of the second experiment).

Our choice of PIs was constrained by the data available, so some relevant factors (mentioned by the developers in their manual analysis) were not captured by those PIs, such as parts' size estimation accuracy, level of experience, and program complexity. However, in case data about those factors is available, our approach can be easily extended to take them into account.

Although our approach and tool are general and can be instantiated for any process, the model and experiment described in the thesis refer only to PSP performance data. We intend to replicate our approach to other processes without having such a well-defined measurement framework as the PSP, but we expect to encounter difficulties regarding data availability, data quality, and standardization.

Table 6.5: Frequency of categories and subcategories for RQ2.

| **Categories and subcategories** | **Absolute frequency** | **Relative frequency** |
|---|---|---|
| Consistent – Match (same causes) | 11 | 21% |
| Consistent - Different level of detail - Deeper manual analysis (tool accurately points out intermediate causes) | 28 | 53% |
| Inconsistent - Developer fault (wrong or missing factors) | 14 | 26% |
| **Total** | **53** | **100%** |

Table 6.6: Comparison of causes of identified performance problems for one subject in the case study.

| **Top-level Indicator** | **Project** | **Causes in manual analysis** | **Causes in automatic analysis** | **Classification** |
|---|---|---|---|---|
| Size Estimation Accuracy | P2 | No relevant cause[1] | None[2] | Consistent – Match (no causes) |
| Time Estimation Accuracy | P2 | “I attribute the bad (time) estimation to the (bad) size estimation” | Size Estimation Accuracy (red) | Consistent – Match (similar causes) |
| Time Estimation Accuracy | P5 | Historical productivity didn’t apply because of more reused code [3] | Design Productivity Stability (red), Design Review Productivity Stability (red) Plan Productivity Stability (yellow), Size Estimation Accuracy (yellow) | Inconsistent - Developer fault (wrong or missing factors)[4] |
| Productivity | P5 | More reused code and more time in Design and Design Review. [5] | Design Productivity (red), Design Review Productivity (red), Plan Productivity (yellow), Code Productivity (yellow) | Consistent - Match. Inconsistent - Developer fault (wrong or missing factors).[4] |
| Productivity | P6 | More code reused. More time in Design and Design Review.[5] | Design Productivity, Design Review Productivity, Code Productivity, Postmortem Productivity, Code Review Productivity and Plan Productivity (red) | Consistent - Different level of detail – Deeper manual analysis (tool accurately points out intermediate causes) |
| Defects Injected | Summary | Most defects injected in Design (mainly of type Function), followed by Code[6]. | Defects Injected in Design (red), Defects Injected in Code (yellow) | Consistent - Different level of detail – Deeper manual analysis (tool accurately points out intermediate causes) |
| Process Yield | P4 | “Bad because I did not perform well (the code and) code review” | Code Review Yield (red) | Consistent – Match (similar causes) |
| Process Yield | P5 | “Bad because I did not perform well (the code and) code review” | Code Review Yield (green) | Consistent – Match (similar causes) |

[1] In this case, the subject mentioned “when I made this estimate I was not aware of how many lines of code I used to program”. However, he did not analyze the two possible immediate causes - problems in the identification of needed parts, or problems in the size estimation of each part – which would point out to relevant improvement actions.

[2] In the training data set, we do not have available the data produced by PSP developers for size estimation and measurement needed for causal analysis (parts and their estimated and actual sizes). If that data was available, causal analysis could be automated.

[3] “As to the error in the program 5, I attribute it to that when I was making this estimate I used method C (time estimate based on historical productivity and size estimate), when the best option was the method D (expert judgment). This is because, unlike in previous programs, program 5 reused a lot of code.”

[4] Significant process changes occur in project 4 (introduction of design templates in the Design phase) and 5 (introduction of execution table analysis in the Design Review phase), which slow down productivity in those phases on all subjects analyzed. By contrast, we did not find a significant correlation between the percentage of reused code and the productivity. In projects P1 to P3, productivity (measured in added and modified LOC/hour) is stable (between 36 or 37 added and modified LOC/hour), with a small % reuse (between 0% and 16%); then there is a steep decrease of productivity in project 4 (where design templates are introduced in the Design phase), with 0% reused code, and productivity of 21 LOC/hour.

[5] “Programs 5 and 6 were where I did reuse more code, and also conducted more design and design review, which explains the low productivity (measured in added and modified LOC/hour).”

[6] “Most of the defects were injected in the design stage, these being mainly errors of type function”.

## 6.2 Controlled experiment

In this section we present the results of a controlled experiment involving 61 software engineering master students, of Instituto Tecnológico de Monterrey, in Mexico, half of whom used ProcessPAIR tool in a PSP performance analysis assignment and other used Process Dashboard tool in the 2016 edition of the Software Quality and Testing course.

Our goal is to assess the benefits of using ProcessPAIR for personal performance analysis, as compared to traditional tools, in terms of user satisfaction, quality of analysis outcomes, and time required to do the analysis.

### 6.2.1 Context

The "Software Quality and Testing" course includes a streamlined version of the official SEI's PSP Fundamental and PSP Advanced training courses. In the 2016 edition, each student has to develop 7 small projects, following increasingly elaborated processes, which incorporate in a proper order relevant software engineering best practices (project estimation, planning and tracking, unit tests, reviews, design practices, etc.).

Several base measures are collected to support some of those practices and also to assess personal performance status and progression. During the course, students used Process Dashboard (Dashboard), a comprehensive PSP support tool for data collection (time, defects, size), project planning and tracking, and data analysis (charts and reports that aid in the analysis of historical data trends) described in section 2.2.5

At the end of the course, the students have to perform the official "PSP Final Report" assignment. In this assignment, students are asked to analyze their personal performance data collected throughout the 7 projects they developed and document their findings and improvement proposals in a report.

To guide their analysis, students are asked to address several questions organized in the following categories (see also the full set of questions presented in Table 2.3):

- Analysis of size estimating accuracy (7 questions);
- Analysis of time estimating accuracy (10 questions);
- Defect and yield analysis (9 questions);
- Quality analysis (4 questions).

### 6.2.2 ProcessPAIR tuning

For better supporting the production of the PSP Final Report with the help of ProcessPAIR, we tuned ProcessPAIR as follows:

- we configured the performance model named “PSP Performance Model for PSP Final Report”, that extends our base performance model for the PSP (see chapter 4), with additional top-level PIs;
- we prepared a web based tutorial (see Appendix A) named “PSP Final Report Guidelines”, explaining how to take advantage of ProcessPAIR to answer the required questions.

### 6.2.3 Experiment design

In this experiment, we aim to prove the following main hypothesis:

*Using ProcessPAIR, PSP students can analyze their performance data and produce their “PSP Final Report” with (i) higher user satisfaction, (ii) higher quality of results, and (iii) less effort, as compared to the traditional approaches, using only Process Dashboard.*

More specifically, the research questions that this experiment aims to answer are:

- *RQ1*: Are students that use ProcessPAIR for performing the “PSP Final Report” assignment more satisfied with the tool support than students that perform that assignment in a traditional way using Process Dashboard?
- *RQ2*: Do students that use ProcessPAIR for performing the “PSP Final Report” assignment produce higher quality reports (as measured by the grades given by the instructor) than students that perform that assignment in a traditional way using Process Dashboard?
- *RQ3*: Do students that use ProcessPAIR for performing the “PSP Final Report” assignment spend less time than students that perform that assignment in a traditional way using Process Dashboard?

To complete the PSP Final Report assignment, the 61 students were randomly split into two groups: a control group, with 31 students, using Process Dashboard, and an experimental group with 30 students using ProcessPAIR. The students in the control group completed their Final Report assignment in a traditional way by inspecting their performance data (charts, reports, etc.) stored in the Process Dashboard tool. The students in the experimental group used ProcessPAIR for analyzing their performance data to complete Final Report assignment.

To measure user satisfaction, we prepared a web-based survey to be answered by the students of both groups after concluding the assignment. The survey contains open text questions plus 14 questions in a five-point Likert scale related to installability, usability, efficiency, usefulness and level of support provided by the tool they used for conducting the performance analysis (see Table 6.7).

Since the students did not have any prior knowledge about ProcessPAIR, before starting the experiment, the tool authors presented a short tutorial about ProcessPAIR (30 minutes) to the selected students by video-conference.

In the next step, the students were assigned a time slot to install ProcessPAIR and make sure it run properly on their performance data. During the installation and initial setup, all the students

Table 6.7: User satisfaction survey questions.

| | |
|---|---|
| A | How easy was it to install the tool on your machine? (please select 1 if you were unable to successfully install) |
| B | How do you rate the tool usability? |
| C | How do you rate the tool efficiency? |
| D | How do you rate the overall usefulness of the tool for performance analysis purposes? |
| E | How do you rate the overall level of support provided by the tool for performance analysis purposes? |
| | How do you rate the level of support provided by the tool for each of the following tasks in the PSP Final Report? |
| F | - Analysis of size estimating accuracy |
| G | - Analysis of time estimating accuracy |
| H | - Defect and yield analysis |
| I | - Quality analysis |
| | How do you rate the level of support provided by the tool for answering the following types of questions in the PSP Final Report? |
| J | - Values and trends of performance indicators |
| K | - Average, maximum and minimum values |
| L | - Relationships and comparisons between indicators |
| M | - Realistic performance goals |
| N | - Process changes to meet the goals |

were accompanied locally by their instructor, with distance assistance by the ProcessPAIR authors. After successful installation, they performed the assignment autonomously, outside classes (they had approximately two weeks, in part time, to conclude the assignment and submit the final report).

### 6.2.4 Results

For each student of both groups, the following data was collected:

- PSP Final Report submitted by the student;
- Process Dashboard file submitted by the student (with performance data from previous assignments and time logs of the final report assignment);
- Grading spreadsheet with the grades assigned by the instructor for the final report, average for the last two programs and final course grade;
- Spreadsheet with results of surveys filled in by the students (as provided by Google forms).

From this data, the following metrics were extracted or computed for each student:

- The time spent in the assignment (extracted from the Process Dashboard time logs);
- Report grade, course grade and last 2 program grades (extracted from the grading spreadsheet);

- Report size, in pages;
- User satisfaction scores (for all questions).

The results obtained regarding user satisfaction, time spent, report size and report grades are summarized in Figure 6.2, Figure 6.3, Figure 6.4, Figure 6.5 and Table 6.8. User comments about ProcessPAIR are listed in Table 6.10.

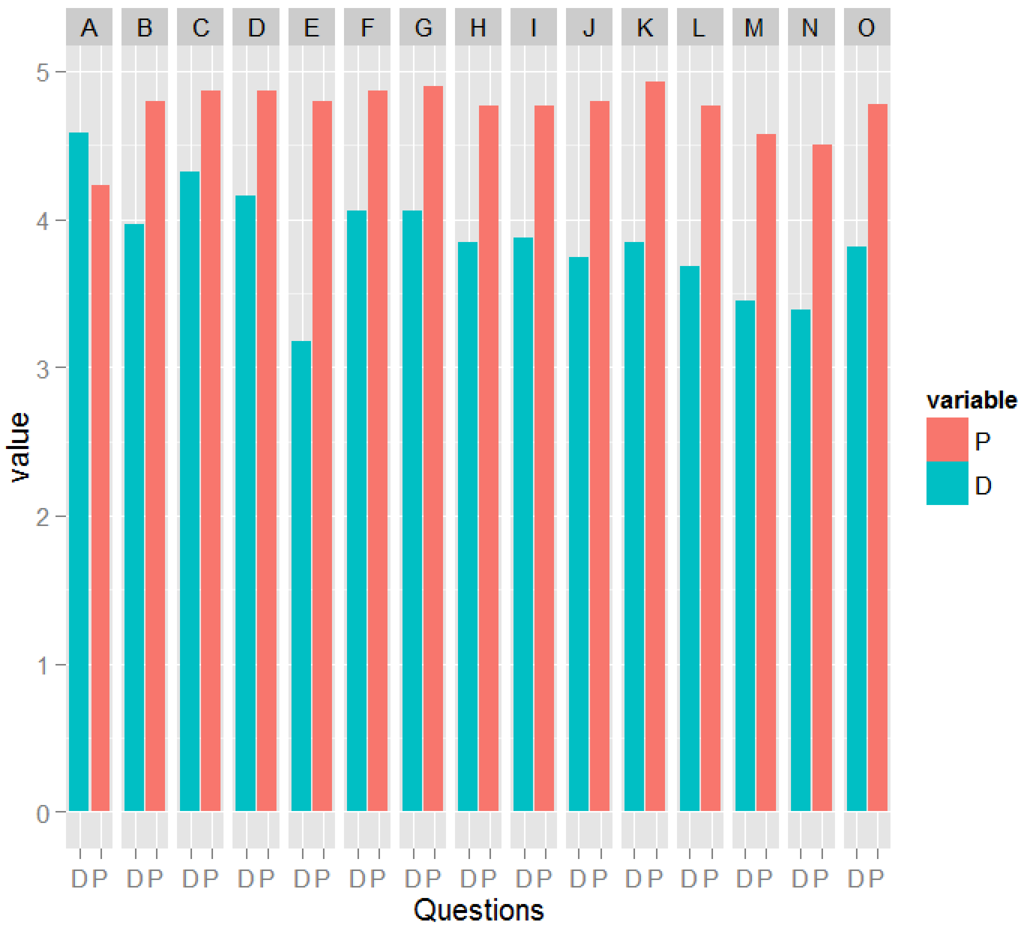


Figure 6.2: Average scores of the user satisfaction survey questions given by ProcessPAIR (P) and Process Dashboard (D) users (Note: column 'O' shows average for all questions).

### 6.2.5 Discussion

The statistics in Table 6.8 show that, on average, ProcessPAIR users were more satisfied with the tool support, produced higher quality reports and spent less time than non-ProcessPAIR users. Comments made by ProcessPAIR users (see Table 6.10) also show a very favorable assessment. To determine whether the differences observed are statistically significant, we performed the t-test for the difference of means (see Table 6.9). Since the variables under analysis did not pass

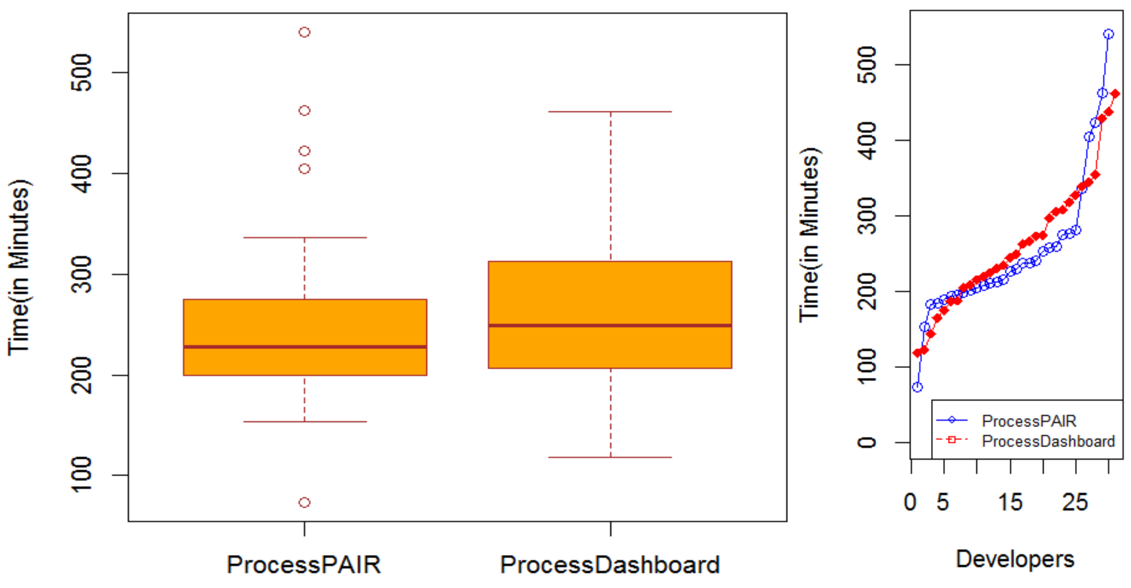


Figure 6.3: Distribution and box plot of time spent (in minutes) by developers in both groups in the PSP Final Report assignment.

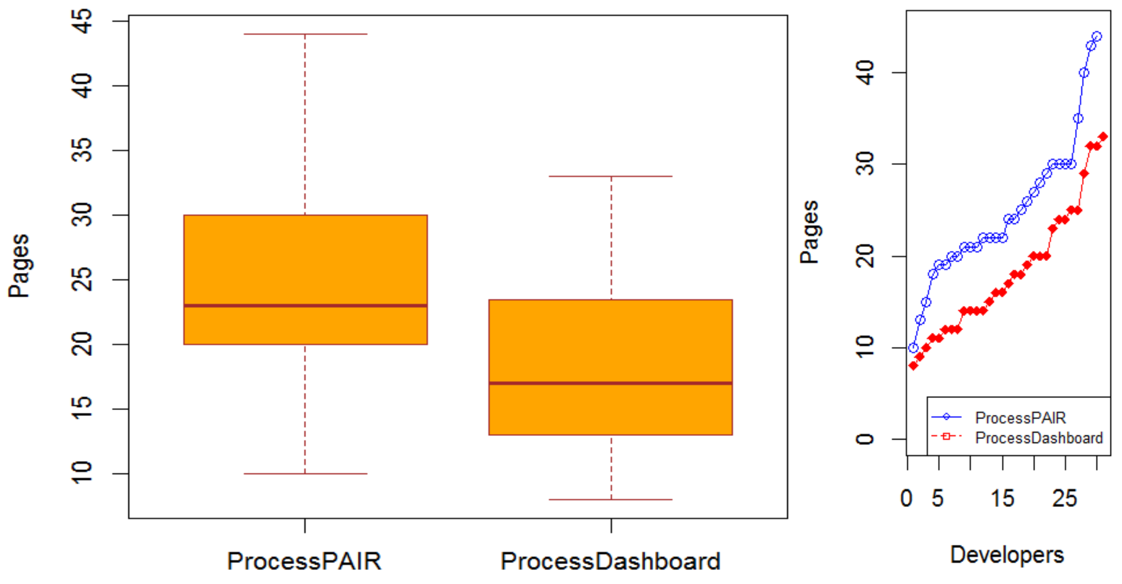


Figure 6.4: Distribution and box plot of report size (in pages) by developers in both groups in the PSP Final Report assignment.

the Shapiro-Wilk normality test for one or both samples (groups), we also performed the Mann-Whitney test for the difference of medians (see Table 6.9). Regarding user satisfaction (RQ1), we conclude that the differences for all questions except installability are statistically significant for a 5% confidence level.

Regarding quality of results (RQ2), the differences in the report grades achieved are statistically significant according to the t-test for the difference of means, but not according the Mann-Whitney test for the difference of medians. By inspecting the distribution of the report grades in

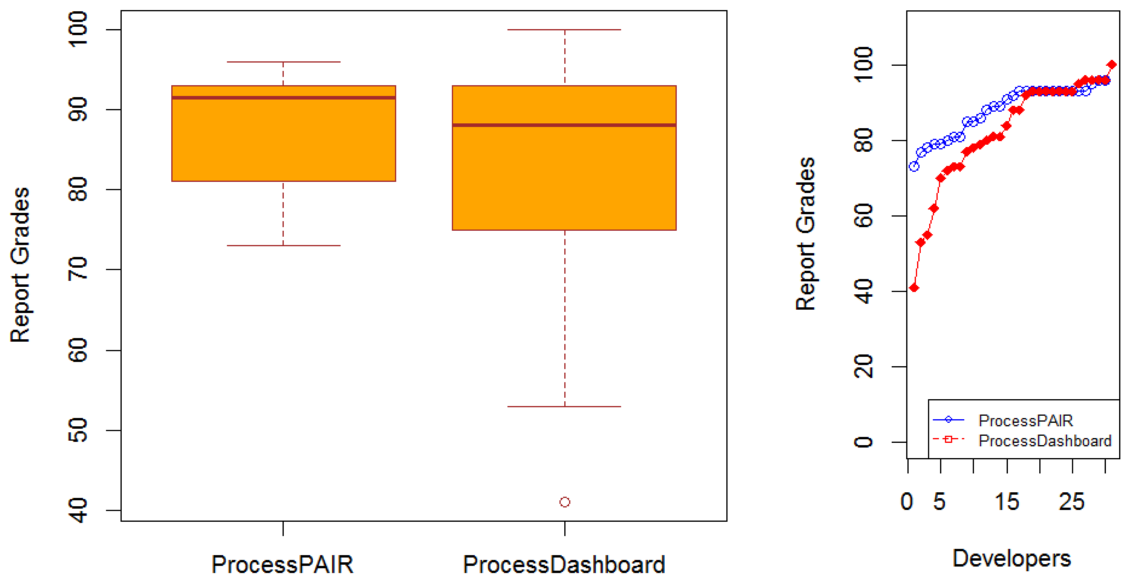


Figure 6.5: Distribution and box plot of grades achieved by developers in both groups in the PSP Final Report assignment.

Figure 6.5, it can be observed that there is a clear difference between the two groups to the left of the median, but not to the right, which is consistent with the tests' results. This suggests that the students who get the lower grades are the ones that benefit most of using ProcessPAIR. To make sure that the difference between the two groups was not caused by an unbalanced split of students between groups, we also checked the grades achieved in recent projects (last programs developed, before the final assignment) by the students in both groups, and found no statistically significant differences (in fact, the students in the experimental group had slightly lower grades in the last two projects).

Regarding the time spent (RQ3), ProcessPAIR users spent on average less time than non-ProcessPAIR users, but the differences observed between both groups are not statistically significant according to both tests. Although ProcessPAIR already has automatic report generation

Table 6.8: Some statistics in the control experiment.

| **Statistic** | **Experimental group (ProcessPAIR)** | **Control group (Process Dashboard)** |
|---|---|---|
| Number of students | 30 | 31 |
| Mean score for all students and questions in user satisfaction survey | 4.78 (in a 1-5 scale) | 3.81 (in a 1-5 scale) |
| Mean report grade | 88.1 (in a 0-100 scale) | 82.5 (in a 0-100 scale) |
| Mean time spent | 252.2 minutes | 262.2 minutes |
| Mean report size | 25.0 pages | 18.2 pages |
| Median time spent | 228.5 | 249.0 |
| Median report size | 23 pages | 16.5 pages |
| Median report grade | 91.5 | 88.0 |

Table 6.9: Results of hypothesis tests.

| **Variables** | **Difference of means statistically significant? (t-test, unequal variances)** | **Difference of medians statistically significant? (Mann-Whitney U Test)** |
|---|---|---|
| Time spent (minutes) | No (p=0.337) | No (p=0.209) |
| Report size (pages) | **Yes** (p=0.00049) | **Yes** (p=0.00058) |
| Report grade | **Yes** (p=0.0293) | No (p=0.173) |
| Survey: Installability (A) | No (p=0.0712) | No (p=0.0630) |
| Survey: Usability, Efficiency, Usefullness, Level of Suport (B,...,N) | **Yes** (p<1E-4 for all questions) | **Yes** (p<1E-3 for all questions) |

features (dispensing the users from the tedious copy-paste of dozens of charts) that was intentionally disabled in the experiment, to have a closer comparison with Process Dashboard. By enabling that feature in the future, we expect to significantly reduce the time needed for performing the assignment.

On the other hand, we observed that students in the experimental group produced significantly longer reports than the students in the control group (see Figure 6.4, Table 6.8 and Table 6.9), which, together with the better grades, suggests that they produced more complete reports in roughly the same time as the students in the control group.

### 6.2.6 Threats to validity

We identified the following threats to validity.

*Different previous knowledge:* The students in the experimental group did not have previous knowledge about the tool for the experiment (ProcessPAIR), whilst the students in the control group had significant prior knowledge of the tool for the experiment (Process Dashboad), because they used it throughout 7 projects. This may have negatively affected the performance of ProcessPAIR users in the experiment, as compared with Process Dashboard users.

