A Model for Analyzing Estimation, Productivity, and Quality Performance in the Personal Software Process

Mushtaq Raza
INESC TEC/Department of Informatics Engineering,
Faculty of Engineering, University of Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
+351920055092
uomian49@yahoo.com

João Pascoal Faria
INESC TEC/Department of Informatics Engineering,
Faculty of Engineering, University of Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
+3512250814 00
jpf@fe.up.pt

ABSTRACT

High-maturity software development processes, making 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. However, there is a lack of tool support for automating that type of analysis, and hence diminish the manual effort and expert knowledge required. So, we propose in this paper a comprehensive performance model, addressing time estimation accuracy, quality and productivity, to enable the automated (tool based) analysis 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.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *software process models, software quality assurance, time estimation.*

General Terms

Management, Measurement, Performance.

Keywords

Personal Software Process; Performance Analysis; Performance Model.

1. INTRODUCTION

Currently, according to [1], the top two software engineering challenges are (1) the increasing emphasis on rapid development and adaptability, and (2) the increasing software criticality and need for assurance. The Team Software Process (TSP) and the accompanying Personal Software Process (PSP) are examples of methodologies that can help individuals and teams improve their performance and produce virtually defect free software on time and budget [2][3][4], addressing current software development 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 management, quality management and process improvement [5].

Software development 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 [6]. Although several tools exist to automate data collection and produce performance charts, tables and reports for manual analysis of TSP/PSP data [7][8][9][10], practically no tool support exists for automating the performance analysis. There are also some studies that show cause-effect relationships among performance indicators [11][12], but no automated root cause analysis is proposed. The manual analysis of performance data for determining root causes of performance problems and devising improvement actions is problematic because of the potentially large amount of data to analyze [6] and the effort and expert knowledge required to do the analysis.

To address those shortcomings, we are developing models and tools to automate the analysis of performance data produced in the context of the TSP/PSP and other high maturity processes, namely, identify performance problems and their root causes and recommend improvement actions. In previous work [13][14] we developed a performance model and a tool to automate the analysis of time estimation performance of PSP developers. The model was validated based on a small data set from our institution. More recently [15][16], we investigated, based on existing PSP data, the factors that affect the productivity of PSP developers. The current paper goal is to present a comprehensive performance model, covering the estimation, quality and productivity aspects, calibrated and validated based on a large PSP data set, to enable the automated (tool based) performance analysis of individual development work with the PSP.

The rest of the paper is organized as follows. Section 2 describes our overall approach. Sections 3 and 4 present the performance model conceived based on literature, expert knowledge and existing PSP data. Section 5 presents an analysis of existing PSP data to validate assumptions in the model. Section 6 presents a case study to illustrate the application of the model and compare the results of model-based and manual analysis. Section 7 presents the conclusions. The appendices contain support tables and charts.

2. OVERALL APPROACH

An overview of the artifacts and steps involved in our approach for automated performance analysis is shown in Fig. 1.

In order to enable the automated identification of performance problems (first part of activity B1), one has to first decide on the relevant performance indicators (PIs) (activity A1) and thresholds (activity A3). In order to enable the automated identification of root causes of performance problems (second part of activity B1), one has to first decide on the relevant cause-effect relationships (activity A2). Together, the performance indicators, performance ranges and cause-effect relationships constitute the performance model. The scope of this paper is the development of such performance model, i.e., activities A1 to A3.

After identifying performance problems and root causes, the next step is to recommend improvement actions to address those causes (activity B2). To enable the automated recommendation of such actions, a catalogue of possible improvement actions has to be set up for each possible root cause (activity A4).

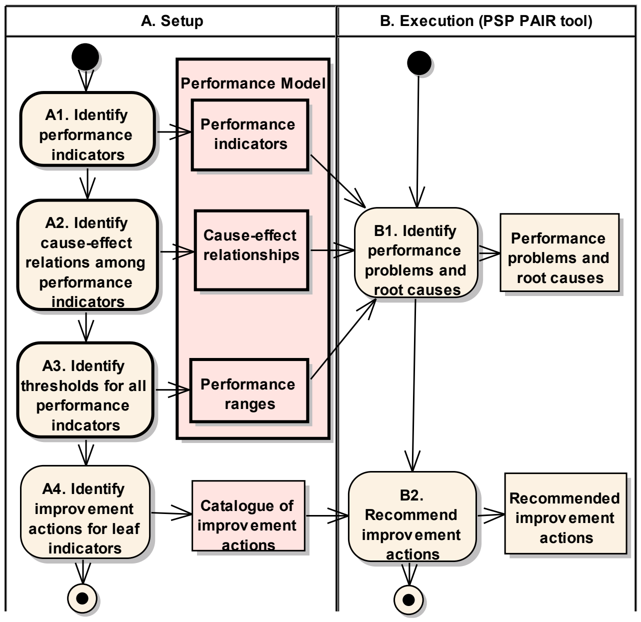

A. Setup
B. Execution (PSP PAIR tool)
Performance Model
A1. Identify performance indicators
Performance indicators
A2. Identify cause-effect relations among performance indicators
Cause-effect relationships
A3. Identify thresholds for all performance indcators
Performance ranges
A4. Identify improvement actions for leaf indicators
Catalogue of improvement actions
B1. Identify performance problems and root causes
Performance problems and root causes
B2. Recommend improvement actions
Recommended improvement actions

Figure 1. UML activity diagram identifying the steps and artifacts in our approach.

3. PERFORMANCE INDICATORS AND DEPENDENCIES

In this and the next section we present the performance model we conceived based on PSP specifications (of base and derived measures, estimation methods, etc.), literature review, expert knowledge, and analysis of PSP data (benchmarking), for enabling the automated performance analysis of PSP developers, namely, identification of performance problems and root causes.

We start by presenting in this section the PIs and dependencies (potential cause-effect relationships) between PIs. We considered the usual three top level performance characteristics in software development—predictability (estimation accuracy), quality and productivity—measured in a way specific to the PSP context, as explained in the next sections. All the PIs introduced in this section are defined in Appendix A.

3.1 Estimation Accuracy

We analyze the time (effort) estimating accuracy, and not the cost or schedule estimation accuracy, because cost measurement is out of the scope of the PSP and schedule planning is seldom performed by PSP developers (because the usual scope is the development of small programs or components of larger programs).

In the PSP, the time estimation error is defined as shown in the top of the summary table in Appendix A. In order to identify the factors that influence the behavior of this PI, one has to understand the PROBE estimation method used in the PSP [2]. In this method, a time estimate is obtained based on a size estimate of the deliverable (in lines of code, function points, etc.), and a productivity estimate (in size units/hour). So, the accuracy of the time estimate will depend on the accuracy of the size and productivity estimates, according to the formula shown in the bottom of Appendix A. Hence, we conclude that *Time Estimation Error* is affected by *Size Estimation Error* and *Productivity Estimation Error* according to a formula, as depicted in Fig. 2. It is worth noting that, in the case of changes or reuse of existing artifacts, only the size of the changes (additions and modifications) is considered.

