A controlled experiment on the effects of PSP training: Detailed description and evaluation

Lutz Prechelt (prechelt@ira.uka.de)
Barbara Unger (unger@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/608-7343
http://wwwipd.ira.uka.de/EIR/

Technical Report 1/1999

April 8, 1999

Abstract

The Personal Software Process (PSP) is a methodology for systematic and continuous improvement of an individual software engineer's software production capabilities. The proponents of the PSP claim that the PSP methods improve in particular the program quality and the capability for accurate estimation of the development time, but do not impair productivity.

We have performed a controlled experiment for assessing these and related claims. The experiment compares the performance of a group of students that have just previously participated in a PSP course to a comparable set of students from a "normal" programming course. This report presents in detail the experiment design and setup, the results of the experiment, and our interpretation of the results.

The results indicate that the claims are basically correct, but the improvements may be a lot smaller than expected. However, we found an important additional benefit from PSP that is not usually mentioned by the PSP proponents: The performance in the PSP group was consistently less variable for most of the many variables we investigated. Less variable performance in a software team greatly reduces the risk in software projects.

Contents

One item could not be deleted because it was missing.
Apple Macintosh System 7 OS error message

Chapter 1

Introduction

Everybody has opinions. I have data.
Watts S. Humphrey

The present report is the definitive and detailed description and evaluation of a controlled experiment comparing students that received PSP (Personal Software Process) training to other students that received other software engineering education.

In the first chapter we will first discuss the general topic of the experiment, then give a broad overview of the purpose and setup of the experiment, and finally describe related work.

Chapter 2 describes the subjects, setup, and execution of the experiment, relying partially on the original experiment materials as printed in the appendix. It also discusses possible threats to the internal and external validity of the experiment.

Chapter 3 presents and interprets in detail the results obtained in the experiment and Chapter 4 presents conclusions. The appendix contains the handouts used in the experiment: personal data questionnaire, estimation questionnaire, task description, work time logging sheet, postmortem questionnaire.

1.1 What is the PSP?

The Personal Software Process (PSP) is a methodology for structuring the work of an individual software engineer introduced by Watts Humphrey in 1995 [3]. At its core is the notion of an individual's software process, that is, the set of procedures used by a single software engineer to do his or her work. The PSP has several goals:

- **Reliable planning capability**, i.e., the ability to accurately predict the delivery time of a piece of work performed by a single software engineer.
- **Effective quality management**, i.e., the ability to avoid introducing defects or other quality-reducing properties into the work products, to detect and remove those that have been introduced anyway, and to improve both capabilities over time. These other quality attributes can be, for instance, the ease of maintenance, reuse, or testing (internal view of the product), or the suitability, flexibility, and ease of use (external view), etc.
- **Defining and documenting the software process**, i.e., laying down in writing the abstract principles and concrete procedures by which one generally creates software. The purpose of process definition is improving the ability of the process to be traced, understood, communicated, measured, or improved.

- **Continuous process improvement**, i.e., the capability to continuously identify the relatively weakest points in one's own software process, develop alternative solutions, evaluating these solutions, and incorporating the best one into the process subsequently.

The PSP may also lead to improved productivity but this is not a primary goal; part of the productivity gains may be offset by the overhead introduced by the PSP, because the means to the above ends are process definition, process measurement, and data analysis, which lead to a number of additional tasks.

The PSP methodology is taught by means of a PSP course. In its standard form, this is a 15-week training program requiring roughly one full day per week. According to the experience of both Watts Humphrey and ourselves, the PSP can hardly be learned without that course, because under the pressure of real-world working conditions, programmers will only be able to accept and execute the overhead tasks once they have experienced their benefits — but the benefits will only be experienced after the overhead tasks have been performed for quite a while. Hence, the course is needed for providing a pressure-free "playground" for learning about the usefulness of PSP techniques.

The practical consequence of the PSP course for an individual software engineer is to obtain a personal software process (*psp*, in non-capital letters). The course provides a set of example methods that serve as a starting point for the development of an individual *psp*. The methods are reasonable default instantiations of the PSP principles and can be tailored to one's individual preferences and work conditions during later *psp* usage.

1.2 Experiment overview

The question asked by this experiment is the following:

> What (if any) differences in behavior or capabilities can be found when comparing software engineers that have received PSP training to software engineers that have received an equivalent amount of "conventional" technical training?

The approach used to answer this question is the following:

- Find participants with similar capabilities and backgrounds, except that one group has had previous PSP training and the other has not.
- Let each participant solve the same non-trivial programming task.
- Observe as many features of behavior (process) and result (product) as possible.
  Examples: total working time, number of compilations, program reliability when program was first considered functional (acceptance test), final program reliability, program efficiency, etc.
- Formulate hypotheses describing which differences might be expected between the groups with respect to the features that were observed.
  Example: PSP-trained participants produce more reliable programs.
- Analyze the data in order to test the hypotheses. Describe additional interesting structure found in the data, if any. Interpret the results.

In our case, the participants were graduate students and the programming task involved designing and implementing a rather uncommon search and encoding algorithm. On average, the task size was effectively more than 1 person day (between 3 and 50 work hours). A total of 55 persons participated in the experiment between August 1996 and October 1998.

1.3 Related work

It is one of the most important principles of the PSP methodology to base decisions on objective measurement data (as opposed to intuitive judgement). Consequently, the PSP course (and also later practitioning of the PSP) builds a collection of data for each participant from which the development of several attributes of process quality and effective developer capability can be seen. Such data has been described and discussed by Humphrey in several articles and reports, e.g. [4]. Most of this data show the development of a certain metric over time, such as the decreasing density of defects inserted in a program. The main and unavoidable drawback of such data is a lack of control: It is impossible to say how much of the effect comes from each of the possible sources, such as: the particular programming problem solved at each time, maturation that would also occur without PSP training (at least given some other training), details of the measurement that cannot be made objective, and, finally, real and unique PSP/*psp* benefits.

The purpose of the present experiment is providing data with a much higher degree of comparability: measuring *psp* effects in a controlled fashion.

We know of no other evaluation work specifically targeting the PSP methodology or the PSP course.

1.4 Why such an experiment?

There are basically two reasons why we need this experiment. First, any methodology, even one as convincing as the PSP, should undergo a sound scientific validation. Not only to see whether it works, but rather to understand the structure of its effects: Which consequences are visible at all? How strong is their influence? How do they interact?

The second reason is more pessimistic: Based on our observations with about a hundred German informatics students we estimate that only about one third of them will regularly use PSP techniques in their normal work after the course and will really form a *psp*. Roughly another third appears to be unable to regularly exercise the self-control required for building and using a *psp*. For the rest, we consider the prospects to be unclear; their PSP future may depend on the kind of environment in which they will work.

Given this estimation it is not clear whether PSP-trained students will be superior to others, even if one is willing to believe that a PSP education *in principle* has this effect. The purpose of the experiment is to assess the average effect as well as look for the results of the above-mentioned dichotomy, if any. If it exists, we may for instance see a larger variance of performance in the PSP group or maybe even a bimodal distribution having two peaks instead of just one.

1.5 How to use this report

This report is meant to provide a most detailed documentation of the experiment and its results. This has several consequences:

- You need not read the whole report from front to back. The contents are logically structured and it should be easy to find specific information of interest using the table of contents.
- When you encounter a term whose definition you have skipped, refer to the table of contents or the glossary for finding it.

- The main text does not try to describe the tasks or questionnaires in any detail but instead relies on the original experiment materials that are printed in the appendix. Please consult the appendix where necessary.

- The results section is rather detailed. We recommend to stick to the text and to refer to the diagrams and their captions only at points of particular interest.

Chapter 2

Description of the experiment

Good judgement comes from experience,
and experience comes from bad judgement.
Anonymous

This chapter will describe the experiment design, the hypotheses to be investigated by the experiment, the experiment procedure, the motivation and background of the participants, and the task to be solved. In the final sections we will discuss possible threats to the internal and external validity of the experiment.

2.1 Experiment design

As mentioned before, the general goal of the experiment is to investigate differences in performance or behavior between

- persons that have received PSP training shortly before and
- "other" people.

The resulting basic experimental design is very simple: it is a two-group, posttest-only, inter-subject design with a single binary independent variable, namely "subject[1] has received PSP training". We will call the two groups P (for "PSP-trained") and N (for "not PSP-trained"). The set of dependent variables to be used is not at all simple, though. To avoid unnecessary repetition, these variables will be introduced step by step during the presentation of the results in Chapter 3.

In order to maximize the power of the experiment in view of the large variations in individual performance that are to be expected, one would ideally want to pair participants with similar expected performance (based on available knowledge about the participants' background) and put one person of each pair into each group. Unfortunately, this is not possible in the present setting because group membership is determined by the previous university career of each participant that can not be controlled by the experimenter. See also the following section.

[1] We will use the terms "subject" and "participant" interchangeably to refer to the persons who participated in our experiment as experimental subjects.

2.2 Hypotheses

As mentioned in the overview in Section 1.2, the general purpose of the experiment is investigating behavior differences (and their consequences) between PSP-trained and non-PSP-trained subjects. To satisfy such a broad ambition, we must collect and analyze data as comprehensively as is feasible (from both a technical/organizational and a psychological point of view).

However, to guide the evaluation of this data it is useful to formulate some explicit expectations about the differences that might occur. These expectations are formulated in this section in the form of hypotheses. The experiment shall investigate the following hypotheses:

Hypothesis H1: Estimation. Since effort estimation is a major component of the PSP course, we expect that the deviations of actual from expected development time are smaller in group P compared to group N.
(Note that this hypothesis assumes that the task can be considered to be from a familiar domain, because otherwise the PSP planning may break down and the results become unpredictable.)

Hypothesis H2: Reliability. Since defect prevention and early defect removal are major goals throughout the PSP course, we expect the reliability of the program for "normal" inputs to be higher in group P compared to group N.

Hypothesis H3: Robustness. Since producing defect-free programs is an important goal during the PSP course, we also expect the reliability of the program for "surprising" sorts of inputs to be higher in group P compared to group N.
(Note that the term "robustness" is somewhat misleading here because robustness against *illegal* inputs was explicitly *not* a requirement in the experiment.)

Hypothesis H4: Release maturity. We expect that group P will typically deliver programs in a relatively more mature state compared to group N. When the requirements are invariant, release maturity can be represented by the fraction of overall development time that comes after the first program release.
(In the experiment, releasing a program will be represented by the request of the participant that an acceptance test be performed with the program.)

Hypothesis H5: Documentation. Since the PSP course puts quite some weight onto making a proper design and design review and onto product quality in general, we expect that there will be more documentation in the final product in group P compared to group N.

Hypothesis H6: Trivial mistakes. Quality management as taught in the PSP course is based on the principle to take even trivial defects seriously, hence we expect a lower number of simple-to-correct defects in group P compared to group N.

Hypothesis H7: Productivity. For programs that are difficult to get right, the PSP focus on early defect detection saves a lot of testing and debugging effort. The overhead implied by planning and defect logging usually does not outweigh these savings. What might outweigh the savings, though, is if participants produce a thorough design documentation that accompanies the program, e.g. in the form of long comments in the source code. Such behavior is also expected to be more likely for PSP-trained participants. Note that since all programs were built to conform to the same requirements, productivity can be defined simply as 1 divided by the total work time. Now for the actual hypothesis: We expect group P to complete the program faster compared to group N, at least if one subtracts the effort spent for documentation.

Hypothesis H8: Quality judgement. The PSP quality management tracks the density of defects in a program as seen during development and even as predicted before development starts. We speculate that this might also lead to more realistic estimates of the defect content and reliability of a final program. Hence, we expect to see more accurate estimates of final program reliability in group P compared to group N.

Speculative hypothesis H9: Efficiency. PSP quality management suggests to prefer simpler solutions over "clever" ones. This might lead to programs that run more efficient or use less memory, but might also lead to programs that run more slowly or require more memory. We hypothesize that we may find differences in speed and memory consumption between groups P and N, although we cannot say in advance which form these differences will have.

Speculative hypothesis H10: Simplicity. The more careful design phase presumably performed by PSP-trained subjects might lead to simpler and shorter code in group P compared to group N.

Note that this set of hypotheses is quite bold and the odds are not the same for each hypothesis: For estimation and reliability, for example, we clearly expect to see some advantage of the P group, while for efficiency expecting any differences is pure speculation. The results for documentation and productivity must be treated with care, because the experiment instructions first and foremost called for reliability, not for productivity or maintainability.

All of these hypotheses are expected to hold more strongly if we look at only the better half of the participants in each group, because then, according to the remark in Section 1.4, the PSP group presumably consists mostly of subjects that indeed use a *psp*. In Section 3.2 on page 25, we will form various subgroups for such comparisons.

As mentioned above, the purpose of the experiment is not limited to formally testing these hypotheses, but includes all other appropriate analyses that might provide insights towards understanding the behavior differences and their effects. Consequently, we will use the hypotheses somewhat loosely in order not to be distracted from possibly more important other observations.

2.3 Experiment format and conduct

After some trial runs in August 1996, the experiment started in February 1997 and finished in October 1998. With a few exceptions, the participants worked during the semester breaks from mid-February to mid-April or mid-July to mid-October. Subjects announced their participation by email and agreed on an appointment, usually starting at 9:30 in the morning.

Each subject could choose the programming language to be C, C++, Java, Modula-2 or Sather-K; two subjects would have preferred Pascal. The compilers used were gcc, g++, JDK, mocka, and sak, respectively. An account was set up for each subject on a Sun Unix workstation. At most three such accounts were in use at any given time. The account was setup so as to protocol activity, in particular each version of the source code that was submitted to the compiler. A subject was told about this monitoring on request. Due to a mistake made by one experimenter (Prechelt) when adapting the setup of the participant accounts to a change in the software environment of our workstations (switch to a new version of the Java development kit), the monitoring mechanism was corrupted and nonfunctional during a significant part of the experiment and many of these compilation protocols were lost. In order to provide as natural a working environment as possible and avoid irritating the participants, no direct observation, video recording, or other kind of visible monitoring was performed.

A subject was allowed to fetch and install auxiliary tools or data by FTP and install it for the experiment. For instance, a few subjects brought their own editor or re-used small parts of previously written programs (e.g. file handling routines). Some, but not all, PSP subjects brought their estimation data and/or PSP tools.

The subject was then given the first three parts of the experiment materials: first the personal information questionnaire, then the task description, and then the estimation questionnaire (see Appendix A). After filling in the latter, the subject was left alone and worked according to a schedule he could choose freely. Only three restrictions were made: First, to always work in this special account; second, to use but one source code file;

and third, to log the working time on a special protocoling sheet we had provided along with the other materials. Each of these restrictions was violated by a few (but not many) of the participants.

The subject was told to ask if he encountered technical problems with the setup or if he felt something was ambiguous in the requirements. Technical problems occurred frequently and were then resolved on the spot. In order not to bias the results, the time required for resolving the problems (between 5 and 30 minutes per participant) is included in the work time. Inclusion is required because some subjects may have chosen to resolve the problem alone. Questions about the requirements were asked by about a dozen participants and in all cases they were referred to read the requirements more closely, because the apparent ambiguity was indeed properly resolved in the description they had received.

The participant was asked not to cooperate with other participants working at the same time or with earlier participants; we have not found any evidence of such cooperation and believe that essentially none has occurred. The subject was further told to ask for an acceptance test at any time if he felt his program would now work correctly.

If the acceptance test failed, the subject was encouraged to analyze the problems, correct the program, and try again. Once the acceptance test was passed (or the subject gave up), the participant was given the postmortem self-evaluation questionnaire. After finishing and returning the questionnaire, the subject was paid (see Section 2.4.1) and all data was copied from the subject's experiment account to a safe place. In a few cases, the subjects asked for (and were granted) some additional time for improving the program after the acceptance test was passed. Typically this was used for cleaning up unused code sections and inserting comments into the program.

2.4 Experimental subjects

This section will describe the background of the experimental subjects. Often in this report we refer to a few of the subjects individually by name: these names are letter/number-combinations in the range from s12 to s102.

2.4.1 Overview

The experiment had 50 participants, 30 of them in the PSP group and 20 in the control group. Our 50 participants break up into the following sorts with respect to their motivation: 40 of them were obliged to participate in the experiment as a part of a lab course they took (29 from two PSP courses and 11 from a Java course, see the description in Sections 2.4.3 and 2.4.4). Note that the obligation included only participation, not success: Even those participants that gave up during the experiment passed their course. One PSP participant, s045, retracted all of his materials from the experiment and will not be included in any of the analyses.

The other 10 participants were volunteers. 8 participants were volunteers from other lab courses in our department. Four of these were highly capable students who expressedly came to "prove that PSP-trained people are not better than others". (s020, s023, s025, and s034. Interestingly, two of these four, s020 and s034, later participated in the PSP course.) 2 participants (s081, s102) are actually "repeaters": they had already participated in the experiment in February/March 1997 as non-PSP subjects and voluntarily participated again in June 1998 and March 1998, respectively, after they had taken the PSP course. Quite obviously, these differences need to be taken into account during data analysis; see Section 3.2 for a description of the actual groups compared in the analysis.

All participants were motivated towards high performance by the following reward system: They would be paid DM 50 (approximately 30 US dollars) for successfully participating in the experiment (i.e., passing the acceptance test). However, each failed acceptance test reduced the payment by DM 10.

2.4.2 Education and experience

The following information excludes the two repeater participants, the one participant from the PSP group who retracted all of his materials from the experiment, and those participants for which the particular piece of information was not available ("missing").

The participants were in their 4th to 17th semester at the university (median 8, see Figure 2.1; please refer to Section 3.1.1 on page 20 for an explanation of the plot). With the exception of two highly capable fourth semesterers (s023 and s050) all of the students were graduate students (after the "Vordiplom").

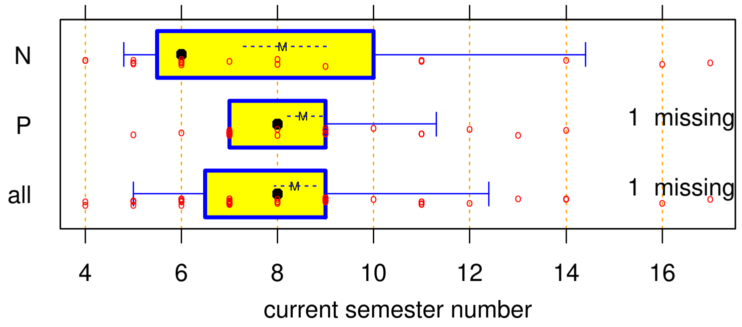


Figure 2.1: Distribution of semester number of subjects in the PSP group (P), the non-PSP group (N), and both together (all).

The participants had a programming experience of 3 to 14 years (median 8 years, see Figure 2.2) with an estimated 10(sic!) to 15000(sic!) actual programming working hours *beyond* the programming exercises performed in the university education (median 600 hours, see Figure 2.3).

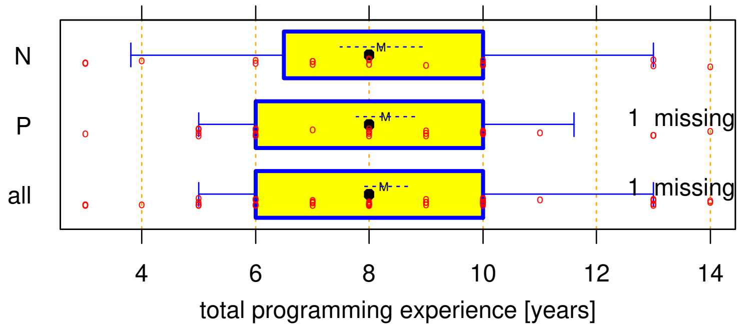


Figure 2.2: Distribution of years of programming experience in the PSP group (P), the non-PSP group (N), and both together (all)

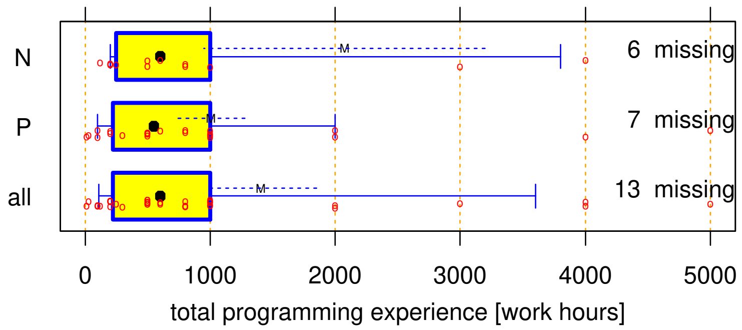


Figure 2.3: Distribution of hours of non-university programming experience in the PSP group (P), the non-PSP group (N), and both together (all). There is one point at 15000 in N.

During that time, each of them had written an estimated total 4 to 2000(sic!) KLOC[2] with a median of 20, see Figure 2.4. The estimated total number of lines the subjects had written in the language they had chosen for

[2]One KLOC is thousand lines of code. For roughly half of the participants this includes comments, for the others it includes only statements.

the experiment was from 0.5 to 100 KLOC (median 5 KLOC, see Figure 2.6). The few extremely high values that occur in most of these variables show that there are a few quite extraordinary subjects in the sample, in particular in the non-PSP group.

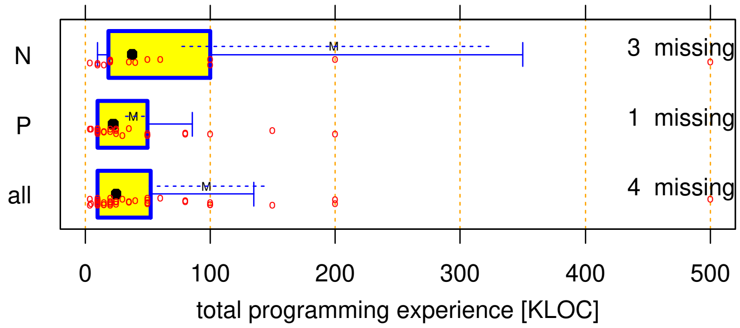


Figure 2.4: Distribution of total KLOC written in the PSP group (P), the non-PSP group (N), and both together (all). There is one point at 2000 in N.

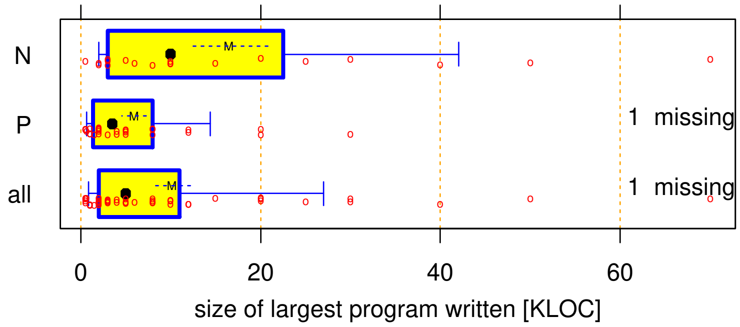


Figure 2.5: Distribution of size of largest program written in the PSP group (P), the non-PSP group (N), and both together (all).

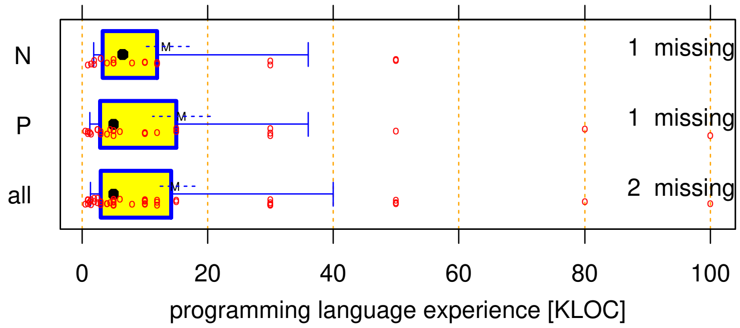
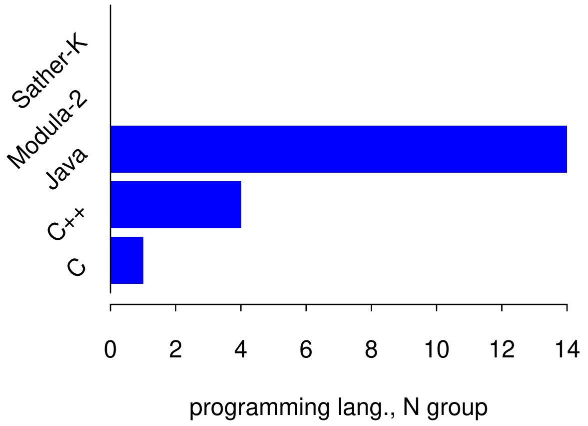


Figure 2.6: Distribution of programming experience (in KLOC) in the programming language used during the experiment by each individual subject in the PSP group (P), the non-PSP group (N), and both together (all).

Looking at these data, our two main groups appear to be reasonably balanced. The apparently somewhat larger values in the N group for total experience in KLOC and size of largest program may be spurious, because the non-PSP participants are more likely to over-estimate their past productivity as we will see in Section 3.3.

There are two subjects (s034 and s043) that appear among the top three 5 times and 3 times, respectively, for the 5 programming experience measures; both are in the N group.

Note that the distribution of programming languages differs between the N and P group, see Figure 2.7. There is a relatively larger fraction of Java users in the N group.



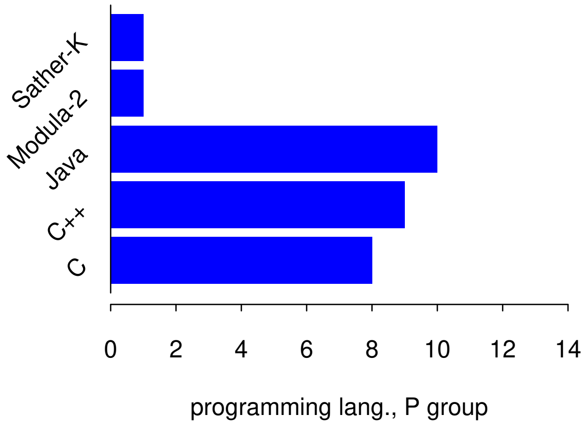


Figure 2.7: Number of participants in the N group (left) and the P group (right) that have used each programming language.

2.4.3 The PSP course (experiment group)

The PSP methodology is taught by means of a PSP course. In its standard form and as taught in our department, this is a 15-week training program consisting of 15 lectures of ninety minutes each, 10 programming exercises, and 5 process exercises. This course requires roughly one full day per week.

Each exercise is submitted to a teaching assistant who carefully checks the correctness of the materials (with respect to process) and marks any problems s/he finds. The materials are handed back to the students who have to resubmit them in corrected form until everything is OK. The focus of the course is on the following topics: working towards (and then according to) a well-defined, written-out software process, learning and using systematic planning and estimation based on personal historical data, and defect prevention and removal by means of managed personal reviews, defect logging, defect data analysis, and performance-data-controlled process changes.

The participants of the P group are from the second and third time we taught the PSP course. For the most part, we used the agenda defined by Humphrey in his book [3] on page 746. The participants needed to submit all exercises in good shape (with resubmissions if corrections were necessary) in order to pass the course. Those few participants who did not pass the course dropped out voluntarily during the semester, nobody was explicitly expelled.

2.4.4 The alternative courses (control group)

The volunteers of the N group (s014, s017, s020, s023, s025, s028, s031, s034) came from various other lab courses.