*Different guidelines available:* Both ProcessPAIR and Process Dashboard have extensive support documentation, but only ProcessPAIR has specific guidelines for producing the “PSP Final Report”. This may have positively affected the performance of ProcessPAIR users in the experiment, as compared with Process Dashboard users. In fact, the usefulness of the online guide was highlighted by students in the comments given in the user satisfaction survey.

Table 6.10: Comments provided by ProcessPAIR users to the question "Please elaborate your opinion about the overall usefulness and level of support for Performance Analysis Purposes".

| |
|---|
| -"ProcessPAIR really does elaborate all the information needed to make the analysis. With the help of the online guide it was easy to know what was happening in each graph and in each set of data." |
| - "The amount of detail in the graphs generated by the platform was awesome." |
| - "It's really helpful because it already gives you all the data analysis that you need for the PSP report." |
| - "Having graphical views for all calculations was very helpful, also the capability to view multiple graphs at the same time." |
| - "It's really easy to visualize your data using the tool, and having data from other projects gives you a reference level for you to compare your own work with so that's great too." |
| - "It was very useful and helpful for the development of the final report." |
| - "The tool is useful to check all your statistics in a quickly way, still need a way to explain a little more the graphs." |
| - "They both worked better than I expected and helped me to make the process of the final report easier." |
| - "I liked ProcessPAIR a lot because it was easier to use than Dashboard, and because it showed the graph for each question, giving me the opportunity to find the answers easier." |
| - "The tool was very useful, it provided a lot of features, a lot of graphs that were really easy to obtain, copy, modify and interact with them, it also had a tab that was like an excel and it marked you in green the areas that you were good at the PSP, and marked you the areas of opportunity in yellow and red, this was useful, because I was able to see where I need to do better." |
| - "It is very useful as it can identify as I was and where I need to improve." (translated from Spanish) |
| - "It pretty much did all the charts and it was really awesome that other student data was used as reference to visually understand where I am. And having the recommendations (the report view tab) was honestly the best assistance given that fact that it was a second option that supported the analysis that we were doing." |
| - "ProcessPAIR was really useful at displaying my dashboard metrics and their average values." |
| - "Analysis was pretty straight forward and easy." |
| - "The performance of the tool is excellent the tutorials help a lot and is easy to understand and elaborate your analysis." |
| - "I could answer all questions without problems." (translated from Spanish) |
| - "I think everything worked perfectly, however the option to take a screenshot should create a screenshot only of the graph, not the entire screen." |
| - "It was completely relevant, giving me the metrics I needed." |
| - "The application served very well to see all my historical data and to prepare the final report." (translated from Spanish) |
| - "It is really useful ProcessPAIR because it gives directly what are you problems and the possible causes. Besides, you can verify if that's true by going to the indicator that shows if you have a problem or not." |
| - "The graphics provided by the tool are to useful for the analysis." |
| - "Many links within the app did not work at all and it did not give useful feedback." |
| - "It really made it easier to understand all the information in the graphs." |
| - "The usefulness is great with this tool because it guides you through the report and also you can analyze more data and statistics of your performance with all the statistics it includes when it analyses your data." |
| - "The tool was very useful to understand my overall performance." |
| - "It helped analyze every answer in a specific way thanks to the help of the graphs that it provided. It was really easy to compare different statistics and understand why things happened." |

Table 6.11: Data regarding total time spent in producing final report, final report size in pages and grades of individual users from both ProcessPAIR and Process Dashboard users.

| ProcessPAIR | | | | Process Dashboard | | | |
|---|---|---|---|---|---|---|---|
| Student# | Time spent (in minutes) | Report size ( in pages) | Report grade (0-100) | Student# | Time spent (in minutes) | Report size (in pages) | Report grade (0-100) |
| 1 | 204 | 22 | 77.0 | 1 | 249 | 18 | 73.0 |
| 2 | 541 | 44 | 89.0 | 2 | 175 | 20 | 78.0 |
| 3 | 463 | 30 | 93.0 | 3 | 273 | 32 | 81.0 |
| 4 | 240 | 21 | 93.0 | 4 | 241 | 32 | 62.0 |
| 5 | 405 | 35 | 91.0 | 5 | 429 | 33 | 41.0 |
| 6 | 227 | 22 | 89.0 | 6 | 118 | 14 | 93.0 |
| 7 | 184 | 21 | 96.0 | 7 | 354 | 25 | 79.0 |
| 8 | 336 | 40 | 88.0 | 8 | 274 | 15 | 84.0 |
| 9 | 183 | 29 | 81.0 | 9 | 220 | 8 | 100.0 |
| 10 | 193 | 22 | 85.0 | 10 | 305 | 20 | 95.0 |
| 11 | 281 | 30 | 86.0 | 11 | 297 | 14 | 70.0 |
| 12 | 212 | 30 | 85.0 | 12 | 230 | 12 | 80.0 |
| 13 | 276 | 21 | 79.0 | 13 | 188 | 9 | 88.0 |
| 14 | 208 | 25 | 79.0 | 14 | 266 | 17 | 77.0 |
| 15 | 275 | 43 | 93.0 | 15 | 225 | 14 | 96.0 |
| 16 | 73 | 24 | 92.0 | 16 | 144 | 10 | 53.0 |
| 17 | 211 | 26 | 78.0 | 17 | 344 | 16 | 72.0 |
| 18 | 260 | 20 | 93.0 | 18 | 263 | 19 | 93.0 |
| 19 | 189 | 10 | 73.0 | 19 | 318 | 29 | 93.0 |
| 20 | 196 | 19 | 81.0 | 20 | 308 | 20 | 93.0 |
| 21 | 199 | 28 | 80.0 | 21 | 235 | 16 | 96.0 |
| 22 | 230 | 15 | 93.0 | 22 | 165 | 12 | 73.0 |
| 23 | 423 | 22 | 93.0 | 23 | 205 | 25 | 96.0 |
| 24 | 202 | 20 | 93.0 | 24 | 438 | 18 | 93.0 |
| 25 | 153 | 24 | 93.0 | 25 | 187 | 14 | 55.0 |
| 26 | 216 | 19 | 93.0 | 26 | 215 | 11 | 81.0 |
| 27 | 253 | 13 | 95.0 | 27 | 327 | 24 | 88.0 |
| 28 | 237 | 27 | 93.0 | 28 | 122 | 11 | 92.0 |
| 29 | 258 | 18 | 93.0 | 29 | 209 | 12 | 93.0 |
| 30 | 237 | 30 | 96.0 | 30 | 462 | 23 | 96.0 |
| - | - | - | - | 31 | 339 | 24 | 93.0 |

# Chapter 7

# Conclusions and future work

## 7.1 Summary of contributions

The main contributions of the research work are the ProcessPAIR method, the ProcessPAIR tool, the PSP performance model for ProcessPAIR, and the ProcessPAIR experiments.

The ProcessPAIR method for automated performance analysis and improvement recommendation in software development is described in chapter 3. We developed and validated the algorithms for automatically evaluating and ranking (prioritizing) the performance problems and their root causes.

The ProcessPAIR support tool is described in chapter 5 and publicly available in (`http://blogs.fe.up.pt/processpair/`. We designed and developed a novel tool (ProcessPAIR) for automating the identification and prioritization of performance problems and their root causes in software development, and showed its successful application for the PSP. Currently, the tool analyzes performance data produced by PSP developers in their projects, as recorded in the PSP Student Workbook or Process Dashboard, and pinpoints the ranked performance problems and their possible root causes.

The PSP performance model for ProcessPAIR is described in chapter 4. We constructed and validated the performance model (defining performance indicators, cause-effect relationships between them, and recommended ranges or thresholds) for predictability, quality, and productivity that can be used to automatically identify and rank performance problems and their root causes of individual PSP developers. We took advantage of the partnership with the Software Engineering Institute (SEI) and accessed PSP data for calibration of the performance model. Since the model is calibrated based on a large data set, it can also be used for benchmarking purposes.

The ProcessPAIR experimental results for both tool and model are described in chapter 6. We conducted experiments in real world contexts for the validation of the developed models and tool and showed the benefits of the approach. The results of the postmortem experiment show that the ProcessPAIR tool is able to accurately identify performance problems of individual PSP developers and potential causes for those problems.

Regarding the results of a controlled experiment, involving 61 software engineering master students, half of whom used ProcessPAIR in a PSP performance analysis assignment, show significant benefits in terms of students' satisfaction, quality of the analysis outcomes, and time required to do the analysis.

## 7.2 Research questions revisited

The results achieved allow us to answer positively to the 5 research questions set in section 1.2

RQ1. *Is it possible to automatically identify performance problems of individual developers, by taking advantage of performance models derived from the performance data of many process users?*

ProcessPAIR tool automatically points out the 'problematic' PIs (colored with red or yellow semaphores). The only manual work needed is the definition, by a process expert, of the relevant PIs for the process under consideration (only once per process), including their optimal values and formulas for calculation from base data. The performance ranges for assessing the values of the PIs are automatically calibrated based on the data set. Both experiments show that ProcessPAIR is able to accurately identify performance problems of individual PSP developers. In the postmortem experiment, in 292 (96%) out of the 302 cases, the results of manual and automatic analysis are found similar.

Potential benefits are: less time spent in performance analysis; benchmarks for comparison are produced automatically.

RQ2. *Is it possible to automatically identify the root causes of the identified performance problems, by taking advantage of performance models derived from the performance data of many process users?*

ProcessPAIR automatically identifies potential root causes by drilling down from problematic top-level PIs to problematic child PIs that affect the former according to a formula or statistical evidence. The only manual work needed is the definition by a process expert (once per process) of the following information for the process under consideration: child PIs; sensitivity formula for relationships when there is an exact formula. Performance ranges for the child PIs are also calibrated automatically by ProcessPAIR as in RQ1 (for top-level indicators).

In the postmortem experiment, ProcessPAIR was able to point out the same causes as the ones found by the developers in their manual analysis. In 21% of the cases, the tool and the developer pointed out the same causes; in 53% of the cases, the tool and the developer pointed out causes at different levels of detail, with the tool accurately pointing out intermediate causes, and the developer pointing out deeper causes. These are cases where the automatic analysis accurately points out factors to focus in subsequent manual analysis, potentially reducing the overall manual effort. In 26% of the cases the results are inconsistent, because of faults in the manual analysis. Such cases might be prevented if our tool was used by the subjects when conducting their performance analysis.

RQ3. *Is it possible to automatically rank the identified performance problems, by taking advantage of performance models derived from the performance data of many process users?*

ProcessPAIR ranks the performance problems based on semaphore (red or yellow) and overall percentile. The semaphore is based on the performance ranges. The percentile is based on the approximate statistical distribution. No manual input is needed. We expect that, by presenting the performance problems properly ranked, developers may focus on the most important performance problems.

RQ4. *Is it possible to automatically rank the identified root causes, by taking advantage of performance models derived from the performance data of many process users?*

ProcessPAIR ranks the identified root causes based on a combination of a percentile coefficient and a sensitivity coefficient that, together, give a cost-benefit estimate of improvement actions. The information needed comes from the data set (statistical distribution, etc.). The only manual work needed is the definition of the sensitivity formulas for PIs related by an exact formula (also needed in RQ2). We expect that, by presenting the potential root causes properly ranked, developers may focus on the most important causes.

RQ5. *By automating the performance analysis as described in RQ1 to RQ4, is it possible to reduce effort and errors and improve user satisfaction as compared to manual analysis?*

In the postmortem experiment, ProcessPAIR was able to accurately identify performance problems and their potential root causes, so we expect that, by using ProcessPAIR, manual work is only needed for identifying deeper causes, reducing the overall effort. Since in a significant number of cases we verified that users pointed out erroneous or meaningless causes, we also expect that, by using ProcessPAIR, errors in the analysis are reduced.

In the controlled experiment, the results show significant benefits in terms of users' satisfaction as compared to previous approaches (average score of 4.78 in a 1-5 scale for ProcessPAIR users, against 3.81 for non-ProcessPAIR users), quality of the analysis outcomes (average grades achieved of 88.1 in a 0-100 scale for ProcessPAIR users, against 82.5 for non-ProcessPAIR users), and time required to do the analysis (average of 252 minutes for ProcessPAIR users, against 262 minutes for non-ProcessPAIR users, but with much room for improvement). The comments regarding overall usefulness and level of support for performance analysis purposes (see Table 6.10) are also very encouraging.

## 7.3 Future work

Currently, ProcessPAIR is available as a standalone Java application. As future work, we intend to deploy ProcessPAIR as a service available on the cloud (SaaS) and integrate it with a cloud-based application lifecycle management tool. This will increase tool accessibility, facilitate metrics collection, and enable the automatic recalibration of the performance models based on the performance data of the users.

Also the performance models are hardcoded as ProcessPAIR extensions. In future, we intend to provide means (graphical user interfaces and configuration files) for defining performance models without programming.

Once deployed on the cloud, we intend to add to ProcessPAIR the capability of recommending detailed improvement actions for the identified causes of performance problems, based on a catalogue of improvement actions constructed using crowdsourcing techniques. Some initial work was already developed to that end, resulting in a Web based platform (WebProcessPAIR, 2016) and a master thesis co-supervised by the author.

Currently, ProcessPAIR is being successfully used by PSP users. In the future, we intend to expand the user base by tailoring ProcessPAIR for other processes, namely, investigate the application of ProcessPAIR for analyzing the adherence to agile practices and the performance of agile teams (using Scrum, XP or TSP).

We also intend to explore statistical and machine leaning techniques to further automate the ProcessPAIR method, e.g., to automatically calibrate optimal values of PIs, and to automatically identify relationships between PIs.

# Appendix A

# Tutorial on PSP Final Report guidelines with ProcessPAIR

This tutorial provides guidelines for producing the PSP Final Report with the help of ProcessPAIR. The requirements for the PSP Final Report are described in the official SEI's "Assignment Kit for Final Report" of the "PSP for Engineers" course.

ProcessPAIR provides important advantages:

- compare your performance data with the performance data of a large number of PSP students (more than 3,000);
- significantly automates the identification of your performance problems and their root causes;
- reduces the effort needed to produce the PSP Final Report;
- prevents errors in the analysis;
- lets you focus your analysis on the identification of deeper causes (not present in the data collected) and remedial actions.

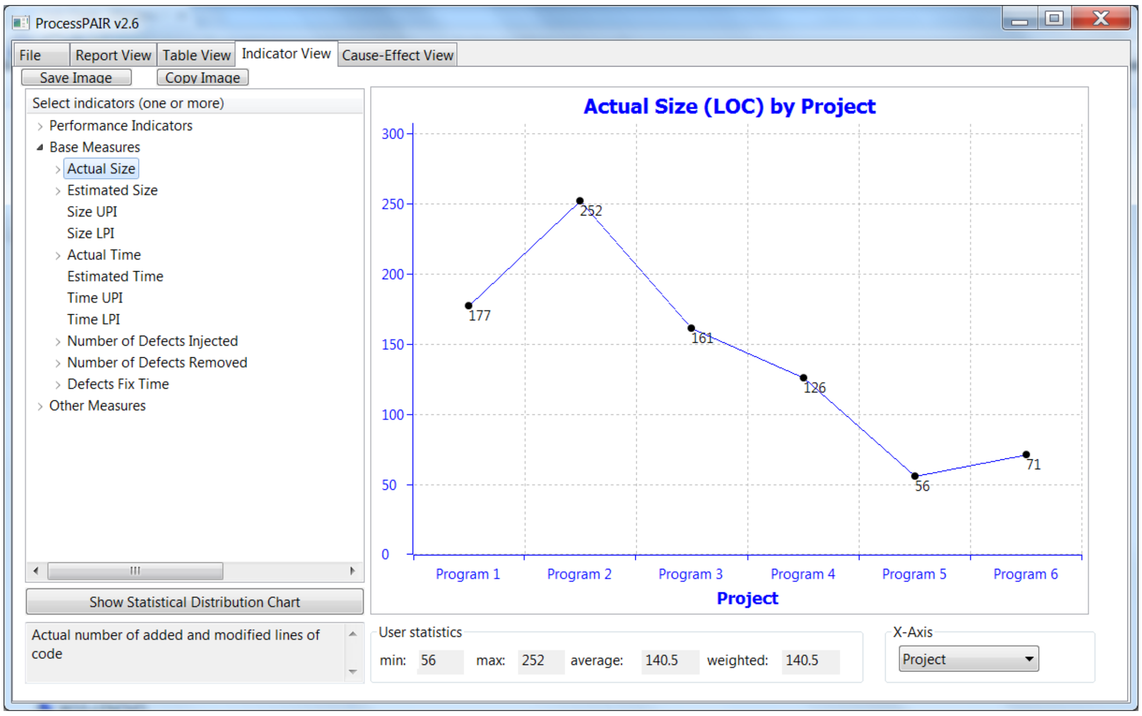
The next sections are structured according to the analysis questions described in the assignment kit. For each question, it is explained how ProcessPAIR helps answering the question. It is assumed that you have recorded your performance data with Process Dashboard or the SEI's PSP Student Workbook. Before proceeding to the next sections, please open ProcessPAIR and upload your performance data file, selecting the "PSP Performance Model for PSP Final Report" model. It is important that you read the guidelines sequentially because several features and techniques are explained on the first occurrence.

- Analysis of size estimating accuracy
- Analysis of time estimating accuracy
- Defect and yield analysis
- Quality analysis

## A.1 Analysis of size estimating accuracy

1. **What are the average, maximum, and minimum actual sizes of your programs in LOC to date?**

   In the "Indicator View", "Base Measures" group, select the "Actual Size" item to obtain the required information, as illustrated in Figure A.1. You can copy the chart to your report, using the "Copy Image" button on the top of the screen or the usual print screen facilities, and add any pertinent comments. In this example, the average, maximum, and minimum actual sizes in LOC to date are 140.5, 252 and 56, respectively. Please notice that these are added and modified lines of code (as explained in the bottom left of Figure A.1).



Actual number of added and modified lines of code

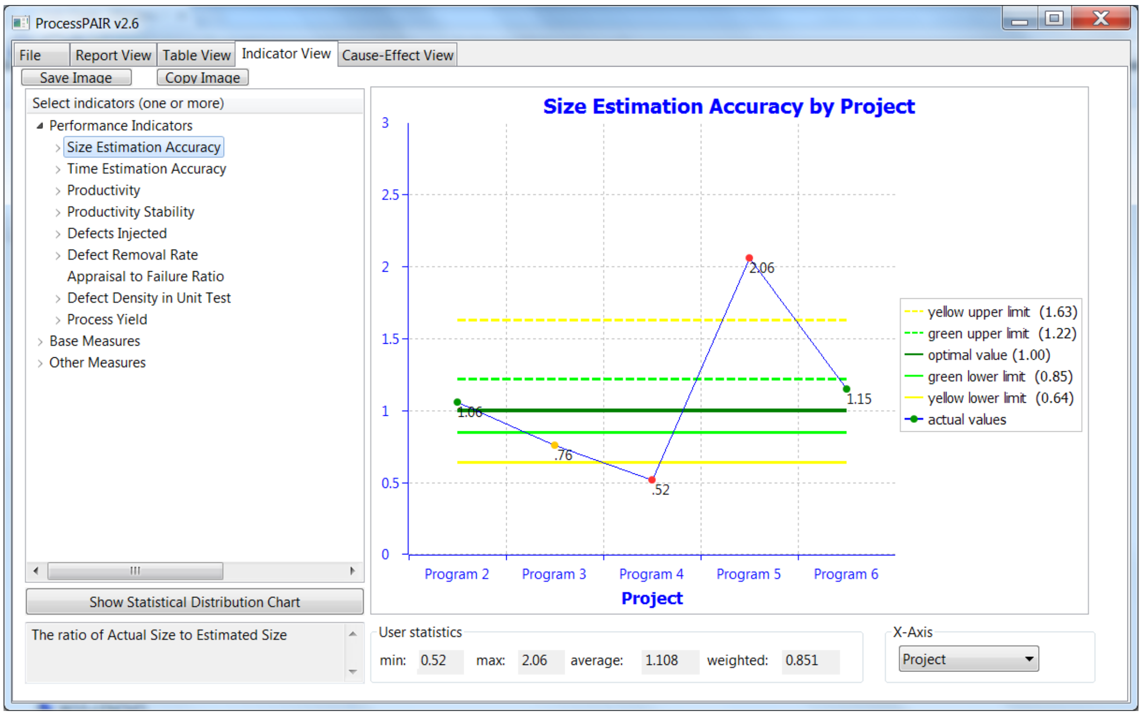
User statistics
min: 56 max: 252 average: 140.5 weighted: 140.5

Figure A.1: Actual size (LOC) by Project.

2. **Excluding assignment 1, what percentage over or under the actual size was the estimated size for each program (for example, if estimated/actual is in %, 85% is 15% under, 120% is 20% over)? What are your average, maximum, and minimum values for these?**

   In the "Indicator View", "Performance Indicators" group, select the "Size Estimation Accuracy" item to obtain the required information, as illustrated in Figure A.2:

   Please notice that ProcessPAIR measures the size estimation accuracy as the ratio of actual to estimated size, being 1 the optimal value. To obtain the percentual size estimation error you have to subtract 1 and multiply by 100%. In Figure A.2, the 2.06 accuracy corresponds to a 106 % (under) estimation error, and the 0.52 accuracy corresponds to a -48% (over) estimation error. The average is 1.108 (10.8%).



The ratio of Actual Size to Estimated Size

User statistics
min: 0.52 max: 2.06 average: 1.108 weighted: 0.851

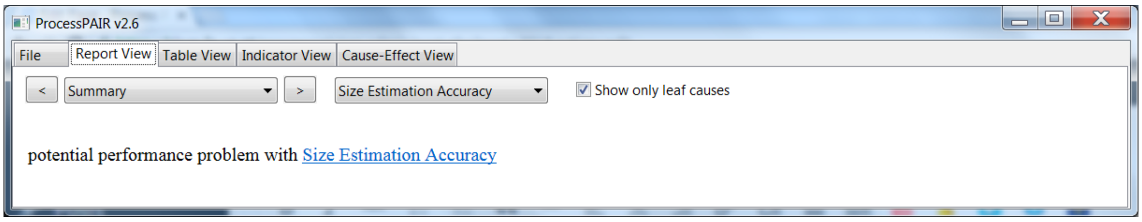
X-Axis
Project

Figure A.2: Size estimation accuracy by Project.

ProcessPAIR computes the weighted average of a ratio by dividing the total value of the numerator for all projects by the total value of the denominator for all projects. In this case, it is the total actual size of programs 2 to 6 divided by the total estimated size of programs 2 to 6. The weighted average is 0.851 (-14.9%). This value is of less interest in this case.

ProcessPAIR also shows in the chart the recommended performance ranges, calibrated based on the performance data of many PSP students. The green range (good performance) corresponds to the 1/3 best values in the calibration data, and goes from 0.85 to 1.22 (-15% to 22%). The 'red' range (poor performance) corresponds to the 1/3 worst values in the calibration data. The yellow range is in between. By comparing your data with the control limits shown in the chart , you can make informed comments about your own performance. In the Figure A.2 you can see, the student was not able to reach a good size estimation performance (only two points green), as compared to the population used for calibration.