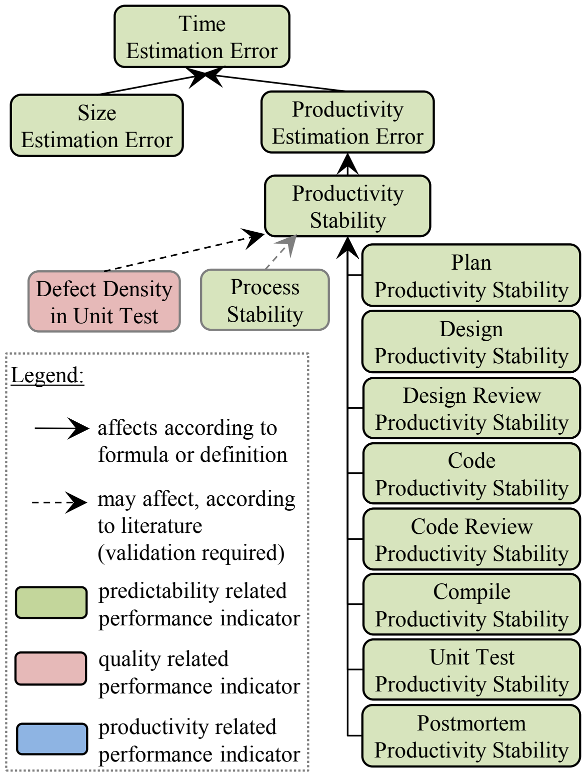

Time Estimation Error
Size Estimation Error
Productivity Estimation Error
Productivity Stability
Defect Density in Unit Test
Process Stability
Plan Productivity Stability
Design Productivity Stability
Design Review Productivity Stability
Code Productivity Stability
Code Review Productivity Stability
Compile Productivity Stability
Unit Test Productivity Stability
Postmortem Productivity Stability
Legend:
affects according to formula or definition
may affect, according to literature (validation required)
predictability related performance indicator
quality related performance indicator
productivity related performance indicator

Figure 2. Performance model for identifying causes of time estimation problems.

In the PROBE method, productivity estimates are based on historical productivity, so their accuracy depends on the stability of the productivity, as indicated in Fig. 2. We calculate the productivity stability up to the project under analysis, based on the actual productivity of that project and past projects of the same developer (see definition in Appendix A).

Since in the PSP time is recorded per process phase, the logical step to follow when an overall productivity stability problem is encountered is to analyze the productivity stability per phase, in order to determine the problematic phase(s). Hence, we indicate in the right-hand side of Fig. 2 a set of PIs for the productivity stability per phase, which together affect the overall productivity stability. Although there isn't a comprehensive formula for this relationship (without using other variables), the fact is that if all the phases have a perfectly stable productivity, then by definition the overall productivity will also be perfectly stable. The converse is also true, i.e., if the overall productivity is not stable, then at least one phase must have an unstable productivity. Hence the relationship is *affects according to definition*. It is worth noting that the scope of the PSP is the development of small programs or components of larger programs, reason why Requirements, High Level Design and System Testing phases are not included, but can be found in the more complete TSP. In the case of projects developed with programming languages or environments without a separate Compile phase, the Compile phase may be absent.

On the other hand, process changes can cause productivity disturbances: usually, after a process change, productivity decreases and later on returns to the original productivity or a new one [16]. This is an important factor when analyzing PSP training data, because the process is changed several times along the training, from PSP0 to PSP2.1. Hence, we indicated in Fig. 2 that *Process Stability* may affect *Productivity Stability*.

In the PSP literature [2], it is claimed that the effort for finding and fixing defects through reviews is much more predictable than through testing, because in the former case the review effort is proportional to the size of the work product under review and defects are immediately located, whilst in the latter case the time needed to locate defects is highly variable. High defect density in tests is considered a common cause of predictability problems. Hence, we indicate in Fig. 2 that the *Defect Density in Unit Test* may affect *Productivity Stability*. In turn, the defect density in unit tests is affected by other factors, as explained in the next section.

3.2 Quality

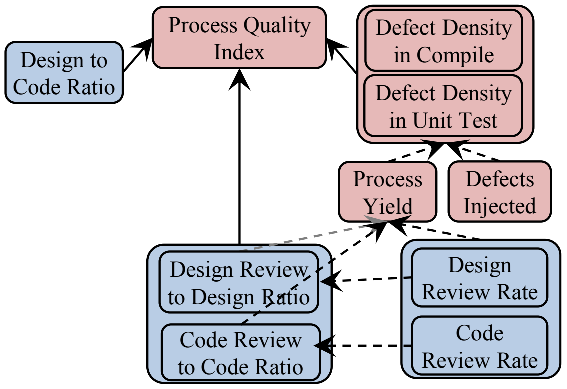
Product quality is usually measured by post-delivery defect density [17]. However, since the scope of the PSP is the development of small programs or components of large programs and information about post-delivery defects is often not available, we concentrate our attention on the data that is collected during the development process (up to the delivery of the program under development to the end user, or the delivery of the component under development to subsequent integration and system testing).

In the PSP literature [2][5], it is proposed an aggregated quality measure - the *Process Quality Index (PQI)*, which, according to [18], constitutes an effective predictor of post-delivery defect density. Hence, we use the *Process Quality Index* as the top-level quality indicator to analyze, as shown in Fig. 3.

The PQI takes into account five components [5]:

- The ratio of design time to coding time (*Design to Code Ratio*), which provides an indication of design quality, and is recommended to be at least 100%;
- The ratio of design review time to design time (*Design Review to Design Ratio*), which provides an indication of design review quality, and is recommended to be at least 50%;
- The ratio of code review time to coding time (*Code Review to Code Ratio*), which provides an indication of code review quality, and is recommended to be at least 50%;
- The ratio of compile defects to a size measure (*Defect Density in Compile*), which provides an indication of code quality, and is recommended to be less than 10 defects/KLOC;
- The ratio of unit test defects to a size measure (*Defect Density in Unit Test*), which provides an indication of program quality, and is recommended to be less than 5 defects/KLOC.

The PQI components are normalized to [0, 1] such that 0 represents poor practice and 1 represents desired practice (according to the recommended values indicated above).



Process Quality Index
Design to Code Ratio
Defect Density in Compile
Defect Density in Unit Test
Process Yield
Defects Injected
Design Review to Design Ratio
Code Review to Code Ratio
Design Review Rate
Code Review Rate

Figure 3. Performance model for identifying causes of quality problems.

The aggregated PQI value is computed by multiplying the normalized values of the components, as shown in Appendix A. Hence, in Fig. 3 we indicate the aforementioned 5 components as factors that directly affect the PQI according to a formula. Next we try to drill down each of the PQI components.

Both the *Defect Density in Compile* and the *Defect Density in Unit Test* are affected by the total density of *Defects Injected* (and found) and the percentage of defects removed before compiling and testing (called *Process Yield* in the PSP). In other words, high defect densities in compile and test may be caused by a large number of defects injected (due to poor defect prevention) and/or a large percentage of defects escaped from previous defect filters (due to poor design and code reviews). Hence, we show in Fig. 3 *Defects Injected* and *Process Yield* as factors that may affect both the *Defect Density in Compile* and the *Defect Density in Unit Test*.