The non-volunteers of the N group all came from an advanced Java course ("component software in Java"); many of them had previously also participated in a basic Java course we had taught the year before. The course was following a compressed schedule such that the course ran over only 6 weeks of the 13-week semester, but required a very high time investment during that time. In terms of the total amount of code produced, this course is quite similar to the PSP course, although it had only 5 larger exercises instead of 10 smaller ones. The course content was highly technical, covering the then-new Swing GUI classes, localization and internationalization, serialization and persistence, reflection and JavaBeans, and distributed programming (Remote Method

Invocation). The programs were submitted to the course teachers and tested in a black-box fashion. Participants needed to score 70% of all possible points (based on correctly implemented functionality) in order to pass the course.

2.5 Task

This section will shortly describe the task to be solved in the experiment and will explain why we chose it. For details, please refer to the original task description on page 66 in the appendix.

2.5.1 Goals for choosing the task

The tasks to be used in our experiment should have the following properties:

1. *Suitable size.* Obviously, the task should not be too small in order to provide enough and interesting data. A trivial task would have too little generalizability. The task was planned to take about 4 or 5 hours for a good programmer, so that most participants would be able to finish within one day, when they started in the morning. (It later turned out that only 28% of the participants were able to finish on the day they started and 46% took more than two days.)
2. *Suitable difficulty.* Most if not all of the participants should be able to complete the task successfully. In particular, it must be possible to solve the task without inventing an algorithm or data structure that requires high creativity.
   Furthermore, the application domain of the task had to be well understandable by all subjects.
   On the other hand, it must be possible to make subtle mistakes or ruin the efficiency so that there can be sufficient differences in the work products among even the successful solutions.
3. *Automatic testability.* In order to test the quality of the solutions thoroughly and objectively it must be possible to run a rather large number of tests without human intervention. In particular, the acceptance test should be automatic and entirely objective.

2.5.2 Task description and consequences

From these requirements, we chose the following task:

> Given a list of long "telephone numbers" and a "dictionary" (list of words), encode each of the telephone numbers by one word or a sequence of multiple words in every possible way according to a fixed, prescribed letter-to-digit mapping. A single digit may stand for itself in the encoding between two words under certain circumstances.
>
> Read the phone numbers and the dictionary from two files and print each resulting encoding to standard output in an exactly prescribed format.

Please see the exact task description on page 66 for the details and for input/output examples.

The above functionality can be coded in about 150 statements with any programming language that has a reasonable string handling capability. Understanding the requirements exactly and producing an appropriate search algorithm is not trivial, but certainly within the capabilities of the participants. Various details give enough room for gross or subtle mistakes, e.g. handling special characters allowed in the phone numbers (slash, dash) or the words (quote, dash), always producing the correct output format, or handling all cases of digit-insertion correctly. The algorithmic nature of the problem is simple to understand for all subjects, regardless of specific backgrounds, and the search algorithm gives room for enormous differences in the resource consumption (both space and time) of the resulting program. The batch job character of the program makes automatic testing possible and the simple structure of the input data even allows for fully automatic generation of test cases once a correct "gold" program has been implemented. This allowed for generating new data for each acceptance test on the fly.

During the evaluation of the experiment it turned out that the differences among the solutions were even larger than expected.

2.5.3 Task infrastructure provided to the subjects

Along with the task description, the following task-specific infrastructure was provided to the participants:

- the miniature dictionary `test.w` and the small phone number input file `test.t` used in the example presented in the task description,
- a file `test.out` containing the correct output for these inputs,
- a large dictionary called `woerter2` containing 73220 words.

The same large dictionary was also used during the evaluation of all programs presented in this report; a fact which we told the participants upon request.

2.5.4 The acceptance test

The acceptance test worked as follows: For each test, a new set of 500 phone numbers was created and the corresponding correct output computed using the gold program. This took only a few seconds. The dictionary used was a random but fixed subset of 20946 words from the `woerter2` dictionary.

Then the candidate program was run with these inputs and the outputs were collected by an evaluation Perl script. This script matches the outputs of the candidate program to the correct outputs and computes the reliability of the candidate program. The evaluation script stops the candidate program if it is too slow: an accumulative timeout of 30 seconds per output was applied plus a 5 minute bonus for loading the dictionary at the start. This means that, for instance, the 40th output must be produced before 25 minutes of wall clock time are over or else the program will be stopped and its reliability judged based on the outputs produced so far. Many programs did indeed run for half an hour or more during the acceptance test; the number of expected outputs varied from 25 to 248 with a typical range of 40 to 80.

At the end of the acceptance test the following data was printed by the evaluation script: The sorted actual output of the candidate program, the sorted expected output (i.e., the output of the gold program), a list of differences in the form of missing correct outputs and additional incorrect outputs, and the resulting reliability in percent.

The exact notion of reliability will be defined in Section 3.4.3 under the name of *output reliability*. A minimum output reliability of 95 percent was required for passing the acceptance test. However, in their final acceptance test with only two exceptions all programs either achieved 100 percent or failed entirely.

2.5.5 The "gold" program

Given this style of acceptance test, the before-mentioned "gold" program obviously plays a rather important role in this experiment. The gold program was developed by Lutz Prechelt together with the development of the requirements. The initial requirements turned out to be too simple, so the rules for allowing or forbidding digits in the encoding were made up and added to the requirements during the implementation of the gold program.

The gold program, called `phonewrd`, was developed using a *psp* in July 1996 during three sessions of about six hours total. The total time splits into 126 minutes of design (including global design, pseudocode development, and test development), 93 minutes of design review, 72 minutes of coding, 38 minutes of code review, and 19 minutes compilation. The program ran correctly upon the first attempt and no defect was ever found after completion — this despite numerous claims of participants that "the acceptance test program is wrong. My program works correctly!". 19 defects were originally introduced in the design and pseudocode, 11 of which were found during design review, 6 defects were introduced during coding and found during code review or compilation. These values include trivial mistakes such as syntactical errors. The program was written in C with refinements[3], which turned out to be a superbly suitable basis for this problem.

The initial program was only modestly efficient (trying 10% of the dictionary for each digit). A few days later it was improved into a more efficient version (called `phonewrd2` trying only 0.01% of the dictionary for each digit) which also worked correctly right from the start. This second version was used throughout the experiment.

2.6 Internal validity

There are two sources of threats to the interval validity of an experiment[4]: Insufficient control of relevant variables or inaccurate data gathering or processing.

2.6.1 Control

Controlling the independent variable means holding all other influential variables constant and varying only the one under scrutiny. Controlling the dozens of possibly relevant variables in human-related experiments is usually done by random sampling of participants into the experiment groups and subsequent averaging over these groups: variation in any other than the controlled variable is then expected to cancel out.

However, random sampling is difficult for software engineering experiments, because they require such a high level of knowledge. The problem becomes particularly pronounced if the independent variable is a specific difference in education: neither can we randomly sample from a large group of possible participants, nor can we freely assign each of them into a group chosen at random. Instead, we are confined to a small number of available subjects with the proper background and, worse yet, typically each of them fits into only one of the groups, because we cannot impose a certain education on the subjects and withhold the other; they choose themselves what they want to learn.

In the given experiment this means that the preferences that let the subjects choose one course and not the other could in principle be related to the results observed. We cannot prove that there is no such effect, but based on our personal knowledge of the individuals in both courses, we submit that we cannot see any severe difference in their average capabilities. In fact, because they liked us as teachers, several participants from the PSP course later also took the other course and vice versa.

[3] http://wwwipd.ira.uka.de/~prechelt/sw/#crefine

[4] Definition from [1]: "*Internal validity* refers to the extent to which we can accurately state that the independent variable produced the observed effect."

2.6.2 Accuracy of data gathering and processing

Inaccurate data gathering or processing is unlikely as there was very little manual work involved in this respect. Instead, most data gathering and almost all data processing was automated. The scripts and programs were carefully developed and tested and their results again scrutinized. Many consistency checks were applied for detecting possible mistakes. One remaining problem is missing data, which is almost inevitable in any large scale experiment. The consequences of missing data, if any, will be discussed for each dependent variable in the results sections below.

2.7 External validity

Three major factors limit the generalizability (external validity) of this experiment: Longer experience as a software engineer, longer experience with *psp* use, and other kinds of work conditions or tasks. (You may perhaps want to skip the rest of this section until you have seen the results and their discussion.)

2.7.1 Experience as a software engineer

One of the most frequent critiques applied to controlled experiments performed with student subjects is that the results cannot be transferred to software professionals.

The validity of this critique depends on the actual task performed in the experiment: if the task requires very specific knowledge such as the capability to properly use complex tools or esoteric notations or uncommon processes, then the critique is probably valid. If, on the other hand, only very general software production abilities are required in the task, a graduate student group performs not much different from a group of professionals: It is known that experience is a rather weak predictor of performance within a group of professionals and in fact soon these same students will be professionals themselves.

The task in this experiment is very general, requiring only knowledge that is taught during undergraduate university education. Hence, we can expect to find relatively little difference in the behavior of our student subjects compared to professionals. We can imagine only two differences that might be relevant.

First, professionals taking a PSP course after some professional experience may often be much more motivated towards actually using PSP techniques, because due to previous negative experiences they have a much clearer conception of the possible benefits than students. Student background often involves only relatively small projects with comparatively little schedule pressure and little need for relying on the work of colleagues. This motivation difference, if present, should pronounce the differences between PSP and non-PSP groups for professionals.

Second, some of the less gifted students may later pick a non-programming job, resulting in a sort of clean-up (smaller variance of performance in the lower part) in a group of professionals compared to a group of students. The possible consequences of this effect, if it exists, on the difference between PSP and non-PSP groups are unclear.

2.7.2 Experience with *psp* use

The present experiment investigates performance and behavior differences shortly after a PSP course. One should be rather careful when generalizing these results to persons that have been using a *psp* for some longer time, say, two years.

It is plausible that in those cases where differences between the PSP group and the non-PSP group were found, these differences will become more pronounced over time. However, for a PSP-adverse work environment it is also conceivable that differences wear off over time (because PSP techniques are no longer used) and it is unclear whether differences may emerge over time where none have been observed in the experiment.

It would definitely be important to run a similar experiment much longer after a PSP (or other) training.

2.7.3 Kinds of work conditions or tasks

As mentioned above, it is plausible that differences due to PSP training may be reduced by a working environment that discourages the sort of data gathering implied by PSP techniques; the level of actual PSP use may just drop. Inversely, the effects might also become more pronounced for instance if the tasks are very difficult to get right, if the work conditions demand accurate communication of technical decisions, or if accurate planning can reduce the stress due to schedule pressure.

Furthermore, some of the PSP participants may have taken the experiment task too lightly and may have under-used their *psp* in comparison to their standard professional working behavior. For instance, some of them did not bring their PSP tools or PSP estimation data.

All of this is unknown, however, so adequate care must be exercised when applying the results of this experiment to such different situations.

Chapter 3

Experiment results and discussion

I don't know the key to success,
but the key to failure is to please everybody.
Bill Cosby

This chapter presents and interprets the results of the experiment. The first section explains the means of statistical analysis and result presentation that we use and explains why they were chosen. Section 3.2 describes how, exactly, the groups to be compared in the analysis were derived from the raw groups of PSP and non-PSP subjects. The following sections present the results (objective performance); the analysis is organized along the hypotheses listed in Section 2.2. (Warning: The amount of detail in the diagrams and captions may overwhelm you. The main text, however, is short and easy to read.) Two final sections describe findings from an analysis of correlations between variables and findings from analyzing the answers of the subjects given in the postmortem questionnaire.

3.1 Statistical methods

In this section we will describe the individual statistical techniques (including the graphical presentations) used in this work for assessing the results. For each technique, we will describe its purpose, the meaning of its results, and its caveats and limitations. The analyses and plots were made with S-Plus 3.4 on Solaris and we will shortly indicate the names of the relevant S-Plus functions in the description as well.

3.1.1 One-dimensional statistics

For most of this report, we will simply compare the values of a single measurement for all of the participants in the PSP group against the participants in the non-PSP group. The simplest form of such a comparison is comparing the arithmetic **mean** of the values in one group against the mean of the values in the other. However, the mean can be very misleading if the data contains a few values that are very different from the rest. A more robust basis for a comparison is hence the **median**, that is, the value chosen such that half of the values in the group are smaller or equal and the other half are greater or equal. In contrast to the mean, the median is not influenced by *how far* away from the rest the most extreme values are located.

Possibly we are not only interested in the average behavior of the groups, but also in the variation (variability, variance) within each group. In our context, smaller variation is usually preferable, because it means more predictable software development. One way of assessing variation is the **standard deviation**. If the underlying data follows a normal distribution (the Gaussian bell curve), about two thirds of the data (68%) will lie within plus or minus one standard deviation from the mean. However, if the data does *not* follow a normal distribution, the standard deviation is plagued by the same problem as the mean: a few far-away values will influence the result heavily — the resulting standard deviations can be very misleading. Software engineering data often has such values and hence the standard deviation is not a reliable measure of variation. Instead, we will often use the interquartile range and similar measures we will explain now in a graphical context.

A flexible and robust way of comparing two groups of values for both average (statisticians speak of "location") and variation (called "spread") is the **boxplot**, more fully called box-and-whisker plot (S-Plus: `bwplot()`). You can find an example in Figure 3.4 on page 28. The data for the PSP group is shown in the upper part, the data for the non-PSP group in the lower part. The small circles indicate the individual values, one per participant. Only their horizontal location is important, the vertical "jittering" was added artificially to allow for discriminating values that happen to be at the same horizontal position. The width and location of the rectangle (the "box") and the T-shaped lines on its left and right (the "whiskers") are determined from the data values as follows. The left edge of the box is located such that 25% of the data values are less than or equal to its position, the right edge is chosen such that 75% of the data values are less than or equal to its position (which means that 25% are greater or equal that value). The position of the left edge is called the **25-percentile** or **25% quantile** or **first quartile**, the right edge is correspondingly called the 75% quantile or third quartile. Similarly, the left and right **whiskers** indicate values such that exactly 10% of the values are smaller or equal (left whisker, 10-percentile) or 10% are larger or equal (right whisker, 90-percentile), respectively. The **fat dot** within the box marks the median, which could also be called 50-percentile, 50% quantile, or second quartile. Note that different percentiles can be the same if there are several identical data values (called **ties**), so that, for instance, whiskers can be missing because they are identical with the edge of the box or the median dot can lie on an edge of the box etc.

Boxplots allow for easy comparison of both spread and location of several groups of data. One can concentrate either on the width of the boxes (called the **inter-quartile range** or **iqr**) or the width of the whole boxplots for comparing spread or can concentrate on particular box edges or the median dots for comparing different aspects of location (namely the location of the **lower half**, **upper half**, or **middle half** of the data points).

Note that for distributions that have only few distinct values (typically all small integers) and therefore contain many ties, differences in the width of the box or the location of any of the quartiles can be misleading because it may change a lot if only a single data value changes. Figure 3.26 on page 41 shows a simple example. The two distributions are similar, but the boxplots look quite different. A similar caveat applies when the number of data values plotted is small, e.g. less than ten.

Our boxplots have one additional feature: the letter M in the plot indicates the location of the mean and the dashed line around it indicates a range of plus or minus one **standard error of the mean**. The latter quantifies the uncertainty with which the mean is estimated from the data and decreases with decreasing standard deviation of the data and with an increasing number of data points. For about 68% of all data samples of the given size taken from the same population, the sample mean will lie within this standard error band. For symmetric distributions, the mean is equal to the median. However, in our data, many distributions are **skewed**, i.e., the data is less dense on one end of the distribution than on the other. In this case, the mean will lie closer to the less dense end than the median.

A string such as "**2 missing**" (e.g. on the left edge of Figure 3.5) indicates that two of the data points in the sample had missing values and hence are not shown in the plot.

When comparing two distributions, say in a boxplot, it is often unclear whether observed differences in location should be considered accidental or real. This question can be assessed by a **statistical hypothesis test**. A test computes the probability that the observed differences of, say, the mean will occur when the underlying distributions in fact have the same mean.

The classical statistical test for comparing the means of two samples is the **t-test** (S-Plus: `t.test()`). Unfortunately, this test assumes that the two samples each come from a normal distribution and that these distributions have the same standard deviation. Most of our data, however, has a distribution that is non-normal in many ways. In particular, our distributions are often unsymmetric. For such data, the t-test may produce misleading results and should thus not be used. Sometimes asymmetric data can be transformed into normally distributed data by taking e.g. the logarithm and the t-test will then produce valid results, but this still requires postulation of a certain distribution underlying the data, which is often not warranted for software engineering data, because too little is known about their composition.

We will use two replacements for the t-test that do not require assumptions about the actual distribution of the data: Bootstrap resampling of means differences and the Wilcoxon test.

The **Wilcoxon rank sum test** also known as the **Mann-Whitney U test** (S-Plus: `wilcox.test()`) compares the medians of two samples and computes the probability that the medians of the underlying distributions are in fact equal. This probability is called the **p-value** and is usually presented as the test result as a number between 0 and 1. When the p-value is sufficiently small, i.e., the difference is probably *not* due to chance alone, we call the difference **significant**. We will call a difference significant if $p < 0.1$. The Wilcoxon test does not make any assumptions about the distribution of the data, except that the data must come from a continuous distribution so that there are never any ties. Fortunately, there is a modified test (the Wilcoxon test with Lehmann normal approximation) that can cope with ties as well and we will use this extension where necessary.

Note that a Wilcoxon test may find the estimated median of a sample a to be larger than that of a sample b even if the actual sample median of a is smaller! Here is an example:

```
a:   1  2  3  4  5  6  7  8  9 10  21  22 23 24 25 26 27 28 29 30 31
b:  10 11 12 13 14 15 16 17 18 19  20  30 31 32 33 34 35 36 37 38 39
```

Obviously b should be considered larger than a and in fact the Wilcoxon test will find exactly this with $p = 0.0076$. (We will write a test result such as this in the following form: Wilcoxon test $p = 0.0076$.) However, the actual sample median happens to be 21 for a but only 20 for b; for other samples this difference could even be arbitrarily large.

If we have no preconception about which sample must have the larger median, if any, we use a so-called **two-sided test**. If, as in the example above, we want to *assume* that, say, either b is larger or both are equal, but want to neglect (in advance) the possibility that a could be larger than b, we will use a **one-sided test**, which is in fact the exactly same thing, except that the p-value is halved. In this experiment, we will most usually use one-sided tests, because our testing is hypothesis-driven.

Our second robust replacement for the t-test is **Bootstrap resampling of differences of means**. The idea of bootstrapping is quite simple: simulation. The only assumption required is that the samples seen are representative for the underlying distribution with respect to the statistic that is being tested — this assumption is of course implicit in all statistical tests. With a computer we can now generate lots of further samples that correspond to the two given ones, by sampling with replacement (S-Plus: `sample(x,replace=T)`). This process is called *resampling*. For instance resampling of the above sample a might yield

```
a:   1  3  3  4  4  6  7  8  9 10 21 22 23 24 25 25 25 27 27 30 30
```

Such a resample can (and usually will) have a different mean than the original one and by drawing hundreds or thousands of such resamples a_r from a and b_r from b we can compute the so-called **bootstrap distribution** of all the differences "mean of a_r minus mean of b_r". Now we can compute what fraction of these differences is, say, greater than zero. Let's assume we have computed 1000 resamples of both a and b and found that only 4 of the differences were greater than zero (which is a realistic value; see Figure 3.1 for an actual example). Then 4/1000 or 0.004 is the p-value for the hypothesis that the mean of the distribution underlying a is actually *larger* than the mean of the distribution underlying b. From this bootstrap-based test, we can clearly reject the hypothesis.[1] Instead of p-values, we can also read arbitrary **confidence intervals** from the bootstrap distribution. In the example, 90% of all bootstrap differences are left of the value -4.0, hence a left 90% confidence interval for the size of the difference would be $(4.0, \infty)$; in other words: the difference is 4 or larger with a probability of 0.9.

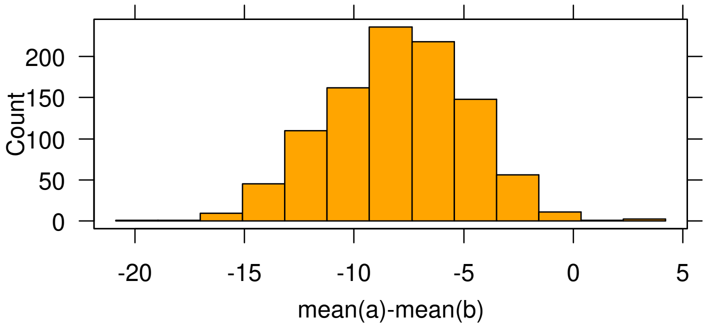


Figure 3.1: Boostrap distribution of the difference of means of the two samples a and b as given in the text.

When we use Bootstrap, we will usually compute 1000 pairs of resamples and indicate the resulting p-value with two digits. For the example above we would write: mean bootstrap $p = 0.00$. Occasionally we will use 10000 resamples and then indicate three digits (mean bootstrap $p = 0.004$)

Sometimes we would like to compare not only means and medians, but also the variability (spread) of two samples. The conventional method of doing this is the **F-test**, which compares the standard deviations. It is related to the t-test and, like the latter, assumes the two samples to come from a normal distribution. Unlike the t-test, which is quite robust against deviations from normality and whose results deteriorate only slowly with increasingly pathological inputs, the F-test is very sensitive to data with deviations from normality. Therefore, the F-test is completely unsuitable for our purposes. Instead we resort to resampling again and compare a robust measure of spread, namely the inter-quartile range mentioned above. We generate pairs of resamples and compute the differences of their interquartile ranges. This way, we compute a **Bootstrap resampling of differences of inter-quartile ranges** in order to arrive at a test for inequality of variability. Note that the inter-quartile range nicely corresponds to the box width in our boxplots. Much like for the test on means, we write the result of a bootstrap-based test on inter-quartile ranges (iqr) like this: iqr bootstrap $p = 0.15$.

3.1.2 Two-dimensional statistics

Sometimes we do not only want to compare groups of individual measurements, but rather want to understand whether or how one variable depends on another. Such a question is usually best be assessed by the familiar x-y-coordinate plot (S-Plus: `xyplot()`); see Figure 3.38 on page 46. Here, each participant is represented by one point and we possibly make two (or more) such plots side by side for comparing different groups of participants.

We will often plot some sort of **trend line** together with the raw data in an x-y plot in order to show the relationship between the two variables plotted. If the assumed relationship is linear, the trend line is a straight line. The most common form of this is the **least squares linear regression** line (S-Plus: `lsfit()`), a line chosen such that the sum of the squares of the so-called **residual errors** (residuals) is minimized. The residual error of a point is the vertical distance from the point to the regression line. While the least squares regression

[1] By the way, the t-test would produce the more pessimistic p-value 0.010 when comparing a and b — an error that is not relevant in this case of a very large difference, but that is important in other cases. If the deviations of the samples from normality are large, the t-test becomes insensitive and can often not detect the differences (statisticians say it has small **power**).

line is very simple to compute, it is sensitive to individual far-away points, much like the mean in the one-dimensional case. Therefore, we will sometimes add two other sorts of trend lines in order to avoid spurious or misleading results. These other trend lines are less common and more difficult to compute, but are often more appropriate because they are more robust against a small number of far-away points. The first of these robust regression lines is the **least distance regression line** or **L1 regression line** (S-Plus: `l1fit()`). It minimizes the absolute values of the residual errors instead of their squares, which reduces the influence of the largest residuals. The L1 regression is computed by linear programming. The second kind of robust regression is the **trimmed least squares regression** (S-Plus: `ltsfit()`), which computes a standard least squares regression after leaving out those, say, 10% of all points that would result in the largest residual errors. The operation of leaving out a certain fraction of extreme points is called **trimming**. Since the identity of these worst points depends on the regression line chosen and the regression line depends on which points are left out, the trimmed regression is extremely difficult to compute — it is implemented by a genetic algorithm. The fraction that is trimmed away is indicated in each case (usually either 10%, 20% or 50%).

If a cloud of points lies close to a line, we can trust the standard least squares regression line. If however there are severe outliers or the correlation is generally not very high, we may assess the presence or absence of a certain trend by looking at all three kinds of trend lines at once: if they all point in the same general direction, the trend is most probably real, but if one of them points up while the other two point down, we should not trust any conclusions about a trend — except possibly the conclusion that there is no trend at all.

In addition to a trend line, we may also compute (and plot, with dotted lines) a **confidence interval** for either the trend line itself or for the data from the underlying distribution; the latter is called a **prediction interval**. A confidence interval indicates the range within which the regression line will be with a given probability if one encounters many different samples of the same size from the same distribution. We typically plot confidence intervals that cover 90% probability. The prediction interval not only includes the regression line but also the individual data points, that is, it describes the variability of the individual points instead of the variability of their mean.

If we do not want to impose a linear relationship onto the two variables plotted, we may use a **loess local regression** line instead (S-Plus: `loess()`). Here, for computing the y-coordinate of each x-coordinate of the trend line, only a fraction of the data points will be used, and these will be weighted such that the weight of each point diminishes rapidly with increasing distance of its x-coordinate. The parameter that controls which fraction of the overall x-axis range will be used at each point of the loess line is called the **span**. The larger the span, the more similar the loess line will be to a straight line, small spans allow for much local variation of the loess line. Dotted lines may be present that indicate a 90% confidence interval around the loess line (S-Plus: `predict(se.fit=T)`).

A final way of robustly assessing relationships between two variables is plotting the **rank** of the values instead of the values themselves. In this case, the smallest value of the whole set is mapped to 1, the second smallest to 2, and so on. In the presence of ties, the average rank is assigned to the whole set of tied values. Ranking ignores the *size* of differences between values and represents only, *how many* other values are in between. This is sometimes useful for variables with very odd distributions, such as reliabilities which tend to be either very close to 100 percent or very close to zero, with gaping holes between these extremes: If there is a trend, we might miss it because the values of interest are too close to each other in a plot that covers the whole range, but after ranking we can clearly see all differences (and non-differences as well).

The **correlation** (more precisely: linear correlation) quantifies how well the relationship of the x and y values in a set of pairs (x_i, y_i) can be described by a straight line. The correlation, usually called r, is a value between -1 and 1. A value of 1 indicates the pairs all lie on a straight line (when plotted in a standard cartesian coordinate system). -1 means just the same, except that the line has negative slope (i.e. is going down to the right) instead of positive slope. For smaller values of r, there are increasing amounts of fluctuation of the points (x_i, y_i) around the best possible line through the point cloud (the regression line). If $r = 0$, the points do not show any trend at all, just as if they were statistically independent. The notion of correlation can be extended to higher dimensions (e.g. describing a set of triples by a plane). The square of the correlation, r^2, describes the **amount of variance explained** by the regression line. For instance, if $r^2 = 0.8$, the overall variance of the y_i is five times as large as the variance of the y_i against the regression line, or, put differently, 80 percent of the overall variance, are "explained" by the model of the data that is described by the regression line.

3.1.3 Presentation of results

Since there are quite many comparisons of 1-dimensional statistics, we use a somewhat standardized format for presenting them. Generally, for each comparison three tests will be computed: A Wilcoxon rank sum test, a bootstrap test for differences of means, and a bootstrap test for differences of interquartile range.

If the results indicate that the differences are not significant, they will often not be mentioned at all. Typically, results will be presented quantitatively in the caption of the figure containing the corresponding plot. The test and its p-value will be indicated in the notation mentioned above and will be preceded by the name of the group that exhibited the better performance. Example:

> The P group has more commented lines (P_{raw}: mean bootstrap $p = 0.05$, Wilcoxon test $p = 0.05$).

The main text will usually mention the results only qualitatively. We use the following expressions in that case: If the p-value is below 0.1, we make a simple statement that there is a difference (e.g. "a is *smaller than* b"). If the p-value is between 0.1 and about 0.2, we call it a *tendency* ("a *tends to be smaller* than b"). For still larger p-values, we either consider the difference very small ("a is *slightly smaller* than b", "a is *hardly smaller* than b") or not present ("a and b are not different"). When the difference is very substantial, it will be called large ("a is *much smaller than* b") and sometimes such statements are backed up by some form of confidence interval for the difference. Where it appears to be warranted, all of these language rules are followed only loosely.