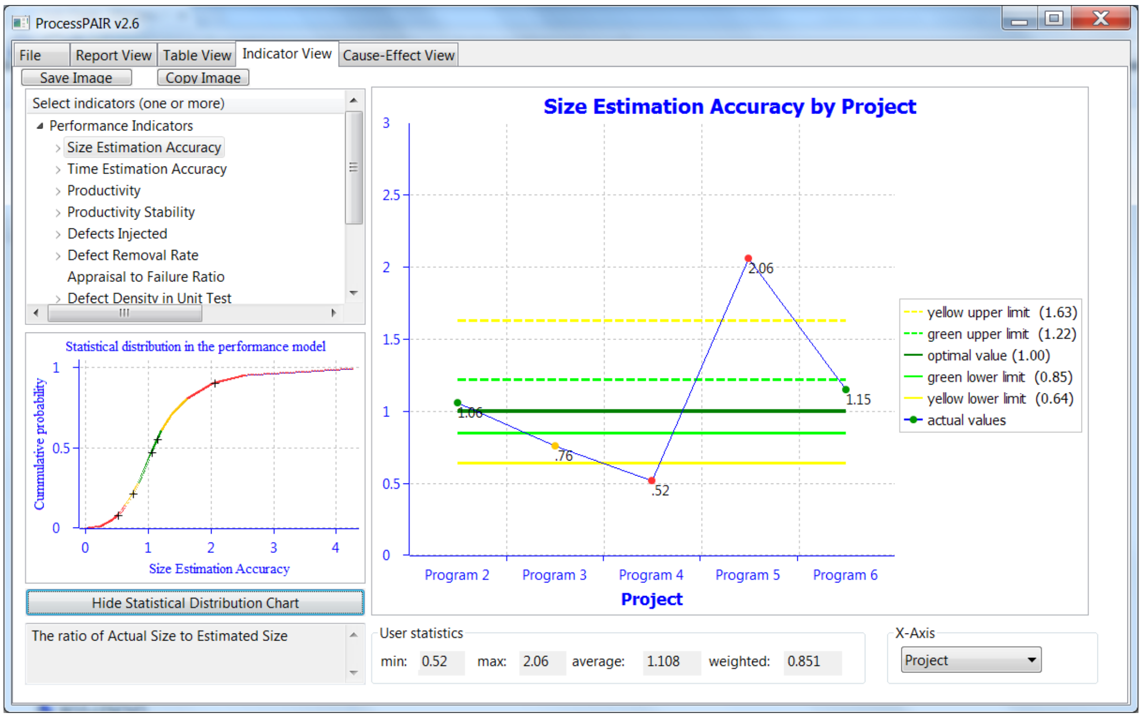
You can also check in the "Report View" if ProcessPAIR identified any clear or potential (moderate) problem regarding your size estimation performance, as illustrated A.3.



potential performance problem with Size Estimation Accuracy

Figure A.3: Report view.

By pressing the "Show Statistical Distribution Chart" button, you can get another view of how your performance compares with the population used for calibration, as illustrated in Figure A.4 (bottom left).



The ratio of Actual Size to Estimated Size

User statistics

min: 0.52 max: 2.06 average: 1.108 weighted: 0.851

X-Axis: Project

Figure A.4: Size estimation accuracy by project.

By selecting both the "Actual Size" and "Estimated Size" in the list of indicators, you can also visually compare the base measures from which the Size Estimation Accuracy is computed, as illustrated in Figure A.5.

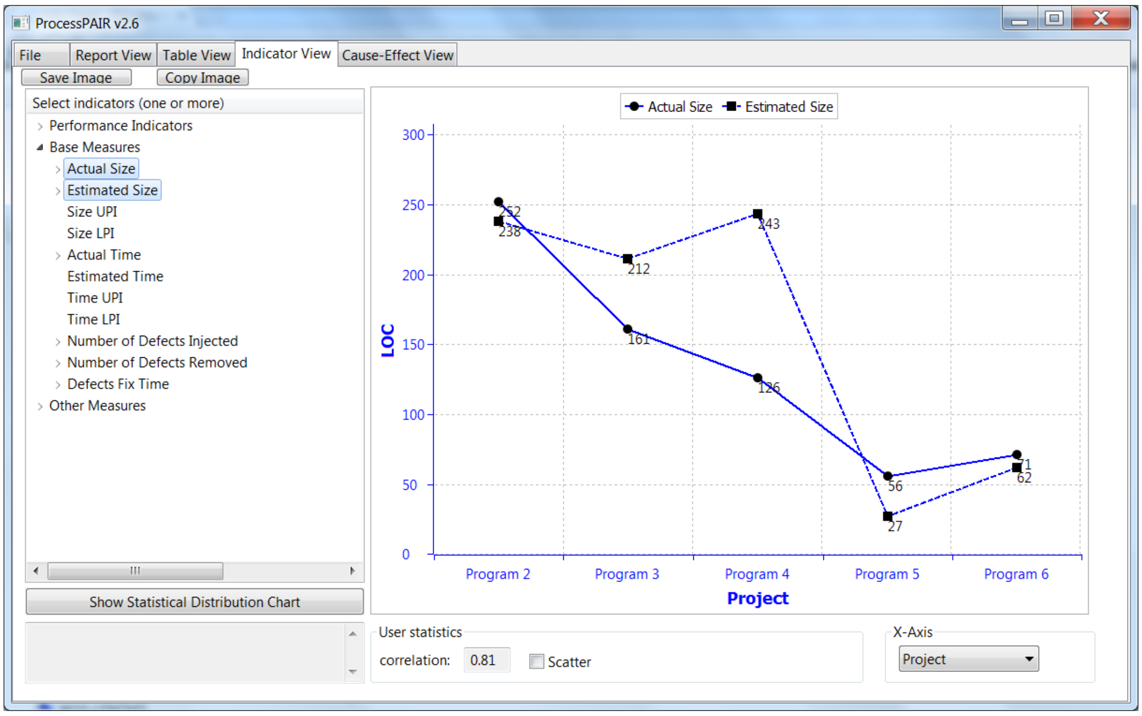
3. **How often was my actual program size within my 70% statistical prediction interval (when you used methods A or B)?**

   The relevant chart is obtained by selecting the Actual Size, Size UPI (upper prediction interval) and Size LPI (lower prediction interval), as shown in Figure A.6. In this example a prediction interval was computed only in program 6. The actual size (round symbol) is within the 70% statistical prediction interval (between LPI and UPI).

4. **Do I have a tendency to add/miss entire objects?**

   The relevant information can be find in Figure A.7. In ProcessPAIR, missing objects are objects that were actually added or modified, but were not planned. Extraneous objects are objects that were planned, but were not actually added or modified. In Figure A.8, missed or extraneous objects occurred only in the first project, so it is fair to conclude that there is no tendency to add/miss entire objects.

5. **Do I have a tendency to misjudge the relative size of objects?**



correlation: 0.81

Figure A.5: Indicator view.

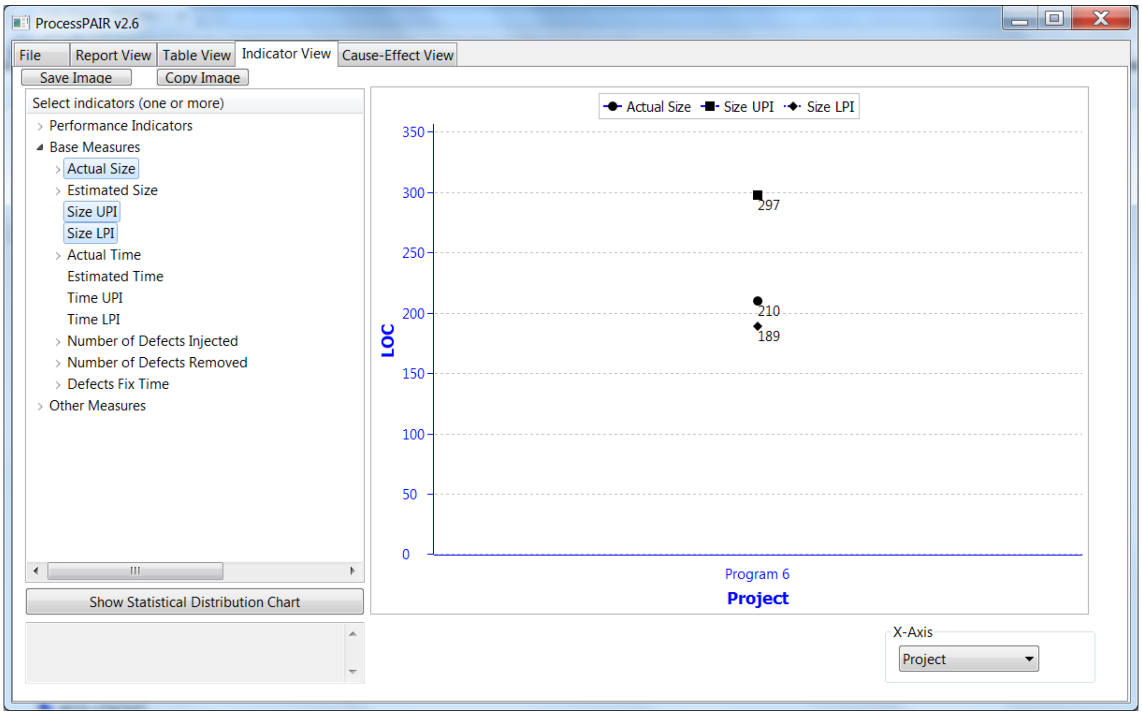
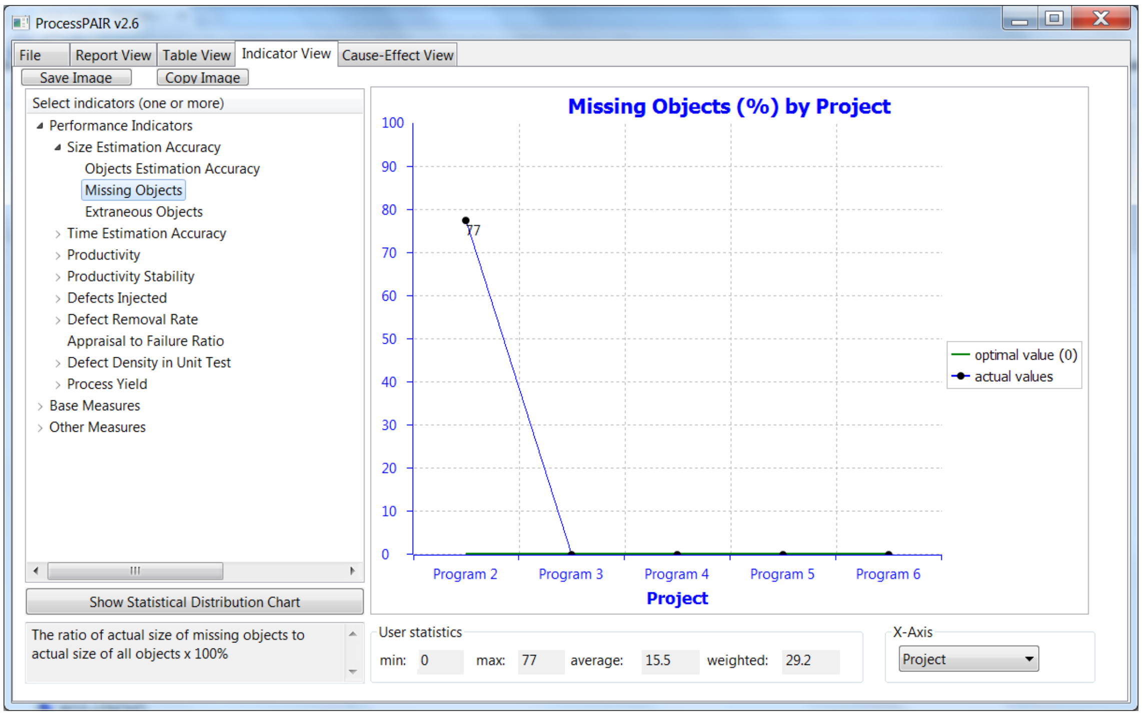


Figure A.6: Indicator view.

The relevant information can be found in Figure A.9. This Figure refers to objects that were correctly identified in the planning phase (i.e., missing or extraneous objects are not included here).

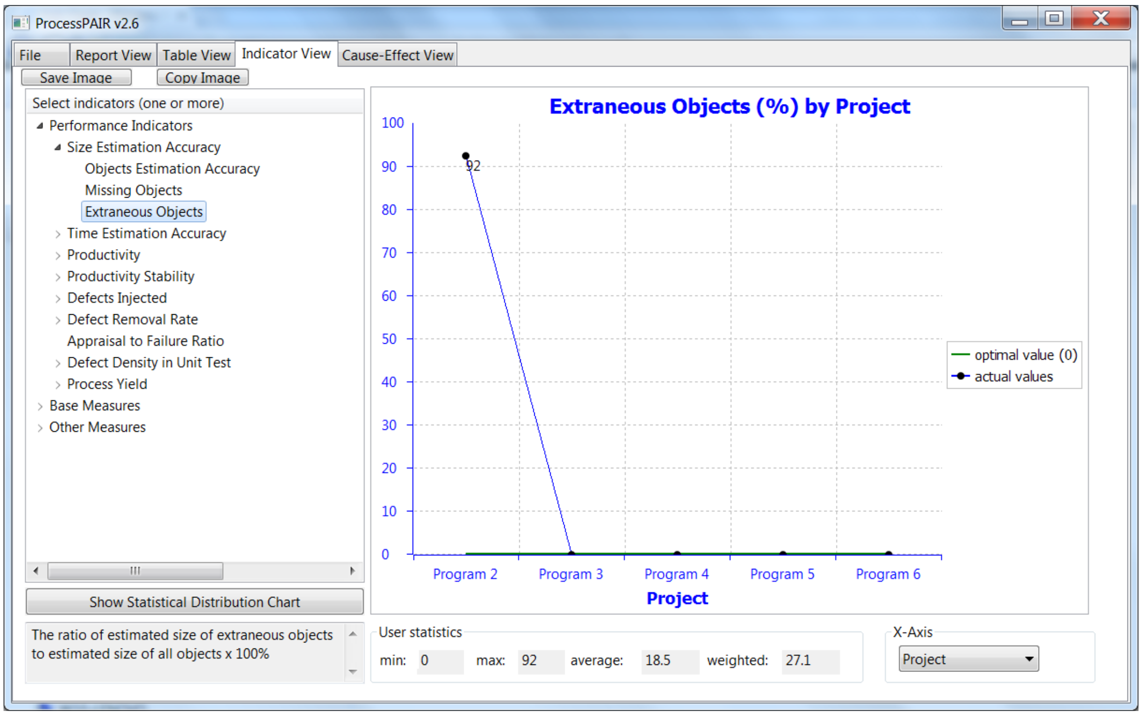


The ratio of actual size of missing objects to actual size of all objects x 100%

User statistics

min: 0 max: 77 average: 15.5 weighted: 29.2

Figure A.7: Indicator view.



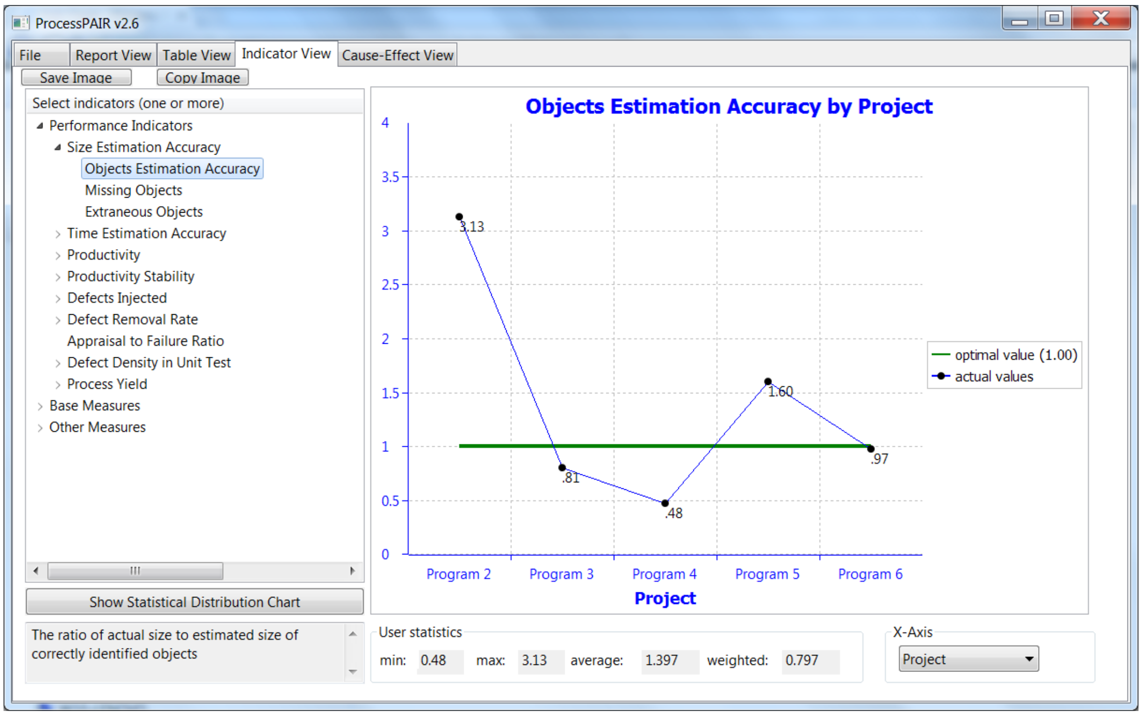
The ratio of estimated size of extraneous objects to estimated size of all objects x 100%

User statistics

min: 0 max: 92 average: 18.5 weighted: 27.1

Figure A.8: Indicator view.

Currently, ProcessPAIR has no benchmarks for this indicator, because of the lack of historical data, but it is reasonable to use ranges similar to the Size Estimation Accuracy, i.e., a green range between 0.85 and 1.22. In this example, although it can be observed a tendency for



The ratio of actual size to estimated size of correctly identified objects

User statistics: min: 0.48 max: 3.13 average: 1.397 weighted: 0.797

X-Axis: Project

Figure A.9: Indicator view.

improvement, there is an oscillatory behavior with room for improvement towards the green range.

By viewing together the Objects Estimation Accuracy and the Size Estimation Accuracy, we can check that they are closely related as expected, as shown in Figure A.10. After program 2, the small deviations occur because size estimates sometimes have a statistical adjustment.

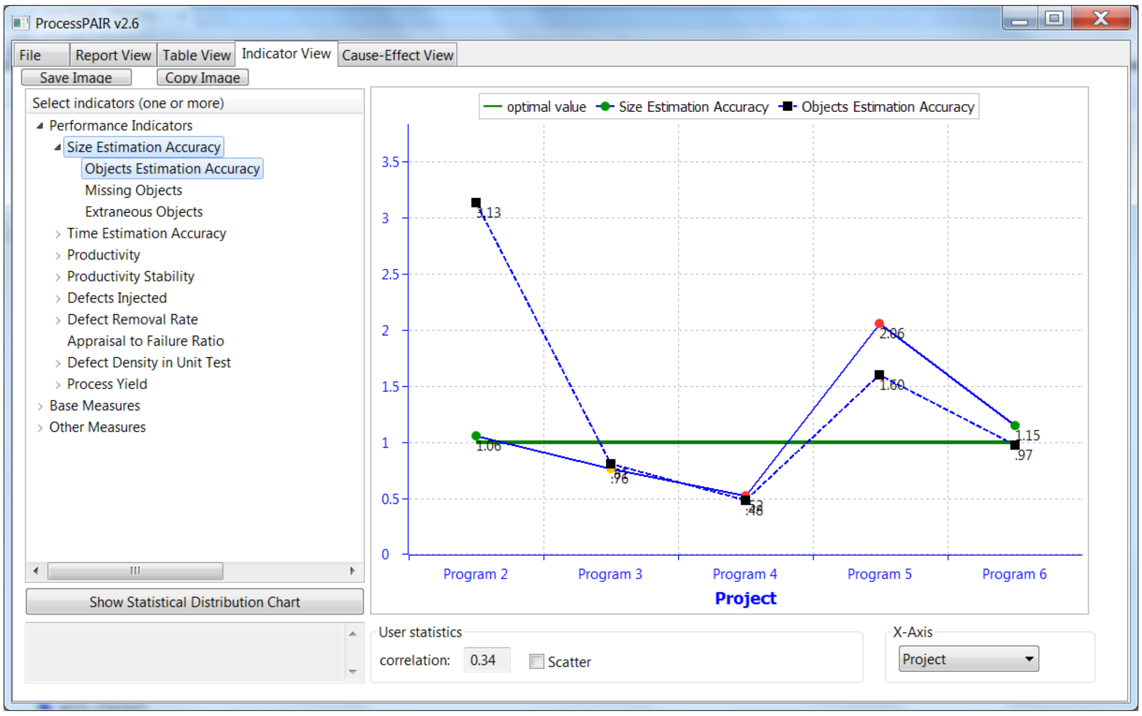
6. **Based on my historical size-estimating accuracy data, what is a realistic size-estimating goal for me?**

   ProcessPAIR lets you compare your performance to the performance achieved by other people, and hence helps establishing realistic goals. In case you already have a good performance (green range), then probably you just have to keep your current performance. Otherwise, moving to the next range may be a realistic goal (from red to yellow, or from yellow to green).

7. **How can I change my process to meet that goal?**

   Before thinking of process changes you should first identify root causes for current performance issues. To help identifying root causes, ProcessPAIR organizes the performance indicators in a tree-like structure, where the child nodes represent factors that may affect the parent nodes, as illustrated in Figure A.11.

   So, by drilling down from top-level indicators to child indicators, you can identify the problematic factors (root causes) and subsequently devise improvement actions (process changes) to address them, as illustrated in Figure A.11.



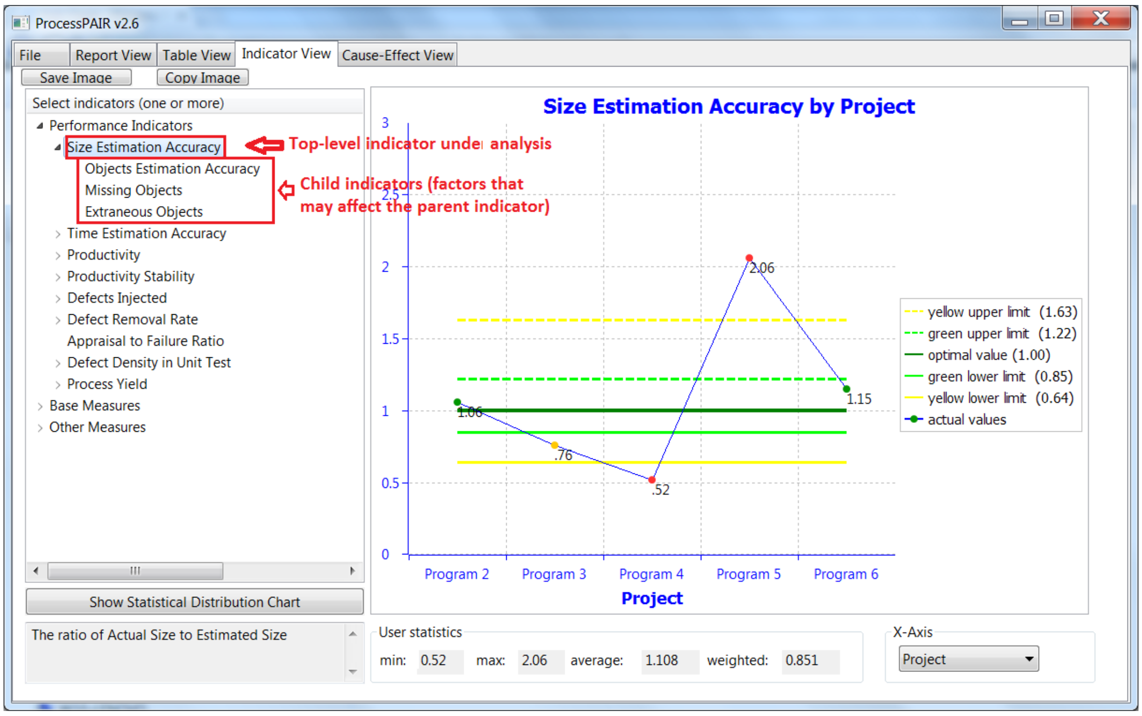
User statistics

correlation: 0.34 Scatter

X-Axis

Project

Figure A.10: Indicator view.