Since in the PSP defects are classified by defect type (based on Orthogonal Defect Classification [19]) and ascribed an injection and a removal phase [2], in the presence of a high density of *Defects Injected,* a defect causal analysis [20] could be performed to determine common causes, such as many defects of the same type or from the same activity. However, the data set used in this paper contains only summary defect data, and not defect type information, reason why we don't decompose *Defects Injected* further in this paper.

According to the PSP literature [2], the time spent in reviewing a work product, measured both in relation with its size or the time spent in developing it, is a leading indicator of the review yield (percentage of defects found in reviews) and consequently of the process yield. In a published study [11], the recommended review rate of 200 LOC/hour or less was found to be an effective rate for individual reviews, identifying nearly two-thirds of the defects in design reviews and more than half of the defects in code reviews. Another study in an industrial setting [21] shows an increase in inspection effectiveness when the review rate is reduced to a value closer to the recommended value. Hence, like other authors [22], we indicate in Fig. 3 that the *Process Yield* may be affected by the *Design Review Rate*, the *Code Review Rate*, the *Design Review to Design Ratio,* and the *Code Review to Code Ratio*.

A too small *Design to Code Ratio* is usually related to a lack of thoroughness of the design artifacts produced (coverage of important design views, coverage of requirements, etc.), or even the total absence of explicit design artifacts. By contrast, at least in the context of PSP training, too high values of the *Design to Code Ratio* may be caused by excessively detailed design artifacts or by lack of experience with the design methods or notations used. However, in both cases, the needed information is usually not available or not amenable for automated analysis, so we don't decompose further this indicator.

Regarding *Design Review to Design Ratio* and *Code Review to Code Ratio*, in case of significant deviations from the recommended value (50%), it is useful to look at other indicators of review quality - *Design Review Rate* and *Code Review Rate.*

3.3 Productivity

Software development productivity is usually measured in function points per time unit or lines of code (LOC) per time unit [23][24][25]. However, both productivity measurement techniques have some limitations. On one hand, the measurement of function points remains subjective even after project completion. On the other hand, productivity measures based on LOC have limitations due to the lack of counting standards, the dependence on the programming language [26] and the inferior economic meaning [27].

In the PSP, productivity is measured in 'size' units per hour. Any size measure can be used as long as it correlates with effort (in order to enable effort estimation based on size estimation) and is automatically measurable. In this paper, we use LOC/hour as the productivity measure, in spite of its limitations, because it is the productivity measure most often used in the PSP practice.

Since in the PSP time is recorded per process phase, the logical step to follow when an overall productivity problem is encountered is to analyze the productivity per phase, in order to determine the problematic phase(s). Hence, we indicate in the right-hand side of Fig. 4 a set of PIs for the productivity per phase, which together affect (according to a formula) the overall productivity. In order to reuse previous PIs, in some cases (as with the *Design to Code Ratio*) we measure the productivity of a phase by comparing the time spent in that phase with the *Code* phase.

It is expectable that the time spent in *Compile* and *Unit Test* is affected by the number of defects to fix. In the case of testing, there may exist a test development effort independent of the number of defects, but even so the majority of the time in the test phase is usually spent in bug fixing. For that reason, we indicate in Fig. 4 the *Defect Density in Compile* and *Defect Density in Unit Test* as factors that may affect the productivity in *Compile* and *Unit Test*, respectively (measured by the ratio with respect to the time in *Code*). Those quality PIs are in turn affected by other indicators as explained in the previous section.

Besides looking at specific phases to determine possible causes of overall productivity problems, it may also be helpful to look at productivity factors that may affect all the phases. In previous studies [15][16], based on the same data set described in section 4, we found significant variations of productivity between developers that could be partially explained by developers experience (measured in KLOC or years) and the level of abstraction of the programming language used. We also found significant variations of productivity per phase between projects that could be partially explained by process variations (process level and stability), and domain complexity. Hence we indicate all those factors as potentially affecting productivity in the left-hand side of Fig. 4. Outside the context of the PSP, similar factors are mentioned in the literature, such as the process used [28], the generation of programming languages used [29], experience of programmers [30] and task complexity [25].

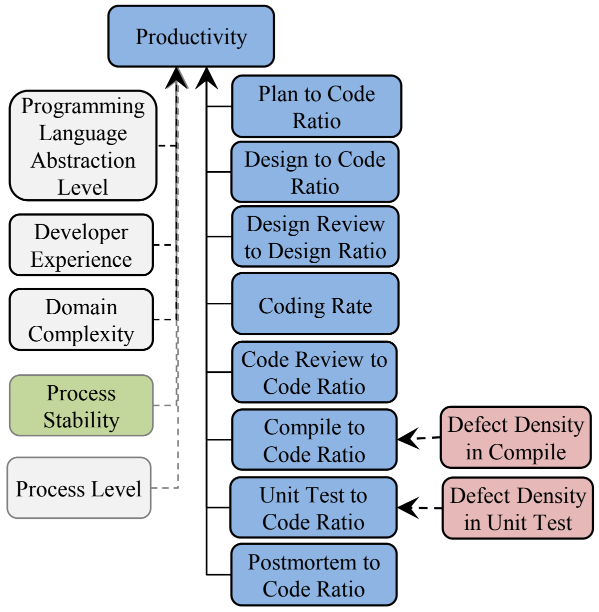


Figure 4 Performance model for identifying causes of productivity problems.

4. DERIVATION OF PERFORMANCE RANGES

In order to recognize performance problems, we defined a set of thresholds and ranges for classifying values of each PI into three categories:

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

To define these ranges we took into account the following criteria:

- recommended values and ranges described in the literature (e.g., [2]), some of which were already mentioned in the previous section;
- the actual distribution of existing PSP data, so that there is an approximately even distribution of data points by the different colors, in a way similar to benchmark-based software product quality evaluation [31];
- thresholds should be rounded to 1 or 2 precision digits.

For determining the actual distribution of existing PSP data, we analyzed a data set from the Software Engineering Institute (SEI), containing data collected during the classic PSP for Engineers I/II training courses. In this training course, targeting professional software engineers, each engineer has to develop 10 small projects (the same for all individuals). The development process is refined throughout the course, with the introduction of additional practices: PSP0 - performance measures and empirical effort estimation; PSP0.1 - empirical size estimates; PSP1 - PROBE estimation method; PSP1.1 - schedule management; PSP2 - design and code reviews and quality management; PSP2.1 - design specification templates. The data set contains 31,140 data points (project submissions) corresponding to 3,114 engineers that performed 10 projects each, during 295 training classes occurred between 1994 and 2005.

In most cases, the 'green' range is located in one of the extremes of the scale, the 'red' range in the other extreme, and the 'yellow' range in the middle. For example, the 'green' range for the *Process Quality Index* is located in the high values of the [0, 1] scale, whilst for the *Defect Density in Unit Test (DDUT)* it is located in the low values of the [0, ∞[scale. In these cases, benchmark-based thresholds were first computed based on rounded terciles and subsequently cross-checked and refined by comparison with literature. For example, according to [5], *DDUT* should be less than 5 defects/KLOC for high performance, but only 20% of the data points lays in the [0, 5] range, so we considered a wider [0, 10] 'green' range instead, containing 31% of the data points.