Generally, the main text contains the main results and their interpretation, while the captions contain the details. Some numeric results are printed in a large table in Section 3.14 on page 55.

3.2 Group formation

Before we start with the actual data analysis we need to overcome some complications which limit the comparability of the groups:

1. Some participants did not finish the task; they gave up either before they even tried an acceptance test (s028, s031, s045, s069) or after one to three failed attempts (s012, s014, s024, s039). It is entirely unclear how one can sensibly compare the performance on finished tasks with that on unfinished ones.

2. The 8 non-PSP volunteer subjects are rather atypical. On the one hand, all 3 failed participants in the non-PSP group are from the volunteer subgroup. On the other hand, three of the most capable participants of the experiment are among the other 5 volunteers, all of which are quite extreme in some aspect of their behavior as is shown in Figure 3.2. We see that of the five successful volunteers two completed extremely fast, one extremely slow, and the other two are also faster than average *although* they compile extremely frequently.

Furthermore, it is conceivable that some of the subjects in the PSP group do not, in fact, use the PSP. We would like to compare subpopulations such that the possible group differences are not blurred by such participants. We use the following approach:

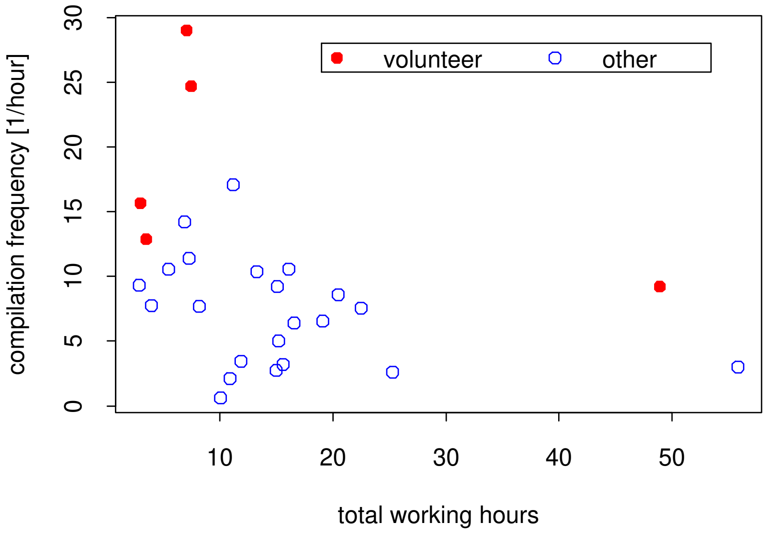


Figure 3.2: Total time for completing the task versus number of compilations per working hour for the non-PSP volunteers and all other participants. 22 subjects are missing in the plot, because their number of compilations is not available.

> *From each group remove that half of the participants which had the least performance and compare only the rest.*

Such an operation is called trimming. This way we can expect "cleaned up" group properties and can also remove the subjects that gave up. Fortunately, the fraction of participants who gave up are the same in the N group (3 out of 19, 16 percent) and the P group (5 out of 29, 17 percent), so that we do not introduce any bias by this operation. Hence, we define our final analysis groups by removing another 6 participants from the N group (ending up with 10 out of 19, or 53%) and another 9 from the P group (ending up with 15 out of 29, or 52%).

The next question is which measure to take for determining performance. We avoid this problem by using several measures and defining one pair of groups by each. During the analysis we will compare all of these group pairs.

Raw groups: We define the group N_{raw} as all of the non-PSP participants minus those that gave up during the experiment (s014, s028, s031). We define the group P_{raw} as all of the PSP participants (except the two volunteers s081 and s102 that had previously participated as non-PSP subjects) minus all those that gave up during the experiment (s012, s024, s039, s045, s069).

Groups trimmed by product reliability: The single most important performance measure as indicated in the task description given to the participants is the reliability of the delivered program. We choose the output reliability on the z3 test here (see Section 3.4.2), because it is the most "tame" of all reliability measures and we expect it to be least likely to produce artifacts. We define the group N_{rel} as all members of group N_{raw} except those 6 with the lowest z3 output reliability (in order: s034, s043, s020, s023, s053, s037). We define the group P_{rel} as all members of group P_{raw} except those 9 with the lowest z3 output reliability (in order: s036, s075, s030, s027, s090, s018, s060, s066, s093). Ties were broken based on z2 output reliability, then m3 output reliability, and finally the output reliability of the first acceptance test.

Groups trimmed by time to completion: Another obvious performance measure is the number of work hours required until passing the acceptance test. We define the group N_t as all members of group N_{raw} except those 6 with the longest work time in the experiment (in order: s053, s017, s065, s056, s068, s037). We define the group P_t as all members of group P_{raw} except those 9 with the longest work time in the experiment (in order: s060, s027, s048, s087, s090, s021, s057, s018, s051).

Groups trimmed by quality discipline: Next we would like to sort out those participants that appear be less disciplined with respect to achieving high software quality than the others. To judge quality discipline, first of all we consider the number of acceptance tests required, because a high number suggests premature "product release" and hence an undisciplined process. (Remember that program correctness was the highest-priority requirement for the experiment.) This coarse measure produces many ties. But there are more aspects of quality discipline. In particular, for the PSP group we are interested in whether actual PSP quality management techniques are used and to what degree. Hence, the next criterion is the presence and completeness of a defect log. There were only 10 defect logs made overall, all by members of the PSP group.[2] We break the remaining ties by the completeness of the handwritten time log that was required from all participants; both defect log and time log were judged on a four-point ordinal scale (missing, dubious or highly incomplete, slightly inaccurate, apparently accurate). Still remaining ties are broken by the output reliability achieved in the first acceptance test. We define the group N_{dpl} as all members of group N_{raw} except those 6 with the lowest quality discipline according to the above ordering (in order: s065, s068, s037, s056, s053, s050). We define the group P_{dpl} as all members of group P_{raw} except those 9 with the lowest quality discipline (in order: s051, s060, s090, s057, s075, s087, s048, s054, s036).

Groups trimmed by product size: The cost of a program is determined not only by its production cost, but also by its maintainability, which is a function of modularity, clarity of structure, design documentation, program length, and other measures. The only one of these that can be objectively measured is program length. Hence we use the number of statement lines of code as a substitute maintainability measure and regard longer programs as harder to maintain. Note that this criterion may fail, because some participants may write overly compact "clever" code that is hard to understand. However, we expect that the opposite case of long-winded and fussy code is more frequent.

We define the group N_{sz} as all members of group N_{raw} except those 6 with the highest number of statement lines in their resulting program (in order: s017, s065, s023, s037, s053, s034). We define the group P_{sz} as all members of group P_{raw} except those 9 with the highest number of statement lines in their resulting program (in order: s090, s057, s018, s027, s015, s072, s084, s054, s087).

Groups trimmed by product efficiency: Finally, one would also prefer more efficient programs over less efficient ones — there were rather huge differences in this respect. We measure efficiency by number of minutes of CPU time required for the test set z3 (see Section 3.4.4). We define the group N_{eff} as all members of group N_{raw} except those 6 with the highest CPU time requirement for test z3 (in order: s068, s062, s023, s047, s059, s034). We define the group P_{eff} as all members of group P_{raw} except those 9 with the highest CPU time requirement for test z3 (in order: s072, s093, s018, s060, s048, s054, s021, s063, s090).

Note that many of the participants are excluded by more than one of the criteria. In particular, s090 is removed all five times and s018, s037, s053, and s060 are excluded four times.

3.3 Estimation

Can the PSP participants really estimate the time required for a task more accurately? As we see in Figure 3.3, the answer is probably yes, although for the given task the difference is not overly consistent. The mis-estimations tend to be less variable in the P group compared to the N group.

Looking at absolute instead of relative deviations from the estimated time and discriminating pessimistic from optimistic estimates, we again find smaller variability in the P group (Figure 3.4), although the difference is not significant. The worst underestimation is much more limited in the P group (about three times the median). The difference in mean or median is not significant.

The estimation questionnaire asked not only for a point estimate of the expected work time but also for a 90% confidence interval. Figure 3.5 compares the actual misestimation to the size of this estimation interval. If the

[2] We think that about three other PSP participants may have created a log but did not deliver it along with their work products.

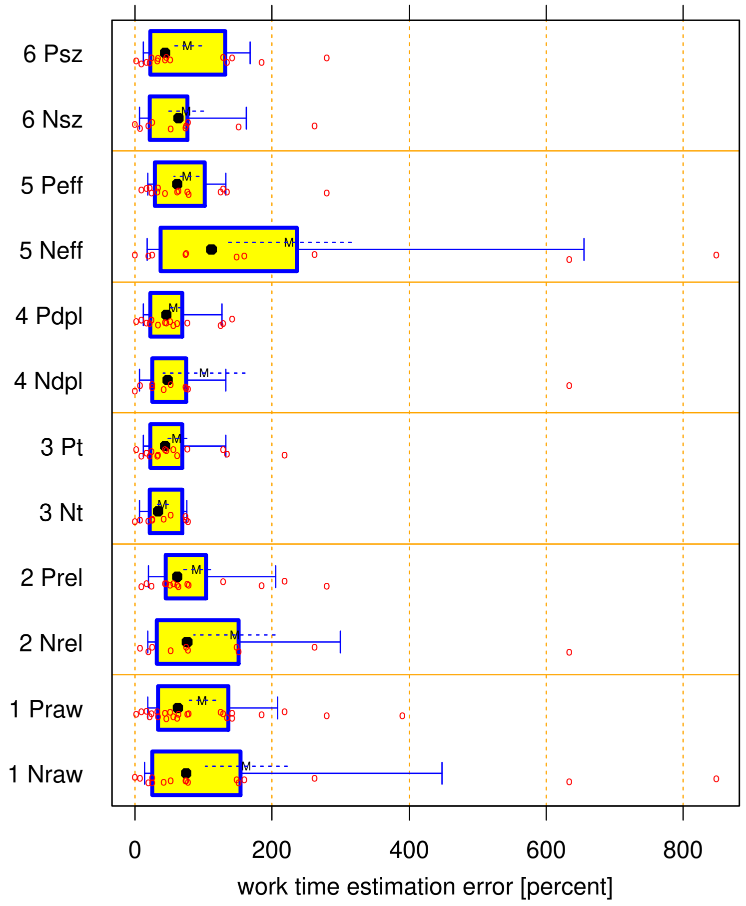


Figure 3.3: $\left|\frac{t_{work}}{t_{estim}} - 1\right|$: Time mis-estimation for various pairs of subgroups. The variability of the estimation accuracy tends to be smaller for the P groups (P_{eff}: iqr bootstrap $p = 0.12$) and the error means and medians tend to be smaller as well (P_{raw}: mean bootstrap $p = 0.18$; P_{eff}: mean bootstrap $p = 0.033$, Wilcoxon test $p = 0.089$).

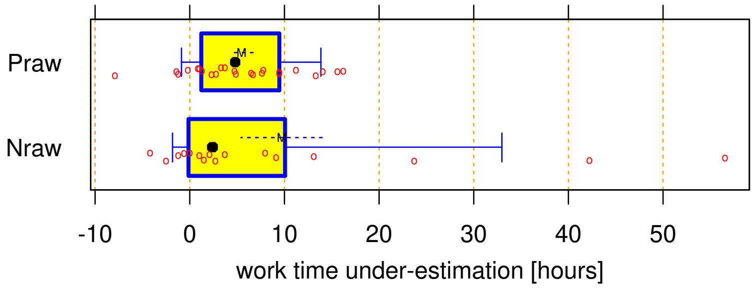


Figure 3.4: $t_{work} - t_{estim}$: Actual work time minus estimated work time. The variability in the P group tends to be smaller (P_{raw}: iqr bootstrap $p = 0.28$) and more symmetric.

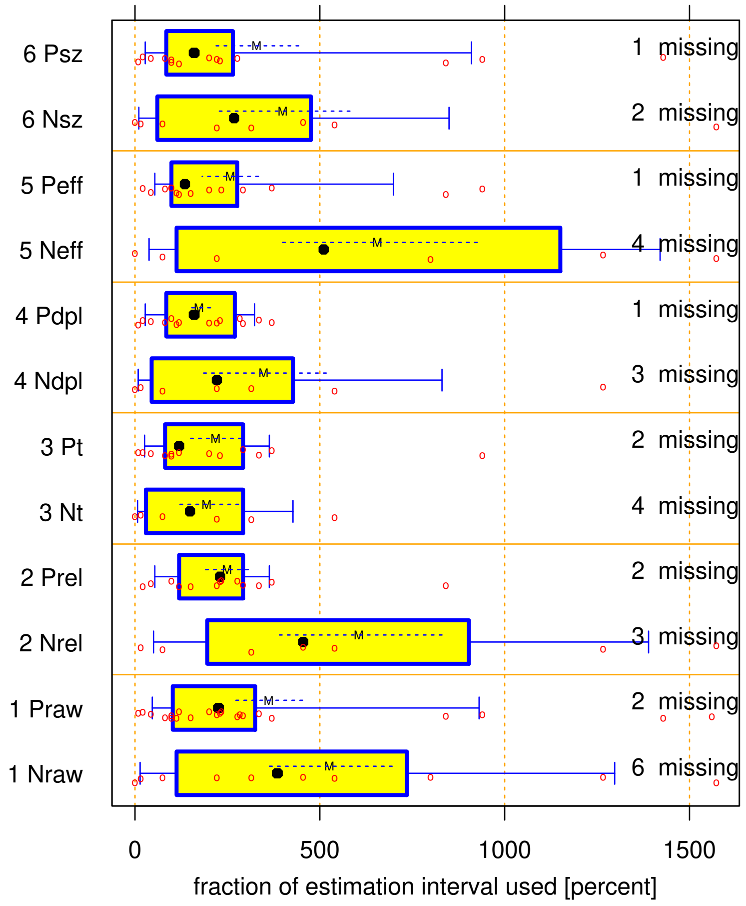


Figure 3.5: $\left|\frac{t_{work}-t_{estim}}{t_{extr}-t_{estim}}\right|$: Fraction of estimated time interval actually used, in percent. t_{extr} is t_{low} for those few estimates that were too high and t_{high} for estimates that were too low. Mean, median, and variability all tend to be smaller in the P group (P_{raw}: mean bootstrap $p = 0.20$, Wilcoxon test $p = 0.25$, iqr bootstrap $p = 0.16$). All differences are more pronounced in the reliable subgroups (P_{rel}: mean bootstrap $p = 0.03$, Wilcoxon test $p = 0.11$, iqr bootstrap $p = 0.06$). The means difference is also a bit more pronounced for the disciplined subgroups (P_{dpl}: mean bootstrap $p = 0.15$).

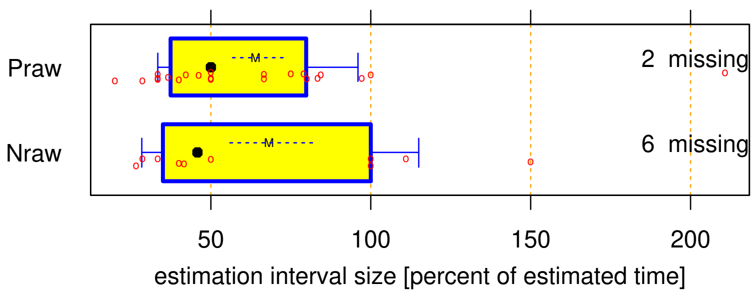


Figure 3.6: $\frac{t_{high}-t_{low}}{t_{estim}}$: Relative size of estimation interval compared to estimated time. The variability of the relative estimation interval size tends to be smaller in the P group (P_{raw}: iqr bootstrap $p = 0.19$).

intervals were good, about 90% of all points should be in the range $0 \ldots 100$ (remember that whisker-to-whisker the boxplot covers 80% of the values). As we see, three quarters of all participants hit the wall quite badly with their estimates. However, both the average and the variability is smaller for the P group. Interestingly, the difference is most pronounced if one ignores the subjects with the less reliable programs, because the bad estimators in P_{rel} happen to produce the worse programs, too, while no such correlation exists in N_{rel}. Many N participants did not even indicate an estimation interval (hence the 6 missing values), presumably because they had no idea how large it might be. The variability in the size of the estimation interval itself (relative to the estimated time) is also smaller in the P group (Figure 3.6).

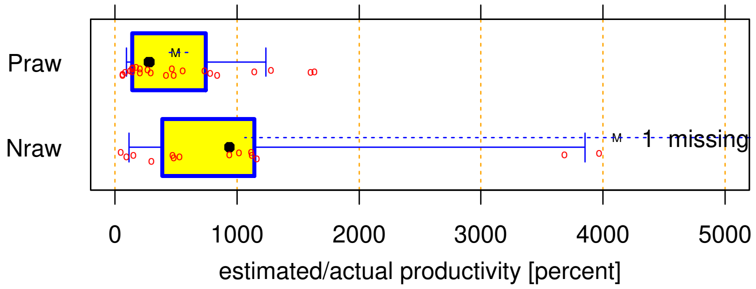


Figure 3.7: $\frac{prod_{estim}}{prod_{actual}}$: Quotient of estimated and actual productivity. Mean, median, and variability are all significantly smaller in the P group (P_{raw}: mean bootstrap $p = 0.000$, Wilcoxon test $p = 0.001$, iqr bootstrap $p = 0.009$).

If one compares the estimated productivity (in lines of code per hour) to the actual productivity (using the participants' own definition of LOC), roughly half of the P group is outside the reasonable range of plus or minus a factor of two around their estimation; see Figure 3.7. In contrast, half of the N group achieved less than one tenth of their expected productivity. With a probability of 0.9, the difference of the means is greater than 120 and the difference of the inter-quartile ranges greater than 96. These values clearly show that many N participants have rather little knowledge about their own productivity.

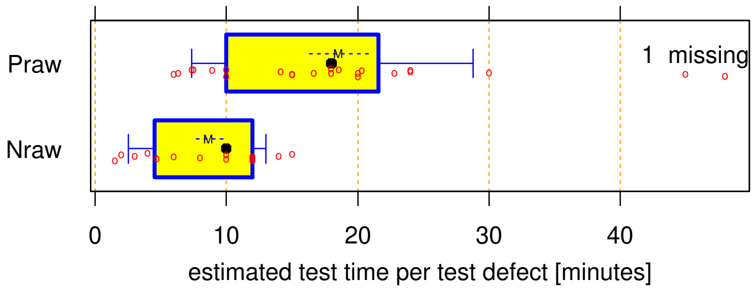


Figure 3.8: $\frac{t_{test,estim}}{N_{err,test}}$. Estimated time per defect found for the test phase. One P subject is missing because his estimated number of defects was zero. Mean and median are smaller in the N group (P_{raw}: mean bootstrap $p = 0.00$, Wilcoxon test $p = 0.00$), the variability also tends to be smaller (iqr bootstrap $p = 0.11$).

One possible source of these misestimations is shown in Figure 3.8: The N participants are far too optimistic about how long it takes to find and remove each defect in the test phase. With a probability of over 80%, their mean estimated time per defect (computed by dividing the estimated time for the test phase by the estimated number of errors to be found in the test phase) is at most half as large as in the P group.

As one last test of the estimation capabilities we can compare what fraction of the estimated total(!) testing time was performed *after* the first acceptance test; it turns out that many participants spent more than the planned testing time even after the program was supposed to be fully tested and correct. The results in Figure 3.9 show less of this effect for the P group and also once again slightly smaller variability (P_{raw}: iqr bootstrap $p = 0.30$).

In summary, the P group is clearly better at estimating the total working time, the productivity, or the time for fixing defects.

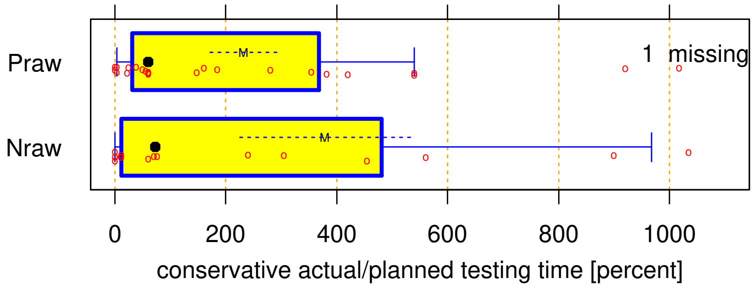


Figure 3.9: $\frac{t_{work} - t_{accept1}}{t_{test,estim}}$: Remaining work time after first acceptance test divided by estimated time for testing phase. This is a rather conservative comparison of actual and expected testing time — much testing time is spent before the first acceptance test. There is one value for N at 2323%. The mean tends to be smaller in the P group (P_{raw}: mean bootstrap $p = 0.19$).

3.4 Reliability and robustness

In this section we will assess the reliability of the delivered programs. Producing a defect-free and hence reliable program was stated as the most important goal in the experiment instructions. Therefore, this comparison should be considered the most important of all.

We will assess the reliability by functional testing, Section 3.4.1 explains why. In Sections 3.4.2 and 3.4.3 we describe the inputs used for the tests and the reliability measures we apply for describing the test results. Sections 3.4.4 and 3.4.5 then present the results of the reliability and robustness tests, respectively. Finally, Section 3.4.6 will assess whether the programming language used by the participants had an influence on reliability. The final subsection summarizes the results.

3.4.1 Black box analysis and white box analysis

The single most important goal set forth for the experiment participants was "write a correct program". There are basically two ways for judging the correctness of a program. The white box method analyzes the program text and determines whether it contains any defects that might produce incorrect behavior. The black box method executes the program on a variety of inputs and records how often the program actually fails.

Both methods have their disadvantages: While the white box method might in principle prove if a program is entirely correct, it cannot directly assess the program's reliability in the presence of defects; the practical consequences of any single defect remain unclear and one can hardly compare one defect to another. Furthermore, finding all defects is very difficult. The black box method, on the other hand, requires that a concrete set of inputs be chosen. Unless the input space can be executed exhaustively (which is usually infeasible even for the most trivial programs), the black box method can never prove that a program is entirely correct, but always returns a quantitative estimate of the program's reliability.

With respect to effort, the white box method is enormously costly. In contrast, in our case the black box method is quite cheap, because we can generate inputs randomly and can generate the corresponding correct outputs using the gold program (see Section 2.5.5).

Another advantage of the black box method is that the relevance of the results is clearer, because the relation of the chosen input distribution to the intended application domain is more easily understood. We will thus exclusively use black box analysis to determine the correctness and reliability of the participant's programs.

3.4.2 The test inputs: a, m, and z

The inputs used both for the acceptance tests during the experiment and also during the experiment evaluation were always generated at random according to a specific probability distribution.

For our purposes, the inputs (called "telephone numbers") are arbitrary strings of no more than 50 characters length, where the characters are chosen from the set

$$C = \{ /, -, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$$

For each acceptance test, a new set of 500 inputs was generated according to probability distribution A described below.

For reliability determination during experiment evaluation we used four sets of inputs m2, m3, m4, and m5 containing 10^2, 10^3, 10^4, and 10^5 examples, respectively, all generated according to probability distribution M described below. We furthermore used four sets of inputs z2, z3, z4, and z5 containing 10^2, 10^3, 10^4, and 10^5 examples, respectively, all generated according to probability distribution Z described below.

Distribution M: The length of the strings is uniformly distributed over the integer range $1 \ldots 50$. Each character from C is randomly chosen with probability 1/12 at each position of each string independently.

Distribution Z: Like M, except that strings without a digit (i.e., consisting only of dashes and slashes) do never occur. To obtain a sample of values from Z, take a sample from M and remove all elements not containing a digit.

Distribution A: Like Z, except that the distribution of input lengths is no longer uniform. Instead, it is as follows. Consider the random variable $z = \lfloor \sqrt{u} + v \rfloor$, where u is uniformly random on $0 \ldots 36$ and v is uniformly random on $0 \ldots 44.5$. u and v are both continuous and independent. Lengths are distributed like z restricted to the range $1 \ldots 50$.

This means that M is a straightforward general distribution of legal input values, Z avoids the possibly difficult case of "telephone numbers" that would produce an empty output encoding, and A additionally makes very short and very long numbers less frequent than those with moderate length.

3.4.3 Reliability measures

Given a mistake of the program, how is the reliability actually computed? We will use two different compound measures of reliability, called *input reliability* and *output reliability*.

Given a particular input, a program may produce no, one, or several outputs. There are several possibilities for the correctness of the actual program outputs O_{act} for a set I of inputs. Consider the following statements:

1. Each output o explicitly refers to exactly one input $i(o)$.
2. Each input i must produce a certain set of expected outputs $O(i)$.
3. We call the set of expected outputs $O_{corr} := \bigcup_{i \in I} O(i)$.
4. We call the expected number $|O_{corr}|$ of outputs E.
5. An actual output $x \in O_{act}$ produced by the program may be a correct output. We call the set of correct outputs $c := O_{act} \cap O_{corr}$.
6. An actual output may be spurious, i.e., wrong. We call the set of wrong outputs $w := O_{act} \setminus O_{corr}$.

7. An expected output may be missing. We call the set of missing outputs $m := O_{corr} \setminus O_{act}$.

8. Any single input may produce exactly the corresponding set of expected outputs or it may fail, i.e, produce at least one missing or wrong output. Failures include outputs for "phantasized" inputs, i.e., outputs which refer to inputs not in I. We call the set of failed inputs $f := \{i : (\exists o \in O_{act} : i = i(o) \wedge (o \notin O(i) \vee i \notin I)\}$

No we define the following two reliability measures: the **output reliability** $rel_{out,I} := \frac{|c|}{E+|w|}$ and the **input reliability** $rel_{in,I} := 1 - f/|I|$. We will usually express these values in percent.

3.4.4 Inputs with nonempty encodings

For the first round of tests, we consider inputs from distribution Z, that is the random but fixed test sets z2, z3, z4, and z5. The digit stands for the size of the test set such that zi has 10^i inputs, i.e., z2 has 100 inputs (10^2), z3 has 1000 inputs (10^3) etc. Some of the programs of successful participants were very slow, hence only z2 and z3 were administered to all programs. Some extremely slow programs were not used with z4 (s034, s072, s093) and these plus also some modestly slow programs were not used with z5 (s020, s021, s023, s027, s034, s036, s047, s048, s054, s059, s060, s062, s063, s065, s068, s072, s090, s093).