The ratio of Actual Size to Estimated Size

User statistics

min: 0.52 max: 2.06 average: 1.108 weighted: 0.851

X-Axis

Project

Figure A.11: Indicator view.

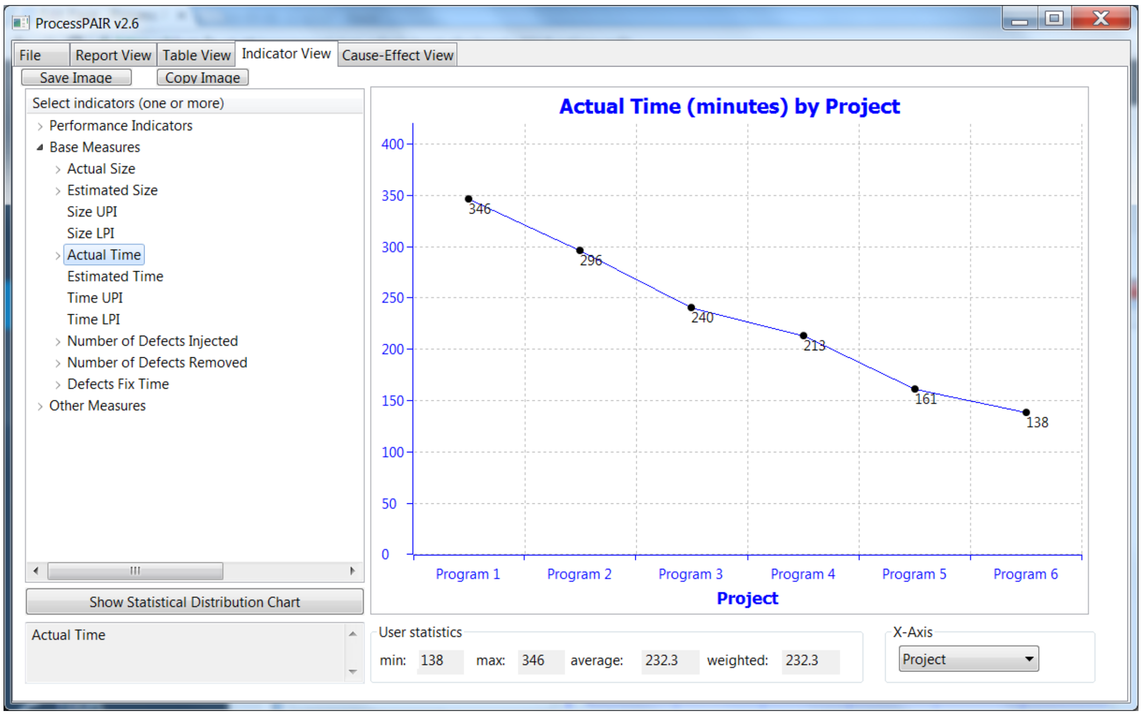
In the case of size estimation accuracy, the child indicators are related with the causes asked in previous questions: tendency to add/miss entire objects; tendency to misjudge the relative size of objects. In the example shown, the relevant cause is the tendency to misjudge the relative size of objects. It can be observed a tendency for improvement, with a very good accuracy in the last project (0.96 or -4% error), but oscillations are still high. Usually such oscillations are reduced with continued practice.

In case you found deeper or different causes for the time estimation problems, you should devise actions for addressing them.

## A.2 Analysis of time estimating accuracy

1. **What are the average, maximum, and minimum times of your assignments to date?**

   In the “Indicator View”, “Base Measures” group, select the “Actual Time” item to obtain the required information, as illustrated in Figure A.12. You can copy the figure to your report, using the “Copy Image” button or the usual print screen facilities, and add pertinent comments. In this example, the average, maximum, and minimum times in the assignments to date are 232.3, 346 and 138 minutes, respectively.

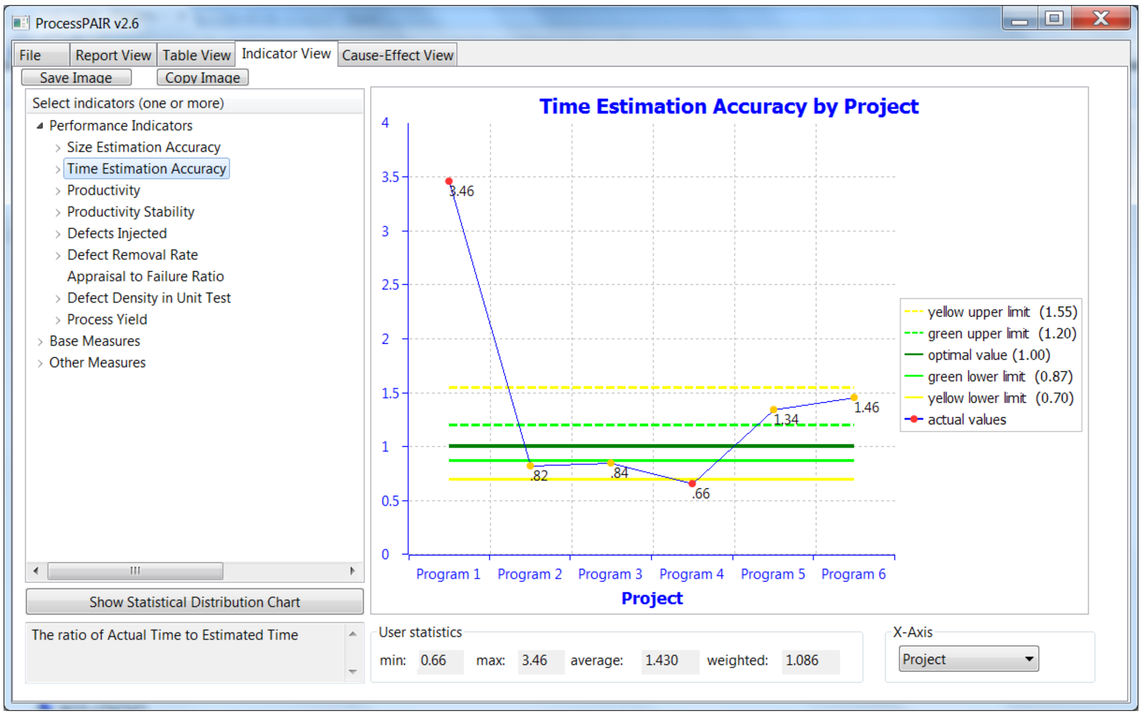


User statistics
min: 138 max: 346 average: 232.3 weighted: 232.3

Figure A.12: Indicator view.

2. **What percentage over or under the actual time was the estimated time for each program (for example, if estimated/actual is in %, 85% is 15% under, 120% is 20% over)? What are your average, maximum, and minimum values for these?**

   In the “Indicator View”, “Performance Indicators” group, select the “Time Estimation Accuracy” item to obtain the required information, as illustrated in Figure A.13.



The ratio of Actual Time to Estimated Time

User statistics

min: 0.66 max: 3.46 average: 1.430 weighted: 1.086

X-Axis

Project

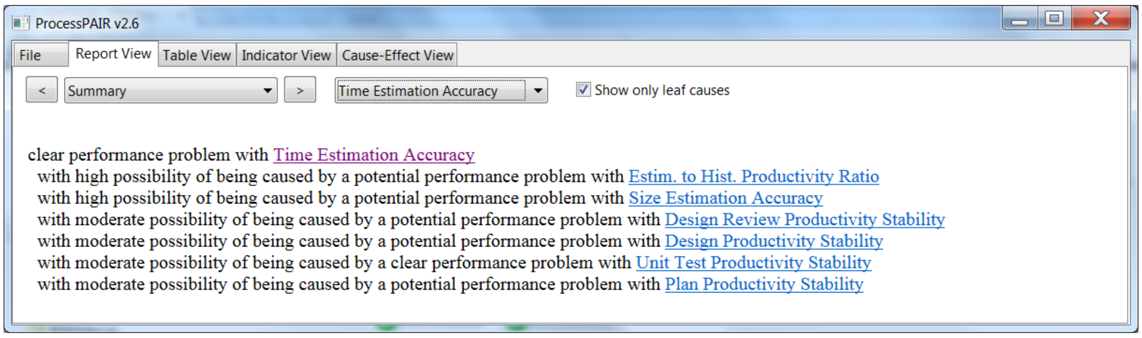
Figure A.13: Indicator view.

Similarly to the size estimating accuracy, you can notice that ProcessPAIR measures the time estimation accuracy as the ratio of actual to estimated time, being 1 the optimal value. To obtain the percentual size estimation error you have to subtract 1 and multiply by 100%. In the Figure A.13, the 3.46 accuracy corresponds to a 246% (under) estimation error, and the 0.66 accuracy corresponds to a -34% (over) estimation error. The (simple) average is 1.430 (43% under estimation error). The weighted average is 1.086 (8.6% under estimation error).

ProcessPAIR also shows in the chart the recommended performance ranges, calibrated based on the performance data of many PSP students. The green range (good performance) corresponds to the 1/3 best values in the calibration data, and goes from 0.87 to 1.20 (-13% to 20%). By comparing your data with the control limits shown in the chart , you can make informed comments about your own performance. In the Figure A.13, there was a significant improvement after the first project, but the student was not yet able to produce good estimates (inside the green range).

You can also check in the “Report View” if ProcessPAIR identified any clear or potential problem regarding your time estimation performance, as illustrated in Figure A.14. In this example ProcessPAIR indicates a clear performance problem, together with some causes (to be investigated later in this section).

By selecting both the “Actual Time” and “Estimated Time” in the list of indicators, you can also visually compare the base measures from which the Time Estimation Accuracy is computed, as illustrated in Figure A.15.



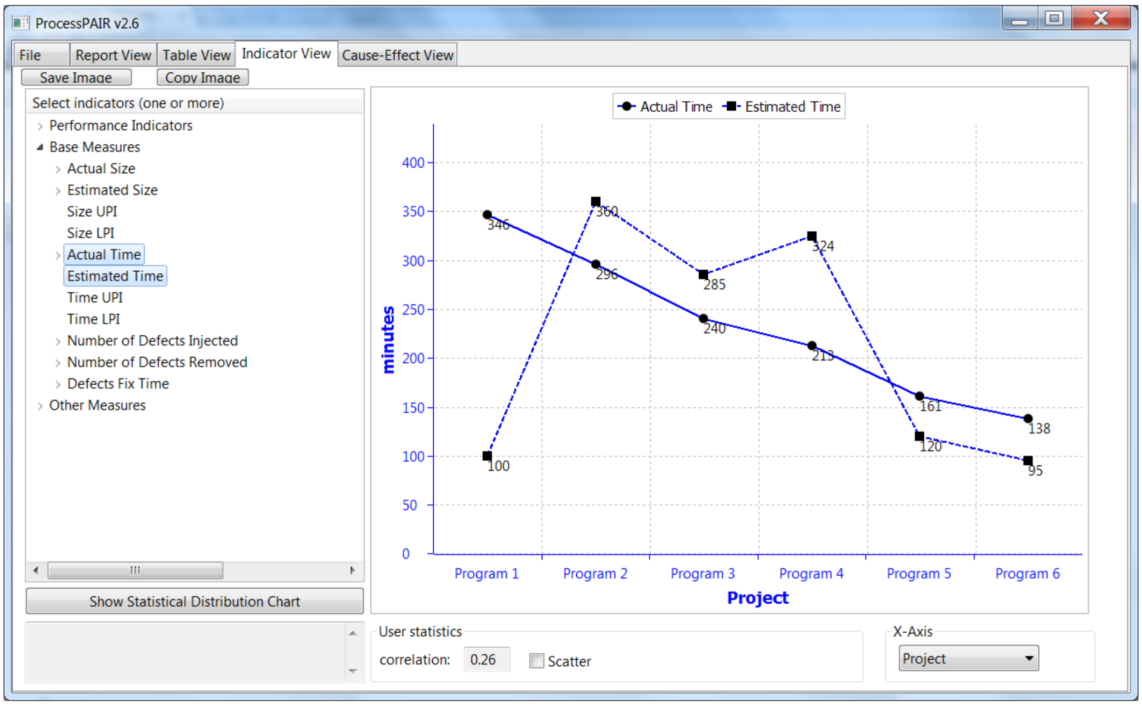
Summary

Time Estimation Accuracy

☑ Show only leaf causes

clear performance problem with Time Estimation Accuracy
with high possibility of being caused by a potential performance problem with Estim. to Hist. Productivity Ratio
with high possibility of being caused by a potential performance problem with Size Estimation Accuracy
with moderate possibility of being caused by a potential performance problem with Design Review Productivity Stability
with moderate possibility of being caused by a potential performance problem with Design Productivity Stability
with moderate possibility of being caused by a clear performance problem with Unit Test Productivity Stability
with moderate possibility of being caused by a potential performance problem with Plan Productivity Stability

Figure A.14: Report view.



correlation: 0.26 ☐ Scatter

X-Axis Project

Figure A.15: Indicator view.

3. **How often was my actual development time within my 70% statistical prediction interval (when you used methods A or B)?**

   The relevant chart is obtained by selecting the Actual Time, Time UPI and Time LPI, as shown in the Figure A.16. In this example a prediction interval was computed only in program 6. The actual time (round symbol) is well within the 70% statistical prediction interval (between lower prediction interval and upper prediction interval).

4. **What are the average, maximum, and minimum values for productivity per program to date in LOC/hr?**

   In the “Indicator View”, “Performance Indicators” group, select the “Productivity” item to obtain the required information, as illustrated in Figure A.17.

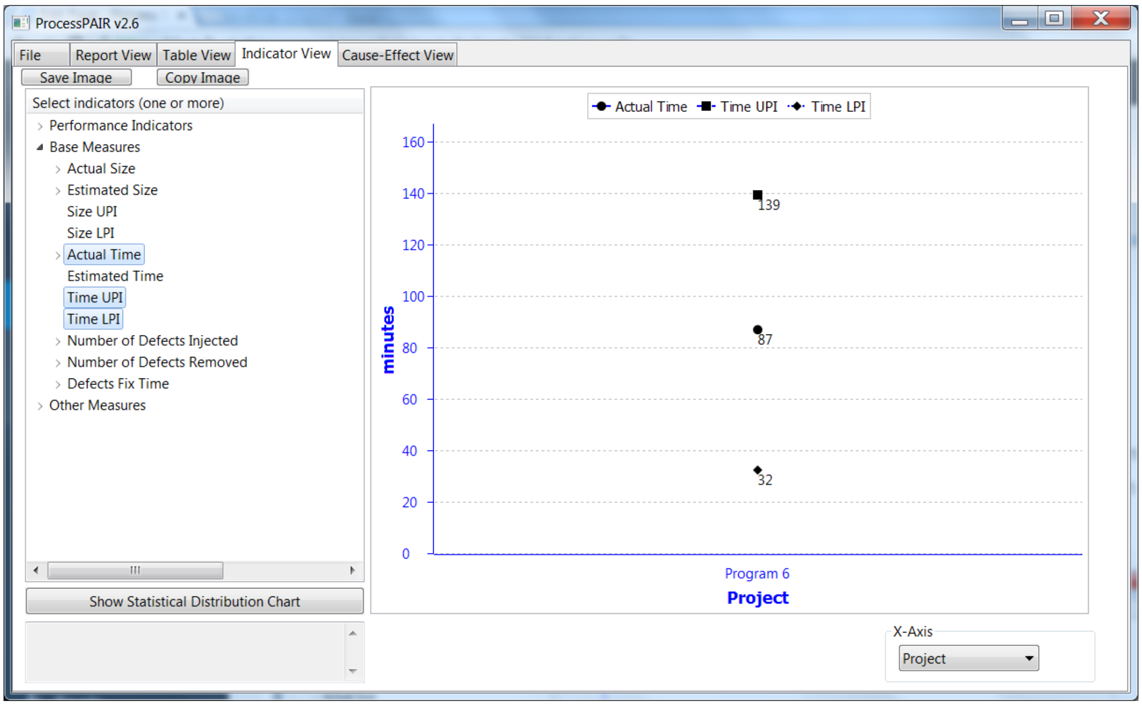
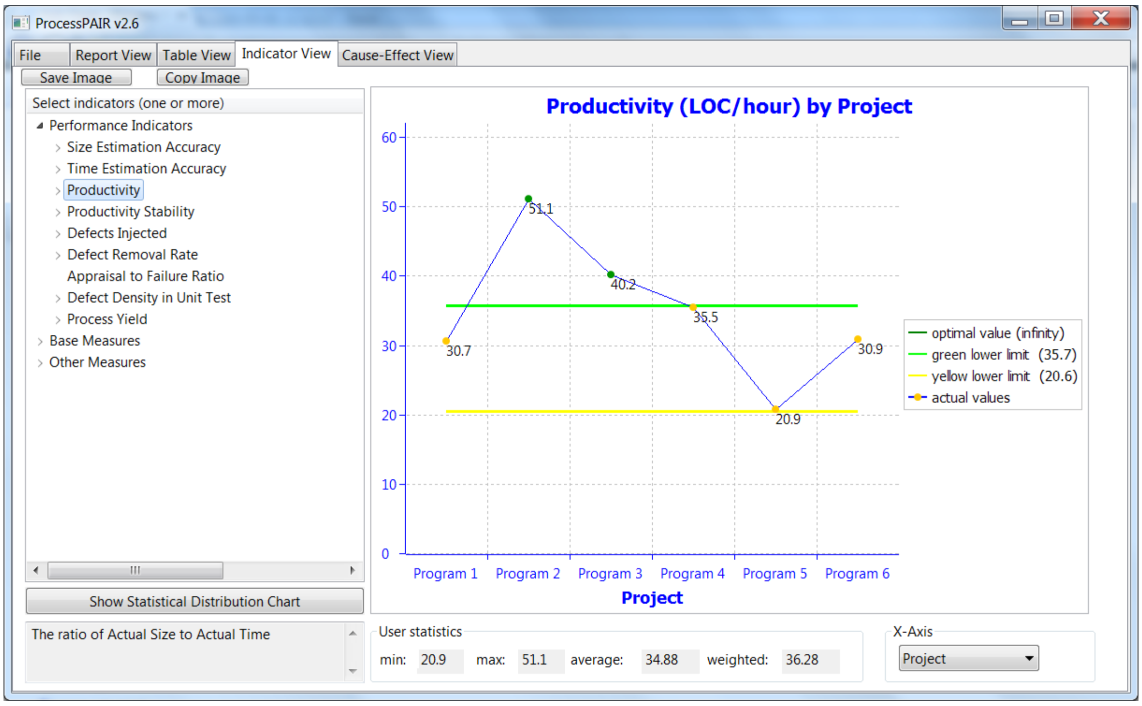


Figure A.16: Indicator view.



The ratio of Actual Size to Actual Time

User statistics

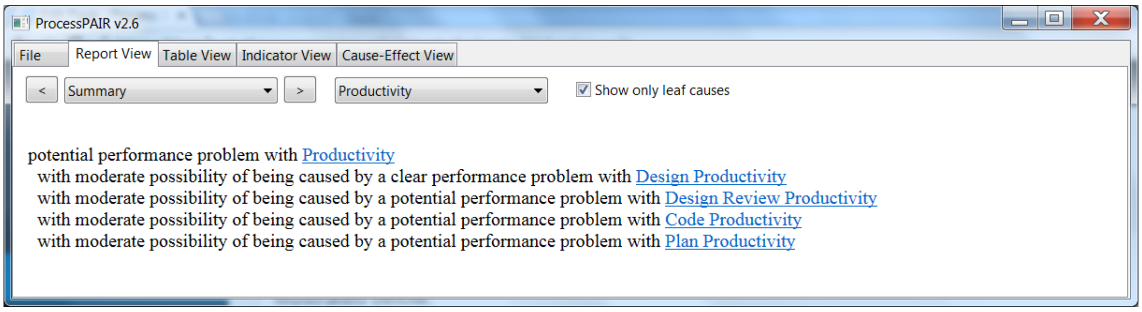
min: 20.9 max: 51.1 average: 34.88 weighted: 36.28

Figure A.17: Indicator view.

You can notice that the weighted average is computed as the ratio between the total size of the programs developed to date and the total time spent in developing those programs.

Similarly to other performance indicators, ProcessPAIR also shows in the chart the recommended performance ranges, calibrated based on the performance data of many PSP students. The green range (good performance) contains the 1/3 best values in the calibration data, and corresponds to a productivity greater or equal to 35.7 LOC/hour. By comparing your data with the control limits shown in the chart , you can make informed comments about your own performance. In the Figure A.17, the productivity is almost always in the green range.

You can also check in the "Report View" if ProcessPAIR identified any clear or potential problem regarding your productivity. In this example, the Report View indicates a potential performance problem with Productivity, as well as the problematic phases, as illustrated in Figure A.18.



potential performance problem with Productivity
with moderate possibility of being caused by a clear performance problem with Design Productivity
with moderate possibility of being caused by a potential performance problem with Design Review Productivity
with moderate possibility of being caused by a potential performance problem with Code Productivity
with moderate possibility of being caused by a potential performance problem with Plan Productivity

Figure A.18: Report view.

By selecting both the "Actual Size" and "Actual Time" in the list of indicators, you can also visually compare the base measures from which the Productivity is computed, as illustrated in the Figure A.19 (with Scatter mode selected in the bottom). As expected, higher sizes are associated with higher development efforts (0.84 linear correlation coefficient).

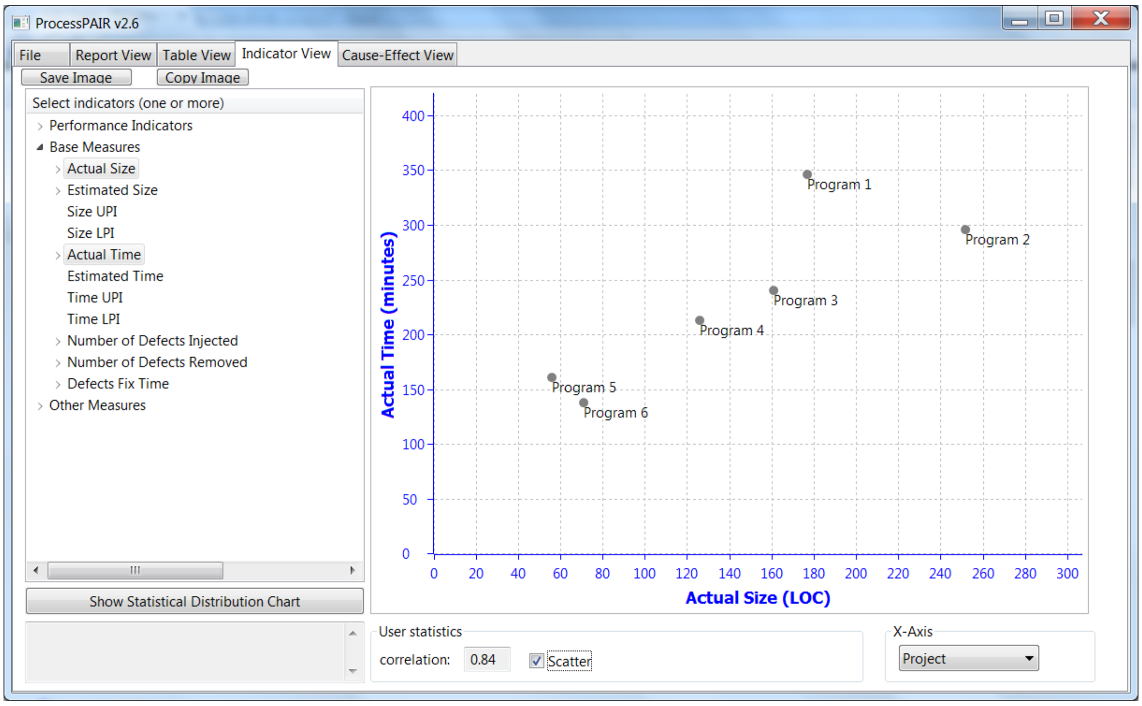
5. **Is my productivity stable? Why or why not?**

You can check the productivity stability by inspecting the "Productivity" indicator, or by inspecting the "Productivity Stability" indicator computed by ProcessPAIR, as illustrated in Figure A.20.