For several other PIs, the 'green' range is located somewhere in the middle, in order to balance conflicting aspects, such as productivity and quality. For example, regarding the *Code Review Rate*, if reviews are performed too fast then the quality of the reviews may suffer (low percentage of defects found); if reviews are performed too slow, then the productivity is negatively affected. In this case, we selected a 'green' range around the recommended value of 200 LOC/hour, containing approximately 1/3 of the data points, and considered 'red' values to exist in both extremes of the scale.

All the performance ranges defined, together with the statistical distribution of the analyzed PSP data points across those ranges, are shown in Appendix A. It should be noted that, in computing the statistical distribution, only the data points with defined values for the PI under consideration were used. For example, for the review related PIs, projects performed in PSP levels less than 2 were excluded, because of the absence of explicit review phases.

5. DEPENDENCIES VALIDATION

The performance models of Figures 2 to 4 indicate several 'may affect' relationships between pairs of PIs suggested from the literature and expert knowledge. In order to validate each '*X may affect Y'* relationship, from the PSP data set previously described, we computed the Spearman's [32] rank correlation coefficient *(r)* 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. 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 (linear or not). The results are shown in Appendix B.

In all cases except one (case *a - Process Stability* versus *Productivity Stability*), the result of the test was positive (i.e., the null hypothesis was rejected), showing a statistically significant correlation between the PIs under analysis.

In 5 other cases the correlation coefficient is very small (less than 0.1), indicating a very weak correlation. The extreme case is *p - Process Level* versus *Productivity* with *r=0.014*. This specific result is not surprising because the progression of PSP levels was designed with the goal of obtaining predictability and quality gains, whilst productivity gains are expected to occur only in later phases (namely integration and system test) beyond the scope of the PSP. In a previous study [16], we found a strong impact of process changes on the productivity of the phases affected by those changes, but when all the phases and factors are put together the impact is greatly diminished. In fact, with the progression of PSP levels, there is an effort shift from tests to reviews and from code to design, but the overall productivity is mostly unchanged.

For an additional examination of the dependencies between PIs *X* and *Y*, Appendix B also shows the conditional means *E(Y|X),* considering the discretization of *X* according to the performance intervals defined in Appendix A. In general, there is a monotonic evolution of *E(Y|X)* for increasing intervals of *X*, consistent with the Spearman's test results, with the exceptions highlighted in the last column (all corresponding to values of *r < 0.1*). In the cases of non-monotonic behavior of *E(Y|X)*, we consider that there isn't enough evidence to validate the dependencies under analysis, so the respective arrows are dimmed in Figures 2 to 4. For the cases of monotonic behavior of *E(YX)*, Appendix C shows the conditional distributions *P(Y|X),* with both variables discretized, for a better visual inspection of the relationship between *X* and *Y*.

6. CASE STUDY

In the end of the PSP training and at regular times afterwards, developers should analyze their personal performance along the series of projects developed, and document their findings and a set of prioritized and quantified process improvement proposals in a Performance Analysis Report (PAR). A goal of our research is to help partially automating this kind of analysis. In this section we describe how the performance of an individual PSP developer (selected based on the availability of his PAR) can be analyzed based on the proposed model. We also compare the results of the model-based analysis with the results of the manual analysis.

In this case, the PSP training sequence (PSP Fundamentals and PSP Advanced) comprised 7 projects. The developer used the Java programming language without an explicit Compile phase.

The evaluation of the 3 top-level PIs for the 7 projects, together with all 'child' PIs defined in our performance model, is shown in Table 1. The main top-level performance problems occur in time estimation (projects P1, P3 and P7) and in productivity (projects P6 and P7). In order to identify the possible causes of those problems, we conducted a top-down analysis, from the top-level PIs to their affecting PIs (following the structure of Figures 2 to 4), restricted to the 'red' colored PIs and projects. The results are shown in Fig. 5. For example, the time estimation problems of projects P3 and P7 have totally different causes: size estimation problems in P3 and productivity instability problems in P7.

A comparison of the conclusions reached in the model-based and manual analysis is presented in Table 2. The conclusions are quite similar, except for the quality performance, because of the much stricter performance ranges considered in the manual analysis. Once automated, the model-based analysis has the advantage of being almost instantaneous, whilst the manual analysis can take significant effort (8 hours in the case studied). Even if it does not eliminate the need for a manual analysis, the model-based analysis can point out the problematic areas to focus in manual analysis.

7. CONCLUSIONS AND FUTURE WORK

We presented a performance model to enable the automated identification of performance problems and their root causes, and reduce the manual effort of performance analysis in the PSP. Since the model is calibrated based on a large data set, it can also be used for benchmarking purposes.

We are currently extending our PSP PAIR (Performance Analysis and Improvement Recommendation) tool [13][14] to support the analysis of all the PIs presented in this paper. The tool analyzes performance data produced by PSP developers in their projects, as recorded in the PSP Student Workbook, and pinpoints performance problems, possible root causes and suggestions for remedial actions. The performance model is given as an XML file for easier configuration and extensibility, but the formulas have to be programmed in tool extensions.

Table 1. Evaluation of the full set of PIs in the case study.

Indicator	P1	P2	P3	P4	P5	P6	P7
Time Estimation Error (|TimeEE|)	**73%**	**34%**	**63%**	**1%**	**28%**	**39%**	**72%**
Size Estimation Error (|SizeEE|)		4%	51%	4%	8%	8%	2%
Productivity Estim.Error (|PEE|)		22%	7%	5%	15%	22%	43%
Productivity Stability (|ProdS|)		32%	20%	17%	21%	52%	63%
Plan Prod. Stab.		80%	34%	1%	113%	33%	34%
Design Prod.Stab.		16%	45%	100%	72%	65%	85%
Design Review Prod. Stab.				19%	65%	73%	59%
Code Prod. Stab.		12%	29%	42%	24%	7%	20%
Code Review Prod. Stab.				218%	99%	3%	48%
Test Prod.Stab.		38%	50%	61%	4%	40%	50%
Potmortem Prod. Stab.		36%	36%	18%	46%	60%	51%
Process Stability (ProcS)	0	0	0	1	2	3	4
Process Quality Index (PQI)			**0.38**	**0.07**	**0.29**	**0.26**	**0.12**
Defect Density in Unit Test (DDUT)	26	8	0	20	15	24	17
Defects Injected (DI)	60	16	25	47	67	122	133
Process Yield (PY)			100%	57%	78%	80%	88%
Productivity (Prod)	**34**	**23**	**22**	**29**	**21**	**12**	**9**
Coding Rate (CR)	85	95	116	138	132	103	88
Plan to Code Ratio (P2C)	0.23	1.30	1.48	1.35	0.61	1.33	1.20
Design to Code Ratio (D2C)	0.52	0.51	0.46	0.35	2.07	1.96	4.54
Design Rev. to Des. Ratio (DR2D)			0.57	0.74	0.37	0.64	0.19
Design Review Rate (DRR)			443	526	171	82	100
Code Review to Code Ratio (CR2C)			0.87	0.45	0.62	0.96	0.54
Code Review Rate (CRR)			134	308	212	107	164
Unit Test to Code Ratio (UT2C)	0.57	1.04	0.69	0.68	0.97	1.21	1.29
Postmortem to Code Ratio (PM2C)	0.21	0.36	0.57	0.63	0.36	0.96	0.73
Program. Lang. Abstr.Lev. (PLAL)	3	3	3	3	3	3	3
Process Level (ProcL)	0	1	2	2	2	2	2
Developer Experience (DevE)	20	20	20	20	20	20	20
Domain Complex. (DomC)	1	2	1	1	1	1	1