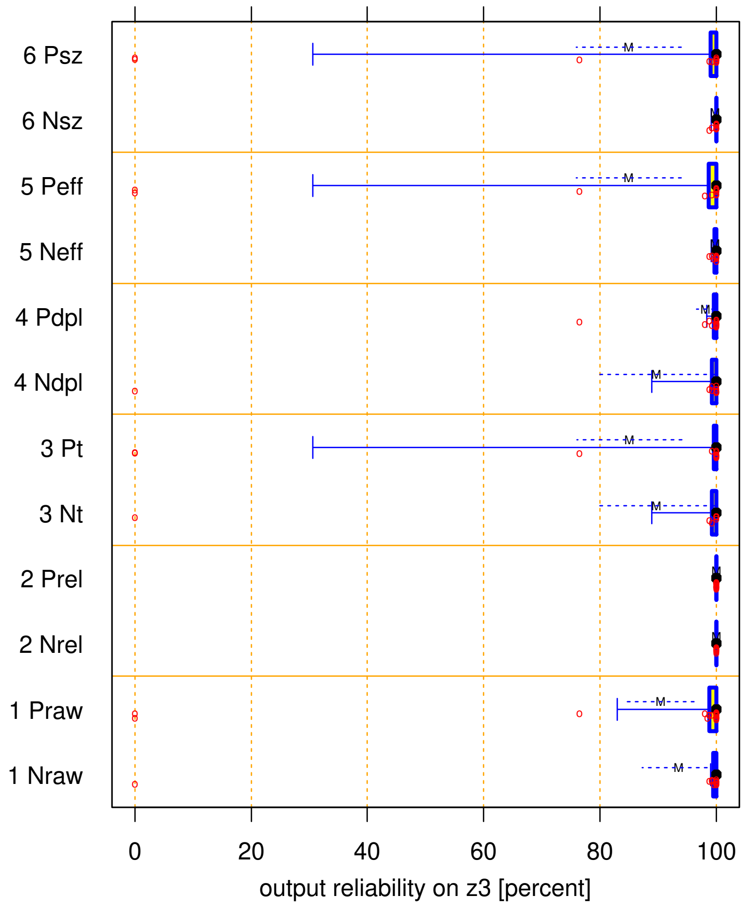


Figure 3.10: $rel_{out,z3}$: Output reliability on the z3 test set. The mean results are better in the N group for some subgroups (N_{sz}: mean bootstrap $p = 0.01$; N_{eff}: mean bootstrap $p = 0.02$), but the differences are not significant for the median (N_{sz}: Wilcoxon test $p = 0.20$; N_{eff}: Wilcoxon test $p = 0.28$). Other differences in median, mean, or iqr are not significant (N_{raw}: mean bootstrap $p = 0.33$, Wilcoxon test $p = 0.32$, iqr bootstrap $p = 0.22$).

As we see in Figure 3.10, the reliability is generally quite high: the median reliability in all subgroups is 100% and even the 0.25 quantile is usually higher than 99%. Apparently the N group has a slight advantage, which we can see more closely in the magnification shown in Figure 3.11: N is better for the N_{sz} and N_{eff} subgroups, which only means that the largest and least efficient P programs were all among the very reliable ones. Overall, the difference is insignificant. As we see, there are a few programs that achieve below 100 percent reliability but are quite close to it. The median reliability is 100 percent in all subgroups, so a majority of the programs is perfect in this test.

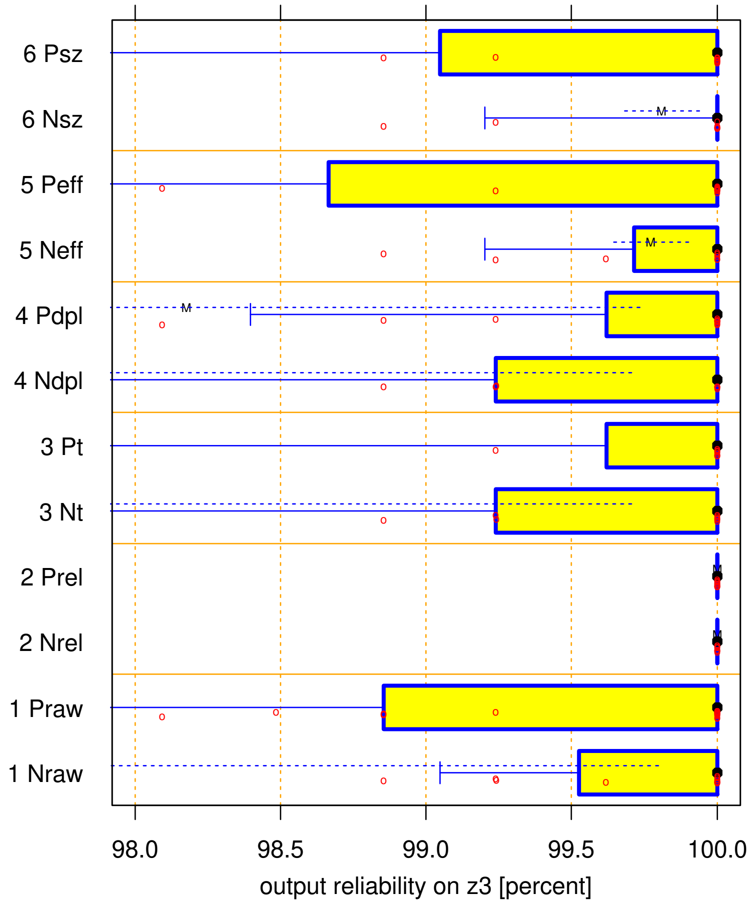


Figure 3.11: 98% to 100% range of output reliability on the z3 test set.

The exact reliability of all programs with a reliability of more than 0 but less than 100 depends a lot on the specific set of inputs chosen. We can reduce this influence by arranging the programs equidistantly according to the reliability ordering instead of by the reliability values themselves, that is, we look at the rank ordering. The results are shown in Figure 3.12. We find that the advantage of the N group depends on only 10 programs that fall between the minimum and the maximum reliability. The difference is again not significant.

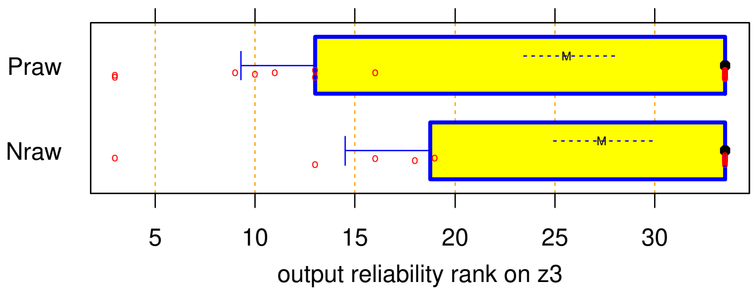


Figure 3.12: rank($rel_{out,z3}$): Rank order of output reliability on the z3 test set. The differences in mean, median, or iqr are not significant (N_{raw}: mean bootstrap $p = 0.30$, Wilcoxon test $p = 0.31$, iqr bootstrap $p = 0.21$).

The reliability results are similar for the other test sets as we can see for z2 in Figure 3.13, for z4 in Figure 3.14, and for z5 in Figure 3.15. The comparisons for the test sets with 10000 or 100000 examples are distorted by the fact that due to time constraints not all programs participated in them, which may bias the results.

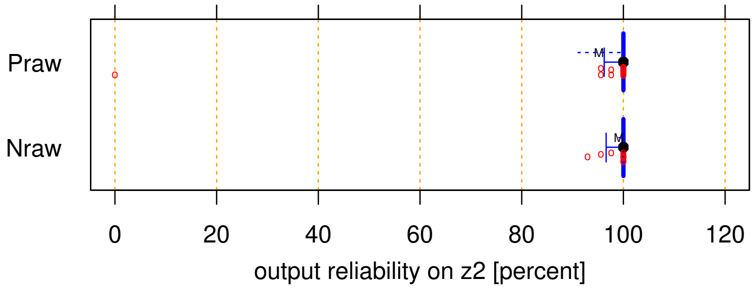


Figure 3.13: $rel_{out,z2}$: Output reliability on the z2 test set. The differences in mean, median, or iqr are not significant (N_{raw}: mean bootstrap $p = 0.24$, Wilcoxon test $p = 0.44$, iqr bootstrap $p = 0.40$).

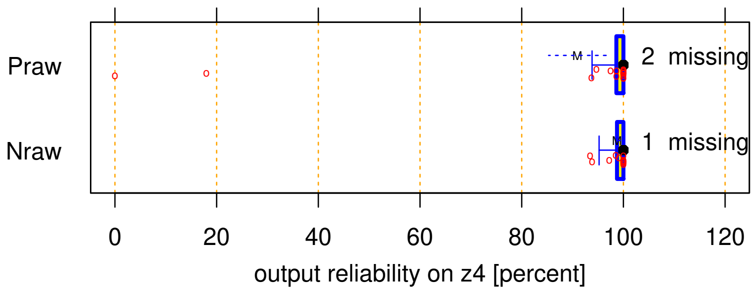


Figure 3.14: $rel_{out,z4}$: Output reliability on the z4 test set. The differences in median and iqr are not significant (N_{raw}: Wilcoxon test $p = 0.49$, iqr bootstrap $p = 0.46$), but the difference in mean is (N_{raw}: mean bootstrap $p = 0.08$). Note that some programs were not measured which may bias the results.

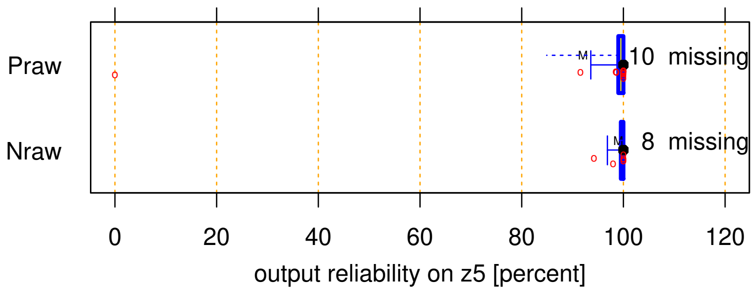


Figure 3.15: $rel_{out,z5}$: Output reliability on the z5 test set. The differences in mean, median, or iqr are not significant (N_{raw}: mean bootstrap $p = 0.18$, Wilcoxon test $p = 0.25$, iqr bootstrap $p = 0.43$).

Summing up, we find a slight tendency that the reliability in the N group is better, but the differences are not significant.

3.4.5 Arbitrary inputs

We now proceed to the more difficult inputs from the M distribution. These contain requests that never occurred in any of the acceptance tests: inputs without digits (but containing slashes and dashes), which should result in an empty encoding. A programmer would not intuitively expect such inputs because the domain metaphor calls the inputs “telephone numbers”.

As for the Z inputs, we have test sets m2, m3, m4, and m5, where mi has 10^i inputs. Again, some of the programs of successful participants were very slow, hence only m2 and m3 were administered to all programs. One extremely slow program was not used with m4 (s072) and several were not used with m5 (s023, s060, s065, s068, s072, s093).

Plotting the distributions for the various subgroups for the 1000-example test produces rather striking results: The P groups are all much better than the N groups (Figure 3.16). However, this plot is visually quite misleading. It is dominated by the fact that there is a huge gap in the reliability values between one group with 10.16% and the next larger value at 89.47%. Nevertheless, the difference in means is at least 25 and the difference in iqr at least 58, both with a confidence of 0.8. For the more disciplined subgroups, these values are 28 for means and 64 for iqr. It is quite satisfying to see that this result, where we find a clear advantage for the P groups, is also one where the more disciplined PSP users are more successful than the P group average. This is an indication that the PSP techniques indeed help considering program designs more carefully and result in writing more robust programs.

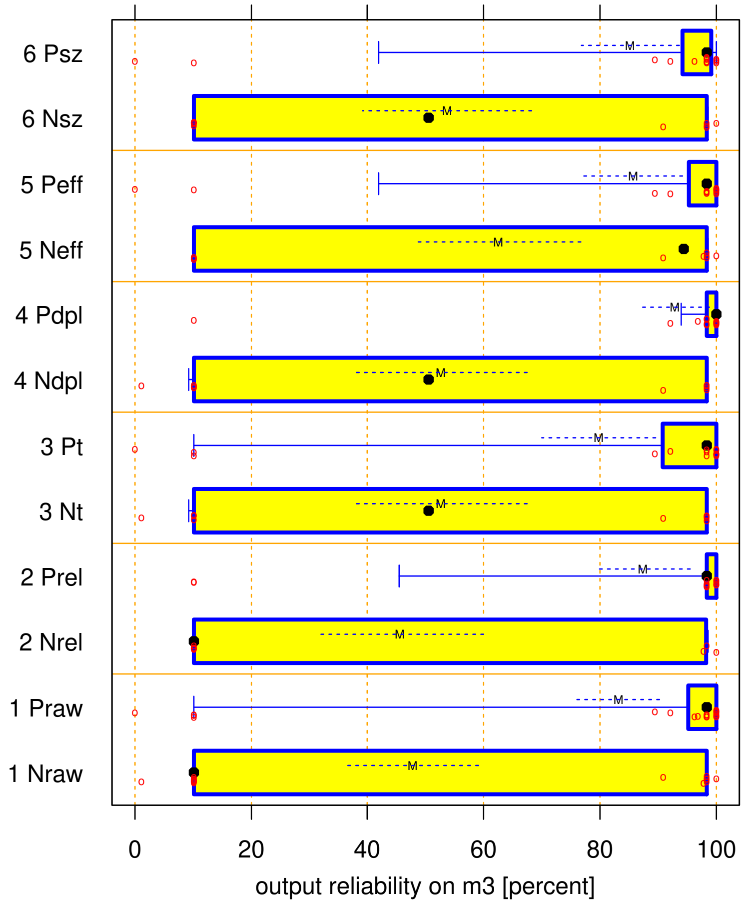


Figure 3.16: $rel_{out,m3}$: Output reliability on the m3 test set. All N subgroups are less reliable than the corresponding P subgroups (P_{raw}: mean bootstrap $p = 0.01$, Wilcoxon test $p = 0.00$) and the variability tends to be larger in N as well (P_{raw}: iqr bootstrap $p = 0.12$). The difference is most pronounced in the more disciplined subgroup (P_{dpl}: mean bootstrap $p = 0.00$, Wilcoxon test $p = 0.00$, iqr bootstrap $p = 0.05$).

A more appropriate and less misleading representation is again the plot that is based on ranks instead of raw values, as is shown in Figure 3.17. We see that all P subgroups have a large advantage over the corresponding N subgroups. The difference from N_{raw} to P_{raw} is at least 7 ranks with a confidence of 0.9; for N_{dpl} to P_{dpl} it is at least 10 ranks.

The other three test sets show a similar result; their rank ordering is shown in Figures 3.18, 3.19, and 3.20.

How come these huge differences between the inputs from distribution M versus those from Z? The reason is that many programs crash when they encounter the first input that does not contain any digit, because the processing of many programs assumes there is at least one word or digit in the output encoding. Most notably, many of the Java programs exit with an “illegal array index” or a similar exception.

3.4.6 Influence of the programming language

Since there are relatively more Java users in the N group (see Figure 2.7), we should check whether using or not using Java is the major factor for the above difference. Since the number of non-Java users is too small in the N group, we simply compare only the Java users in both groups. The result for the m3 output reliability ranks is shown in Figure 3.21.

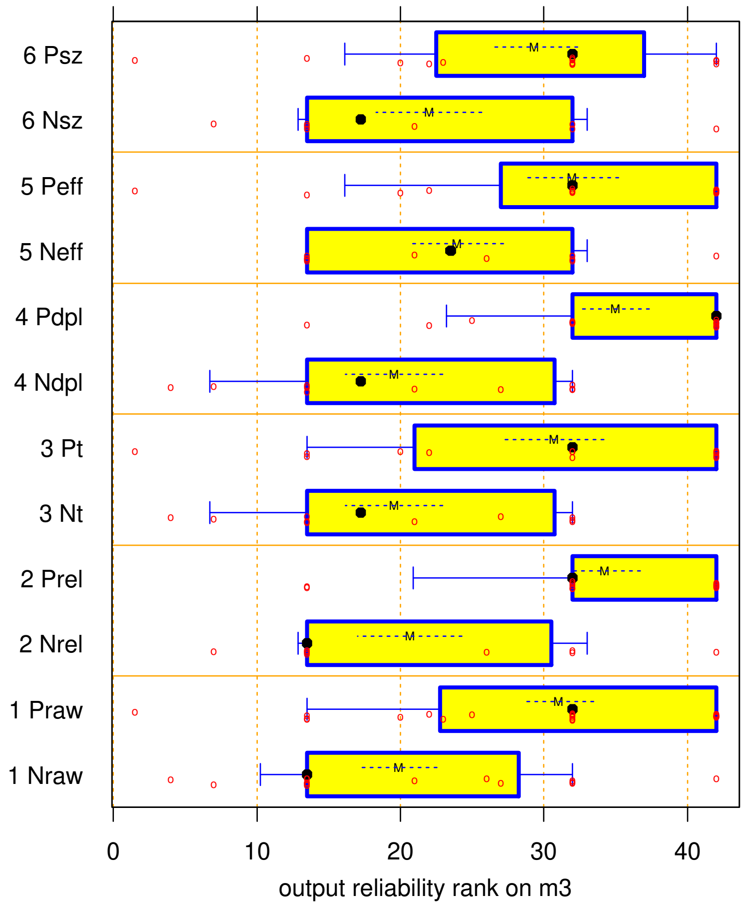


Figure 3.17: rank($rel_{out,m3}$): Rank order of output reliability on the m3 test set. All P groups rank significantly higher than the corresponding N groups (e.g. P_{raw}: mean bootstrap $p = 0.001$, Wilcoxon test $p = 0.003$). The difference is most pronounced in the disciplined and reliable subgroups (P_{dpl}: mean bootstrap $p = 0.000$, Wilcoxon test $p = 0.001$).

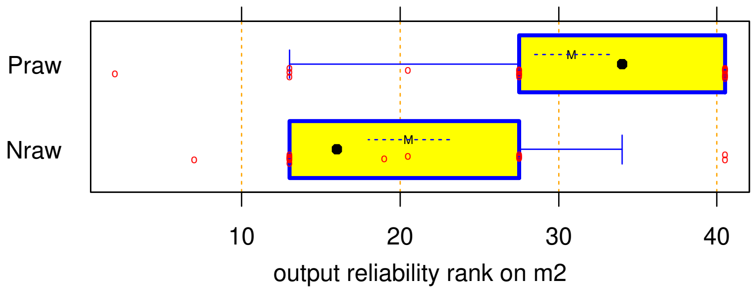


Figure 3.18: rank($rel_{out,m2}$): Rank order of output reliability on the m2 test set.

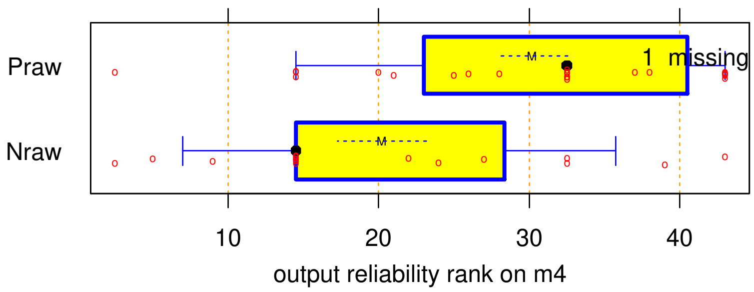


Figure 3.19: rank($rel_{out,m4}$): Rank order of output reliability on the m4 test set.

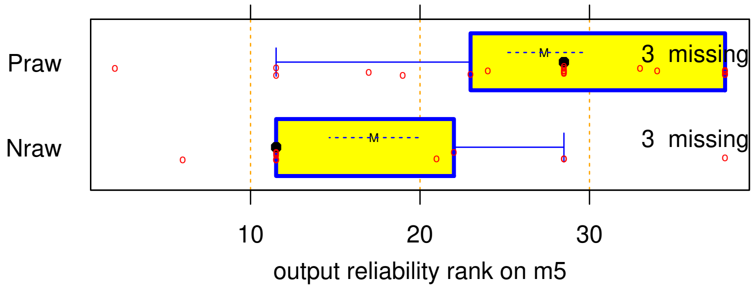


Figure 3.20: rank($rel_{out,m5}$): Rank order of output reliability on the m5 test set.

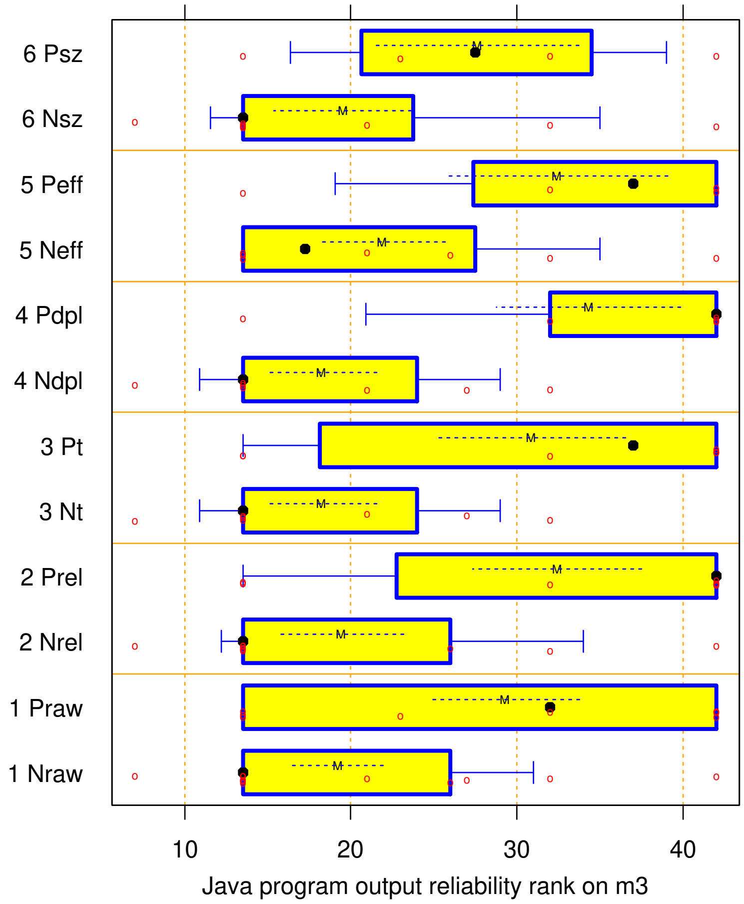


Figure 3.21: rank($rel_{out,m3}$)$|_{Java}$: Rank order of output reliability on the m3 test set, where only the ranks assigned to Java programs are considered. All P groups rank significantly higher than the corresponding N groups (P_{raw}: mean bootstrap $p = 0.02$, Wilcoxon test $p = 0.04$). The difference is most pronounced in the disciplined and reliable subgroups (P_{dpl}: mean bootstrap $p = 0.01$, Wilcoxon test $p = 0.02$).

As we see, all PSP Java subgroups still have a large advantage over the corresponding non-PSP Java subgroups, but the differences are not larger than for all programs. Specifically, the 0.9-confidence minimum rank difference from N_{raw} to P_{raw} has decreased from at least 7 ranks to at least 4. For N_{dpl} to P_{dpl} it has decreased from at least 10 ranks to at least 8. The values are directly comparable, because we have just left the rank numbers of the non-Java programs out of the comparison instead of computing the ranking for the Java programs only. However, most of the decrease comes from the growth of the confidence intervals that is due to the smaller number of data points in this comparison. Nevertheless, these results indicate that the validity of the experiment is not endangered by the fact that more Java programs crash and the fraction of Java users is higher in the N groups. Our conclusion still holds that the P groups produced the more robust programs.

For the more "tame" z4 data, there is still a slight advantage for the P group, but the difference is not significant (raw reliability is shown in Figure 3.22, ranks thereof in Figure 3.23). A still smaller difference is found for the z3 data.

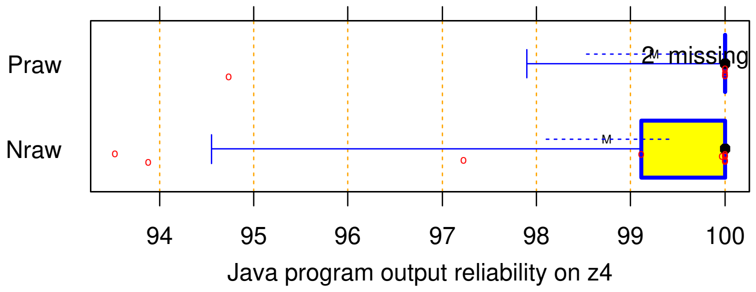


Figure 3.22: $rel_{out,z4}|_{Java}$: Output reliability on the z4 test set for the Java programs only. The P group tends to be better (P_{raw}: mean bootstrap $p = 0.29$, Wilcoxon test $p = 0.17$).

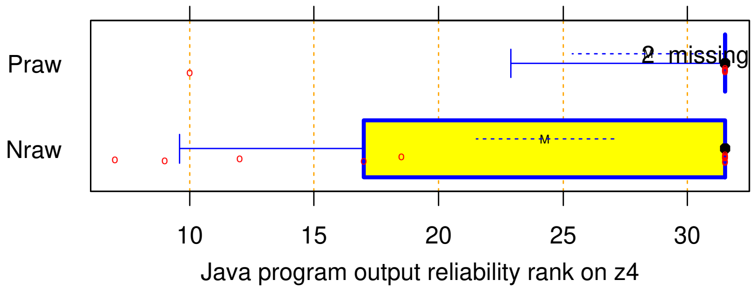


Figure 3.23: rank($rel_{out,z4}$)$|_{Java}$: Rank order of output reliability on the z4 test set for the Java programs only. The P group tends to be better (P_{raw}: mean bootstrap $p = 0.15$, Wilcoxon test $p = 0.17$).

Summing up these two results, we find that the tendency of Java programs to crash should not be attributed to the language, but rather to its careless use in the N group.

3.4.7 Summary

For "normal" inputs, the two groups show hardly any difference in reliability. However, there is a significant advantage for the PSP group when it comes to unexpected kinds of inputs. This difference becomes even more pronounced if we consider only that half of the participants of each group with the higher quality discipline. Together, these observations suggest that using PSP techniques does indeed improve the correctness of programs.

3.5 Release maturity

Do non-PSP programmers have a stronger tendency to deliver software that is not yet reliable? In the context of this experiment there are three ways of assessing this question: the success in the first acceptance test (quantified by either the acceptance rate or the average reliability achieved), the overall number of acceptance tests required, and the fraction of working time that took place *after* the first acceptance test. We will look at all of these.

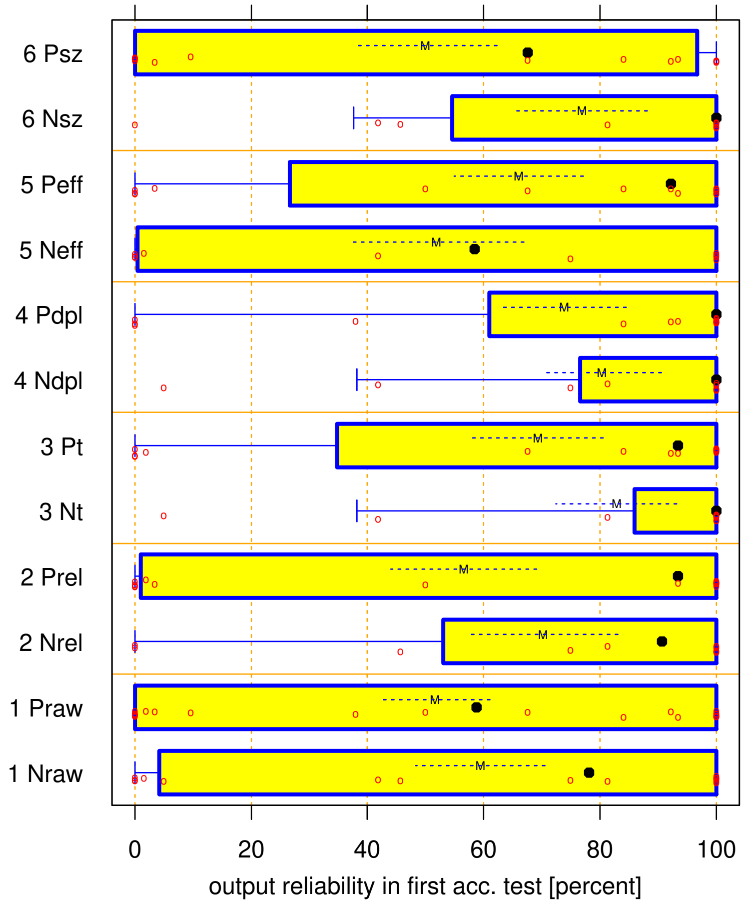


Figure 3.24: $rel_{out,accept1}$: Output reliability achieved in the first acceptance test attempted by each participant. The P group tends to be better for P_{eff}, the N groups elsewhere, but none of the differences are significant, with one exception: In the N group, the smaller half of the programs was more reliable than in the P group (N_{sz}: mean bootstrap $p = 0.05$, Wilcoxon test $p = 0.06$, iqr bootstrap $p = 0.06$).