In ProcessPAIR, the "Productivity Stability" of a project measures how close the productivity in that project is to the historical productivity (of previous projects), and is given by the ratio of the productivity in that project to the historical productivity, being 1 the optimal value.

Similarly to other performance indicators, ProcessPAIR also shows in the chart the recommended performance ranges, calibrated based on the performance data of many PSP students. In this case, the green range corresponds to a productivity stability between 0.80 and 1.19. By comparing your data with the control limits shown in the chart , you can make informed comments about your own performance. In the Figure A.20, the productivity stability is almost always inside the green range.

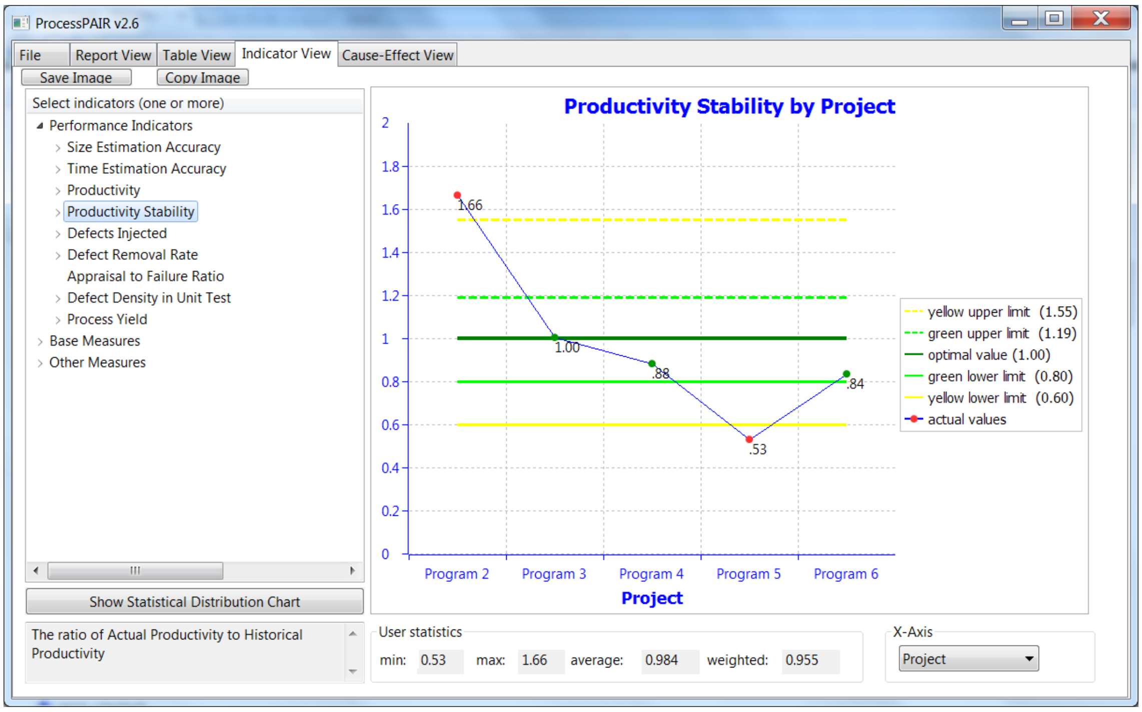
In PSP training there is usually a productivity decrease in the middle of the training, when some process changes are introduced, followed by a productivity recovery as the new processes are



correlation: 0.84 ✓ Scatter

X-Axis: Project

Figure A.19: Indicator view.



The ratio of Actual Productivity to Historical Productivity

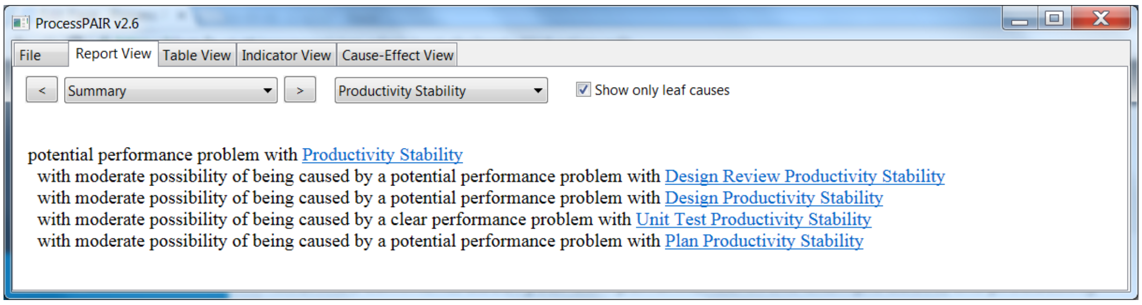
min: 0.53 max: 1.66 average: 0.984 weighted: 0.955

X-Axis: Project

Figure A.20: Indicator view.

practiced. Since the process changes usually affect specific phases, ProcessPAIR also analyzes the Productivity Stability of each process phase.

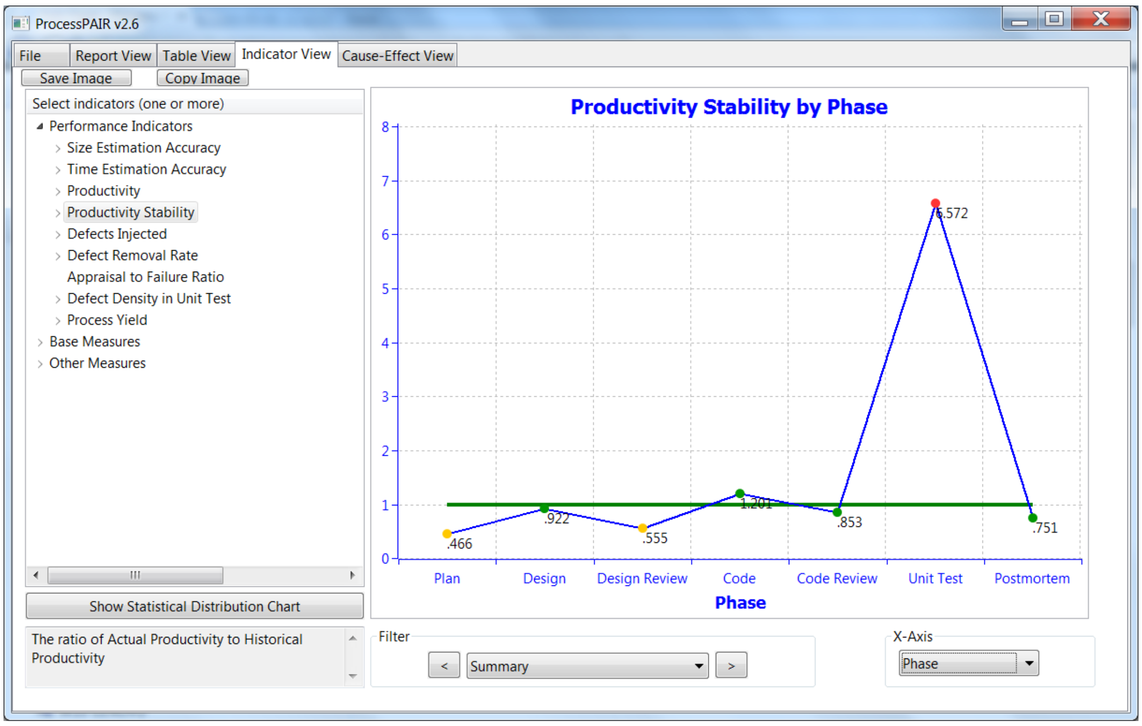
The simplest way to identify the phases that are causing productivity instability problems is to look at the “Report View”, as illustrated in Figure A.21. In this example, there is a moderate problem with the overall productivity stability, caused by instability in the Design Review, Design, Unit Test and Plan phases (by decreasing order of importance). This is a typical situation in PSP training, because of the changes introduced in the design phase (introduction of design templates), design review phase (introduction of design verification techniques), and plan phase (introduction of size estimation, introduction of quality planning, etc.). The instability in the Unit Test phase usually occurs because of the decrease of defects entering the unit test phase.



potential performance problem with Productivity Stability
with moderate possibility of being caused by a potential performance problem with Design Review Productivity Stability
with moderate possibility of being caused by a potential performance problem with Design Productivity Stability
with moderate possibility of being caused by a clear performance problem with Unit Test Productivity Stability
with moderate possibility of being caused by a potential performance problem with Plan Productivity Stability

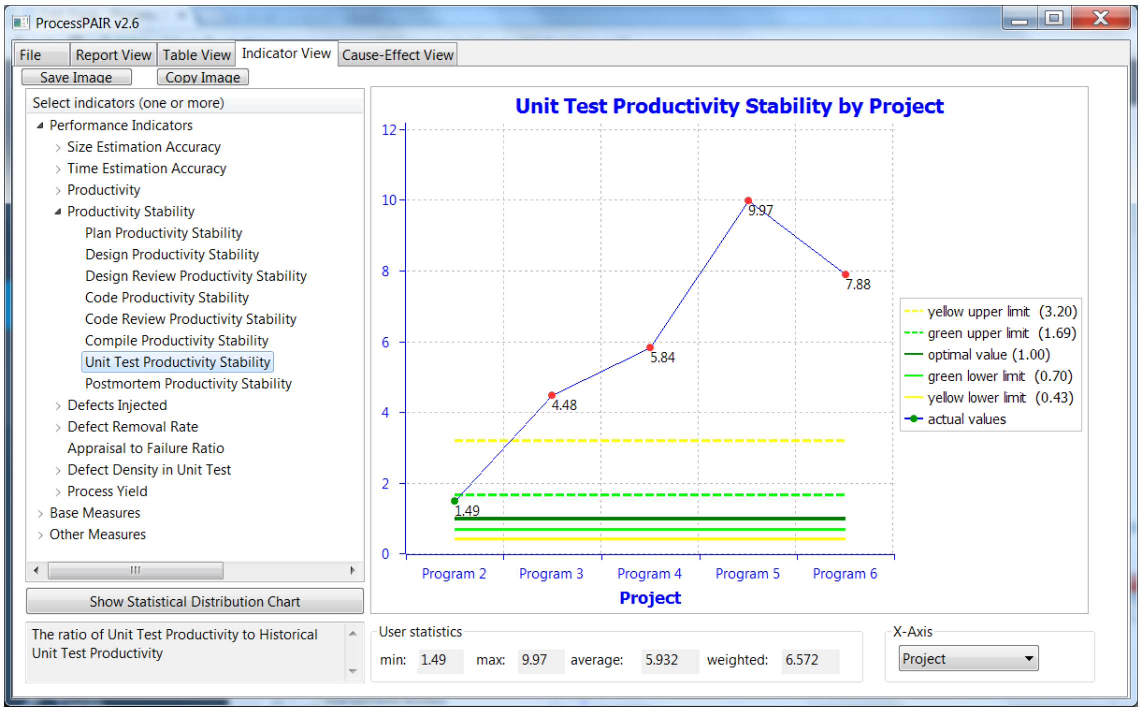
Figure A.21: Report view.

To manually identify the problematic phases causing productivity instability problems you can look at different charts in the “Indicator View”, e.g., by switching the X-Axis to “Phase” (Figure A.22 ), or by inspecting the child indicators of the Productivity Stability (Figure A.23, for the Unit Test phase).



The ratio of Actual Productivity to Historical Productivity

Figure A.22: Indicator view.


ProcessPAIR v2.6
File
Report View
Table View
Indicator View
Cause-Effect View
Save Image
Copy Image
Select indicators (one or more)
Performance Indicators
Size Estimation Accuracy
Time Estimation Accuracy
Productivity
Productivity Stability
Plan Productivity Stability
Design Productivity Stability
Design Review Productivity Stability
Code Productivity Stability
Code Review Productivity Stability
Compile Productivity Stability
Unit Test Productivity Stability
Postmortem Productivity Stability
Defects Injected
Defect Removal Rate
Appraisal to Failure Ratio
Defect Density in Unit Test
Process Yield
Base Measures
Other Measures
Show Statistical Distribution Chart
Unit Test Productivity Stability by Project
1.49
4.48
5.84
9.97
7.88
yellow upper limit (3.20)
green upper limit (1.69)
optimal value (1.00)
green lower limit (0.70)
yellow lower limit (0.43)
actual values
Program 2
Program 3
Program 4
Program 5
Program 6
Project
X-Axis
Project

The ratio of Unit Test Productivity to Historical Unit Test Productivity

User statistics

min: 1.49 max: 9.97 average: 5.932 weighted: 6.572

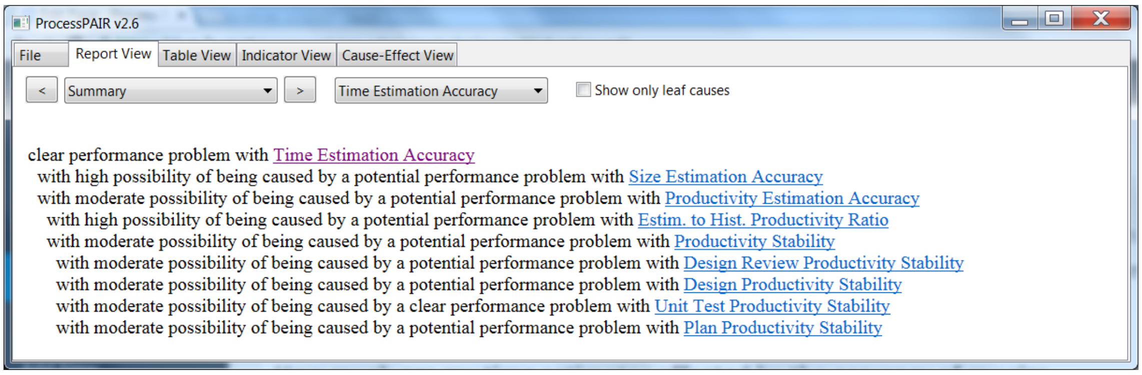
Figure A.23: Indicator view.

6. **How can I stabilize my productivity?**

As explained before, ProcessPAIR helps identifying the problematic process phases that are causing productivity instability problems. Based on that information, you are in better position to devise relevant improvement actions. In many cases, instability is caused by process changes and consequently can be addressed by repeated practice with a stable (unchanged) process.

7. **How much are my time estimates affected by the accuracy of my size estimates?**

The simplest way to answer this question is to look at the Report View, as illustrated in Figure A.24 (with "Show only leaf causes" unchecked, to visualize intermediate causes).



clear performance problem with Time Estimation Accuracy
  with high possibility of being caused by a potential performance problem with Size Estimation Accuracy
  with moderate possibility of being caused by a potential performance problem with Productivity Estimation Accuracy
    with high possibility of being caused by a potential performance problem with Estim. to Hist. Productivity Ratio
    with moderate possibility of being caused by a potential performance problem with Productivity Stability
      with moderate possibility of being caused by a potential performance problem with Design Review Productivity Stability
      with moderate possibility of being caused by a potential performance problem with Design Productivity Stability
      with moderate possibility of being caused by a clear performance problem with Unit Test Productivity Stability
      with moderate possibility of being caused by a potential performance problem with Plan Productivity Stability

Figure A.24: Report view.

The causal analysis conducted by ProcessPAIR, follows the following rational: problems with the Time Estimation Accuracy may be caused by problems with Size Estimation Accuracy or by problems with Productivity Estimation Accuracy (discrepancies between estimated and actual productivity). In the example Figure A.24, it is identified a clear performance problem with Time Estimation Accuracy, caused mainly ('high possibility') by problems with Size Estimation Accuracy, and in second place ('moderate possibility') by problems with Productivity Estimation Accuracy. Hence, we conclude that, in this example, time estimation accuracy is highly affected by the accuracy of size estimates.

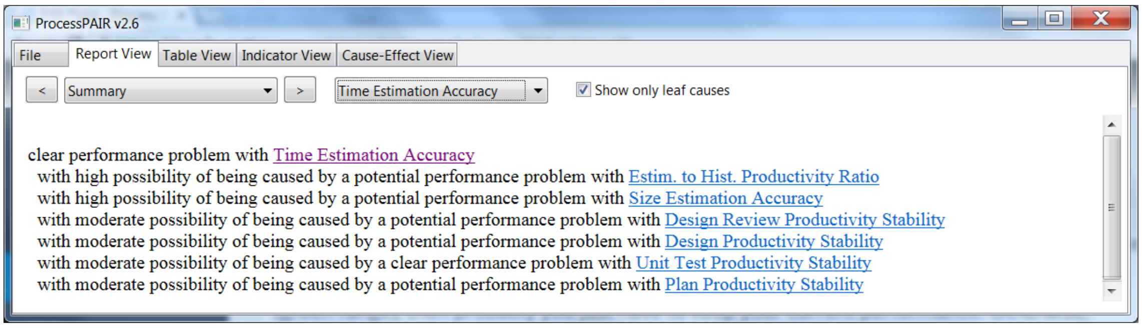
ProcessPAIR further drills down the causal analyses according to the following rational: productivity estimation problems may occur because of productivity instability problems (making historical productivity not reliable for estimation) or because historical productivity is not used in estimates. In turn, productivity stability problems may be caused by productivity instability in specific phases.

8. **Based on my historical time-estimating accuracy data, what is a realistic time-estimating goal for me?**

   ProcessPAIR lets you compare your performance to the performance achieved by other people, and hence helps establishing realistic goals. In case you already have a good performance (green range), then probably you just have to keep your current performance. Otherwise, moving to the next range may be a realistic goal (from red to yellow, or from yellow to green).

9. **How can I change my process to meet that goal?**

   The simplest way to answer this question is to first look at the causes suggested in the Report View, as illustrated in Figure A.25 ("Show only leaf causes" checked).



clear performance problem with Time Estimation Accuracy
with high possibility of being caused by a potential performance problem with Estim. to Hist. Productivity Ratio
with high possibility of being caused by a potential performance problem with Size Estimation Accuracy
with moderate possibility of being caused by a potential performance problem with Design Review Productivity Stability
with moderate possibility of being caused by a potential performance problem with Design Productivity Stability
with moderate possibility of being caused by a clear performance problem with Unit Test Productivity Stability
with moderate possibility of being caused by a potential performance problem with Plan Productivity Stability

Figure A.25: Report view.

In the example Figure A.25 , the first two causes are much more important than the other ones, so it is a good idea to focus on that causes. For addressing the first one, the historical productivity of previous projects should be used in time estimating for future projects (with the appropriate PROBE method). For addressing the second one, the relevant actions should have already been indicated in a previous section of the report.

In case you found deeper or different causes for the time estimation problems, you should devise actions for addressing them.

## A.3 Defect and yield analysis

1. **Which defect type accounts for the most time spent in compile? In test? In which phase was each type of defect injected most often?**

   In the "Indicator View", "Base Measures" group, select the "Fixtime of Defects Removed in Compile" and "Fixtime of Defects Removed in Unit Test" items, and "Defect Type" for the X-Axis, to obtain the answers to the first two questions, as illustrated in Figure A.26.

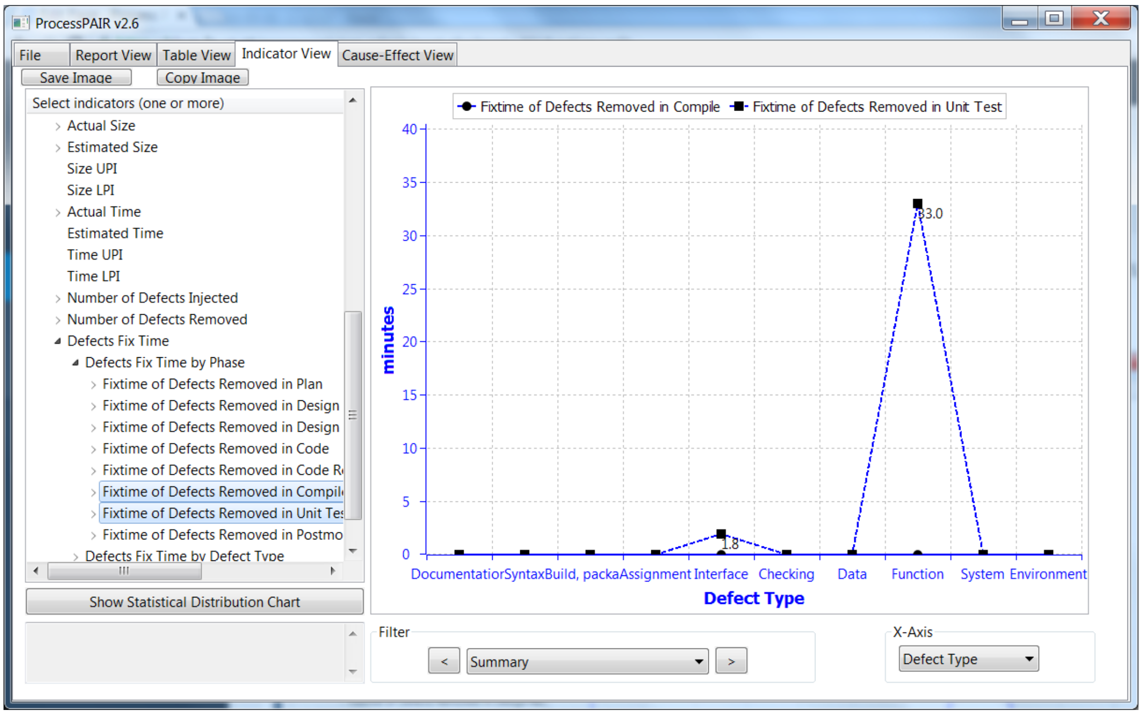

ProcessPAIR v2.6
File
Report View
Table View
Indicator View
Cause-Effect View
Save Image
Copy Image
Select indicators (one or more)
Actual Size
Estimated Size
Size UPI
Size LPI
Actual Time
Estimated Time
Time UPI
Time LPI
Number of Defects Injected
Number of Defects Removed
Defects Fix Time
Defects Fix Time by Phase
Fixtime of Defects Removed in Plan
Fixtime of Defects Removed in Design
Fixtime of Defects Removed in Design
Fixtime of Defects Removed in Code
Fixtime of Defects Removed in Code R
Fixtime of Defects Removed in Compil
Fixtime of Defects Removed in Unit Tes
Fixtime of Defects Removed in Postmo
Defects Fix Time by Defect Type
Show Statistical Distribution Chart
Fixtime of Defects Removed in Compile
Fixtime of Defects Removed in Unit Test
minutes
0
5
10
15
20
25
30
35
40
33.0
1.8
Documentation
Syntax
Build, packa
Assignment
Interface
Checking
Data
Function
System
Environment
Defect Type
Filter
<
Summary
>
X-Axis
Defect Type

Figure A.26: Indicator view.

   In this example there are no defects found in the Compile phase. In the Unit test phase, the defects of type Function take most of the time, followed by the defects of type Interface. Also notice that it is shown the average fix time for all projects.

   The phases where defects of type Function and Interface were injected most often (third question), can be seen in the Figure A.27. Those types of defects are injected mostly in the Code phase, followed by the Design and the Unit Test phases.

2. **What type of defects do I inject during design and coding?**

   Figure A.28 provides an answer in terms of number of defects. In this example, the most important type is Function, followed by Interface and Assignment.