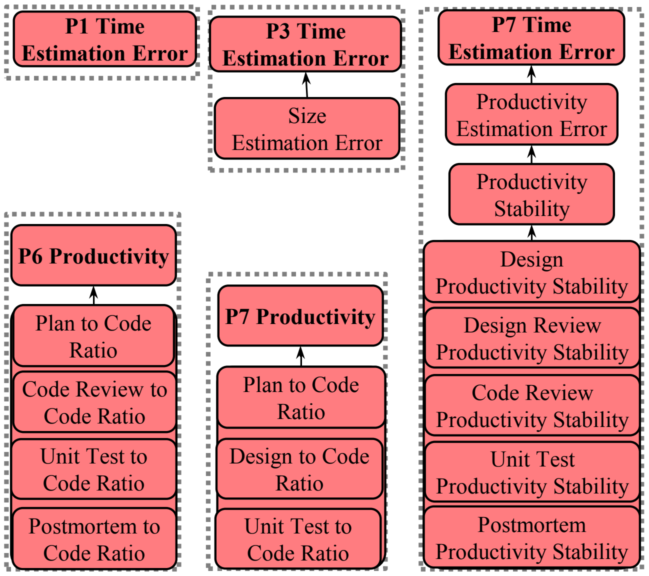


Figure 5 Summary of major performance problems and root causes identified in the case study based on our model.

Table 2. Comparison of major problems and root causes identified in manual and model-based analysis.

Manual Analysis (PAR)	Model-Based Analysis
Poor time <u>estimation</u> accuracy, with time underestimation in P3 caused by size underestimation, and time underestimation in P7 due mainly to an inneficient and unstable DLD process.	Significant time <u>estimation</u> problems in 3 projects (P1, P3, P7), caused in P3 by a size estimation problem, and in P7 by productivity instability in several phases (DLD, DLDR, CR, UT, PM).
Product <u>quality</u> problems, with average DDUT well above the recommended value of 5, caused by a high number of defects injected.	No significant process (PQI) and product (DDUT) <u>quality</u> problems, as compared to benchmarks.
<u>Productivity</u> problems, namely at DLD phase, caused by an inneficient DLD process (long design specification documents following PSP templates).	Significant <u>productivity</u> problems in two projects, caused by slow performance in several process phases (PLAN, DLD, CR, UT and PM), notably in DLD.

As future work we plan to develop techniques for ranking the identified root causes of performance problems, to guide improvement efforts, by a combination of *sensitivity ranking* and *percentile ranking*. E.g., Fig. 5 identifies three non-prioritized root causes for the poor productivity in project P7—poor productivity in Plan, Design and Unit Test phases (as given by the P2C, D2C and UT2C ratios). But the data in Table 1 shows that the D2C value is the main contributor to the poor productivity in P7, so should be ranked first (*sensitivity ranking)*. The same conclusion is reached by noting that the D2C value goes much more into the red range than the P2C and UT2C values (*percentile ranking*).

We also intend to incorporate defect causal analysis techniques (namely, decompose *Defects Injected* per injection phase and, if data is available, per defect type), develop separate calibrations for different problem domains and programming languages (to increase the precision of the analysis), build a comprehensive catalogue of improvement actions to recommend for each root cause, conduct further experiments, and extend the approach for analyzing performance data produced in the context of other processes (namely TSP and Scrum with TSP combinations) and tools (namely cloud-based project management environments).

8. AKNOWLEDGMENTS

The authors would like to acknowledge the SEI, in particular W. Nichols and J. Over, for facilitating the access to the PSP data for performing this study and for helpful discussions. This work is partially funded by the Portuguese Foundation for Science and Technology (*FCT - Fundação para a Ciência e a Tecnologia*), under research grant SFRH/BD/85174/2012, and by the 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.

9. REFERENCES

[1] Bohem, B. 2011. Some Future Software Engineering Opportunities and Challenges. In *The Future of Software Engineering*, Springer-Verlag, 1-32.

[2] Humphrey, W. 2005. *PSP^{sm}: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional.

[3] Davis, N. and Mullaney, J. 2003. *The Team Software Process (TSP) in Practice: A Summary of Recent Results*. CMU/SEI-2003-TR-014.

[4] Rombach, D., Münch, J., Ocampo, A., Humphrey, W., and Burton, B. 2008. Teaching disciplined software development. *Journal of Systems and Software* 81(5): 2008, 747-763.

[5] Pomeroy-Huff, M., Cannon, R., Chick, T., Mullaney, J., and Nichols, W. 2009. *The Personal Software Process^{SM} (PSP^{SM}) Body of Knowledge (Version 2.0)*. CMU/SEI-2009-SR-018.

[6] Burton, D. and Humphrey, W. 2006. Mining PSP Data. In *TSP Symposium 2006 Proceedings*.

[7] The Software Process Dashboard Initiative home page. http://www.processdash.com/.

[8] Philip, J., Kou, H., Agustin, J., Christopher, C., Moore, C., Miglani, J., Zhen, S., Doane, W. 2003. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In *ICSE 2003*. Portland, Oregon.

[9] Shin, H., Choi, H., and Baik, J. 2007. Jasmine: A PSP Supporting Tool. In *Proc. of the Int. Conf. on Software Process* (ICSP 2007), LNCS 4470, Springer-Verlag, 73-83.

[10] Nasir, M. and Yusof, A. 2005. Automating a Modified Personal Software Process. *Malaysian Journal of Computer Science*, vol. 18, 11–27.

[11] Kemerer, C., and Paulk, M. 2009. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering*, vol. 35, Issue 4, 534-550.

[12] Shen, W., Hsueh, N., Lee, W. 2011. Assessing PSP effect in training disciplined software development: A Plan–Track–Review model. *Inform. and Soft.Technology* 53, 137–148.

[13] Duarte, C., Faria, J., and Raza, M. 2012. PSP PAIR: Automated Personal Software Process Performance Analysis and Improvement Recommendation. In *Proc. of the 8th Int. Conf. on the Quality of Information and Communications Technology*, IEEE CPS, Lisbon, Portugal.

[14] Duarte, C., Faria, F., Raza, M., Henriques, P. 2012. Model and Tool for Analyzing Time Estimation Performance in PSP. In *TSP Symposium 2012*, CMU/SEI-2012-SR-015, 21-40.