Figure 3.24 shows the output reliability of the first delivered program, that is, the reliability in the first acceptance test, whether that was successful or not. As we see, there is clearly no advantage for the P group. The slight advantage of several N subgroups is not significant, except for one: The N subgroup with the smallest programs (N_{sz}) is significantly more successful in the first acceptance test than the rest of the group while no such effect is present in P_{sz}, thus provoking a significant difference between N_{sz} and P_{sz}. The success rate gives a similar result: there were 7 out of 16 successful first acceptance tests in the N group (44%) and 8 out of 24 (33%) in the P group. There is an advantage for the N group but the difference is not significant (Fisher exact $p = 0.53$). The cumulative success rate in subsequent acceptance tests is also quite similar for both groups as shown in Figure 3.25

The amount of work that has to be expended after the first attempt at an acceptance test also shows no clear group differences: Both groups spent an average of 17% of their work time after the first test (Figure 3.27) and, except for two outliers in the N group, also a similar absolute amount of time (Figure 3.28). About a quarter of all subjects spend more than 5 hours completing the program they thought was already correct.

Summing up, there is no visible difference in release maturity between the two groups in this experiment.

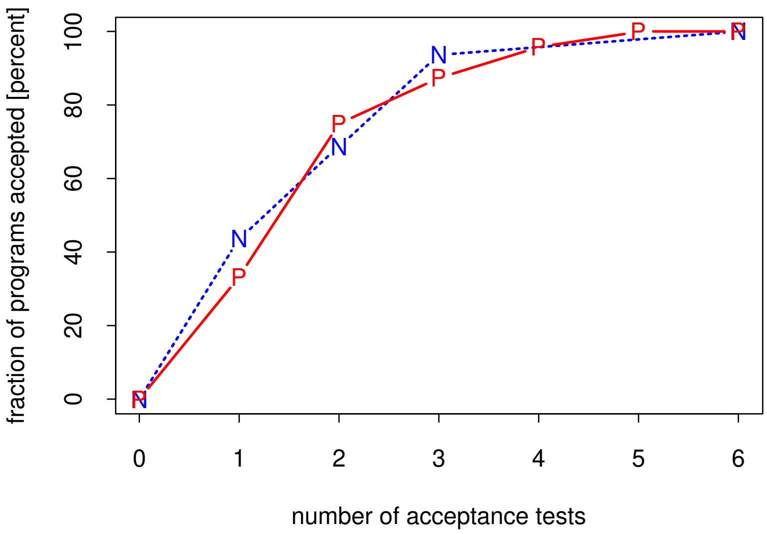


Figure 3.25: Fraction of programs that have been accepted in the two groups depending on the number of acceptance tests performed. The plot is cumulative: once a program was accepted, it is counted in all subsequent acceptance tests as well.

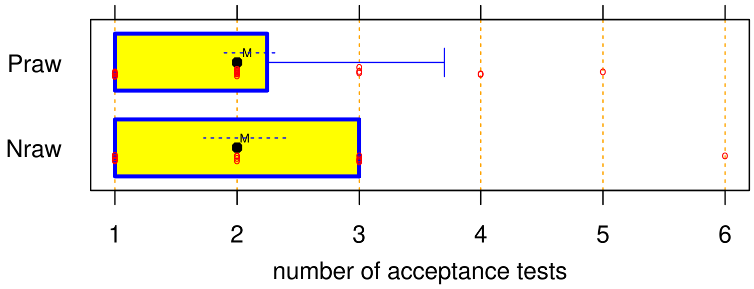


Figure 3.26: N_{accept}: Number of acceptance tests performed by each participant. The differences are not significant.

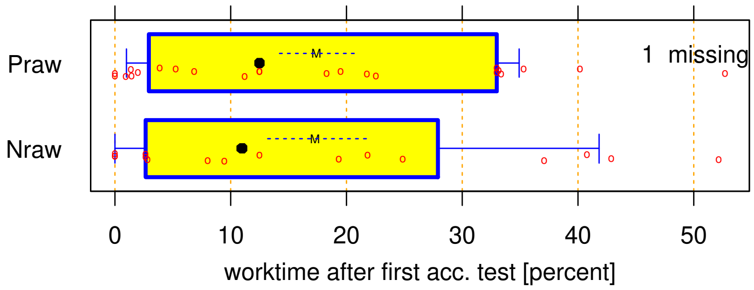


Figure 3.27: $\frac{t_{work} - t_{accept1}}{t_{work}}$: Fraction of total working time spent after the first acceptance test. The differences are not significant.

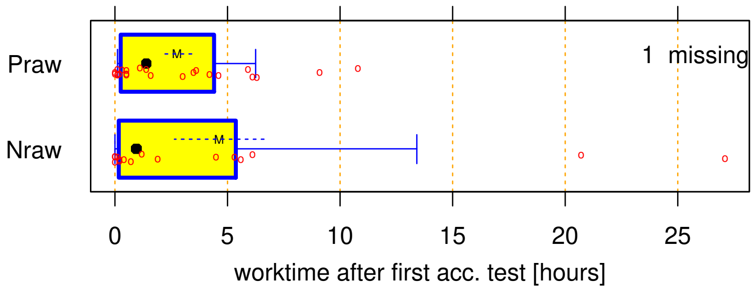


Figure 3.28: $t_{work} - t_{accept1}$: Number of working hours spent after the first acceptance test.

3.6 Documentation

It is obviously impossible to judge the quality of design or program documentation objectively, except by means of another controlled experiment for evaluating the quality. Such an experiment, however, would require multiple subjects for each of the programs from the present experiment and we can obviously not afford such effort.

However, the deliverables in this experiment were program code only and many programmers' source code is notorious for having hardly any comments. Hence, we may use the amount of commenting in the source code as a substitute for the quality of documentation, because the difference will mostly consist of none versus some, not so much of differences in content quality.

For objectively assessing the amount of commenting, we consider the lines (called "lines of code", LOC) making up the program source code and partition the set of lines into the following subsets:

1. *Empty LOC* are all lines containing only white space and also all lines containing only comment consisting of non-letters and non-digits (such as lines of asterisks etc.).
2. *Pure statement LOC* are lines containing meaningful syntactical elements of the programming language, but no comment.
3. *Commented LOC* ($LOC_{commented}$) are lines containing both meaningful syntactical elements of the programming language and comment.
4. *Comment LOC* ($LOC_{comment}$) are lines containing only comment that are not empty lines. Note that this may include statements that were commented out of the program.

Furthermore, we call the union of pure statement LOC and commented LOC *statement LOC* (LOC_{stmt}).

We now express the amount of documentation in both absolute numbers of lines and as a percentage relative to the number of statement LOC. Note that the latter can in principle be larger than 100.

The relative amount of comment lines (see Figure 3.29) shows no difference of location between the full groups, but two of the N subgroups are inferior: in the fast subgroup (N_t) as well as the more disciplined subgroup (N_{dpl}), the N subjects appear to sacrifice documentation for speed. No such effect is present in the corresponding P subgroups.

The relative amount of commented statement lines (see Figure 3.30), is larger in the full P group than in the N group and also in several subgroups (in particular again in the faster subgroup). However, the P group median of 2 percent[3] is still quite small: Only very few (difficult?) statements were commented.

For the absolute numbers of comment lines (Figure 3.31) or commented lines (Figure 3.32), the results are roughly the same as for the percentages.

Summing up, although the PSP participants do not provide more source code documentation overall, they appear to have a somewhat higher willingness to explain a few specific (presumably difficult) statements. In particular, the faster subjects do not comment less than average, which is typical for the non-PSP participants.

3.7 Trivial mistakes

We expect the P group to be more careful with respect to avoiding simple, easy-to-correct mistakes, because the PSP course shows that the resulting defects may sometimes be quite harmful. While the experiment provides

[3] The raw data was rounded to integral percent values

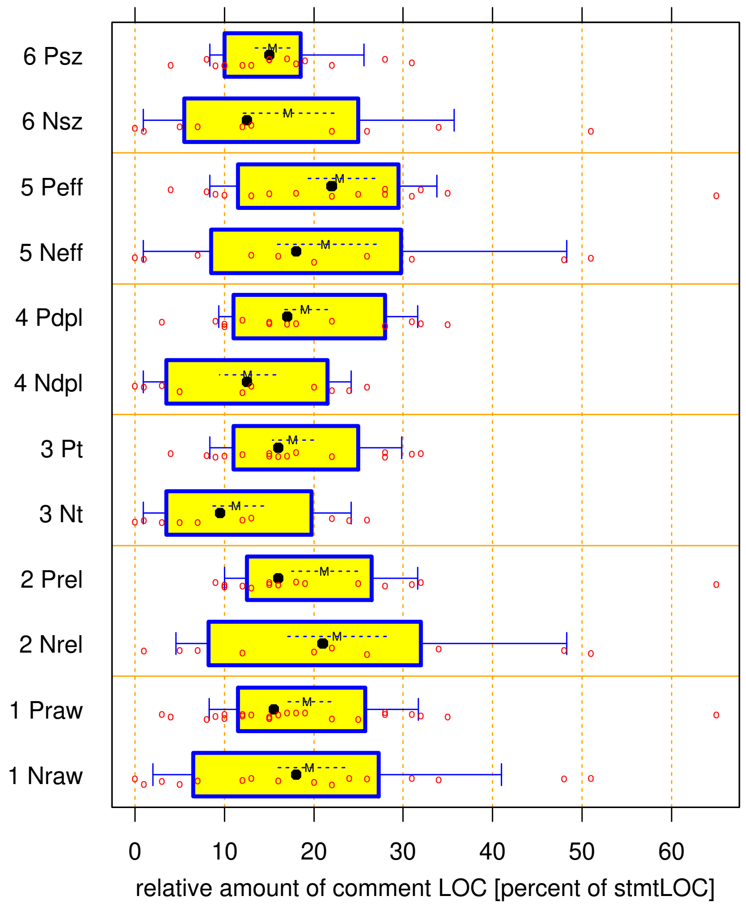


Figure 3.29: $\frac{LOC_{comment}}{LOC_{stmt}}$: Fraction of comment lines in delivered programs (measured in percent of the number of statement LOC). The amount is smaller for N in the fast and the disciplined subgroups (P_t: mean bootstrap $p = 0.04$, Wilcoxon test $p = 0.05$); P_{dpl}: mean bootstrap $p = 0.05$, Wilcoxon test $p = 0.08$).

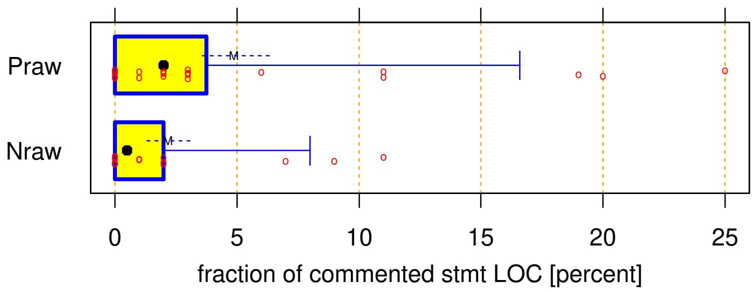


Figure 3.30: $\frac{LOC_{commented}}{LOC_{stmt}}$: Fraction of commented statement LOC in delivered programs (measured in percent of the number of total statement LOC). Several P groups have significantly more commented lines, in particular the full group (P_{raw}: mean bootstrap $p = 0.05$, Wilcoxon test $p = 0.05$) and again the faster subgroup (P_t: mean bootstrap $p = 0.10$, Wilcoxon test $p = 0.06$).

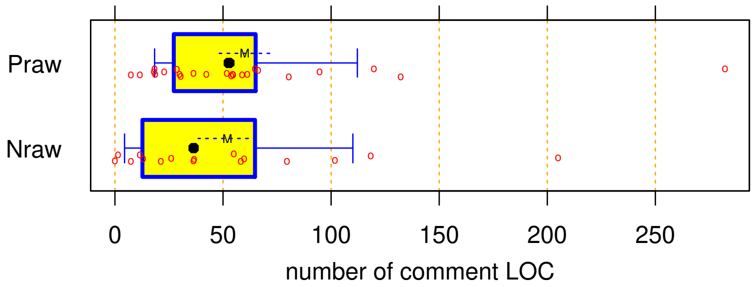


Figure 3.31: $LOC_{comment}$: Absolute number of pure comment lines in delivered programs. There is significantly more comment in the P groups for the more disciplined subgroup (P_{dpl}: mean bootstrap $p = 0.04$, Wilcoxon test $p = 0.07$) and the faster subgroup (P_t: mean bootstrap $p = 0.01$, Wilcoxon test $p = 0.03$); the difference is not significant for the full group (P_{raw}: mean bootstrap $p = 0.33$, Wilcoxon test $p = 0.23$).

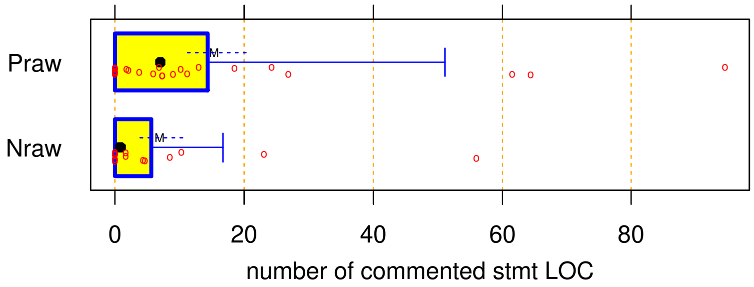


Figure 3.32: $LOC_{commented}$: Absolute number of commented statement LOC in delivered programs.

no direct insight into the frequency of simple programming errors in general, we have observations (at least for many of the participants) of the number of compilations they executed and the number of these compilations that went through without a compiler error message ("clean compiles"). A high fraction of clean compiles is a good indicator of making few trivial mistakes and an overall small number of compilations can be interpreted similarly (although this is less reliable).

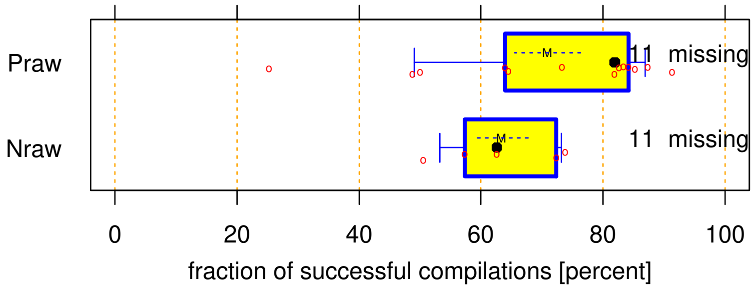


Figure 3.33: $\frac{N_{comp,OK}}{N_{comp}}$: The fraction of compilations without a compiler error message. This data is missing for many participants. For the rest, the P group tends to be more successful (P_{raw}: mean bootstrap $p = 0.14$, Wilcoxon test $p = 0.12$).

The fraction of compilations that were successful, i.e., that did not produce an error message from the compiler is shown in Figure 3.33. The values are indeed somewhat higher for the P group, except for one outlier with only 25% success. The median success rate of 82% for the P group is quite good, in particular given the much smaller number of overall compilations shown in Figure 3.34.

Let us now normalize the number of compilations by the total work time or by the program size. The compilation frequency (number of compilations per hour) is clearly higher in the N group, also suggesting less care for trivial mistakes (Figure 3.35).

The average compilation density (number of compilations per program statement) as shown in Figure 3.36 is about twice as high in the N group compared to the P group. The small absolute values of this statistic indicate

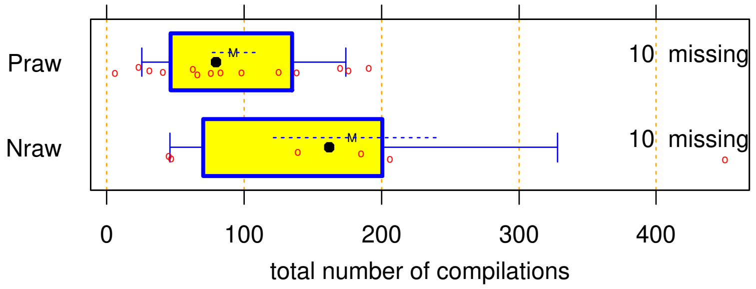


Figure 3.34: N_{comp}: Total number of compilations. This data is missing for many participants. For the rest, the P group compiles less often (P_{raw}: mean bootstrap $p = 0.05$, Wilcoxon test $p = 0.06$).

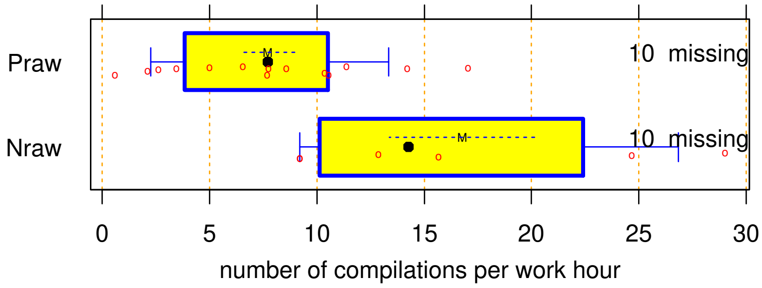


Figure 3.35: $\frac{N_{comp}}{t_{work}}$: Number of compilations per work hour. This data is missing for many participants. For the rest, the P group compiles much less frequently (P_{raw}: mean bootstrap $p = 0.00$, Wilcoxon test $p = 0.01$).

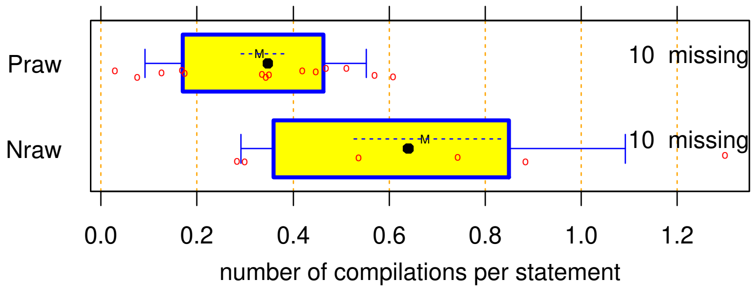


Figure 3.36: $\frac{N_{comp}}{LOC_{stmt}}$: Number of compilations per resulting program statement. This data is missing for many participants. For the rest, the P group compiles much less frequently (P_{raw}: mean bootstrap $p = 0.00$, Wilcoxon test $p = 0.04$).

that subjects in both groups either preferred a rather incremental development style or had lots of compilations during the test phase.

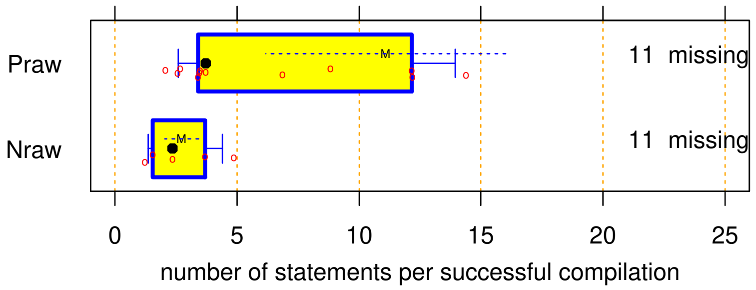


Figure 3.37: $\frac{LOC_{stmt}}{N_{comp,OK}}$: Number of resulting program statements per successful compilation. There is one outlier at 68 in the P group. This data is missing for many participants. For the rest, the P group produces many more statements between any two clean compiles (P_{raw}: mean bootstrap $p = 0.00$, Wilcoxon test $p = 0.03$).

The same proportion holds for the number of successful compilations per program statement (see Section 3.6). Looking at the inverse of this value, the number of final program statements written per successful compilation (Figure 3.37), we find that many of the N participants obviously did quite a lot of trial and error changes (presumably in the later stages of the development). The P group median and mean are roughly 50% and 300% larger, respectively, than those of the N group. One P subject achieved the remarkable value of 68 LOC per clean compile and three quarters of the P subjects are as good or better than the best quarter of the N subjects. The maximum value in the N group is only 5 lines! On the other hand, even in the P group half of the participants achieve less than 4 lines.

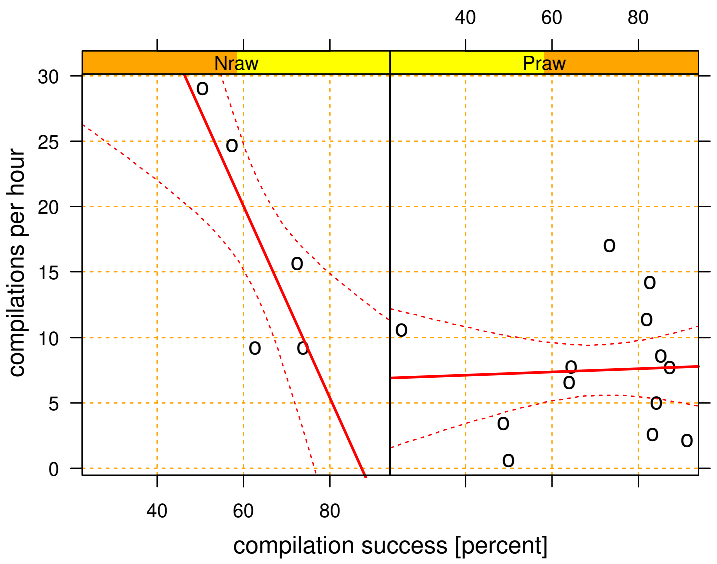


Figure 3.38: Number of compilations per work hour ($\frac{N_{comp}}{t_{work}}$) by fraction of successful compilations ($\frac{N_{comp,OK}}{N_{comp}}$). This data is missing for many participants. The plot includes a linear regression trend line with its 80% confidence band.

The plot of compilation frequency (number of compilations per hour) versus compilation success (fraction of clean compiles) as shown in Figure 3.38 indicates that the compilation frequency of the N group is dramatically higher for subjects with low compilation success. The P subjects seem to choose their compilation frequency much more by themselves. Note that the effect in the N group is much too large to be explainable by the additional recompilations of unsuccessful program versions alone. As we will see now, the reason is probably a generally much less careful development by these subjects.

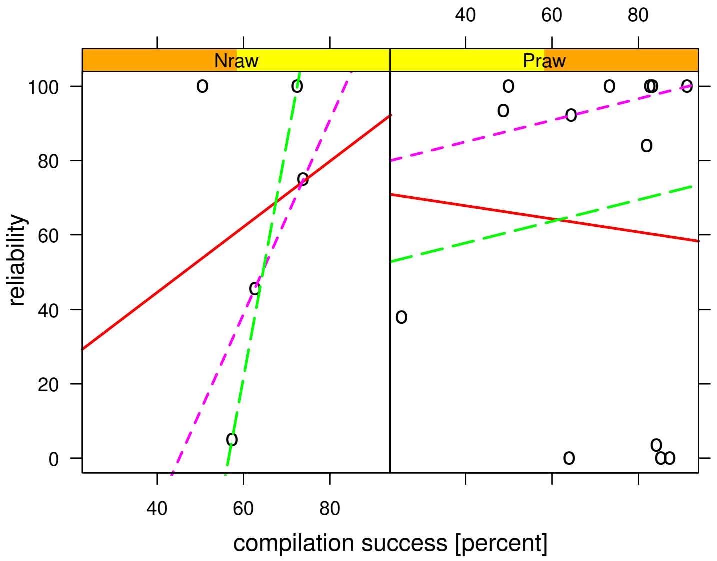


Figure 3.39: Output reliability in first acceptance test ($rel_{out,a1}$) depending on the compilation success ($\frac{N_{comp,OK}}{N_{comp}}$) in percent. This data is missing for many participants. The continuous red line is a normal least squares linear regression line, the dashed magenta line is a robust L1 least distances regression line, and the long-dashed green line is a resistant 10%-trimmed least squares regression line.

Interestingly, there is a dependency of the reliability achieved in the first acceptance test on the compilation success. Figure 3.39 shows a scatter plot of these two values, decorated with various trend lines. The plot shows that for the N group all of these trend lines indicate decidedly increasing reliability for increased compilation success. In contrast, the P group shows no consistent trend at all. Some of this effect is also visible in similar plots based on the reliability of the final program (not shown here).

Summing up, as far as the data is available we find more care for the seemingly trivial details of program development in the P group compared to the N group. Furthermore, we find some indications that lack of such care, as exhibited by some members of the N group, may directly relate to problems later in development (here: the first acceptance test).

3.8 Productivity

Although improving the productivity is *not* a major goal of the PSP course, the PSP education may increase the productivity as a side effect of defect prevention and earlier defect detection. In this experiment, there are basically two possible views of productivity we can assess. First, as all programs solve the same task, we can simply define productivity as the inverse of the total working time required. Second, we may also use the more common view of productivity as the number of LOC written per working hour; see Section 3.6.

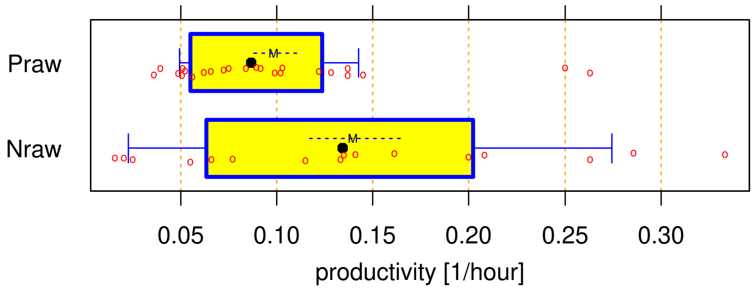


Figure 3.40: $\frac{1}{t_{work}}$: Productivity measured as the inverse of total working time (number of programs per hour, one could say). The productivity is higher in the N group (N_{raw}: mean bootstrap $p = 0.06$, Wilcoxon test $p = 0.10$, and even more pronouncedly in the subgroups N_{sz}, N_{dpl}, and N_t), but is more stable (less variable) in the P group (P_{raw}: iqr bootstrap $p = 0.05$).

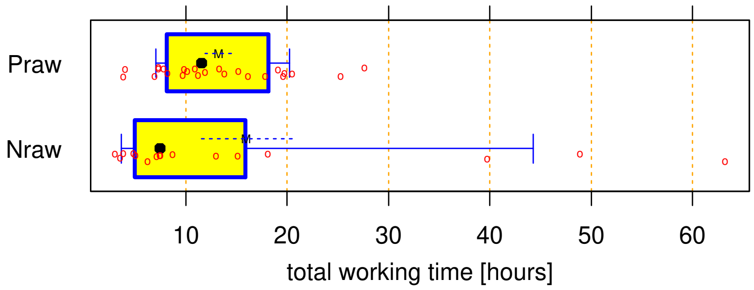


Figure 3.41: t_{work}: Total number of working hours (which can be interpreted as cycle time). The N group has a lower median (N_{raw}: Wilcoxon test $p = 0.10$), but a slightly higher mean due to some very slow subjects (P_{raw}: mean bootstrap $p = 0.28$).