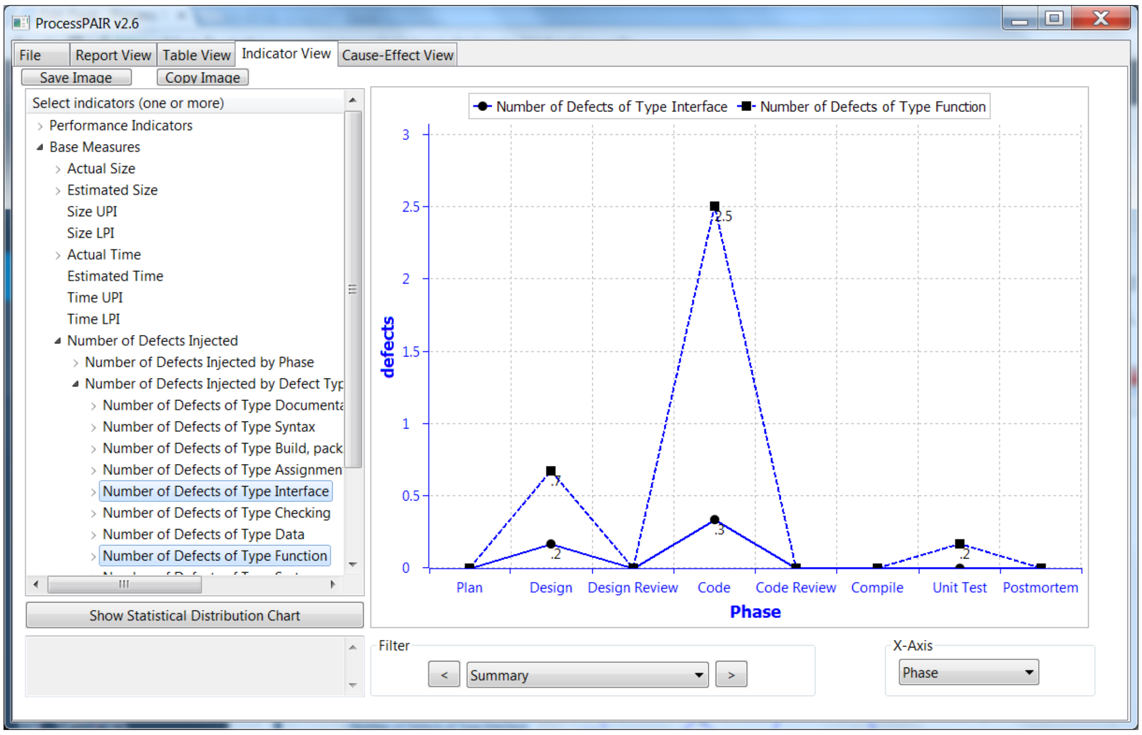

ProcessPAIR v2.6
File
Report View
Table View
Indicator View
Cause-Effect View
Save Image
Copy Image
Select indicators (one or more)
Performance Indicators
Base Measures
Actual Size
Estimated Size
Size UPI
Size LPI
Actual Time
Estimated Time
Time UPI
Time LPI
Number of Defects Injected
Number of Defects Injected by Phase
Number of Defects Injected by Defect Typ
Number of Defects of Type Documenta
Number of Defects of Type Syntax
Number of Defects of Type Build, pack
Number of Defects of Type Assignmen
Number of Defects of Type Interface
Number of Defects of Type Checking
Number of Defects of Type Data
Number of Defects of Type Function
Show Statistical Distribution Chart
Number of Defects of Type Interface
Number of Defects of Type Function
defects
Phase
Plan
Design
Design Review
Code
Code Review
Compile
Unit Test
Postmortem
Filter
Summary
X-Axis
Phase

Figure A.27: Indicator view.

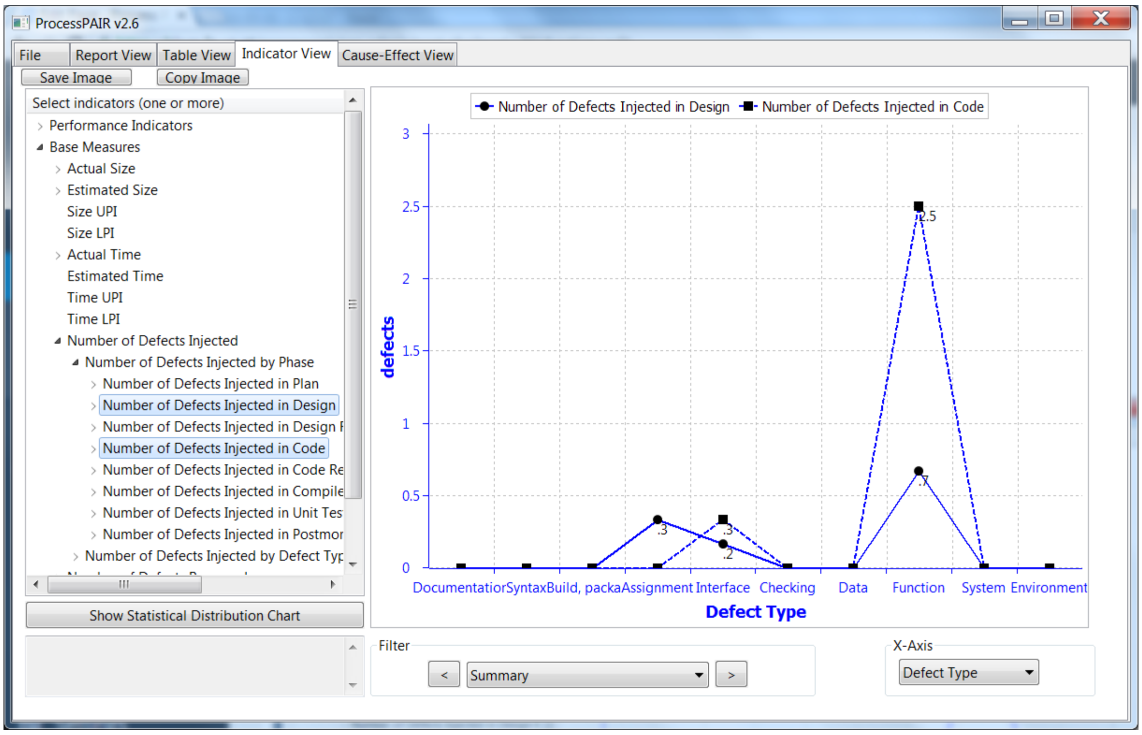
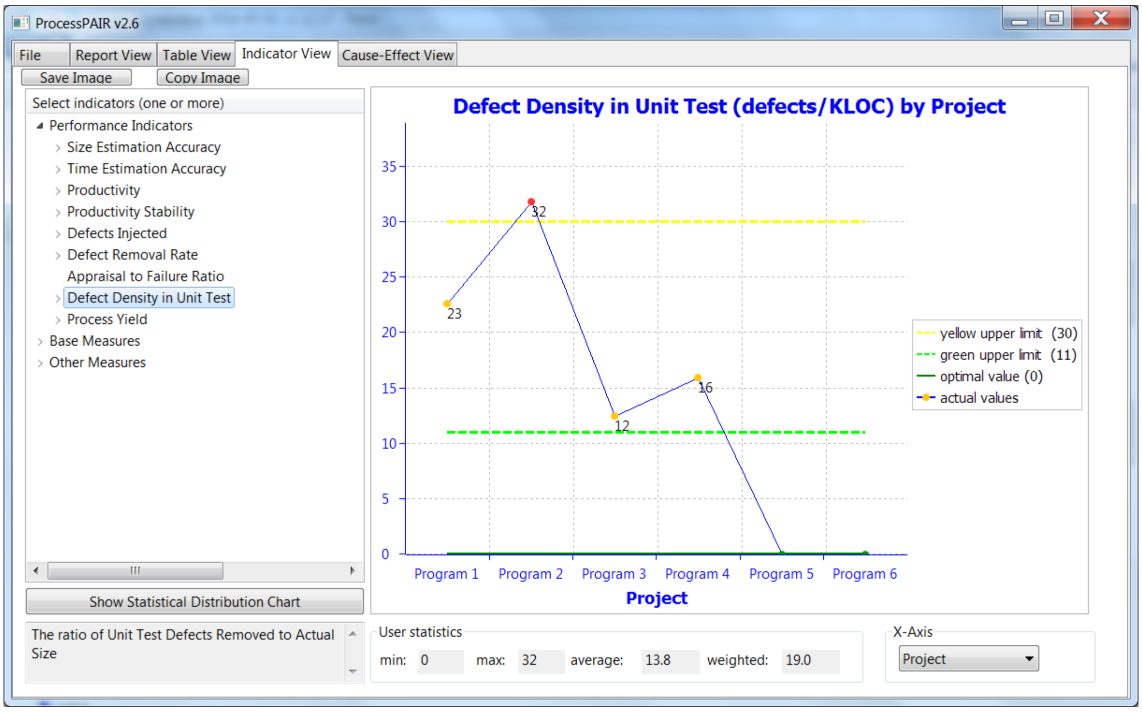

ProcessPAIR v2.6
File
Report View
Table View
Indicator View
Cause-Effect View
Save Image
Copy Image
Select indicators (one or more)
Performance Indicators
Base Measures
Actual Size
Estimated Size
Size UPI
Size LPI
Actual Time
Estimated Time
Time UPI
Time LPI
Number of Defects Injected
Number of Defects Injected by Phase
Number of Defects Injected in Plan
Number of Defects Injected in Design
Number of Defects Injected in Design F
Number of Defects Injected in Code
Number of Defects Injected in Code Re
Number of Defects Injected in Compile
Number of Defects Injected in Unit Tes
Number of Defects Injected in Postmor
Number of Defects Injected by Defect Typ
Show Statistical Distribution Chart
Number of Defects Injected in Design
Number of Defects Injected in Code
defects
Defect Type
Documentation
Syntax
Build, packa
Assignment
Interface
Checking
Data
Function
System
Environment
Filter
Summary
X-Axis
Defect Type

Figure A.28: Indicator view.

3. **What trends are apparent in defects per size unit (e.g., KLOC) found in reviews, compile, and test?**

Figures A.29 A.30 show the relevant information. In this example there is no Compile phase.

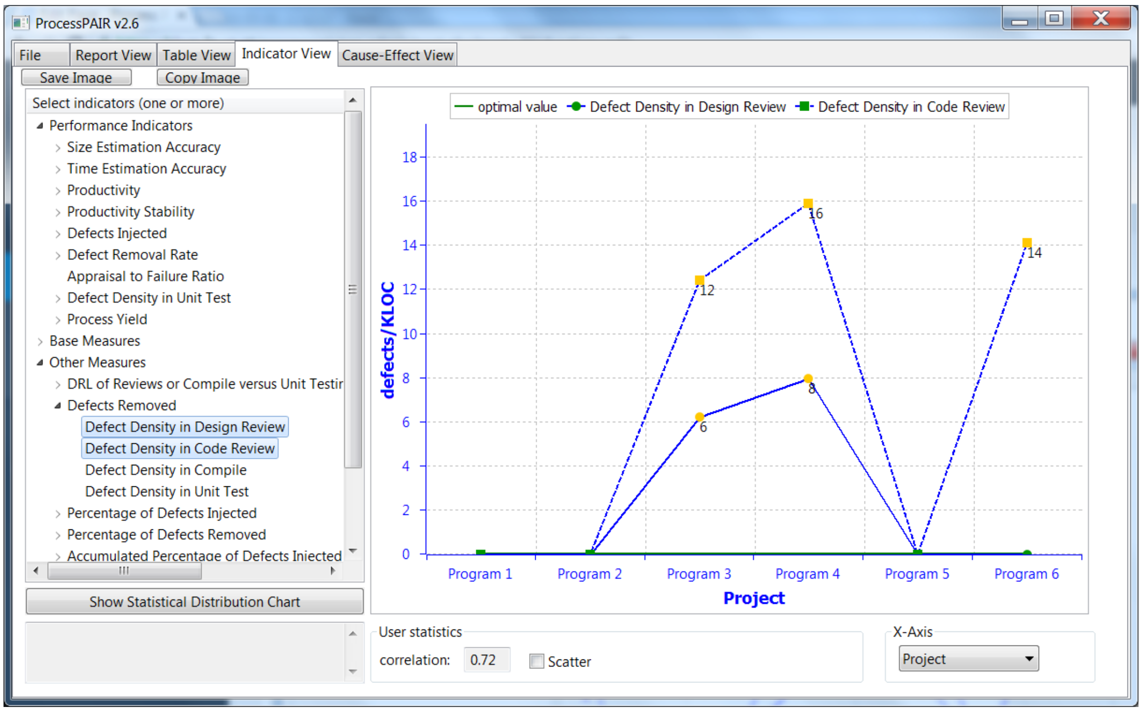


The ratio of Unit Test Defects Removed to Actual Size

User statistics

min: 0 max: 32 average: 13.8 weighted: 19.0

Figure A.29: Indicator view.


ProcessPAIR v2.6
File
Report View
Table View
Indicator View
Cause-Effect View
Save Image
Copy Image
Select indicators (one or more)
Performance Indicators
Size Estimation Accuracy
Time Estimation Accuracy
Productivity
Productivity Stability
Defects Injected
Defect Removal Rate
Appraisal to Failure Ratio
Defect Density in Unit Test
Process Yield
Base Measures
Other Measures
DRL of Reviews or Compile versus Unit Testir
Defects Removed
Defect Density in Design Review
Defect Density in Code Review
Defect Density in Compile
Defect Density in Unit Test
Percentage of Defects Injected
Percentage of Defects Removed
Accumulated Percentage of Defects Injected
Show Statistical Distribution Chart
optimal value
Defect Density in Design Review
Defect Density in Code Review
defects/KLOC
Program 1
Program 2
Program 3
Program 4
Program 5
Program 6
Project
X-Axis
Project

User statistics

correlation: 0.72 Scatter

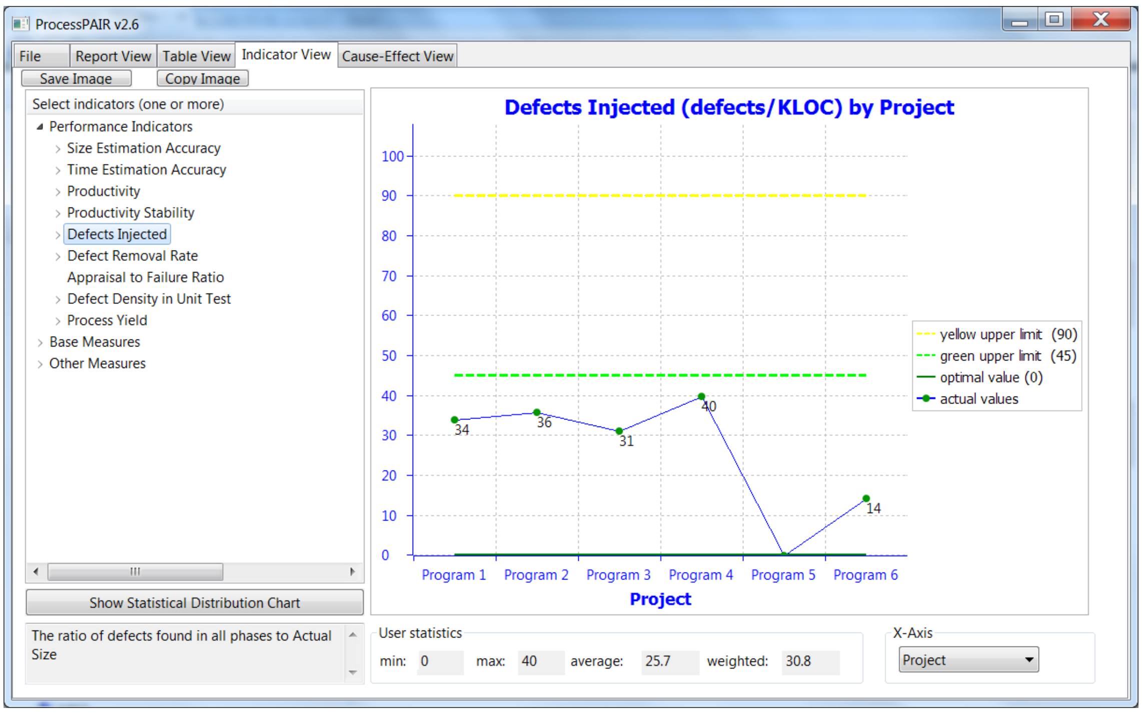
Figure A.30: Indicator view.

In this example (Figure A.29), there is a trend for decreasing the defects found in unit test per size unit, with 0 defects in the last two projects. This may be caused by defects being found in earlier phases or because of fewer defects being injected. Usually, fewer defects found in unit test also means that fewer defects remain in the delivered program, so the delivered program is of higher quality.

Regarding defects per size unit found in code and design reviews, there is no clear trend (see Figure A.30).

4. **What trends are apparent in total defects per size unit?**

   The relevant chart is shown in Figure A.30. In this example there is a trend for decreasing in the last two projects. Anyways, the total defects per size unit are within the green region (less than 45 defects per KLOC) in all projects. So, the performance is good and improving.



The ratio of defects found in all phases to Actual Size

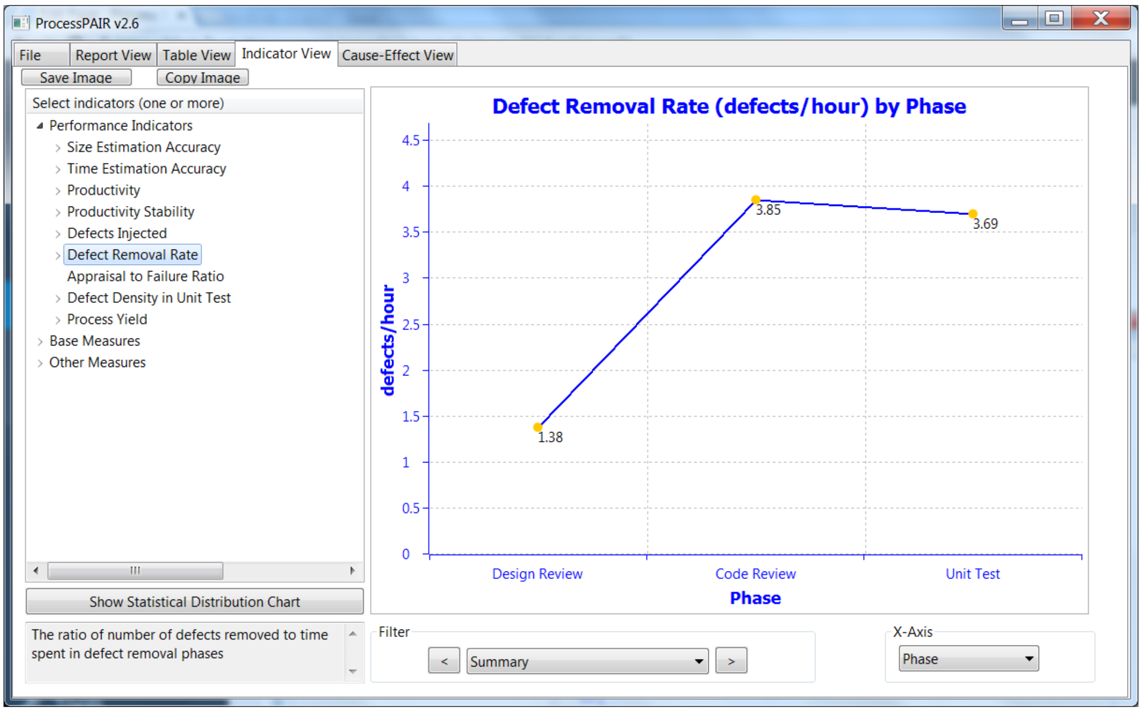
User statistics
min: 0 max: 40 average: 25.7 weighted: 30.8

Figure A.31: Indicator view.

5. **How do my defect removal rates (defects removed/hour) compare for design review, code review, compile, and test?**

   The relevant chart is shown in Figure A.32. In this example there is no Compile phase. The Code Review phase (with 3.85 defects found per hour on average) is slightly more efficient in finding defects than the Unit Test phase (with 3.69 defects found per hour on average). The Design Review phase is less efficient (with 1.38 defects found per hour).

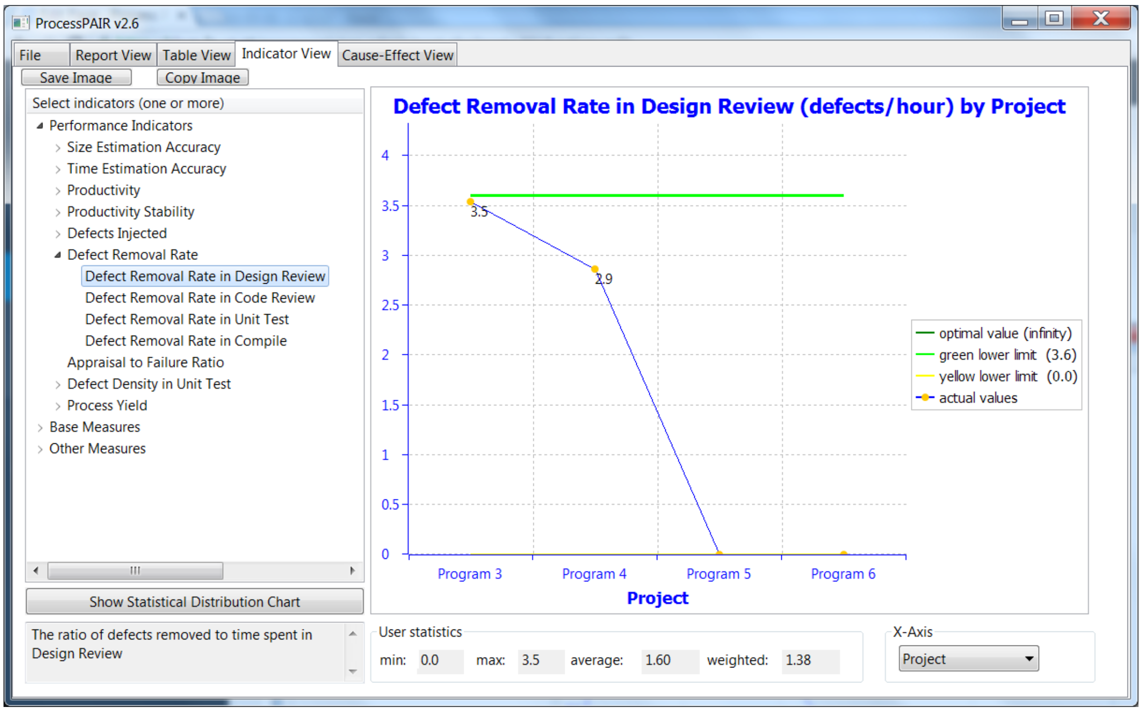
   The Figure A.33, A.34 and A.35 allow a better comparison with benchmarks. As indicated in the figures, a good performance for the defect removal rate is a value equal or greater than 8.0 defects/hour in Code Review and a value equal or greater than 4.9 defects/hour in Unit Test (less efficient than Code Review). In this example, compared to the benchmarks, the weighted average performance in Code Review (3.85 compared to 8.0) is worse than the weighted average



The ratio of number of defects removed to time spent in defect removal phases

Figure A.32: Indicator view.