[15] Raza, M., Faria, J., Henriques, P., and Nichols, W. 2013. Factors Affecting Productivity Performance in PSP Training. In *TSP Symposium 2013.*, CMU/SEI-2013-SR-022, 35-45.

[16] Raza, M., Faria, J. 2014. Factors Affecting Personal Software Development Productivity: A Case Study with PSP Data. In *IASTED SE 2014*.

[17] Jones, C. 2000. *Software Assessments, Benchmarks, and Best Practices*. Addison Wesley.

[18] Humphrey, W. 2009. *The Software Quality Profile*. White Paper, SEI.

[19] Chillargee, R., Bhandari, I., et. al. 1992. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Trans. on Software Eng.*, Vol. 18, Issue 11, 943-956.

[20] Card, D. 2005. Defect Analysis: Basic Techniques for Management and Learning. *Advances in Computers*, vol.64, 259-295. Elsevier.

[21] Ferreira, A., Machado, R., Costa, L., Silva, J., Batista, R., and Paulk, M. 2010. An Approach to Improving Software Inspections Performance. In *Proc. of the 2010 IEEE Int. Conf. on Soft.Maintenance*, 1-8, ISBN 978-1-4244-8640-4.

[22] Tamura, S. 2009. Integrating CMMI and TSP/PSP: Using TSP Data to Create Process Performance Models. CMU/SEI-2009-TN-033.

[23] Wagner, S. and Ruhe, M. 2008. A Systematic Review of Productivity Factors in Software Development. In *Proc. of 2nd Int. Workshop on Software Productivity Analysis and Cost Estimation (SPACE 2008)*.

[24] Maxwell, K. and Forselius, P. 2000. Benchmarking Software Development Productivity. *IEEE Software*, 17(2), 80-88.

[25] Goparaju, P., Farooq, A., Patnaikc, S. 2012. Measuring Productivity of Software Development Teams. *Serbian Journal of Management* 7 (1) (2012), 65-75.

[26] Card, D. 2006. The Challenge of Productivity Measurement. In *Proc. of the Pacific Northwest Software Quality Conference*, Portland, OR.

[27] Jones, C. 2010. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. McGraw-Hill.

[28] Scacchi, W. 1995. Understanding Software Productivity. *Software Engineering and Knowledge Engineering: Trends for the Next Decade*. World Scientific Press.

[29] Comstock, C., Jiang, Z., and Naudé, P. 2007. Strategic Software Development: Productivity Comparisons of General Development Programs. *Int. Journal of Computer and Information Engineering* 1:8 2007, 486-491.

[30] Banker, R. and Kauffman, R. 1991. Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study. *MIS Quarterly*, Sept 1991, 14(3):374-401

[31] Alves, T. 2012. *Benchmark-based Software Product Quality Evaluation*. Doctoral Thesis. University of Minho.

[32] Navidi, W. 2011. Statistics for Engineers and Scientists, Third Edition, McGraw-Hill.

APPENDIX A - PERFORMANCE INDICATORS

Indicator	Formula	Performance Ranges			Distribution of PSP Data		
		Green	Yellow	Red	G	Y	R
Time Estimation Error (|TimeEE|)*	$\left\lvert\frac{Actual\ Time - Estimated\ Time}{Estimated\ Time}\right\rvert$	[0, 20%]	]20%, 40%]	]40%,, ∞[	41%	26%	33%
Size Estimation Error (|SizeEE|)	$\left\lvert\frac{Actual\ Size - Estimated\ Size}{Estimated\ Size}\right\rvert$	[0, 20%]	]20%, 45%]	]45%,, ∞[	36%	29%	35%
Productivity Estimat. Error (|PEE|)	$\left\lvert\frac{Actual\ Productivity - Estim.\ Productiv.}{Estimated\ Productivity}\right\rvert$	[0, 20%]	]20%, 40%]	]40%,, ∞[	35%	27%	38%
Productivity (In)Stability(|ProdS|)	$\left\lvert\frac{Current\ Productivity - Hist. Product.}{Historical\ Productivity}\right\rvert$ *w/H.P.=TotalSize/TotalEffort for past projects*	[0, 20%]	]20%, 40%]	]40%,, ∞[	34%	29%	37%
Process Stability (ProcS)	Number of previous projects performed with same process level as current one.	>=2	1	0	40%	30%	30%
Process Quality Index (PQI)	$\min(\frac{D2C}{1},1) \times \min(\frac{DR2D}{0.5},1) \times \min(\frac{CR2C}{0.5},1) \times \min(\frac{2 \times 10}{DDC+10},1) \times \min(\frac{2 \times 5}{DDUT+5},1)$	[0.25, 1]	[0.06, 0.25[	[0, 0.06[	27%	37%	35%
Defect Density in Unit Test (DDUT)	$\frac{\#Defects\ found\ and\ removed\ in\ Unit\ Test}{Actual\ Size\ (KLOC)}$	[0, 10]	]10, 30]	]30, ∞[	31%	36%	33%
Defect Density in Compile (DDC)	$\frac{\#Defects\ found\ and\ removed\ in\ Compile}{Actual\ Size\ (KLOC)}$	[0, 10]	]10, 40]	]40, ∞[	30%	37%	33%
(Density of) Defects Injected (DI)	$\frac{\#Defects\ found\ in\ all\ phases}{Actual\ Size\ (KLOC)}$	[0, 50]	]50, 100]	]100, ∞[	39%	34%	28%
Process Yield (PY)	$\frac{\#Defects\ removed\ before\ Compile\ \&\ Test}{\#Defects\ injected\ before\ Compile\ \&\ Test}$	[70%,100%]	[50%, 70%[	[0, 50%[	28%	35%	37%
Productivity (Prod)*	$\frac{Actual\ Size\ (LOC)}{Actual\ Time\ (hours)}$	[35, ∞[	[20, 35[	[0, 20[	34%	34%	32%
Coding Rate (CR)	$\frac{Actual\ Size\ (LOC)}{Code\ Time\ (hours)}$	[80, ∞[	[45, 80[	[0, 45[	35%	32%	33%
Plan to Code Ratio (P2C)	$\frac{Plan\ Time}{Code\ Time}$	[0.15, 0.4]	[0.05, 0.15[∪]0.4, 0.8]	[0, 0.05[∪]0.8, ∞[	32%	41%	26%
Design to Code Ratio (D2C)	$\frac{Design\ Time}{Code\ Time}$	[0.5, 1.5]	[0.2, 0.5[∪]1.5,2.0]	[0,0.2[∪]2.0, ∞[	33%	35%	32%
Design Review to Design Ratio (DR2D)	$\frac{Design\ Review\ Time}{Design\ Time}$	[0.3, 0.5]	[0.1, 0.3[∪]0.5, 0.8]	[0, 0.1[∪]0.8, ∞[	28%	52%	20%
Design Review Rate (DRR)	$\frac{Actual\ Size\ (LOC)}{Design\ Review\ Time\ (hours)}$	[200, 400]	[115,200[∪]400,700]	[0, 115[∪ [700, ∞[	30%	35%	35%
Code Review to Code Ratio (CR2C)	$\frac{Code\ Review\ Time}{Code\ Time}$	[0.3, 0.5]	[0.1, 0.3[∪]0.5, 0.6]	[0, 0.1[∪]0.6, ∞[	31%	41%	29%
Code Review Rate (CRR)	$\frac{Actual\ Size\ (LOC)}{Code\ Review\ Time\ (hours)}$	[150, 300]	[100,150[∪]300,500]	[0, 100[∪]500, ∞[	31%	34%	35%
Compile to Code Ratio (C2C)	$\frac{Compile\ Time}{Code\ Time}$	[0, 0.06]	]0.06, 0.2]	]0.2, ∞[	31%	39%	30%
Unit Test to Code Ratio (UT2C)	$\frac{Unit\ Test\ Time}{Code\ Time}$	[0, 0.3]	]0.3, 0.7]	]0.7, ∞[	30%	33%	37%
Postmortem to Code Ratio (PM2C)	$\frac{Postmortem\ Time}{Code\ Time}$	[0.15, 0.4]	[0.05, 0.15[∪]0.4, 0.8]	[0, 0.05[∪]0.8, ∞[	38%	42%	20%
Program. Lang. Abstr.Level (PLAL)	1 (Assembly languages), 2 (C, C++, Pascal, Fortran, VB), 3 (Java, C#, VB.Net)	3	2	1	24%	76%	0%
Process Level (ProcL)	0 (for PSP0 and PSP0.1) , 1 (for PSP1 and PSP1.1), or 2 (for PSP2 and PSP2.1)	2	1	0	40%	30%	30%
Developer Experience (DevE)	KLOC developed in the programming language used	[20, ∞[	[4, 20[	[0, 4[	36%	31%	33%
Domain Complexity (DomC)	1 (numerical problems), 2 (text processing problems)	1	2		80%	20%	0%