We find that the average productivity as measured by inverse working time alone is actually *lower* in the P group (Figure 3.40). However, the variability is once again much smaller in the P group. This may mean that in real software development the productivity of a *team* of PSP participants might actually be similar to or even better than that of a team of non-PSP participants, because the unpredictability of productivity is a major risk factor for the cycle time in a team software development. This is also nicely illustrated by the cycle time produced by the individuals alone (Figure 3.41): If you pick one subject from each group, chances are the N subject is faster (lower median). However, if you pick a whole team, chances are the P group is faster (lower mean). This weird effect happens because all of the really bad cycle times, which could put a project at risk, occurred in the N group (despite it being smaller in size!)

If we take program length into account and measure productivity by the number of statements written per hour as shown in Figure 3.42, the difference almost disappears (N_{raw}: mean bootstrap $p = 0.30$, Wilcoxon test $p = 0.42$) but the larger variance of the N group remains. As a consequence of this higher variance, the faster N subgroup is significantly more productive than the faster P subgroup (N_t: mean bootstrap $p = 0.09$, Wilcoxon test $p = 0.12$). Conversely, P would be slightly faster if we looked at the slower half.

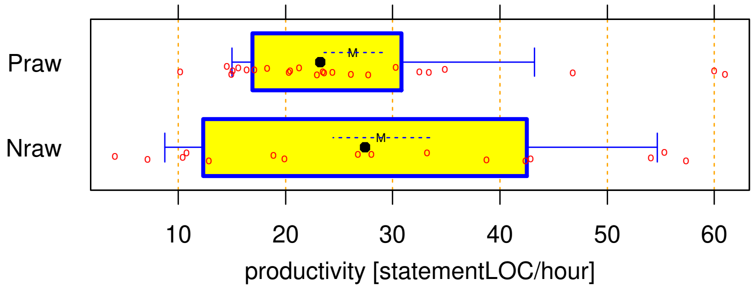


Figure 3.42: $\frac{LOC_{stmt}}{t_{work}}$: Productivity measured as the number of statement LOC written per hour. The differences of mean and median are not significant, but the variability is smaller in the P group (P_{raw}: iqr bootstrap $p = 0.04$).

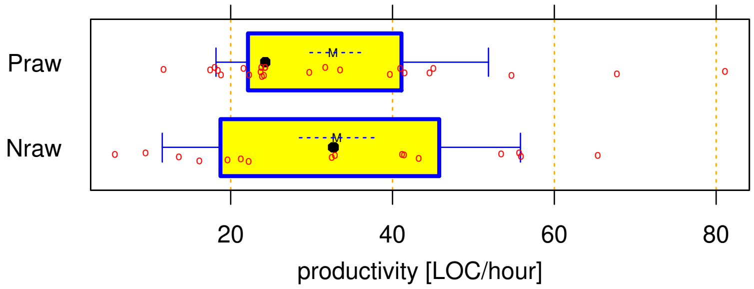


Figure 3.43: $\frac{LOC_{stmt}+LOC_{comment}}{t_{work}}$: Productivity measured as the number of total LOC (including comments) written per hour. The differences of mean and median are not significant (N_{raw}: mean bootstrap $p = 0.42$, Wilcoxon test $p = 0.49$), but the variability tends to be lower in the P group (P_{raw}: iqr bootstrap $p = 0.14$).

If we count comments as LOC as well (in addition to the statements), the difference disappears entirely, but the lower variability remains; see Figure 3.43.

Summing up, productivity tended to be somewhat lower (rather than higher) in the P group, but also more stable (less variable), hence inducing much less development risk.

3.9 Quality judgement

The PSP course teaches estimation not only of development times, but also of an expected number of defects. Therefore, it is plausible that the PSP participants make more accurate judgement of the quality of their own *delivered* program.

To assess this effect, two questions in the postmortem questionnaire asked for the expected number of defects remaining in the delivered program and the estimated frequency of program failures. We cannot determine the actual number of defects, but will compare the reliability estimation to an actual value.

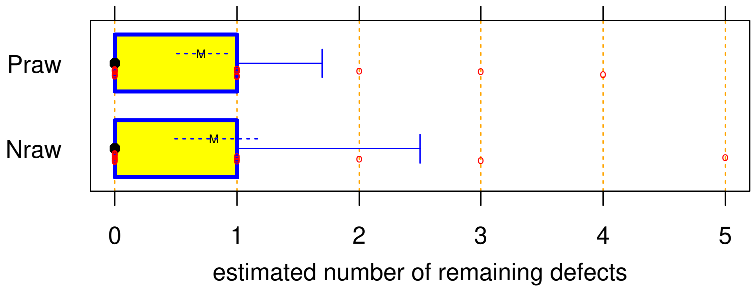


Figure 3.44: $n_{defects,estim}$: The subjects' postmortem estimation of the number of defects in the delivered program. The differences are not significant.

First of all, the P group on average was not more optimistic of their programs' remaining defect content than the N group (Figure 3.44). As for the reliability (Figure 3.45), the P group tended to be slightly more optimistic — and rightly so: a plot of the quotient of estimated and actual input reliability shows that more than half of

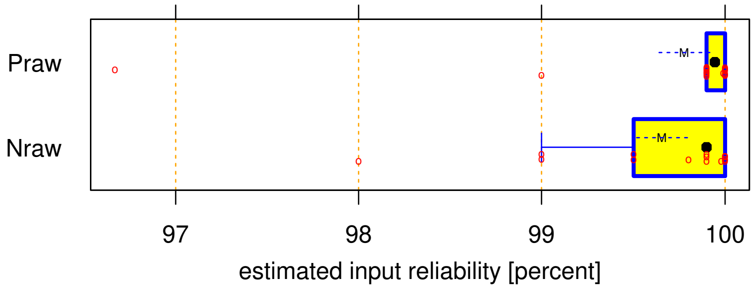


Figure 3.45: $rel_{in,estim}$: Subject's postmortem estimation of the input reliability of the delivered program. The differences of mean and median are not significant (mean bootstrap $p = 0.25$, Wilcoxon test $p = 0.22$), but the variability is clearly smaller in the P group (iqr bootstrap $p = 0.03$).

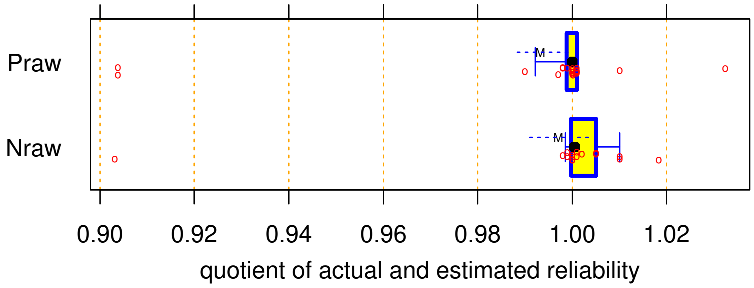


Figure 3.46: $\frac{rel_{in,z3}}{rel_{in,estim}}$: Quotient of actual input reliability for the "tame" z3 test set and subject's postmortem estimate of input reliability. Median, and variability tend to be smaller in the P group (P_{raw}: mean bootstrap $p = 0.33$, Wilcoxon test $p = 0.14$, iqr bootstrap $p = 0.17$).

the P group's estimates are within a factor of 0.0015 (1.5 per thousand) of the actual value, while a few more of the N participants were somewhat too pessimistic about their programs' reliability (Figure 3.46), resulting in a somewhat lower variability of the estimation accuracy for the P group.

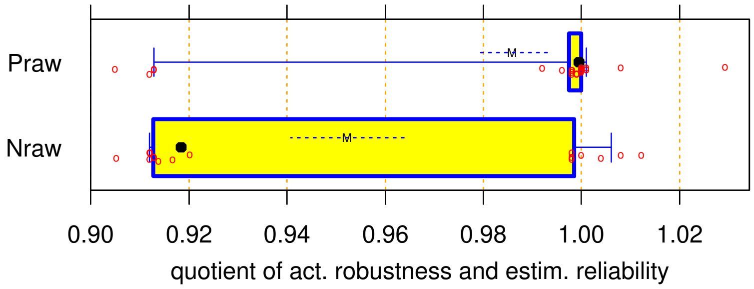


Figure 3.47: $\frac{rel_{in,m3}}{rel_{in,estim}}$: Quotient of input reliability for the "difficult" m3 test set and subject's postmortem estimate of input reliability. Note that the huge difference in box size is misleading; it is due to the large empty gap between 0.99 and 0.92. Still, the P group estimates are significantly better (P_{raw}: mean bootstrap $p = 0.01$, Wilcoxon test $p = 0.03$) and less variable (P_{raw}: iqr bootstrap $p = 0.07$).

In contrast, most estimates were too optimistic when judged against the robustness test set m3 — and much more so for the N group than for the P group (Figure 3.47), due to the lack of robustness of many N programs .

Some participants in both groups estimated their program to contain one or even several defects, but still be at least 99.9% reliable.

Summing up, the quality judgement is indeed both more accurate and more reliable in the P group.

3.10 Efficiency

Without any real expectation what to find we are also interested in comparing the efficiency of the programs created by our two groups. We assess efficiency in terms of run time as well as memory consumption. We can use only the z2 and z3 test runs, as all others may be biased, either due to programs that crashed prematurely or due to missing values for the most inefficient programs; see Sections 3.4.4 and 3.4.5.

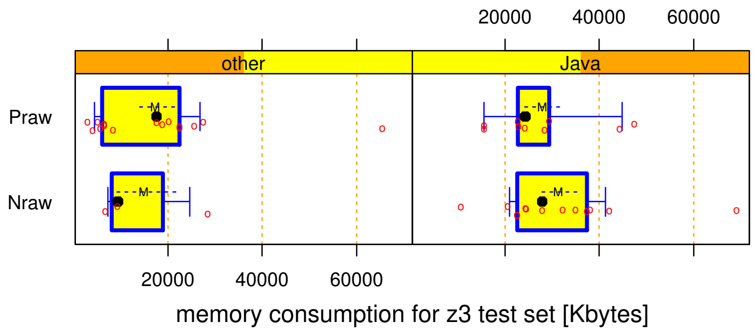


Figure 3.48: mem_{z3}: Memory consumption on the z3 test set compared for the P group (top row) versus the N group (bottom row), separately for the Java programs (right diagram) and the programs written in other languages (left diagram). The differences are significant neither for the Java programs nor for the programs in other languages.

Obviously for these measures it would be misleading to ignore the programming language of the programs, because Java programs as such have (at least as of JDK 1.2) both much higher run time and higher memory consumption than equivalent programs in any of the other languages. We could try and put all of the latter into one bin and the Java programs into another and compare these subgroups separately as shown in Figure 3.48. Unfortunately, there are only 3 non-Java programs for the N group (2 C++ programs and 1 C program) and we cannot draw useful conclusions from such a small set. Therefore, in the following we will compare only the Java programs from the N group to the Java programs from the P group.

As we see in Figure 3.48, there is a slight trend that P group memory consumption might be smaller, but the difference is insignificant. A very similar trend occurs for the z2 test. Note that the average programs use about three times as much memory as the greediest one.

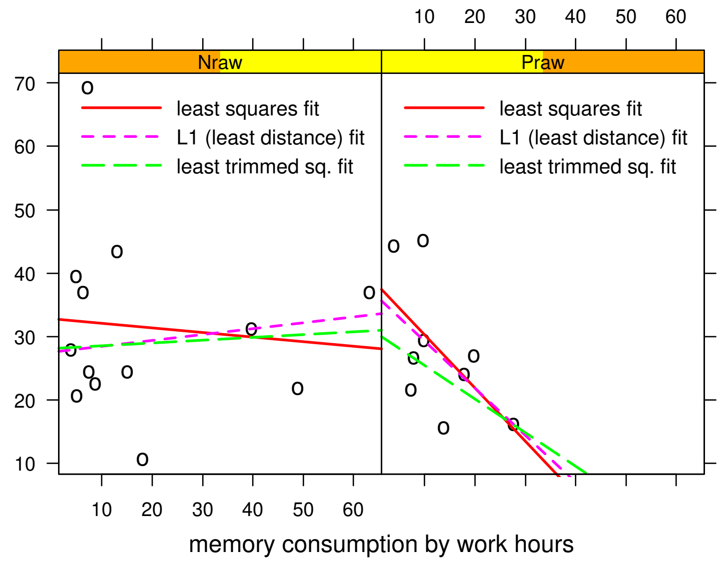


Figure 3.49: Memory consumption mem_{z3} in Megabytes (vertical) depending on total subject working hours t_{work} (horizontal). The continuous red line is a standard least squares linear regression line, the dashed magenta line is a robust L1 least distances regression line, and the long-dashed green line is a resistant 20%-trimmed least squares regression line.

There is a trend that the P group Java programs consume less memory when the participant worked on them longer. No such trend is visible for the N group; see Figure 3.49. Unless this effect is spurious, it may mean that PSP participants tended to actively work towards lower memory consumption and invested time to achieve it, whereas the memory requirements of the non-PSP participants "just happened".

For the run times, there is no salient group difference whatsoever for the z3 test runs (Figure 3.50) nor for the z2 test runs.

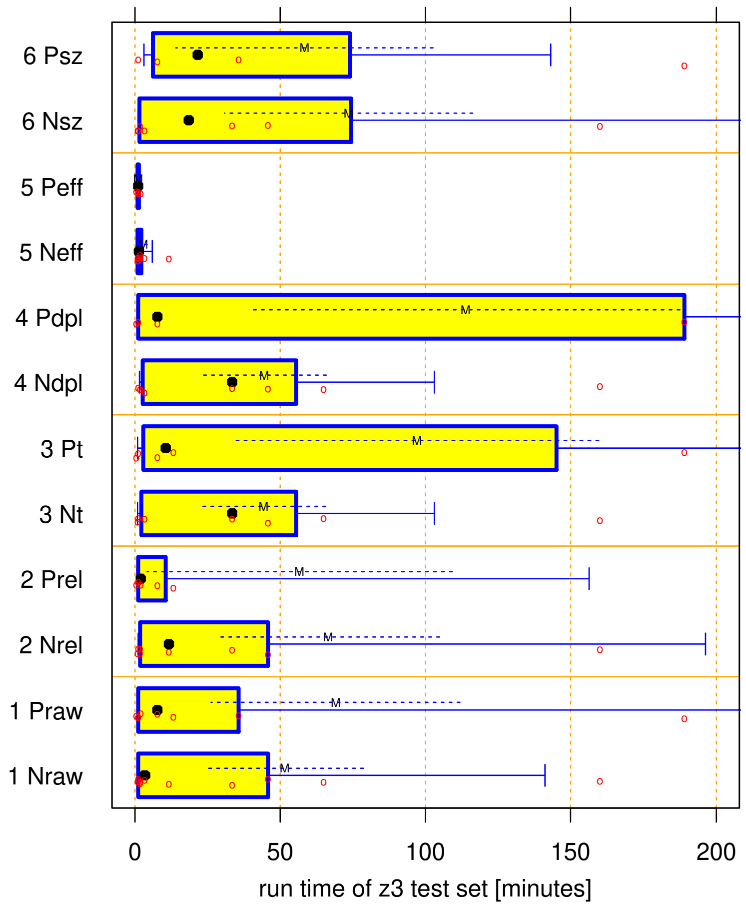


Figure 3.50: $t_{run,z3}$: Run time (in minutes) for the Java programs on the z3 test set. There is one outlier in the N Java group at 342 and one in the P Java group at 371. The overall group differences are not significant.

Summing up, no differences in program efficiency between the two groups could be found, neither for run time, nor for memory consumption.

3.11 Simplicity

As still no good general measures for software complexity (or simplicity) are available, we compare the groups on two of the simplest and most useful alternative measures instead, namely program length (see the definition of LOC in Section 3.6) and the number of decisions in the program (McCabe cyclometric complexity). Both of these measures mostly reflect size, which usually is the dominant aspect of program complexity.

As we see in Figure 3.51, the programs are noticeably longer in the P group. Figure 3.52 shows that the median number of decisions in the program is larger in the P group as well, but the means are the same. The variance once again tends to be larger in the N group, however.

It is not at all clear how to interpret these results. More statement lines per decision (as in the P group) may result in less dense, easier-to-read code but could also be the result of fussy, inelegant coding style. Summing up, the results are inconclusive with respect to simplicity.

3.12 Analysis of correlations

All of the above analyses were more or less hypothesis-driven: we searched for specific expected effects. However, we also performed an entirely data-driven analysis in order to possibly find further, surprising effects.

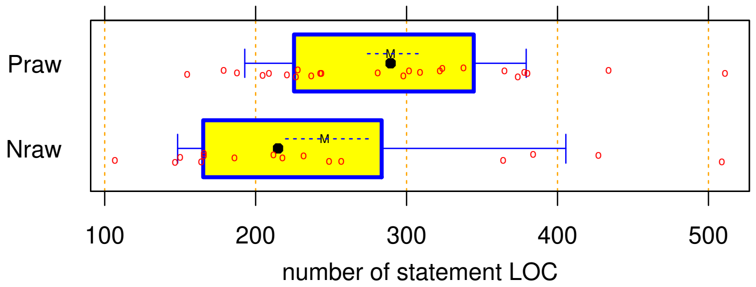


Figure 3.51: LOC_{stmt}: Total number of statement lines of code in the delivered programs. The N programs are significantly smaller (N_{raw}: mean bootstrap $p = 0.09$, Wilcoxon test $p = 0.04$). Similar differences occur in the subgroups as well, least pronounced for the efficient programs (N_{raw}: mean bootstrap $p = 0.41$, Wilcoxon test $p = 0.20$).

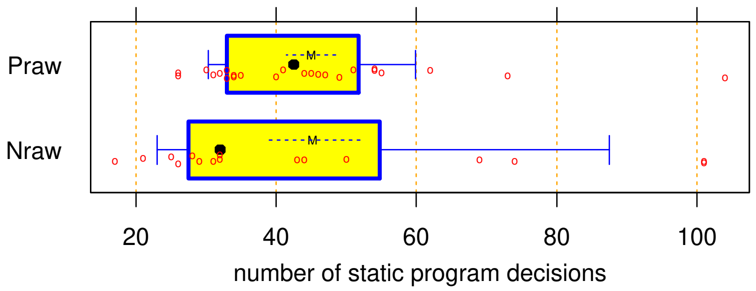


Figure 3.52: $n_{decisions}$: Total number of program decisions in the source program. This value is an approximation to the McCabe cyclometric complexity and was determined by a simple keyword count. It is the number of `if`, `while`, `catch`, `break` keywords etc. Note that counting `break` is correct in `switch` statements but unnecessary elsewhere. The N group tends to have a smaller median (N_{raw}: Wilcoxon test $p = 0.13$), but the same mean (P_{raw}: mean bootstrap $p = 0.49$) and slightly higher variability (P_{raw}: iqr bootstrap $p = 0.27$).

We computed all pairwise correlations and pairwise rank correlations of 34 different variables measured in the experiment and tabulated the results ordered by the absolute size of the correlation. This process found 103 correlations with a size greater than 0.4. We investigated all of them and sorted out the many obvious ones as well as the spurious ones. As an example of an obvious correlation, the high correlation between input and output reliabilities is uninteresting, the high correlation between programming experience in years and the size of the largest program ever written is quite expected, etc. On the other hand, we found, for example, a rank correlation of 0.59 between the number of compilations and the run time of the m3 reliability test. This is spurious and misleading, because the m3 run times ranks reflect two dissimilar subpopulations, namely the successful and the crashed runs. Much of what this correlation reflects is just noise.

However, after sorting out the garbage, we found a few pearls as well.

3.12.1 How time is spent

The length of the program written is a predictor of the total working time (and similarly of the time to the first acceptance test). See Figure 3.53 for a plot of the relationship. Put differently, the people who took longer for solving the task typically spent their time actually writing and debugging more statements.

Since most of the subjects' time was spent removing defects from the programs, this is an impressive indicator

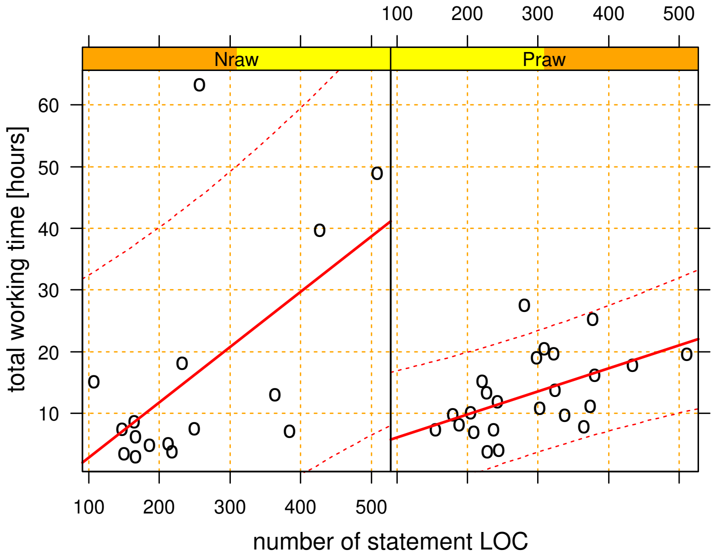


Figure 3.53: Total working time t_{work} depending on the length LOC_{stmt} of the final program. The relationship is closer for the P group.

that there is something as a constant defect density in programs — one should strive for compact programs, at least for this programming task.

An alternative explanation would be to assume that the less capable programmers (who would take longer in any case) are also the ones writing the longer programs, because they are not able to produce compact code. A mix of both explanations is probably the most accurate model for understanding this data.

In any case, the relationship is closer in the P group (correlation $r = 0.57$) than in the N group ($r = 0.51$), which represents another instance of more stable, predictable performance due to PSP training.

3.12.2 Better documentation saves trivial mistakes

One rather interesting relationship was found in the answers in the postmortem questionnaire. The subjective evaluation of the quality of the description of procedures embedded in the program correlates with the fraction of compilations that were successful ($r = -0.57$). The better the subjects judged their own descriptions, the higher was their fraction of compilations without compiler error messages, which increases from about 49% for participants with “bad” descriptions to about 84% for “very good” ones. However, the trend is not statistically significant and may hence be spurious. The same effect, only weaker, is found for the design descriptions. The difference between the groups is unclear, because there are only five valid data points in the N group for this measure.

3.12.3 The urge to finish

We also found an interesting correlation between four correlations. There are obviously correlations between the estimated working time and either the actual working time and the time to the first acceptance test. We can build one-parameter linear models to predict the latter two variables from the estimated time: $t_{work} = c_{tot} \cdot t_{estim}$ and $t_{accept1} = c_{acc} \cdot t_{estim}$. We compute the parameters separately for the N and P groups. Comparing the optimal c_{tot} parameters for these models, we find the following:

$$t_{work} = 1.50 \cdot t_{estim} \text{ explains 82\% of the variance in P}$$

$$t_{work} = 2.49 \cdot t_{estim} \text{ explains 54\% of the variance in N}$$

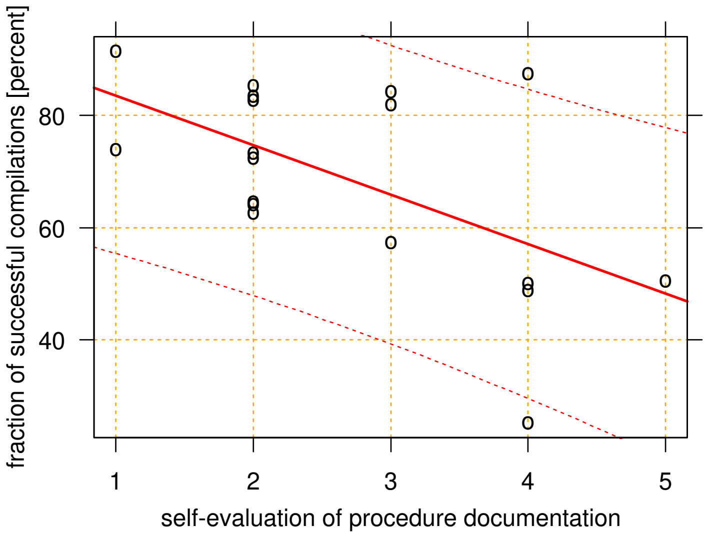


Figure 3.54: The fraction of successful compilations $\frac{N_{comp,OK}}{N_{comp}}$ depending on the self-evaluation of the quality of the procedure descriptions (documentation) in the code on a scale from 1 (very good) to 5 (bad).

These values just reflect the fact (already discussed in Section 3.3) that the estimations are better and their errors are less variable in the P group. However, we also find the following for c_{acc}:

$$t_{accept1} = 1.20 \cdot t_{estim} \text{ explains 86\% of the variance in P}$$

$$t_{accept1} = 1.65 \cdot t_{estim} \text{ explains 52\% of the variance in N}$$

The new insight here is that, for the P group, the quality of the model for $t_{accept1}$ is higher than that for t_{work}, whereas there is no such effect in the N group. A possible interpretation is as follows: The P group feels more urged to finish within the predicted time than the N group and has a stronger inclination to pick the time for the first acceptance test close to the predicted total work time. This is obviously not a good idea, calling for an acceptance test should be driven by product reliability, not by expired working time.

3.13 Subjects' experiences

In this section we analyze what the participants reported in their postmortem questionnaire.

The first section of that questionnaire revolved around the quality of the delivered program, concerning its documentation, complexity, extensibility, and reliability. There are hardly any differences between the groups in the answers to most of these questions. An exception is the documentation of procedures and difficult individual statements. The participants in the P group judged their own description of the procedures better than the N group participants theirs, as shown in Figure 3.55.

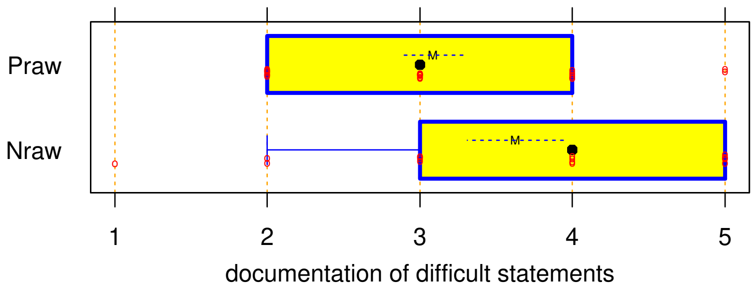


Figure 3.55: Self-evaluation of the quality of the documentation of individual difficult statements in the program, on a scale from 1 ("very good") to 5 ("bad"). Both mean and median are better in the P group (mean bootstrap $p = 0.06$, Wilcoxon test $p = 0.06$.)

A similar, slightly weaker difference is found for the description of the procedures. Although the difference for statement documentation is statistically significant, it may be accidental, because it is not backed up by similar differences in any of the other questions about documentation quality (program usage, program purpose, design ideas, data types, variables). The questions about unnecessary program complexity in data types, control flow, or algorithm and about understandability and extensibility of the program also show no clear group differences.

Asked for the number of defects they believed were still in their program, a majority of participants from both groups believed there were none. The rest of the distributions are also very similar. Ironically, a large fraction of participants answered "no defects" here but then in the immediately following question admitted they expected their program to fail sometimes. This happened with 31% of the answers in the N group and 37% in the P group. See Section 3.9 for a discussion of the estimated reliability.

The second section of the postmortem questionnaire asked where the participants saw their greatest weaknesses and which means they consider suitable for improvement.

For the question "where could you most reduce the number of defects?", the most frequent answer was "design". The profile of the answers looks quite similar in the two groups. Asked *how* defects could best be reduced, the most frequent answer was "design review" in both groups. However, the second most frequent answer, "code reviews", was given much more often in group N.

As for increasing productivity, again the most frequent answer was "more design" and was given about as often in one group than in the other.