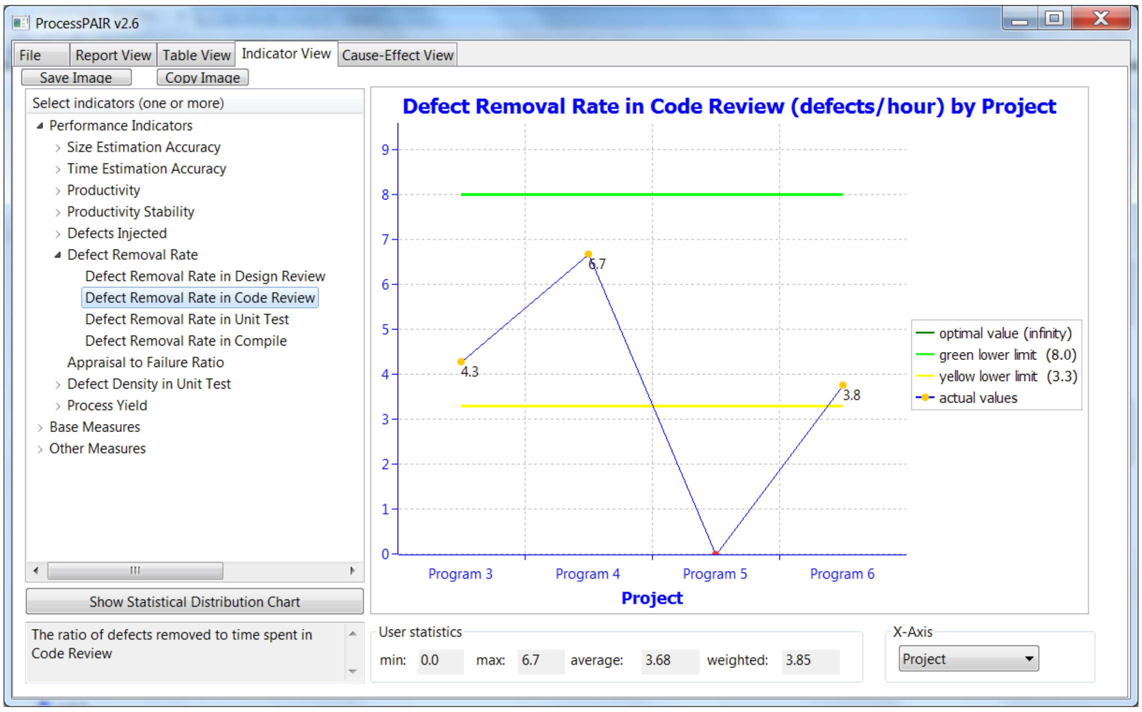
performance in Unit Test (3.69 compared to 4.9). So there is much more room for improvement in the Code Review phase.



The ratio of defects removed to time spent in Design Review

User statistics
min: 0.0 max: 3.5 average: 1.60 weighted: 1.38

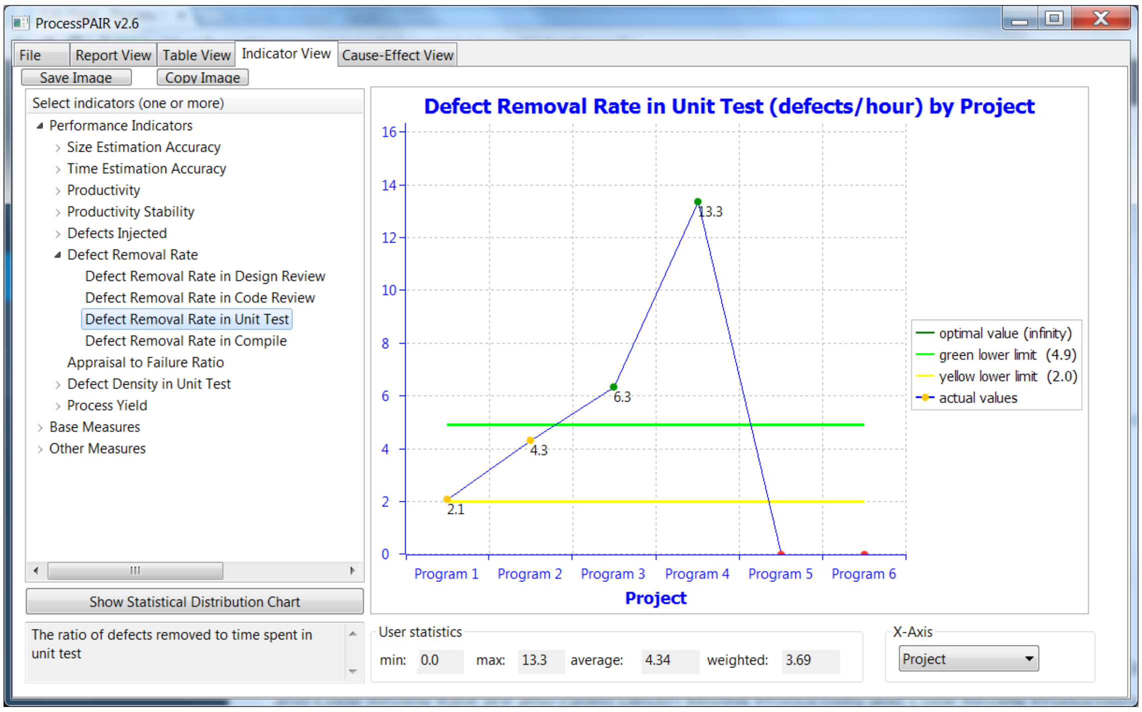
Figure A.33: Indicator view.



The ratio of defects removed to time spent in Code Review

User statistics
min: 0.0 max: 6.7 average: 3.68 weighted: 3.85

Figure A.34: Indicator view.



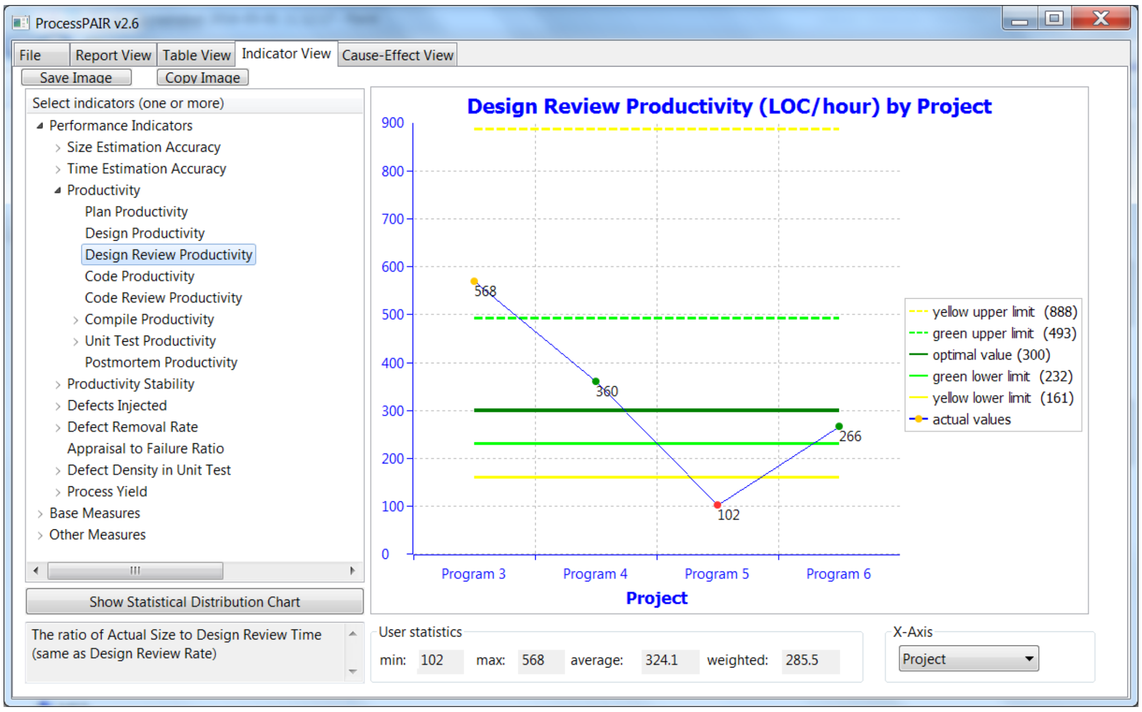
The ratio of defects removed to time spent in unit test

User statistics
min: 0.0 max: 13.3 average: 4.34 weighted: 3.69

Figure A.35: Indicator view.

6. **What are my review rates (size reviewed/hour) for design review and code review?**

   The relevant Figures are A.36 and A.37. In ProcessPAIR the Design Review Rate and Code Review Rate are also called Design Review Productivity and Code Review Productivity, respectively. In this example, the average Design Review Rate is 324 LOC/hour, which is within the green region. The average Code Review Rate is 310 LOC/hour, which is also within the green region. In both cases there is a trend towards getting close to the optimal value.

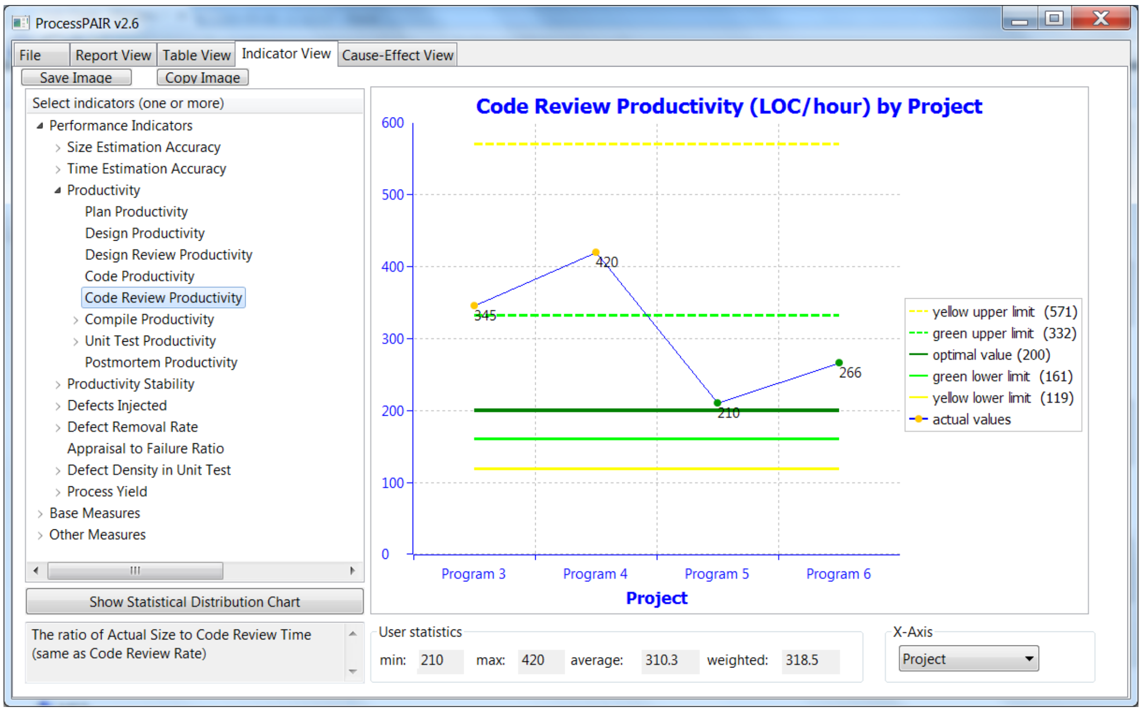


The ratio of Actual Size to Design Review Time (same as Design Review Rate)

User statistics

min: 102 max: 568 average: 324.1 weighted: 285.5

Figure A.36: Indicator view.



The ratio of Actual Size to Code Review Time (same as Code Review Rate)

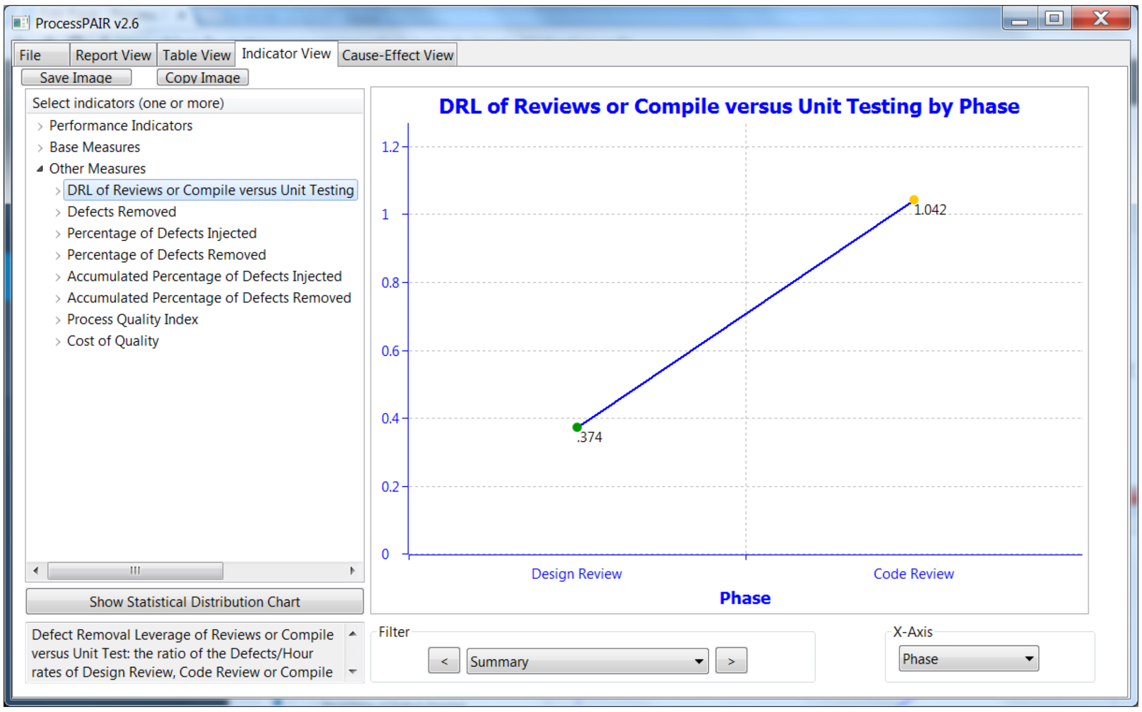
User statistics

min: 210 max: 420 average: 310.3 weighted: 318.5

Figure A.37: Indicator view.

7. **What are my defect-removal leverages for design review, code review, and compile versus unit test?**

   The relevant values are shown in the Figure A.38. In this example there is no Compile phase. On average, the defect removal leverage is 0.374 for design review versus unit test and 1.042 for code review versus unit test. The interpretation was already done in a previous question.


ProcessPAIR v2.6
File
Report View
Table View
Indicator View
Cause-Effect View
Save Image
Copy Image
Select indicators (one or more)
Performance Indicators
Base Measures
Other Measures
DRL of Reviews or Compile versus Unit Testing
Defects Removed
Percentage of Defects Injected
Percentage of Defects Removed
Accumulated Percentage of Defects Injected
Accumulated Percentage of Defects Removed
Process Quality Index
Cost of Quality
Show Statistical Distribution Chart
DRL of Reviews or Compile versus Unit Testing by Phase
1.2
1
0.8
0.6
0.4
0.2
0
1.042
.374
Design Review
Code Review
Phase
Filter
Summary
X-Axis
Phase

Defect Removal Leverage of Reviews or Compile versus Unit Test: the ratio of the Defects/Hour rates of Design Review, Code Review or Compile

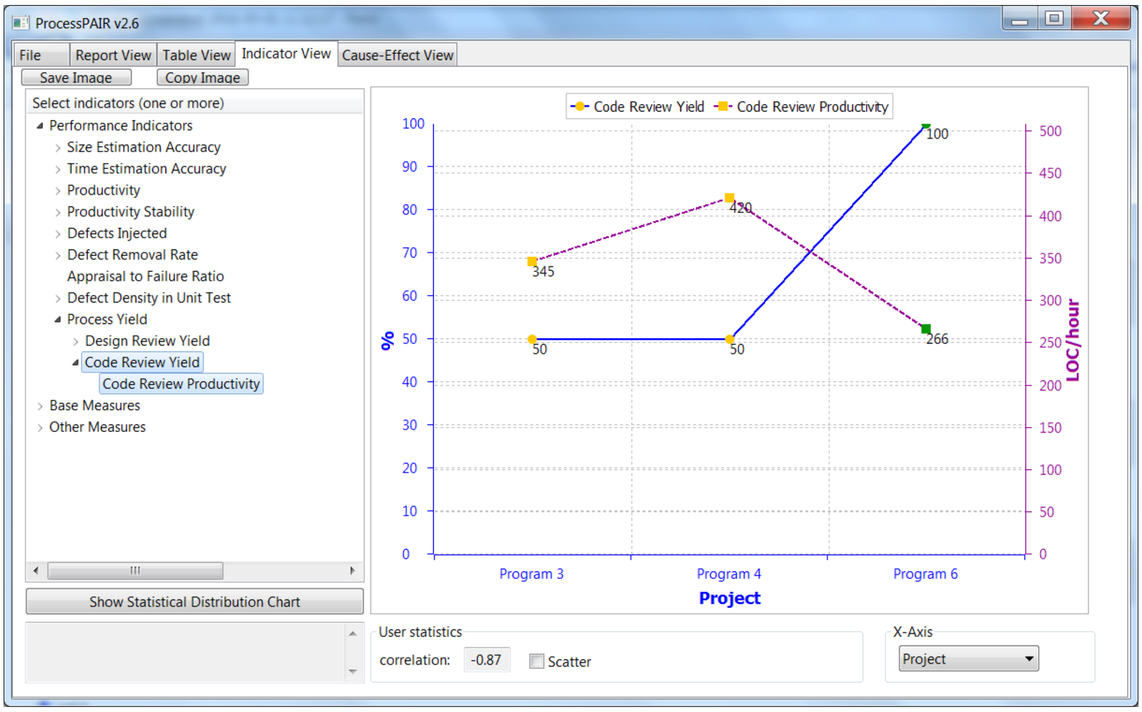
Figure A.38: Indicator view.

8. **Is there any relationship between yield and review rate (size reviewed/hour) for design and code reviews?**

   The relevant information is shown in Figure A.39 and A.40 . The review yield is undefined when there are 0 defects entering the review phase. In this example, that happens with program 5 (for code and design reviews) and program 4 (for design review only). Regarding the Code Review Yield versus the Code Review Rate, there is a negative correlation (-0.87) as expected (because slower reviews are usually associated with higher yields). However, since there are only 3 data points, the correlation is not statistically significant. As for the Design Review Yield versus Design Review Rate, the trend is different from expected (a slower review rate is associated with a smaller yield), but with only two data points the correlation coefficient is meaningless (always +1 or -1).

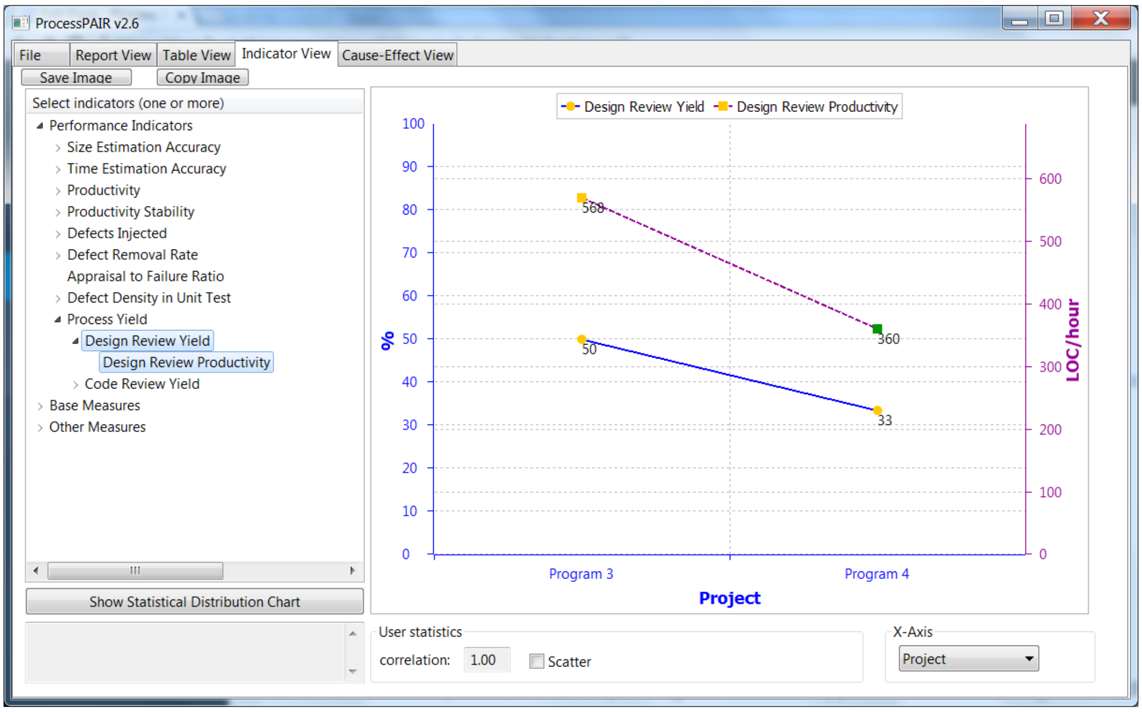
9. **Is there a relationship between yield and A/FR?**

   The relevant information is shown in Figure A.41. In this example there is a very good positive correlation (0.95) between the Process Yield and the Appraisal to Failure Ratio (A/FR). The correlation is also statistically significant (for a 5% confidence level, computed with a web



User statistics
correlation: -0.87
Scatter
X-Axis
Project

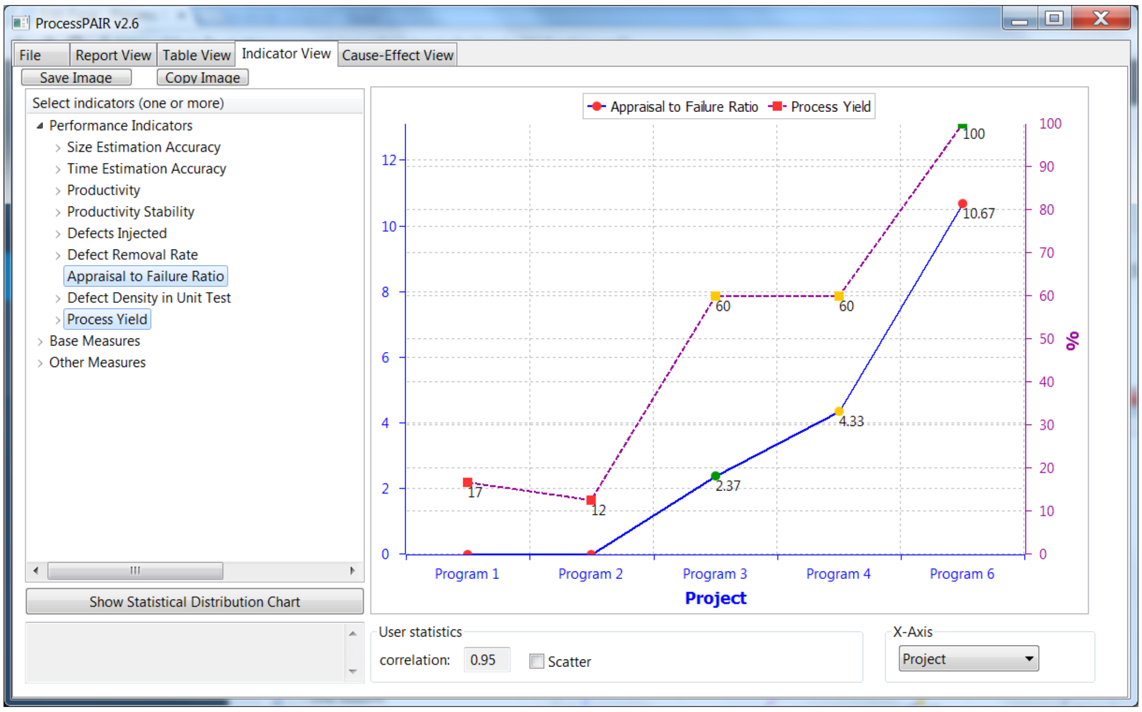
Figure A.39: Indicator view.