*Relationships: $TimeEE = \frac{SizeEE+1}{PEE+1} - 1$; $Prod = \frac{CR}{P2C+D2C(1+DR2D)+1+CR2C+C2C+UT2C+PM2C}$.

APPENDIX B - CORRELATION TESTS

Id	Affected Indicator (Y)	Affecting Indicator (X)	Spearman correlation tests (H_0: r=0) (1)					Conditional means E(Y | X) for increasing intervals of X (7)					
			H_1 (2)	n (3)	r (4)(8)	p (5)(8)	Reject H_0? (6)	XS	S	M	L	XL	Monotonically increasing/decreasing according to H_1?
a	Productivity (In)Stability	Process Stability	r<0	24574	-0.006	0.189	***No***	-	40.2	38.8	41.5	-	***No***
b	Productivity (In)Stability	Defect Density in Unit Test	r>0	24568	0.032	2.3e-7	Yes	-	46.4	38.9	40.8	-	***No***
c	Def. Density in Unit Test	Process Yield	r<0	9612	-0.398	<2e-16	Yes	-	22.7	16.4	6.7	-	Yes
d	Def. Density in Unit Test	Defects Injected	r>0	27648	0.647	<2e-16	Yes	-	10.6	25.6	63.0	-	Yes
e	Def. Density in Compile	Process Yield	r<0	9612	-0.389	<2e-16	Yes	-	25.2	16.2	5.5	-	Yes
f	Def. Density in Compile	Defects Injected	r>0	27648	0.692	<2e-16	Yes	-	10.7	32.1	83.1	-	Yes
g	Process Yield	Design Review Rate	r<0	9371	-0.226	<2e-16	Yes	63.1	61.7	56.8	51.3	45.4	Yes
h	Process Yield	Code Review Rate	r<0	9548	-0.247	<2e-16	Yes	62.1	61.6	58.2	51.8	43.7	Yes
i	Process Yield	Des. Review to Design Ratio	r>0	9362	0.060	4e-9	Yes	45.2	52.5	54.6	54.5	53.3	***No***
j	Process Yield	Code Review to Code Ratio	r>0	9546	0.309	<2e-16	Yes	32.6	44.4	54.5	60.0	63.5	Yes
k	Des. Review to Des. Ratio	Design Review Rate	r<0	9677	-0.392	<2e-16	Yes	0.87	0.65	0.54	0.44	0.33	Yes
l	Code Review to Code Ratio	Code Review Rate	r<0	9881	-0.615	<2e-16	Yes	1.47	0.75	0.59	0.41	0.26	Yes
m	Productivity	Prog. Language Abstract. Level	r>0	12507	0.201 (0.164)	<2e-16	Yes	-	-	32.1	43.1	-	Yes
n	Productivity	Developer Experience	r>0	15481	0.161	<2e-16	Yes	-	28.7	35.3	37.8	-	Yes
o	Productivity	Domain Complexity	r<0	27855	-0.075 (-0.061)	2e-13 (<2e-16)	Yes	-	33.3	30.6	-	-	Yes
p	Productivity	Process Level	r>0	27855	0.014 (0.010)	0.011 (0.012)	Yes	-	33.3	32.7	32.1	-	***No***
q	Productivity	Process Stability	r>0	27855	0.053	<2e-16	Yes	-	33.1	31.3	33.5	-	***No***
r	Unit Test to Code Ratio	Defect Density in Unit Test	r>0	27625	0.480	<2e-16	Yes	-	0.43	0.75	1.21	-	Yes
s	Compile to Code Ratio	Defect Density in Compile	r>0	27625	0.630	<2e-16	Yes	-	0.077	0.177	0.365	-	Yes

Notes:

(1) H_0 denotes the null hypothesis considered in each test (*r=0*).

(2) H_1 denotes the alternative hypothesis considered in each test (r>0 or r<0 depending on the type of dependency expected).

(3) *n* denotes the number of data points. Only the data points with defined values for the variables under analysis were considered.

(4) *r* denotes the Spearman's correlation coefficient.

(5) *p* is a probability that indicates the statistical significance of the correlation coefficient (the lower the value of *p* the higher the significance) in the one-tailed test (because of the nature of H_1).

(6) We reject the null hypothesis if $p<0.05$ (5% significance level).

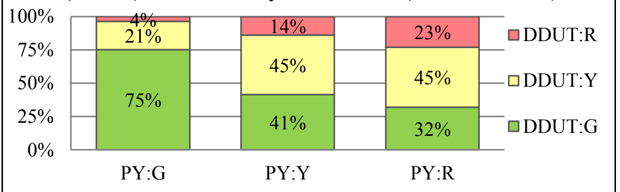
(7) The intervals for X are the ones defined for performance ranges in Appendix A, arranged by increasing values (XS - extra small, S - small, M - medium, L - large, XL - extra-large). For example, for DRR the intervals are XS=[0, 115[, S=[115,200[, M=[200, 400], L=]400,700] and XL=[700, ∞[.

(8) Between parentheses it is presented the Kendall's tau-b rank correlation results, for the cases in which X has a few discrete values (originating many ties). The Kendall's tau-b coefficient is usually recommended over the Spearman's coefficient in the presence of a large number of ties.