3.14 Mean/median/iqr overview table

The following multipage table summarizes some of the results again in numeric form. It contains data for the following variables

1. amount of work time mis-estimation (see also Figure 3.3 on page 28)
2. reliability on the "tame" z inputs (see also e.g. Figure 3.11 on page 34)
3. reliability on the difficult m inputs ("robustness", see also e.g. Figure 3.16 on page 36)
4. reliability in the first acceptance test ("release maturity", see also Figure 3.24 on page 40)
5. comment density in the delivered program (see also Figure 3.29 on page 43)
6. productivity (inverse of total work time, see also Figure 3.40 on page 47)
7. length of the delivered program (see also Figure 3.51 on page 52)

For each of these variables, the data for all subgroups (as described in Section 3.2 on page 25) is presented. For each subgroup, three different statistics are shown: mean, median, and interquartile range (iqr, as described in Section 3.1.1 on page 20). For each statistic, the value of both the P group and the N group is shown, the ratio of these values, and an 80-percent confidence interval for this ratio (computed by Bootstrap resampling with 1000 iterations, which will typically result in a 2-digit accuracy).

The rightmost column indicates which group was "better", where smaller iqr, underestimation, and program length is considered better, and larger reliability, comment density, and productivity. The name of the better group is a dash if the ratio is within the range 0.99 to 1.01, is printed in a small font if the confidence interval includes 1.00, and is printed in boldface if it does not. The latter is equivalent to statistical significance on a 0.1 level. Note that this test uses a ratio rather than a difference criterion and so will sometimes give results different from the tests presented in the main text and captions of the previous sections. Overall, however, the conclusions are the same.

groups	statistic	P	N	ratio P/N	80% confidence interval	better
VARIABLE $\lvert t_{work}/t_{estim} - 1\rvert$ (time estimation inaccuracy):						
P_{raw}/N_{raw}	mean	98.4	163	0.605	0.374…1.08	P
.	median	62.6	74.5	0.84	0.495…1.76	P
.	iqr	103	129	0.799	0.307…1.69	P
P_{rel}/N_{rel}	mean	89.3	145	0.615	0.36…1.14	P
.	median	61.7	76.1	0.811	0.407…1.53	P
.	iqr	58.5	119	0.492	0.16…1.59	P
P_t/N_t	mean	60.4	39.8	1.52	0.978…2.29	N
.	median	43.9	33.5	1.31	0.621…2.24	N
.	iqr	46.4	47.4	0.98	0.57…3.27	P
P_{dpl}/N_{dpl}	mean	55.9	101	0.552	0.298…1.54	P
.	median	46	47.2	0.974	0.575…1.84	P
.	iqr	46.4	49.8	0.933	0.456…2.49	P
P_{eff}/N_{eff}	mean	75.3	225	0.335	0.195…0.687	**P**
.	median	61.7	112	0.553	0.215…1.25	P
.	iqr	72.9	199	0.366	0.0947…0.916	**P**
P_{sz}/N_{sz}	mean	76.7	74.5	1.03	0.611…1.88	N
.	median	43.9	63.2	0.694	0.44…2.04	P
.	iqr	109	55.4	1.96	0.38…3.13	N
VARIABLE $rel_{out,z3}$ (reliability for "tame" inputs):						
P_{raw}/N_{raw}	mean	90.4	93.6	0.966	0.862…1.08	N
.	median	100	100	1	1…1	—
.	iqr	1.15	0.476	2.41	0…∞	N
P_{rel}/N_{rel}	mean	100	100	1	1…1	—
.	median	100	100	1	1…1	—
.	iqr	0	0	—	—	—
P_t/N_t	mean	85	89.7	0.948	0.799…1.15	N
.	median	100	100	1	1…1.01	**P**
.	iqr	0.38	0.76	0.5	0…176	P
P_{dpl}/N_{dpl}	mean	98.2	89.7	1.09	0.973…1.25	P
.	median	100	100	1	1…1.01	**P**
.	iqr	0.38	0.76	0.5	0…∞	P
P_{eff}/N_{eff}	mean	84.9	99.8	0.851	0.732…0.969	**N**
.	median	100	100	1	0.996…1	—
.	iqr	1.33	0.286	4.66	0…∞	N
P_{sz}/N_{sz}	mean	85	99.8	0.851	0.733…0.97	**N**
.	median	100	100	1	1…1	—
.	iqr	0.953	0	∞	0.655…∞	N
VARIABLE $rel_{out,m3}$ (reliability for difficult inputs):						
P_{raw}/N_{raw}	mean	83.2	47.8	1.74	1.3…2.49	**P**
.	median	98.4	10.2	9.69	1.04…9.76	**P**
.	iqr	4.78	88.3	0.0541	0.018…1.02	P
P_{rel}/N_{rel}	mean	87.4	45.6	1.92	1.38…3.14	**P**
.	median	98.4	10.2	9.69	1.02…9.84	**P**
.	iqr	1.58	88.1	0.0179	0.012…∞	P
P_t/N_t	mean	79.8	52.6	1.52	1.07…2.33	**P**
.	median	98.4	50.5	1.95	1…9.84	**P**

continued…

groups	stat-istic	P	N	ratio P/N	80% confidence interval	bet-ter
.	iqr	9.2	88.3	0.104	0.0179…1.36	P
P_{dpl}/N_{dpl}	mean	92.9	52.6	1.76	1.29…2.7	**P**
.	median	100	50.5	1.98	1.02…9.84	**P**
.	iqr	1.58	88.3	0.0179	0.0166…0.231	**P**
P_{eff}/N_{eff}	mean	85.7	62.5	1.37	1.03…1.94	**P**
.	median	98.4	94.4	1.04	1…9.69	**P**
.	iqr	4.73	88.3	0.0536	0.0179…1.23	P
P_{sz}/N_{sz}	mean	85.2	53.7	1.59	1.17…2.41	**P**
.	median	98.4	50.5	1.95	1…9.69	**P**
.	iqr	5.02	88.3	0.0569	0.0157…0.98	**P**
VARIABLE $rel_{out,accept1}$ (release maturity):						
P_{raw}/N_{raw}	mean	51.7	59.4	0.87	0.628…1.23	N
.	median	58.8	78.2	0.752	0.109…1.92	N
.	iqr	100	95.9	1.04	0.908…2.22	N
P_{rel}/N_{rel}	mean	56.6	70.2	0.806	0.541…1.15	N
.	median	93.4	90.7	1.03	0.0337…1.33	P
.	iqr	99.1	47	2.11	0.92…5.35	N
P_t/N_t	mean	69.3	82.8	0.837	0.638…1.1	N
.	median	93.4	100	0.934	0.841…1.03	N
.	iqr	65.3	14	4.66	0.365…∞	N
P_{dpl}/N_{dpl}	mean	73.8	80.3	0.919	0.717…1.16	N
.	median	100	100	1	0.922…1.23	—
.	iqr	39	23.4	1.67	0.186…∞	N
P_{eff}/N_{eff}	mean	66	51.8	1.27	0.88…2.05	P
.	median	92.2	58.4	1.58	0.841…65.2	P
.	iqr	73.3	99.6	0.736	0.246…1.69	P
P_{sz}/N_{sz}	mean	50	76.9	0.651	0.457…0.91	**N**
.	median	67.6	100	0.676	0.0337…1.03	N
.	iqr	96.7	45.4	2.13	1.06…∞	**N**
VARIABLE $LOC_{comment}/LOC_{stmt}$ (comment density):						
P_{raw}/N_{raw}	mean	19.2	19.6	0.984	0.744…1.33	N
.	median	15.5	18	0.861	0.64…1.44	N
.	iqr	14.2	20.8	0.687	0.351…1.16	P
P_{rel}/N_{rel}	mean	21.2	22.6	0.938	0.674…1.45	N
.	median	16	21	0.762	0.536…1.68	N
.	iqr	14	23.8	0.589	0.256…1.23	P
P_t/N_t	mean	17.7	11.3	1.56	1.11…2.45	**P**
.	median	16	9.5	1.68	0.941…4	P
.	iqr	14	16.2	0.862	0.412…1.94	P
P_{dpl}/N_{dpl}	mean	19	12.6	1.51	1.07…2.29	**P**
.	median	17	12.5	1.36	0.75…4.25	P
.	iqr	17	18	0.944	0.515…1.85	P
P_{eff}/N_{eff}	mean	22.9	21.3	1.07	0.754…1.64	P
.	median	22	18	1.22	0.65…2.14	P
.	iqr	18	21.2	0.847	0.405…1.68	P
P_{sz}/N_{sz}	mean	15.4	17.1	0.901	0.63…1.44	N
.	median	15	12.5	1.2	0.591…2.17	P

continued…

groups	statistic	P	N	ratio P/N	80% confidence interval	better
.	iqr	8.5	19.5	0.436	0.228...0.955	**P**
			VARIABLE $1/t_{work}$ (productivity):			
P_{raw}/N_{raw}	mean	0.0984	0.14	0.704	0.546...0.929	**N**
.	median	0.0867	0.134	0.646	0.48...1.01	N
.	iqr	0.0683	0.139	0.493	0.272...0.865	**P**
P_{rel}/N_{rel}	mean	0.0991	0.142	0.698	0.518...0.966	**N**
.	median	0.0893	0.148	0.602	0.408...1.38	N
.	iqr	0.0547	0.148	0.369	0.206...0.743	**P**
P_t/N_t	mean	0.127	0.198	0.641	0.524...0.787	**N**
.	median	0.103	0.181	0.571	0.463...0.849	**N**
.	iqr	0.0465	0.113	0.412	0.252...1.37	P
P_{dpl}/N_{dpl}	mean	0.119	0.179	0.665	0.511...0.873	**N**
.	median	0.102	0.151	0.675	0.441...0.849	**N**
.	iqr	0.0503	0.114	0.443	0.263...1.92	P
P_{eff}/N_{eff}	mean	0.111	0.134	0.834	0.575...1.27	N
.	median	0.0917	0.0959	0.956	0.459...2.22	N
.	iqr	0.059	0.174	0.34	0.173...0.949	**P**
P_{sz}/N_{sz}	mean	0.114	0.182	0.625	0.471...0.831	**N**
.	median	0.099	0.181	0.548	0.403...0.823	**N**
.	iqr	0.0665	0.129	0.514	0.279...1.12	P
			VARIABLE LOC_{stmt} (program length):			
P_{raw}/N_{raw}	mean	290	246	1.18	1...1.4	N
.	median	290	215	1.35	1.04...1.69	**N**
.	iqr	119	118	1.01	0.495...1.94	—
P_{rel}/N_{rel}	mean	288	235	1.23	0.985...1.56	N
.	median	302	199	1.52	1.12...1.86	**N**
.	iqr	106	74.5	1.43	0.359...2.63	N
P_t/N_t	mean	255	204	1.25	1.06...1.46	**N**
.	median	237	176	1.35	1.1...1.48	**N**
.	iqr	106	52	2.04	0.537...4.33	N
P_{dpl}/N_{dpl}	mean	276	236	1.17	0.937...1.45	N
.	median	244	189	1.29	1.04...1.82	**N**
.	iqr	133	76.8	1.73	0.465...3.19	N
P_{eff}/N_{eff}	mean	277	267	1.04	0.849...1.29	N
.	median	243	222	1.09	0.788...1.46	N
.	iqr	106	166	0.638	0.298...2.14	P
P_{sz}/N_{sz}	mean	235	175	1.34	1.21...1.49	**N**
.	median	228	166	1.37	1.15...1.49	**N**
.	iqr	55.5	52	1.07	0.601...3.59	N

Chapter 4

Conclusion

It is a capital mistake to theorize before one has data.
Sherlock Holmes

4.1 Summary of results

Put very shortly, our experiment for comparing PSP-trained and non-PSP-trained subjects has found the following significant differences between these groups:

1. The estimations of total required working time are much better in the PSP group; the same holds for related measures such as estimated productivity measured in lines of code per hour. The PSP group is also better when estimating the average time for finding and fixing a defect and when judging the reliability of the program they had produced.

2. The productivity tends to be lower in the PSP group. However, it is also more constant (less variable) than in the non-PSP group.

3. The program reliability for unexpected (but legal) inputs is better in the PSP group, in particular when we compare the more disciplined halfs of each group. For more standard inputs, the differences are not significant, but there is a slight tendency that the non-PSP group is better.

4. The PSP group exhibits more care towards avoiding simple mistakes (such as compiler errors) and we find for the non-PSP group that the frequency of such simple mistakes correlates with decreasing program reliability in the first acceptance test, i.e., with a tendency to deliver unreliable first program versions.

5. For many of the observed variables we find that the variability in the PSP group is smaller than in the non-PSP group. Hence a team of PSP programmers would be easier to manage because its performance is more predictable.

4.2 Possible reasons

We should mention that we believe the PSP methodology is useful and the PSP course is a valid method of teaching its ideas as well as an initial set of basic *psp* techniques. We believed so before this experiment and we still do. From this perspective, the results of the experiment are disappointing: In some areas we found no advantage for the PSP group and in others the difference appears to be smaller than expected. Why did we not find further and more pronounced differences? We will discuss several possible explanations.

PSP is useless: The first and foremost possibility is obviously that the PSP might be not very useful. This would leave the results as they stand and not call for further interpretation. When taking this perspective one would conclude that our results generalize to most programming situations and that learning and using the PSP methodology is a waste of time. However, this point of view makes it almost impossible to explain the very positive results from within PSP courses (our own, those of Humphrey and the SEI, or those of many other successful instructors). We thus reject this explanation.

PSP is useless for this task: The PSP methodology might be of little help for the specific task to be solved in this experiment and could still be highly useful in other situations. This explanation is sort of an excuse. What could be so special about our task? Admittedly, the experimental task was quite different from the tasks in the PSP course. However, a similar statement is true for the control group as well. Most importantly, the PSP claims to be general, so even severe peculiarities in the task should not invalidate the PSP advantages. Hence, we reject this explanation as well.

Bad teaching: When Watts Humphrey first heard about these results and that we did not find dramatic differences, he immediately said "I suggest that you look at your teaching methods." However, we have no reason to believe that our PSP teaching is unsuccessful. First, the subjective experience of the students is so positive that both of the PSP courses ranked first among all course evaluations in our department for the respective semester. Second, the objective results of our students during the course are quite similar to those published by Humphrey for his own courses [4]. For instance, the average test defect density decreases from about 20 to 40 test defects found per KLOC in the first three programming exercises to under 10 in the tenth exercise. We believe that our teaching is definitely OK.

Incorrect PSP use: The second sentence in Humphrey's above-mentioned statement was "The students may, in fact, not being properly using the PSP". Such improper use could happen in two ways. Either they could use the PSP not at all (at least in parts) or they could use it in an incorrect way. We consider the second explanation unlikely for our students.

Lack of PSP use: This leaves the one explanation which we think is correct. Most of our PSP participants used little or none of the PSP techniques in their actual experiment work and probably also had no well-developed PSP mindset. This impression is corroborated by statements that some of our participants made after the experiment, for instance

> *"I should really have performed a code review, but I didn't."*
> *"I started to code too early although I had learned in the PSP course that this is a bad idea."*
> *"I have not used any of the PSP techniques and that made me suffer."*

We cannot quantify the adherence to PSP techniques well. We did not ask for such information in the postmortem questionnaire, because we felt that the resulting information would be unreliable. However, the small fraction of PSP participants who created time/defect logs (10 out of 29, 34%) suggests that the adherence was generally quite low.

In this context, we made an interesting observation: Out of the 5 PSP participants who gave up on the experiment, as many as 4 (or 80%) had created a time/defect log! The fraction is so much higher than average that the difference is probably not accidental (Fisher exact $p = 0.13$). This suggests that the least capable programmers have a much higher tendency to cling on to PSP techniques, which is corroborated by the average total working times of the successful PSP participants: Those with no time/defect logs had an average of 12 hours, those with logs had an average of 17 hours; the difference is significant (mean bootstrap $p = 0.02$). Presumably, adapting PSP techniques helps the more, the less capable a programmer is. If this is the case, it explains why the variability is consistently lower in the PSP group for almost all variables compared to the non-PSP group.

4.3 Consequences

We conclude that the PSP is indeed a helpful methodology and the cost of learning it is probably worthwhile. However, one must not overestimate the influence of the PSP on the average performance of the software engineers. Our results show that the improvements observed over time within each single PSP course must not be generalized, because that is too optimistic as a prediction of live software practice. On the other hand, we found a very important benefit from PSP use that has been completely neglected so far: The variability across individual software engineers is reduced by PSP training. Small variability is highly desirable in practical software organizations, because projects profit most from improvements of their worst parts.

We recommend adapting the PSP and consider it to be a risk reduction technique as much as it is a productivity or quality improvement technique.

Acknowledgements

Thanks to Oliver Gramberg for his contributions to the PSP course, to Holger Hopp, Thomas Schoch, Christian Nester, and Georg Grütter for guinea-pigging the experimental setup, and to Watts Humphrey for inventing the PSP in the first place. Thanks most of all to our participants (most of which, although not volunteers, liked the experiment and gave their best).

Appendix A

Experiment materials

I found in taking the PSP course that the intuition and experience
of someone else I respected a lot were sometimes wrong;
the "someone else" was me.
I now believe that data on actual practice
is a far better guide for what processes work than anyone's intuition.
Jim Alstad

This appendix contains the original materials given to the subjects (except for somewhat different page breaks). The materials were handed out in parts:

1. a questionnaire about personal background information
2. the description of the task and the work time logging sheet
3. a questionnaire which asked for an estimation of expected working time for the task and some other attributes,
4. a postmortem questionnaire (handed out after delivery of the program) for grading quality attributes of one's own program and suggesting improvement areas for one's own process.

There are two versions of the materials in this report: The original German version as given to the subjects (Sections A.6 to A.10) and an English translation (Sections A.1 to A.5). Note that in the translation of the task description the relative weight of the parts of a sentence or paragraph could not always be perfectly maintained. The page breaks differ from the original materials in both the German and the English version.

Dear participant,

thank you very much that you agreed to participate in our experiment. The purpose of this experiment is comparing different methods for producing defect-free programs. Thus, if you have had training in PSP or Cleanroom, remember what these methods suggested for the production of defect-free software.

It is *not* the purpose of the experiment to evaluate the performance of the individual participants. Therefore, your name will not be recorded on the experiment materials; we record only a subject number and a group designation.

A.1 Experiment procedure

The experiment will be conducted as follows: Each participant receives a programming task that is to be solved and handed in as error-free as possible. Of course it is nice if you are also fast to complete the task, but that is of less importance.

The experiment sequence is as follows:

1. fill in the general questionnaire
2. read the task description,
3. fill in the task questionnaire,
4. perform the task and pass the acceptance test
5. fill in the postmortem questionnaire

Prepare yourself for at least half a day, more probably a whole day, for the experiment!

If you solve the task correctly you will receive a copy-card valued DM 50. We will test your solution by a well-defined and objective acceptance test, in which you have to achieve a correctness of at least 95 percent. This acceptance test simulates the acceptance testing of a real customer — if your program fails in the acceptance test, a contractual penalty must be paid. This penalty is DM 10 in our case. We will execute the acceptance test whenever you ask for it; if you fail, the value on your copy-card will drop to DM 40, then DM 30 etc.

Make sure your program is correct. The programming task is not very difficult; you can surely manage to write a correct program if you try hard enough. To pass the acceptance test upon first try is within your capabilities, so accept the challenge and take away the full DM 50.

Please read the instructions for the questionnaires and the task carefully. When questions arise, always ask them immediately.

At some points during the experiment you need to record the time of day. Please be exact and do not forget it. If you find you forgot anyway, estimate the time retrospectively and mark your entry as an estimation by prepending “ca.”.

A.2 Questionnaire – personal information

Participant: ________________________
insert number

Group: ________________________
will be added later

In order to maintain anonymity in the experiment data, you will be identified by a number only. Please write this number in the top left corner of each questionnaire *and also in your program (e.g. in a header).*

1. Your semester number? ________ semester

2. How many years of programming experience do you have? ________ year(s)

3. How much programming experience do you have in KLOC? ________ KLOC

4. In how many different programming languages have you written *more* than one program? ________ languages

5. How long (in work hours) have you programmed overall outside of your university education? ________ hours

6. How long (in KLOC) is the biggest program (or your part of a program) you have written so far? ________ KLOC

7. In which programming language will you solve the task? ____________________________
programming language

8. How much programming experience do you have in the language chosen in question 7? ________KLOC

9. Have you programmed a lot recently?

☐ Yes, I took a lab course with programming exercises.
☐ Yes, for other reasons.
☐ Well, it wasn't so much
☐ No

10. Do you know how to use Emacs?

☐ yes
☐ a bit
☐ no

11. How do you define "Lines of Code" (LOC)?

☐ Each line counts as a LOC, even if it is empty or contains comment only.
☐ All code lines and all comment lines count as LOC, empty lines do not.

- [ ] All code lines count as LOC, empty lines and comment lines do not.
- [ ] My own definition:

____

____

Now carefully read the task description. It is important that you understand it thoroughly. Then fill in the field for the time of day at the bottom of page 7.

A.3 Task description

Attention: Please follow these instructions *super accurately*, or else both you and we will become frustrated... First read through the whole of it in order to get an overview. Concentrate on the details only upon second reading.

The following mapping from letters to digits is given:

E	J N Q	R W X	D S Y	F T	A M	C I V	B K U	L O P	G H Z
e	j n q	r w x	d s y	f t	a m	c i v	b k u	l o p	g h z
0	1	2	3	4	5	6	7	8	9

We want to use this mapping for encoding telephone numbers by words, so that it becomes easier to remember the numbers.

Functional requirements

Your task is writing a program that finds, for a given phone number, all possible encodings by words, and prints them. A phone number is an *arbitrary(!)* string of dashes `-`, slashes `/` and digits. The dashes and slashes will not be encoded. The words are taken from a dictionary which is given as an alphabetically sorted ASCII file (one word per line).

Only exactly each encoding that is possible from this dictionary and that matches the phone number exactly shall be printed. Thus, possibly nothing is printed at all. The words in the dictionary contain letters (capital or small, but the difference is ignored in the sorting), dashes `-` and double quotes `"`. For the encoding only the letters are used, but the words must be printed in exactly the form given in the dictionary. Leading non-letters do not occur in the dictionary.

Encodings of phone numbers can consist of a single word or of multiple words separated by spaces. The encodings are built word by word from left to right. If and only if at a particular point no word at all from the dictionary can be inserted, a single digit from the phone number can be copied to the encoding instead. Two subsequent digits are never allowed, though. To put it differently: In a partial encoding that currently covers k digits, digit $k+1$ is encoded by itself if and only if, first, digit k was not encoded by a digit and, second, there is no word in the dictionary that can be used in the encoding starting at digit $k+1$.

Your program must work on a series of phone numbers; for each encoding that it finds, it must print the phone number followed by a colon, a single(!) space, and the encoding on one line; trailing spaces are not allowed.

All remaining ambiguities in this specification will be resolved by the following **example**. (Still remaining ambiguities are intended degrees of freedom.)

Dictionary (in file `test.w`):

```
an
blau
Bo"
Boot
bo"s
da
Fee
fern
```

```
Fest
fort
je
jemand
mir
Mix
Mixer
Name
neu
o"d
Ort
so
Tor
Torf
Wasser
```

Phone number list (in file `test.t`):

```
112
5624-82
4824
0721/608-4067
10/783--5
1078-913-5
381482
04824
```

Program start command (*must* have this form!):

```
telefonkode test.w test.t
```

Corresponding correct program output (on screen):

```
5624-82: mir Tor
5624-82: Mix Tor
4824: Torf
4824: fort
4824: Tor 4
10/783--5: neu o"d 5
10/783--5: je bo"s 5
10/783--5: je Bo" da
381482: so 1 Tor
04824: 0 Torf
04824: 0 fort
04824: 0 Tor 4
```

Any other output would be wrong (except for different ordering of the lines).

Wrong outputs for the above example would be e.g.
`562482: Mix Tor`, because the formatting of the phone number is incorrect,
`10/783--5: je bos 5`, because the formatting of the second word is incorrect,
`4824: 4 Ort`, because in place of the first digit the words `Torf, fort, Tor` could be used,
`1078-913-5: je Bo" 9 1 da` , since there are two subsequent digits in the encoding,

`04824: 0 Tor` , because the encoding does not cover the whole phone number, and
`5624-82: mir Torf` , because the encoding is longer than the phone number.

The above data are available to you in the files `test.w` (dictionary), `test.t` (telephone numbers) and `test.out` (program output).

Quantitative requirements

Length of the individual words in the dictionary: 50 characters maximum.
Number of words in the dictionary: 75000 maximum
Length of the phone numbers: 50 characters maximum.
Number of entries in the phone number file: unlimited.

Quality requirements

Work as carefully as you would as a professional software engineer and deliver a correspondingly high grade program. Specifically, thoroughly comment your source code (design ideas etc.).

The focus during program construction shall be on correctness. Generate *exactly* the right output format right from the start. Do not generate additional output. We will automatically test your program with hundreds of thousands of phone numbers and it should not make a single mistake, if possible — in particular it must not crash. Take youself as much time as is required to ensure correctness.

Your program must be run time efficient in so far that it analyzes only a very small fraction of all dictionary entries in each word appending step. It should also be memory efficient in that it does not use 75000 times 50 bytes for storing the dictionary if that contains many much shorter words. The dictionary must be read into main memory entirely, but you must not do the same with the phone number file, as that may be arbitrarily large.

Your program need *not* be robust against incorrect formats of the dictionary file or the phone number file.

Other constraints

Please use only a single source program file, not several source modules, and give that file the name telefonkode.c (or whatever suffix is used in your programming language). Otherwise the evaluation will become too difficult for us. If you need to quickly try something out you may of course use a separate file for that purpose.

A large dictionary, which you can use for testing purposes, is available as the file `woerter2`.

For compiling your program please use the command `make telefonkode`, a suitable Makefile is present in your account. Java users can use this command only in the window entitled “for make and run”.

What’s next?

You have now read the task description. If you have any questions about it, ask them right now. The same is true for all other questions: ask them immediately. When you want to perform an acceptance test, just tell us. If the task is clear now, fill in the time of day and read the following pages.

Time of day:___________________________

From now on you should log any interruptions of your work. Note start and end time of each interruption. The reason is not mandatory, but may be useful for reconstructing missing times. Interruptions are, for instance, questions or going to the restroom.

Start of interruption	**End of interruption**	**Duration of interruption**	**Reason**

Participant: ______________
insert number

A.4 Questionnaire – Estimation

Group: ______________
will be added later

Imagine that your company is asked to implement the "phone number memorization utility". Your boss has to prepare a cost estimate and asks you, how much work that will be (since you are also the programmer planned to implement the program). He asks you to estimate the effort in terms of LOC as wel as time units. Furthermore, you shall estimate the number of defects as a characterization of complexity and difficulty. In addition, provide a range in which you expect the actual values to lie with a probability of 90 percent.

1. Estimate how long you will need for solving the task (starting with the time stamp you just wrote and ending with the complete, working, well-tested program).

   ____
   time in minutes

   ____
   time range [min–max]

2. Estimate your productivity measured in LOC/hour

   ____
   LOC/hour

   ____
   productivity range [min–max]

3. How many defects will you find during compilation and how many defects will you find during test?

   a) during compilation

   ____
   number of defects

   b) during test

   ____
   number of defects

   How long will designing, coding, compiling, and testing your program take? These intervals are defined as follows: Coding time is the time required for transforming a design into code; compiling time is the time from the first start of the compiler until the first successful compilation; and testing time is the time required for thoroughly testing the compiled program. Often program development is not linear. If, for example, you code only partially, then compile, then code further, then compile, then test etc. you have to add the times for the individual coding, compilation, and testing phases.

4. How long will you take for design and coding?

   ____
   in minutes

5. How long will you take for compiling your program?

   ____
   in minutes

6. How long will you take for testing your program?

   ____
   in minutes

time of day:__________________________

Now that you have answered the questions, please throw this questionnaire in the box. Then start programming.