User statistics
correlation: 1.00
Scatter
X-Axis
Project

Figure A.40: Indicator view.

calculator). This correlation is as expected, because spending more time in 'appraisal' activities (design and code reviews) is usually associated with a higher process yield (percentage of defects found before compile and test).



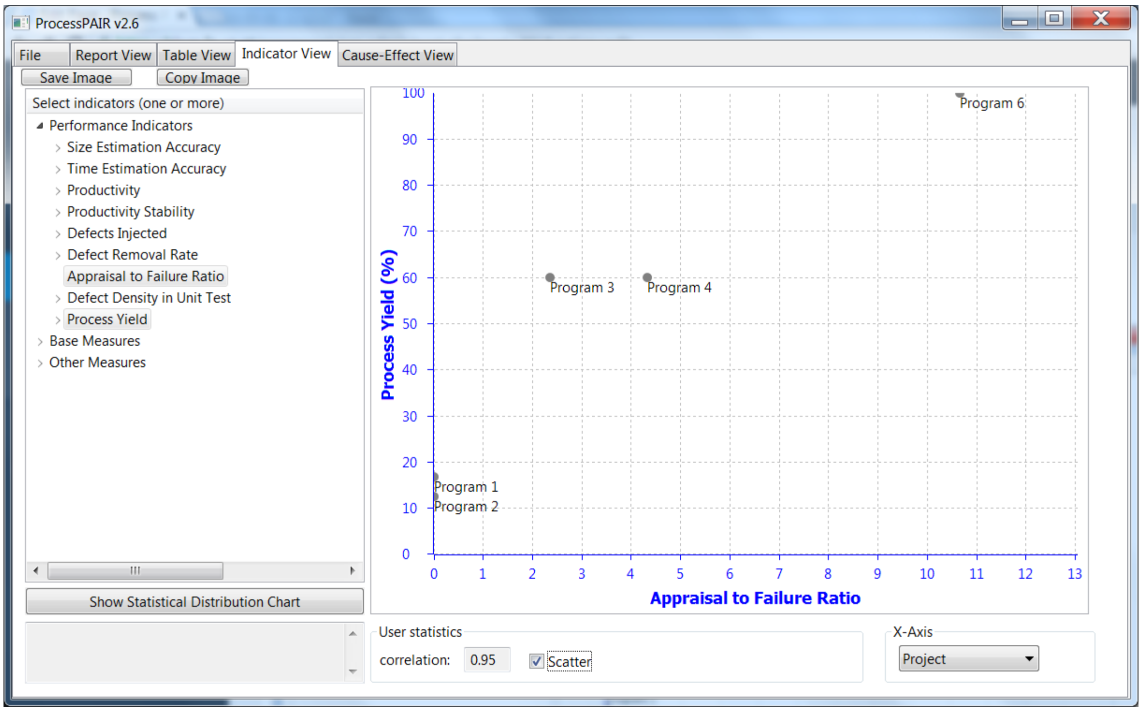
User statistics

correlation: 0.95 ☐ Scatter

X-Axis Project

Figure A.41: Indicator view.

You can also see the same information in Figure  A.42 (Scatter check box selected).



User statistics

correlation: 0.95 ☑ Scatter

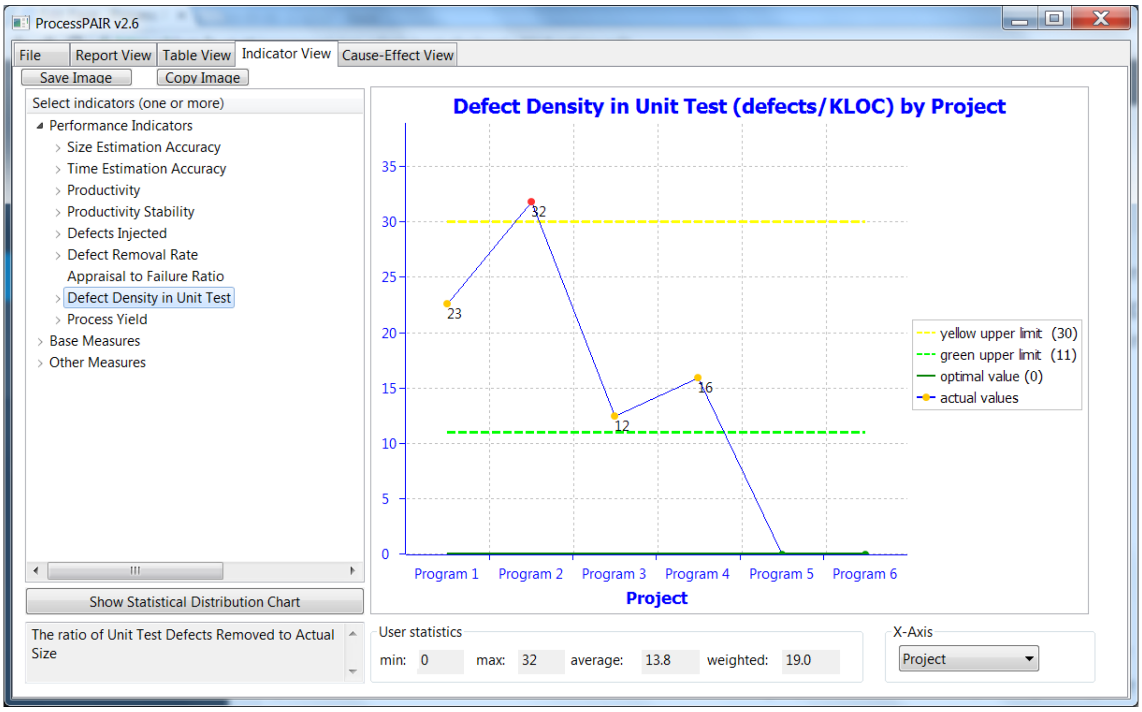
X-Axis Project

Figure A.42: Indicator view.

## A.4 Quality analysis

1. **How much did the quality of the programs entering unit test change? Why?**

   In PSP, the quality of programs entering unit test is usually assessed based on the defects per size unit found in unit test, so the relevant information is shown in Figure A.43. As already analysed before, in this example there is a trend for decreasing the defects found in unit test per size unit, starting with an average of 27 defects/KLOC in the first two programs, an average of 14 defects/KLOC in programs 3 and 4, and finally 0 defects/KLOC in the last two projects, which is a very good improvement trend.



The ratio of Unit Test Defects Removed to Actual Size

User statistics

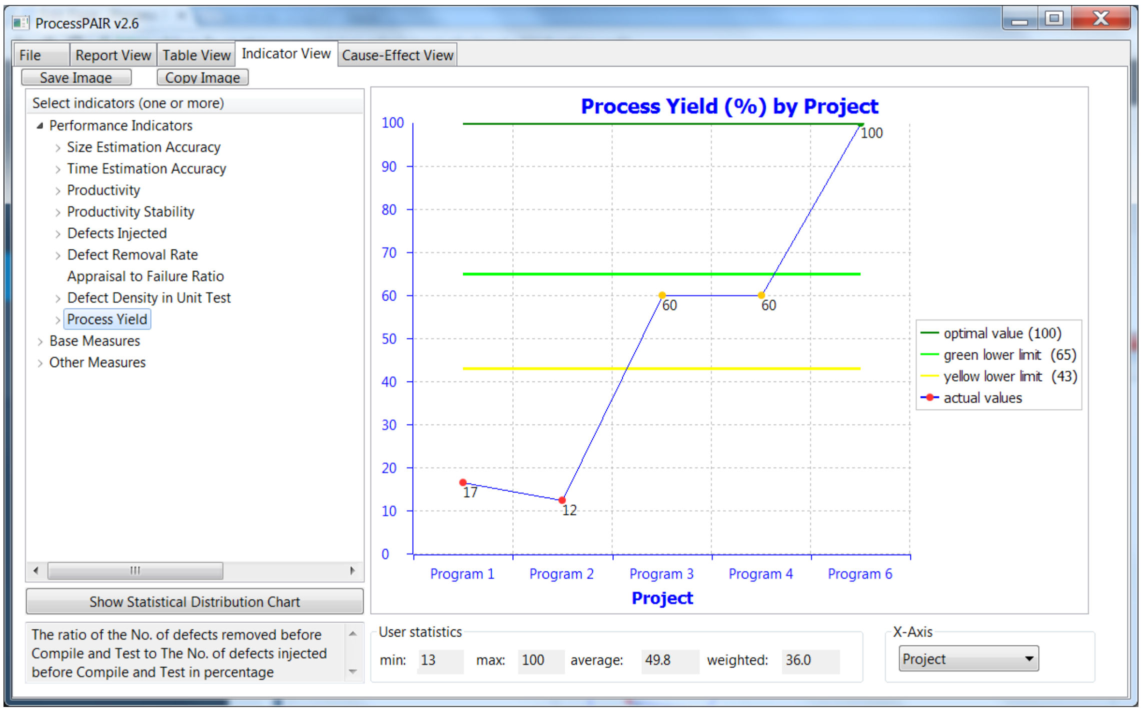
min: 0 max: 32 average: 13.8 weighted: 19.0

Figure A.43: Indicator view.

2. **Am I finding my defects in design and code reviews? Why or why not?**

   The relevant information is shown in Figure A.44 and for a more detailed information, the design review and code review yields, as shown in Figure A.45. The yield is not defined when there are no defects entering a phase. In this example, regarding the process yield, there is a very good evolution from very small values in the first two programs (without review phases) to 60% in programs 3 and 4 (when reviews are introduced) and finally 100% in program 6 (in program 5 the yield is undefined because of no defects recorded). The Code Review Yield follows a similar trend. Regarding the Design Review Yield, it is undefined in programs 5 and 6, and the values observed in the previous two programs still have room for improvement (namely to reach the 78% boundary).

3. **Based on my historical data, what are some realistic quality goals for me?**

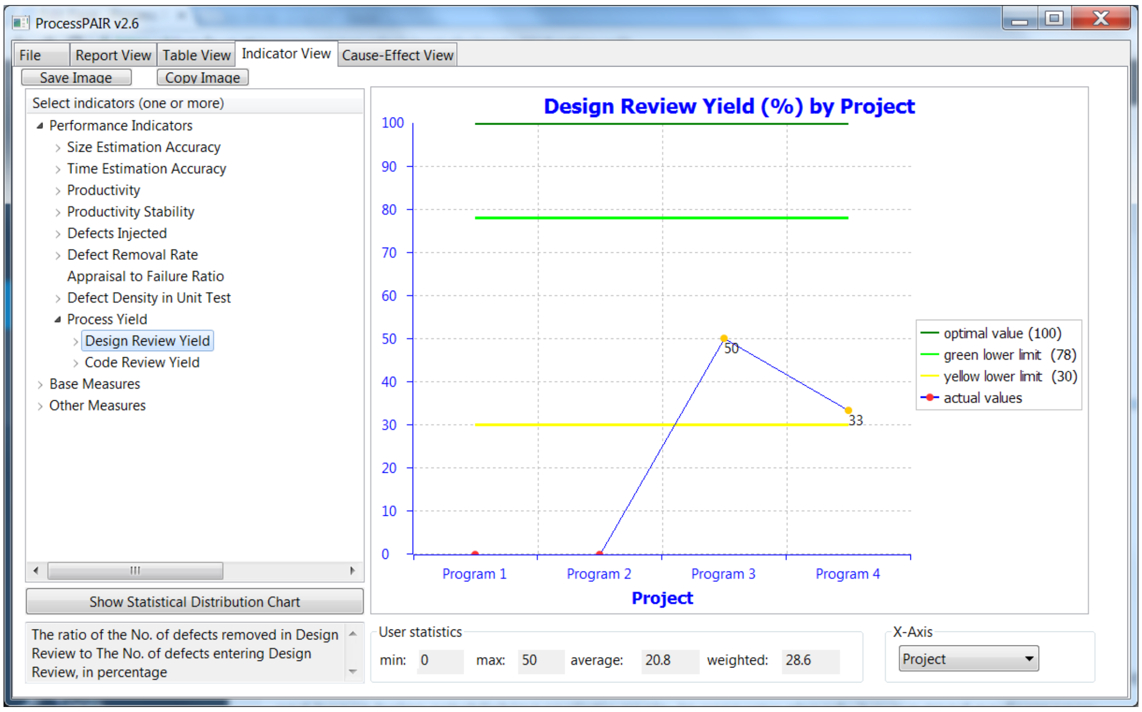


The ratio of the No. of defects removed before Compile and Test to The No. of defects injected before Compile and Test in percentage

User statistics

min: 13 max: 100 average: 49.8 weighted: 36.0

Figure A.44: Indicator view.



The ratio of the No. of defects removed in Design Review to The No. of defects entering Design Review, in percentage

User statistics

min: 0 max: 50 average: 20.8 weighted: 28.6

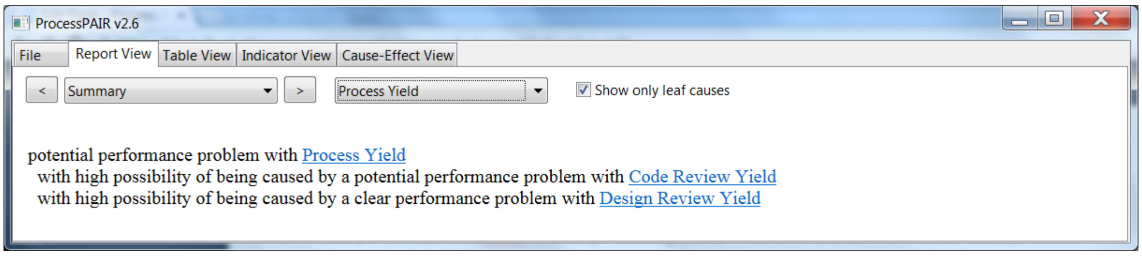
Figure A.45: Indicator view.

ProcessPAIR lets you compare your performance to the performance achieved by other people, and hence helps establishing realistic goals. In case you already have a good performance (green range), then probably you just have to keep your current performance. Otherwise, moving to the next range may be a realistic goal (from red to yellow, or from yellow to green).

In this example, it seems realistic to be in the green region for the defects found per size unit in unit test (<= 11 defects/KLOC) and process yield (>= 65% in the benchmarks used).

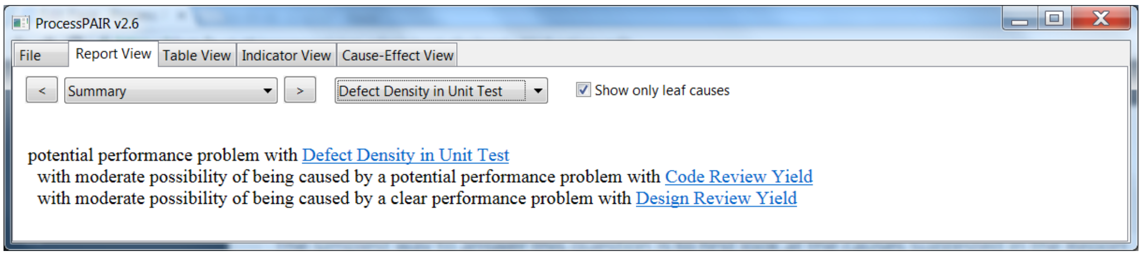
4. **How can I change my process to meet those goals?**

The simplest way to answer this question is to first look at the causes suggested in the Report View, as illustrated in Figure A.46 and A.47 ("Show only leaf causes" checked).



potential performance problem with Process Yield
with high possibility of being caused by a potential performance problem with Code Review Yield
with high possibility of being caused by a clear performance problem with Design Review Yield

Figure A.46: Report view.



potential performance problem with Defect Density in Unit Test
with moderate possibility of being caused by a potential performance problem with Code Review Yield
with moderate possibility of being caused by a clear performance problem with Design Review Yield

Figure A.47: Report view.

In this example, it is indicated a potential performance problem with both the process yield and the defect density in unit test, having in both cases the same root causes: potential performance problem with the Code Review Yield and clear performance problem with the Design Review Yield.

Although the Code Review Yield is better than the the Design Review Yield, the first priority indicated by ProcessPAIR (based on a cost-benefit estimate) is to improve the Code Review Yield. Since the Code Review Rate is already within the recommended range, the improvement actions should be based on other review best practices (e.g., improving review checklists to focus on the types of defects that escaped from code reviews).

In case you found deeper or different causes for the problems identified, you should devise actions for addressing them.

# References

Charu C. Aggarwal. *Data Classification: Algorithms and Applications*. Chapman & Hall/CRC, 1st edition, 2014. ISBN 1466586745, 9781466586741.

Lukas Alperowitz, Dora Dzvonyar, and Bernd Bruegge. Metrics in agile project courses. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 323–326. ACM, 2016.

Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

Tiago Miguel Laureano Alves. Benchmark-based software product quality evaluation. 2012.

AssignmentKit. Assignment kit for final report, personal software process for engineers: Part ii, PSP academic material 4-1, 2006. URL `http://www.sei.cmu.edu/tsp/tools/academic/index.cfm`.

Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

Barry Boehm. Some future software engineering opportunities and challenges. In *The Future of Software Engineering*, pages 1–32. Springer, 2011.

L. Breiman. *Classification and Regression Trees*. The Wadsworth statistics/probability series. Wadsworth International Group, 1984. ISBN 9780534980542. URL `https://books.google.pt/books?id=mlZgQgAACAAJ`.

J Campos. Risk management and failure mode and effect analysis for product development. *Rapid Innovation LLC*, 2012.

David N Card. Defect-causal analysis drives down error rates. *IEEE Software*, 10(4):98–99, 1993.

David N Card. Defect analysis: Basic techniques for management and learning. *Advances in Computers*, 65:259–295, 2005.

Samprit Chatterjee and Ali S Hadi. *Regression analysis by example*. John Wiley & Sons, 2015.

Mary Beth Chrissis, Mike Konrad, and Sandra Shrum. *CMMI for development: guidelines for process integration and product improvement*. Pearson Education, 2011.

Watts S. Humphrey Daniel Burton. Mining PSP Data. 2006.

Process Dashboard. Tuma Solutions LLC, Process Dashboard. URL `http://www.processdash.com/`.

T. Daughtrey. *Fundamental Concepts for the Software Quality Engineer*. ASQ Quality Press, 2002. ISBN 9780873895217. URL `https://books.google.pt/books?id=dFmXBaLy0YMC`.

Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

Jerome H Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.

John C Gower. A general coefficient of similarity and some of its properties. *Biometrics*, pages 857–871, 1971.

Hackystat. Hackystat Development Team, Hackystat. URL `https://code.google.com/archive/p/hackystat/`.

DM Hamby. A review of techniques for parameter sensitivity analysis of environmental models. *Environmental monitoring and assessment*, 32(2):135–154, 1994.

H James Harrington et al. *Business process improvement: The breakthrough strategy for total quality, productivity, and competitiveness*, volume 1. McGraw-Hill New York, NY, 1991.

Watts Humphrey and James Over. *Leadership, Teamwork, and Trust: Building a Competitive Software Capability*. Addison-Wesley Professional, 2010.

Watts S Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley Professional, 1997.

Watts S Humphrey. Team Software Process (TSP). *Encyclopedia of Software Engineering*, 2000.

Watts S Humphrey. *PSP (sm): a self-improvement process for software engineers*. Addison-Wesley Professional, 2005.

C. Jones. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. McGraw-Hill Education, 2009. ISBN 9780071621625. URL `http://books.google.pt/books?id=CJd__8ANvtQC`.

Capers Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc., 2000.

Marcos Kalinowski, David N Card, and Guilherme H Travassos. Evidence-based guidelines to defect causal analysis. *IEEE Software*, 29(4):16–18, 2012.

Chris F Kemerer and Mark C Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *Software Engineering, IEEE Transactions on*, 35(4): 534–550, 2009.

Ron S Kenett and Emanuel Baker. *Software process quality: management and control*. CRC Press, 1999.

Max Kuhn and Kjell Johnson. *Applied predictive modeling*. Springer, 2013.

Young Hoon Kwak and Frank T Anbari. Benefits, obstacles, and future of six sigma approach. *Technovation*, 26(5):708–715, 2006.

Steve McConnell. *Software estimation: demystifying the black art*. Microsoft press, 2006.

Mushtaq and João Pascoal Faria. A Model for Analyzing Estimation, Productivity, and Quality Performance in the Personal Software Process. In *Proceedings of the 2014 International Conference on Software and System Process*, ICSSP 2014, pages 10–19, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2754-1. doi: 10.1145/2600821.2600828. URL `http://doi.acm.org/10.1145/2600821.2600828`.

MHNM Nasir and Azwina M Yusof. Automating a modified personal software process. *Malaysian Journal of Computer Science*, 18(2):11–27, 2005.

W Navidi. Statistics for scientist and engineers, 2011.

William Cyrus Navidi. *Statistics for engineers and scientists*. McGraw-Hill Higher Education, 2008.

Lutz Prechelt and Barbara Unger. *A controlled experiment on the effects of PSP training: Detailed description and evaluation*. Univ., Fak. für Informatik, 1999.

PSPWorkBook. Software Engineering Institute (SEI) customer relations, self-study PSP material. URL `http://www.sei.cmu.edu/tsp/tools/student/`.

Mushtaq Raza, João Pascoal Faria, and Rafael Salazar. Empirical Evaluation of the Process-PAIR Tool for Automated Performance Analysis. In *The 28th International Conference on Software Engineering and Knowledge Engineering*, SEKE 2016, Research Inc. and Knowledge Systems Institute Graduate School, 2016. KSI. ISBN 1-891706-39-X. URL `http://dblp.uni-trier.de/rec/bib/conf/seke/2016`.

Dieter Rombach, Jürgen Münch, Alexis Ocampo, Watts S Humphrey, and Dan Burton. Teaching disciplined software development. *Journal of Systems and Software*, 81(5):747–763, 2008.

Guoping Rong, He Zhang, Shan Qi, and Dong Shao. Can software engineering students program defect-free?: an educational approach. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 364–373. ACM, 2016.

Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

George AF Seber and CJ Wild. Nonlinear regression, 1989.

Cosma Shalizi. Classification and regression trees. *Statistics*, pages 36–350, 2009.

Wen-Hsiang Shen, Nien-Lin Hsueh, and Wei-Mann Lee. Assessing PSP effect in training disciplined software development: A plan–track–review model. *Information and Software Technology*, 53(2):137–148, 2011.

Hyunil Shin, Ho-Jin Choi, and Jongmoon Baik. Jasmine: a PSP supporting tool. In *International Conference on Software Process*, pages 73–83. Springer, 2007.

Raymund Sison. Personal software process (PSP) assistant. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 8–pp. IEEE, 2005.

L.M. Surhone, M.T. Tennoe, and S.F. Henssonow. *Ishikawa Diagram*. Betascript Publishing, 2010. ISBN 9786131409950. URL `https://books.google.pt/books?id=hqotkgAACAAJ`.

Shurei Tamura. Integrating CMMI and TSP/PSP: Using TSP data to create Process Performance Models. Technical report, DTIC Document, 2009.

Supachai Thisuk and Sakgasit Ramingwong. Wbps: A new web based tool for personal software process. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2014 11th International Conference on*, pages 1–6. IEEE, 2014.

UML2.5. Unified Modeling Language$^{TM}$, 2015. URL `http://www.omg.org/spec/UML/2.5/`.

Stefan Wagner, Melanie Ruhe, and AG Siemens. A systematic review of productivity factors in software development. *language*, 1989, 1980.

WebProcessPAIR. Webprocesspair: Recommendation system of improvement actions, 2016. URL `https://repositorio-aberto.up.pt/handle/10216/85826`.

Dave Zubrow, Bob Stoddard, Rawdon Young, and Kevin Schaaf. A practical approach for building CMMI process performance models. In *SEPG North America Conference*, 2009.