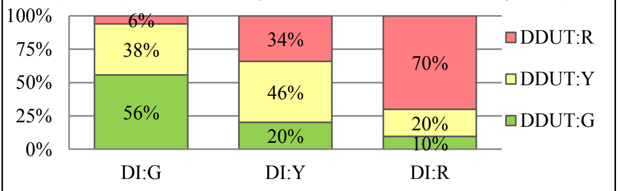
APPENDIX C - CONDITIONAL DISTRIBUTION CHARTS

Key: R:Red, Y:Yellow, G:Green, +:high interval, -:low interval

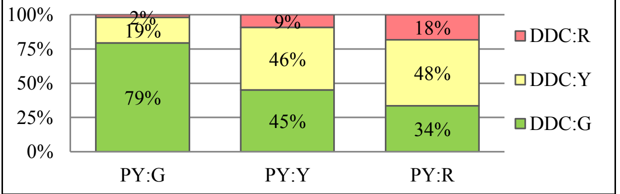
c) Prob(Defect Density in Unit Test | Process Yield)



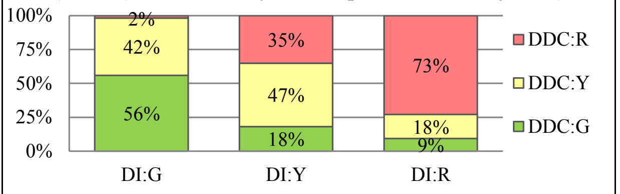
d) Prob(Defect Density in Unit Test | Defects Injected)



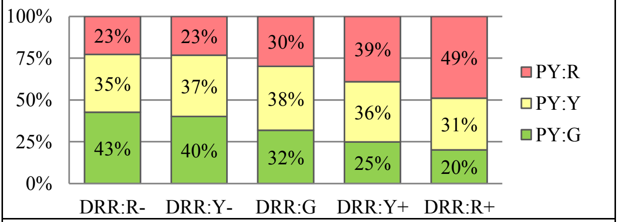
e) Prob(Defect Density in Compile | Process Yield)



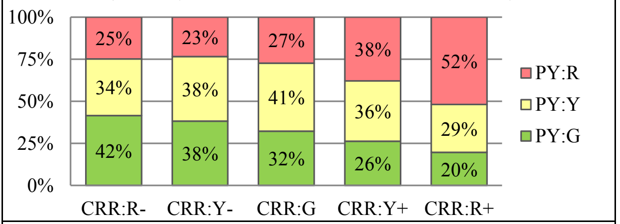
f) Prob(Defect Density in Compile | Defects Injected)



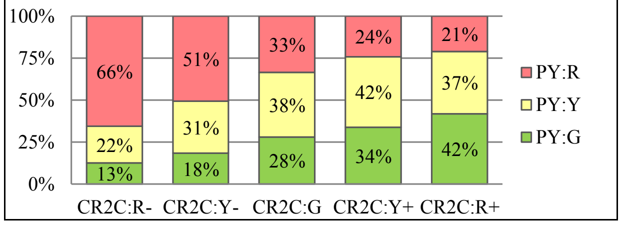
g) Prob(Process Yield | Design Review Rate)



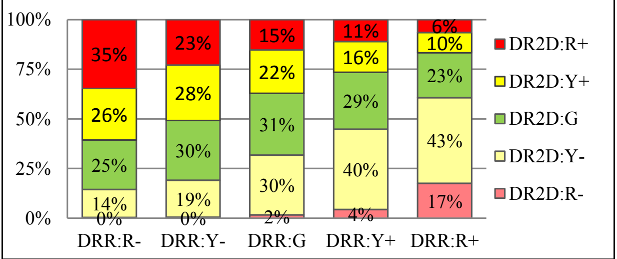
h) Prob(Process Yield | Code Review Rate)



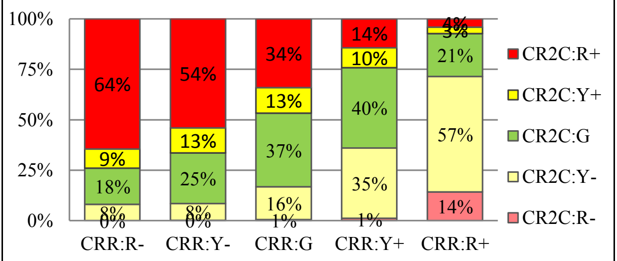
j) Prob(Process Yield | Code Review to Code Ratio)



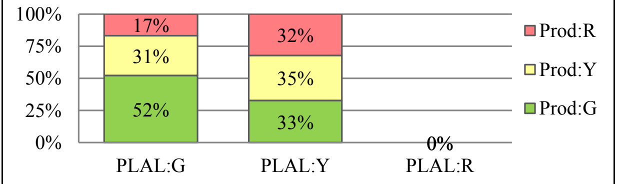
k) Prob(Design Review to Des. Ratio|Design Review Rate)



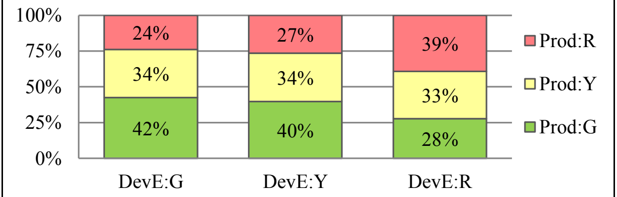
l) Prob(Code Review to Code Ratio | Code Review Rate)



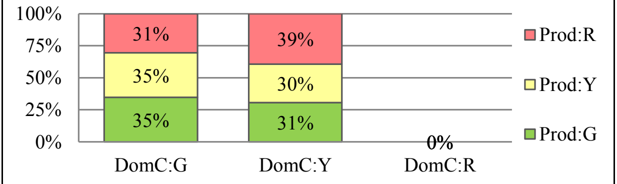
m) Prob(Productivity | Prog. Language Abstraction Level)



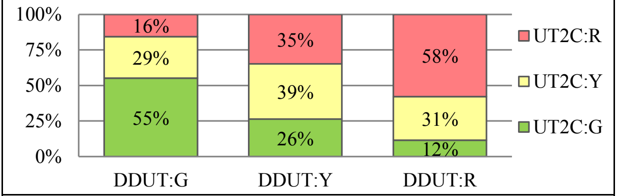
n) Prob(Productivity | Developer Experience)



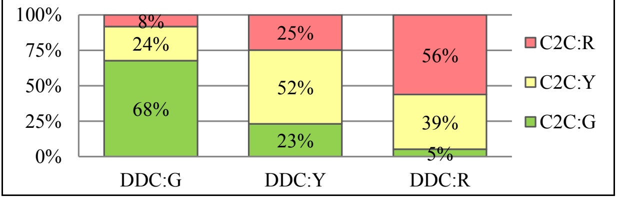
o) Prob(Productivity | Domain Complexity)



r) Prob(Unit Test to Code Ratio|Defect Dens. in Unit Test)



s) Prob(Compile to Code Ratio | Defect Density in Compile)


DDC:G
DDC:Y
DDC:R
C2C:R
C2C:Y
C2C:G