Now please start programming!

A.5 Questionnaire – Self-evaluation

Participant: ____________________
insert number

Group: ____________________
will be added later

Now that your implementation is finished completely. Please write down the time of day here.

Time of day:________________________

Evaluate your finished program according to the following criteria:

Scale: (1) very good (2) good (3) OK
(4) weak (5) bad

The documentation (should be informative and precise but not verbose)

Program usage: How well can a new user handle the program; will s/he know how to call it? __________

Program purpose: Can somebody who does not know the program find out from the documentation what the program does? __________

Design ideas: How well can design decisions be understood? __________

Purpose of procedures: How well documented is the purpose of individual procedures? __________

Purpose of data types: How well documented is the purpose of the program's data types? __________

Purpose of variables: How easily can the purpose of variables be recognized? __________

Code in procedures: How well documented are difficult code sequences in procedures? __________

The complexity (should be as low as possible)

Data structures: How complex are the data structures that are used? __________

Control flow: How easily can control flow be followed? __________

Algorithms: How understandable/complex are the algorithms that are used? __________

Evaluate overall quality

Understandability: How readable is the whole program, e.g. with respect to changes to be made by other programmers? ___________

Extensibility: How well-suited is the program for later changes and extensions? ___________

Correctness: How well does the program satisfy its requirements? ___________

How many defects do you expect have remained in the program despite the tests? ________________

How reliable do you expect your program to be? 1 failure for every ________________ test cases.

Where do you expect the *largest* potential for improvement if you want to improve the speed or quality of your program development:

☐ defect reduction

- ☐ reducing design defects
- ☐ reducing implementation defects
- ☐ reducing defects during translation of design into code
- ☐ ____

____

☐ How could you reduce your defects?

- ☐ Perform design reviews
- ☐ Perform code reviews
- ☐ use formal methods
- ☐ use different programming language ________________________________ which, or what needs to be changed in the current one
- ☐ use tools ________________________________ which?
- ☐ ____

____

☐ achieve higher productivity by

- ☐ better programming languages ________________________________ which, or what needs to be changed in the current one
- ☐ more design (more precise designs, more detailed designs)
- ☐ leave out unimportant process steps, e.g. defect protocols or overly precise designs
- ☐ more tool use, e.g. ________________________________
- ☐ more programming practice
- ☐ a better programming education

That’s all. Thanks for participating!

One more request: Please do not talk about the experiment to other persons, unless these have already participated themselves. In particular, don’t describe the task. If somebody knows the task, s/he will unavoidably start thinking about possible solutions, which might massively distort the results if that person later participates in the experiment. In that case, several months of preparation work we did would be wasted.

Liebe Versuchsteilnehmerin, lieber Versuchsteilnehmer,

vielen Dank, daß Sie sich bereiterklärt haben, an unserem Experiment teilzunehmen. Dieses Experiment soll dazu dienen, verschiedene Methoden zur Herstellung fehlerfreier Programme zu vergleichen. Wenn Sie also einmal PSP oder Cleanroom gelernt haben, erinnern Sie sich daran, was diese Methoden für die Herstellung fehlerfreier Software vorschlagen.

Das Experiment dient *nicht* dazu, die einzelnen Versuchspersonen in ihren Leistungen zu bewerten. Deshalb werden auf den Unterlagen keine Namen festgehalten, sondern nur Nummern und eine Gruppenbezeichnung.

A.6 Versuchsablauf

Das Experiment wird folgendermaßen ablaufen: Jede Versuchsperson bekommt eine Programmieraufgabe gestellt, die möglichst fehlerfrei gelöst und abgegeben werden soll. Natürlich ist es schön, wenn das auch noch schnell geht, aber die Zeit spielt eine untergeordnete Rolle. Der Ablauf des Versuches ist wie folgt:

1. den allgemeinen Fragebogen ausfüllen,
2. die Aufgabenbeschreibung lesen,
3. den Fragebogen zur Aufgabe ausfüllen,
4. die Aufgabe lösen und den Akzeptanztest bestehen,
5. den Schlußfragebogen ausfüllen.

Stellen Sie sich darauf ein, daß das Experiment mindestens einen halben, eher einen ganzen Tag in Anspruch nimmt!

Wenn Sie die Aufgabe korrekt lösen, erhalten Sie als Ausgleich eine Copy-Card im Wert von bis zu DM 50. Wir prüfen Ihre Lösung durch einen wohldefinierten und objektiven Akzeptanztest, bei dem Sie eine Korrektheit von 95 Prozent erreichen müssen. Dieser Akzeptanztest simuliert den Abnahmetest durch einen echten Softwarekunden — wird der Abnahmetest nicht bestanden, wird eine Vertragsstrafe fällig! Diese beträgt bei uns DM 10. Wir führen den Akzeptanztest immer dann durch, wenn Sie danach fragen; beim Durchfallen sinkt also der Wert ihrer Copy-Card erst auf DM 40, dann auf DM 30 und so weiter.

Achten Sie also auf die Korrektheit Ihres Programms. Die Aufgabe ist nicht übermäßig schwierig; Sie können es mit Sicherheit schaffen, ein korrektes Programm zu schreiben, wenn Sie sich genug Mühe geben. Den Akzeptanztest auf Anhieb zu Bestehen, liegt durchaus im Rahmen ihrer Möglichkeiten, also nehmen Sie die Herausforderung an und kassieren Sie die DM 50 komplett!

Bitte lesen Sie die Anweisungen zu den Fragebögen und der Aufgabe genau durch. Falls Fragen auftauchen, klären Sie sie immer sofort.

An einigen Stellen im Experiment müssen Sie die Uhrzeit protokollieren. Bitte seien Sie möglichst exakt und vergessen Sie es nicht. Falls es doch einmal dazu kommen sollte, schätzen Sie die Uhrzeit nachträglich und markieren sie durch vorangestelltes „ca.“ als geschätzt.

Versuchsperson: ____________________
Hier Nummer eintragen

A.7 Fragebogen – persönliche Angaben

Gruppe: ____________________
Wird später eingetragen

Damit die Versuchsdaten anonym bleiben, bekommen Sie nur eine Nummer. Bitte scheiben Sie diese Nummer oben rechts auf jeden Fragebogen *und auch in Ihr Programm (z. B. in den Header).*

1. In welchem Semester sind Sie? ________ Semester

2. Wieviele Jahre Programmiererfahrung haben Sie? ________ Jahr(e)

3. Wieviel Programmiererfahrung haben Sie in KLOC? ________ KLOC

4. In wie vielen verschiedenen Programmiersprachen haben Sie schon *mehr* als ein Programm geschrieben? ________ Programmiersprachen

5. Wie lange (in Arbeitsstunden) haben Sie insgesamt außerhalb Ihrer unmittelbaren Studienausbildung programmiert? ________ Stunden

6. Wie lang (in KLOC) ist das größte Programm (bzw. Ihr Anteil an einem Programm), das Sie bisher geschrieben haben? ________ KLOC

7. In welcher Programmiersprache werden Sie die Aufgabe lösen? ____________________
Programmiersprache

8. Wieviel Programmiererfahrung haben Sie in der (in Frage 7) gewählten Sprache? ________KLOC

9. Haben Sie in der letzten Zeit viel programmiert?

- [ ] Ja, ich habe ein Praktikum mit Programmierübungen gemacht
- [ ] Ja, ich habe aus anderen Gründen viel programmiert
- [ ] Naja, soviel war's nicht
- [ ] Nein

10. Kennen Sie sich mit dem Emacs aus?

- [ ] ja
- [ ] wenig
- [ ] nein

11. Wie definieren Sie das Maß „Lines of Code" (LOC)?

- [ ] Zu LOC zählt jede Zeile, auch wenn sie nur einen Kommentar enthält oder eine Leerzeile ist.
- [ ] Als LOC zählen alle Code-Zeilen und alle Kommentarzeilen, Leerzeilen jedoch nicht.
- [ ] Als LOC zählen nur Code-Zeilen, Leerzeilen und Kommentarzeilen jedoch nicht.

☐ Eigene Definition:

____

____

Lesen Sie sich bitte die Aufgabenstellung gut durch. Es ist wichtig, daß Sie diese verstanden haben. Danach füllen Sie unten auf Seite 7 das Feld für die Uhrzeit aus.

A.8 Aufgabenstellung

Achtung: Befolgen Sie diese Anleitung bitte *supergenau*, sonst gibt es nur Frust auf beiden Seiten. . .
Lesen Sie erst einmal alles komplett durch, um ein Gesamtbild der Aufgabe zu bekommen, und konzentrieren Sie sich erst beim anschließenden zweiten Lesen auf die Details.

Gegeben sei die folgende Abbildung der Buchstaben des Alphabets auf Ziffern:

E	J N Q	R W X	D S Y	F T	A M	C I V	B K U	L O P	G H Z
e	j n q	r w x	d s y	f t	a m	c i v	b k u	l o p	g h z
0	1	2	3	4	5	6	7	8	9

Wir wollen diese Abbildung benutzen, um Telefonnummern durch Wörter zu kodieren, damit wir sie uns leichter merken können.

Funktionale Anforderungen

Ihre Aufgabe ist es, ein Programm zu schreiben, das zu einer eingegebenen Telefonnummer alle möglichen Kodierungen durch Wörter findet und ausgibt. Eine Telefonnummer ist eine *beliebige(!)* Kette aus Querstrichen `-`, Schrägstrichen `/` und Ziffern. Die Quer- und Schrägstriche werden nicht kodiert. Die Wörter sind einem Wörterbuch zu entnehmen, das als alphabetisch sortierte ASCII-Datei vorliegt (ein Wort pro Zeile).

Nur genau jede laut diesem Wörterbuch mögliche Kodierung, die exakt auf die Telefonnummer paßt, soll ausgegeben werden. Gegebenenfalls wird also gar nichts ausgegeben. Die Wörter im Wörterbuch können Buchstaben (Groß- und Kleinbuchstaben, die aber gleich sortiert werden), Querstriche `-` und Anführungszeichen `"` enthalten. Für die Kodierung werden nur die Buchstaben beachtet, jedoch sollen Wörter genau in der Form ausgegeben werden, wie sie im Wörterbuch stehen. Führende Sonderzeichen kommen im Wörterbuch nicht vor.

Kodierungen von Telefonnummern können aus einem einzelnen Wort oder aus mehreren Wörtern hintereinander (durch Leerzeichen getrennt) bestehen. Die Kodierungen werden von vorne nach hinten wortweise aufgebaut. Wenn und nur wenn an einer Stelle überhaupt kein Wort aus dem Wörterbuch eingesetzt werden kann, darf stattdessen auch eine einzelne Ziffer aus der Telefonnummer direkt übernommen werden, niemals jedoch zwei Ziffern hintereinander. Mit anderen Worten: In einer teilfertigen Kodierung, die bisher k Ziffern abdeckt, steht für Ziffer $k + 1$ dann und nur dann die Ziffer selbst, wenn erstens für Ziffer k keine Ziffer steht und zweitens kein Wort im Wörterbuch steht, das sich ab Ziffer $k + 1$ beginnend einsetzen läßt.

Ihr Programm soll gleich eine ganze Liste von Telefonnummern abarbeiten und für jede gefundene Kodierung die Telefonnummer gefolgt von Doppelpunkt, einem(!) Leerzeichen und der Kodierung auf einer Zeile ausgeben; nachfolgende Leerzeichen sind nicht erlaubt.

Alle verbleibenden Mehrdeutigkeiten in der Spezifikation werden von folgendem **Beispiel** ausgeräumt (übrige Mehrdeutigkeiten sind gewollte Freiheitsgrade).

Wörterbuch (in der Datei `test.w`):

```
an
blau
Bo"
```

```
Boot
bo"s
da
Fee
fern
Fest
fort
je
jemand
mir
Mix
Mixer
Name
neu
o"d
Ort
so
Tor
Torf
Wasser
```

Telefonnummernliste (in der Datei `test.t`):

```
112
5624-82
4824
0721/608-4067
10/783--5
1078-913-5
381482
04824
```

Programmaufruf (muß unbedingt diese Form haben!):

```
telefonkode test.w test.t
```

Zugehörige korrekte Programmausgabe (auf den Bildschirm):

```
5624-82: mir Tor
5624-82: Mix Tor
4824: Torf
4824: fort
4824: Tor 4
10/783--5: neu o"d 5
10/783--5: je bo"s 5
10/783--5: je Bo" da
381482: so 1 Tor
04824: 0 Torf
04824: 0 fort
04824: 0 Tor 4
```

Jede abweichende Ausgabe wäre falsch (ausgenommen andere Reihenfolge der Zeilen).

Falsche Ausgaben im obigen Beispiel wären also z.B.
`562482: Mix Tor`, weil das Format der Telefonnummer verkehrt ist.
`10/783--5: je bos 5`, weil das Format des zweiten Worts verkehrt ist.
`4824: 4 Ort`, weil bei der ersten Ziffer die Wörter `Torf, fort, Tor` eingesetzt werden können,
`1078-913-5: je Bo" 9 1 da` , da in der Kodierung zwei Ziffern aufeinanderfolgen,
`04824: 0 Tor` , weil die Kodierung nicht die ganze Telefonnummer abdeckt und
`5624-82: mir Torf` , weil die Kodierung länger ist als die Telefonnummer.

Die obigen Daten stehen Ihnen in den Dateien `test.w` (Wörterbuch), `test.t` (Telefonnummern) und `test.out` (Programmausgaben) zur Verfügung.

Quantitative Anforderungen

Länge der einzelnen Wörter im Wörterbuch: maximal 50 Zeichen.
Anzahl Wörter im Wörterbuch: maximal 75000
Länge der Telefonnummern: maximal 50 Zeichen
Anzahl Einträge in der Telefonnummerndatei: beliebig.

Qualitätsanforderungen

Arbeiten Sie so sorgfältig, wie sie es auch als professioneller Softwareingenieur tun würden und liefern Sie ein entsprechend hochwertiges Programm ab. Kommentieren Sie insbesondere Ihren Quellkode gründlich (Entwurfsideen etc.).

Das Hauptaugenmerk bei der Erstellung des Programms soll auf der Korrektheit liegen. Erzeugen Sie von Anfang an *genau* das richtige Ausgabeformat. Machen Sie keinerlei zusätzliche Ausgaben. Wir werden Ihr Programm maschinell mit mehreren Hunderttausend Telefonnummern testen und es sollte wenn irgend möglich keinen einzigen Fehler machen — insbesondere keinesfalls abstürzen. Nehmen Sie sich soviel Zeit wie nötig, um die Korrektheit sicherzustellen.

Ihr Programm muß laufzeiteffizient sein insofern, daß bei jedem Wortanfügungsschritt nur ein sehr kleiner Teil der Wörterbucheinträge betrachtet wird. Es sollte auch speichereffizient sein, indem nicht 75000 mal 50 Bytes für das Wörterbuch benutzt werden, wenn viele viel kürzere Wörter darin vorkommen. Das Wörterbuch muß komplett in den Hauptspeicher eingelesen werden, die Telefonnummerndatei darf dies jedoch keinesfalls, da sie beliebig groß werden kann.

Ihr Programm braucht *nicht* robust zu sein gegen fehlerhafte Formate von Wörterbuch oder Telefonnummernliste.

Sonstige Randbedingungen

Bitte benutzen Sie die ganze Zeit nur eine Programmdatei, also nicht mehrere Module, und geben Sie ihr den Namen telefonkode.c (oder welchen Suffix auch immer ihre Programmiersprache braucht). Andernfalls wird die Auswertung für uns zu kompliziert. Wenn Sie zwischendurch kurz etwas ausprobieren müssen, können sie dafür natürlich auch eine andere Datei benutzen.

Ein großes Wörterbuch, das Sie zu Testzwecken verwenden können, steht Ihnen als `woerter2` zur Verfügung.

Zum Übersetzen Ihres Programms benutzen Sie bitte das Kommando `make telefonkode`, einen passenden Makefile haben wir für Sie vorbereitet. Java-Benutzer müssen beachten, daß dieses Kommando aus technischen Gründen ausschließlich in dem mit „Nur hier Java“ betitelten Fenster funktioniert.

Wie geht's weiter?

Sie haben nun die Aufgabenstellung gelesen. Wenn Sie dazu noch Fragen haben, stellen Sie sie bitte jetzt sofort. Dies gilt auch für alle anderen Fragen. Sofort alle Unklarheiten beseitigen. Wenn Sie irgendwann einen Akzeptanztest durchführen wollen, sagen Sie uns einfach Bescheid. Wenn die Aufgabenstellung klar ist, tragen Sie nun bitte die Uhrzeit ein und lesen noch die nächsten Seiten.

Uhrzeit:__________________________

Ab diesem Zeitpunkt sollen Sie jede Unterbrechung mitprotokollieren. Schreiben Sie immer bei Beginn und Ende einer Unterbrechung die Uhrzeit auf. Der Grund der Unterbrechung ist nicht unbedingt erforderlich, ist aber nützlich, um evtl. fehlende Uhrzeiten nachzuvollziehen. Als Unterbrechungen zählen z. B. Fragen, Gang auf die Toilette.

Beginn der Unterbrechung	**Ende der Unterbrechung**	**Dauer der Unterbrechung**	**Grund**

Versuchsperson: ____________
Hier Nummer eintragen

A.9 Fragebogen – Selbsteinschätzung

Gruppe: ____________
Wird später eingetragen

Stellen Sie sich vor, Ihre Firma bekommt den Auftrag „Programmieren einer Telefonnummer-Merkhilfe". Ihr Chef muß einen Kostenvoranschlag machen und will von Ihnen wissen, wieviel Arbeit das ist (da Sie auch der Programmierer sein werden, der das implementieren soll). Sie sollen den Aufwand sowohl in LOC als auch in Zeiteinheiten angeben. Darüberhinaus soll die Abschätzung der Fehler eine ungefähre Einschätzung der Komplexität und des Schwierigkeitsgrades ermöglichen. Geben Sie jeweils zusätzlich einen Bereich an, in dem das Ergebnis Ihrer Einschätzung nach mit einer Wahrscheinlichkeit von 90% liegen wird.

1. Was schätzen Sie:
   Wie lange werden Sie ungefähr für das Lösen der Aufgabe brauchen (gerechnet ab dem letzten Uhrzeit-Protokollieren bis zum fertigen, lauffähigen und ausgetesteten Programm)?

   ____
   Zeit in Minuten

   ____
   Zeitbereich [von–bis]

2. Wie hoch schätzen Sie ihre Produktivität in LOC/Stunde?

   ____
   LOC/Stunde

   ____
   Produktivitätsbereich [von–bis]

3. Wieviele Fehler werden Sie beim Übersetzen (Kompilieren) beseitigen, und wieviele Fehler werden Sie im Test finden?

   a) beim Übersetzen

   ____
   Fehleranzahl

   b) beim Testen

   ____
   Fehleranzahl

   Wie lange werden Sie jeweils zum Entwerfen und Kodieren, Kompilieren und Testen Ihres Programmes brauchen? Die einzelnen Zeiten sind wie folgt definiert: Kodierzeit ist die Zeit, die man benötigt, um einen Entwurf in Kode umzusetzen; das Übersetzen dauert von der ersten Benutzung des Compilers bis zum ersten fehlerfreien Übersetzen; und das Testen ist die Zeit, die benötigt wird, das übersetzte Programm auszutesten. Bitte beachten Sie: Da ein Programm meist nicht linear bearbeitet wird, etwa zuerst nur kodieren, übersetzen und dann testen, sondern z. B. teilweise kodieren, übersetzen, weiter kodieren, nochmals übersetzen und dann erst austesten. Falls das Ihre übliche Arbeitweise ist, müssen Sie die Werte der einzelnen Kodier-, Übersetzungs- und Testphasen jeweils aufaddieren.

4. Wie lange werden Sie zum Entwerfen und Kodieren brauchen?

   ____
   in Minuten

5. Wie lange werden Sie zum Übersetzen Ihres Programmes brauchen?

   ____
   in Minuten

6. Wie lange werden Sie zum Testen Ihres Programmes brauchen?

   ____
   in Minuten

aktuelle Uhrzeit:____________

Nachdem Sie die Fragen beantwortet haben, werfen Sie nun diesen Fragebogen in die Box. Bitte fangen Sie dann an zu programmieren.

Bitte programmieren

Sie jetzt die Aufgabe!

Versuchsperson: ______________________
Hier Nummer eintragen

A.10 Fragebogen – Eigenbeurteilung

Gruppe: ______________________
Wird später eingetragen

Bitte tragen Sie, nachdem sie die Implementierung vollständig abgeschlossen haben, hier die Uhrzeit ein.

Uhrzeit:______________________

Nachdem Sie nun Ihr Programm fertig haben, versuchen Sie es an folgenden Kriterien zu beurteilen:

Bewertungsskala: (1) sehr gut (2) gut (3) ausreichend
(4) schwach (5) schlecht

Im Bereich der Dokumentation (sollte informativ und genau, aber nicht ausschweifend sein)

Programmverwendung: Wie gut kann ein fremder Nutzer mit dem Programm umgehen; weiß er, wie er es aufrufen muß? __________

Programmzweck: Kann jemand, der das Programm nicht kennt, an der Dokumentation erkennen, was das Programm macht? __________

Entwurfsideen: Wie gut kann man Entwurfsentscheidungen nachvollziehen? __________

Zweck der Prozeduren: Wie gut ist der Zweck der einzelnen Prozeduren dokumentiert? __________

Zweck der Datentypen: Wie gut ist dokumentiert, wozu die einzelnen Datentypen dienen? __________

Zweck der Variablen: Wie gut erkennt man den Zweck von Variablen? __________

Kode in Prozeduren: Wie gut sind einzelne schwierige Kodestücke dokumentiert? __________

Im Bereich der Komplexität (sollte so gering wie möglich sein)

Datenstrukturen: Wie komplex sind die verwendeten Datenstrukturen? __________

Kontrollfluß: Wie gut ist der Kontrollfluß zu verfolgen? __________

Algorithmen: Wie verständlich/komplex sind die verwendeten Algorithmen? __________

Beurteilen Sie die Gesamtqualität

Verständlichkeit: Wie verständlich ist das Gesamtprogramm, z. B. wenn es von fremden Programmierern inspiziert werden muss? ___________

Erweiterbarkeit: Wie gut ist das Programm für Änderungen und Erweiterungen geeignet? ___________

Korrektheit: Wie gut entspricht das Programm den gestellten Anforderungen? ___________

Wieviele Fehler, vermuten Sie, sind trotz Test im Programm verblieben? ________________

Wie zuverlässig schätzen sie Ihr Programm ein? 1 Versagen pro ________________ Testfälle

Wo erwarten Sie Ihr *größtes* Verbesserungspotential, wenn Sie schneller und/oder bessere Software entwickeln möchten:

- ☐ Fehlerreduktion
  - ☐ Entwurfsfehler verringern
  - ☐ Implementierungsfehler verringern
  - ☐ Fehler bei der Umsetzung von Entwurf in Implementierung verringern
  - ☐ ____

- ☐ Wie könnten Sie die Fehler reduzieren?
  - ☐ Entwurfsdurchsichten machen
  - ☐ Kodedurchsichten machen
  - ☐ formale Methoden einsetzen
  - ☐ andere Programmiersprache einsetzten ________________ (welche, bzw. was muß an der aktuellen geändert werden)
  - ☐ Werkzeuge einsetzen ________________ (welche?)
  - ☐ ____

- ☐ höhere Produktivität anstreben durch
  - ☐ bessere Programmiersprachen ________________ (welche, bzw. was muß an der aktuellen geändert werden)
  - ☐ mehr Entwurf (genauere Entwürfe, ausführlichere Entwürfe)
  - ☐ unwichtige Zwischenschritte weglassen, z. B. Fehlerprotokolle oder zu genaue Entwürfe
  - ☐ mehr Werkzeuge einsetzen, z. B. ________________
  - ☐ mehr Übung im Programmieren
  - ☐ eine bessere Programmierausbildung

Das wars. Danke fürs Mitmachen!

Wir haben noch eine dringende Bitte an Sie: Sprechen Sie nicht mit anderen Personen über das Experiment, wenn diese nicht selbst schon an dem Experiment teilgenommen haben. Beschreiben Sie insbesondere niemandem die Art der Problemstellung der Programmieraufgabe. Wer sie kennt, fängt unweigerlich an, über mögliche Lösungen nachzudenken, was unsere Experimentergebnisse total verfälschen würde, wenn die betreffende Person später am Experiment teilnimmt. Dann hätten wir monatelange Vorbereitungen umsonst gemacht.

Appendix B

Glossary

This glossary contains only the most important terms. Please refer to the main text for the exact definitions or descriptions, as (and if) indicated in the short glossary description. The definitions of related terms are often found at nearby points in the text.

bootstrap resampling	a general statistical technique based on stochastic simulation (see Section 3.1.1 on page 20)
iqr	the inter-quartile range, a specific measure of variability of data (see Section 3.1.1 on page 20)
iqr bootstrap $p = 0.1$	a p-value computed by a bootstrap resampling of differences of interquartile ranges (see Section 3.1.1 on page 20)
LOC	lines of code. Either refers to a subject's own definition of the term (estimation) or to a common standard definition as indicated (measurement)
mean bootstrap $p = 0.1$	a p-value computed by a bootstrap resampling of differences of means (see Section 3.1.1 on page 20)
median	A value such that 50% of the values are smaller and 50% are larger (see Section 3.1.1 on page 20)
M	in boxplots: symbol that marks the arithmetic mean
N	control group of non-PSP-trained subjects (see Section 2.4.1 on page 11). Usually refers to N_{raw} as well as its subgroups.
N_{raw}	control group with unsuccessful participants removed (see Section 3.2 on page 25)
P	experiment group of PSP-trained subjects (see Section 2.4.1 on page 11). Usually refers to P_{raw} as well as its subgroups.
p-value	the probability (computed by a statistical test) that a certain observation is only due to chance (see Section 3.1.1 on page 20)
participant	A person who has participated in the experiment, also called *subject*
P_{raw}	experiment group with unsuccessful participants removed (see Section 3.2 on page 25)

$prod_{estim}$	productivity (in lines of code written per work hour) as estimated by the subject
PSP	The Personal Software Process methodology (see Section 1.1 on page 4)
psp	An individual personal software process (see Section 1.1 on page 4)
quantile	A value just larger than a certain fraction of all values (see Section 3.1.1 on page 20)
rank	the order number of a value when the sample is sorted (see Section 3.1.2 on page 23)
regression line	a trend line computed by a mathematical procedure (see Section 3.1.2 on page 23)
significant	A difference is called significant if its p-value is too small to be incidental (see Section 3.1.3 on page 25)
subject	A person who has participated in the experiment, also called *participant*
t_{estim}	total working time as estimated by the subject
t_{work}	actual total working time of subject
variability	or variation or variance or spread: how much the values in a sample differ from each other (see Section 3.1.1 on page 20)
Wilcoxon test $p = 0.1$	a p-value computed by a Wilcoxon rank sum test (see Section 3.1.1 on page 20)

Bibliography

[1] Larry B. Christensen. *Experimental Methodology*. Allyn and Bacon, Needham Heights, MA, 6th edition, 1994.

[2] Bradley Efron and Robert Tibshirani. *An introduction to the Bootstrap*. Monographs on statistics and applied probability 57. Chapman and Hall, New York, London, 1993.

[3] Watts Humphrey. *A Discipline for Software Engineering*. SEI Series in Software Engineering. Addison Wesley, Reading, MA, 1995.

[4] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[5] Julien L. Simon. *Resampling: The new statistics*. Duxbury Press, Belmont, CA, 1992. http://www.statistics.com.