This is a preprint of an article published in Lech Madeyski, "Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites", Software Process: Improvement and Practice, vol. 13, no. 3, pp. 281–295, 2008. http://dx.doi.org/10.1002/spip.382

Impact of Pair Programming on Thoroughness and Fault Detection Effectiveness of Unit Test Suites

Lech Madeyski

Institute of Applied Informatics, Wroclaw University of Technology,
Wyb.Wyspianskiego 27, 50370 Wroclaw, POLAND
Lech.Madeyski@pwr.wroc.pl, http://madeyski.e-informatyka.pl/

Abstract. Pair programming is regarded as one of the practices that can make testing more rigorous, thorough and effective. Therefore, we examined pair programming vs. solo programming with respect to both, thoroughness and fault detection effectiveness of test suites. Branch coverage and mutation score indicator were used as measures of how thoroughly tests exercise programs, and how effective they are, respectively. It turned out that the pair programming practice did not significantly affect branch coverage ($U = 471.5$, non-significant, $r = -.03$), and mutation score indicator ($U = 422.0$, non-significant, $r = -.12$). These results are consistent with the results of selective analysis in which projects with a limited number of assertions are excluded. Analysis of covariance was performed to get a more sensitive measure of our experiment effect as well as to reduce pre-existing differences among subjects. The obtained results do not support anecdotal opinion regarding the positive impact of pair programming on thoroughness or fault detection effectiveness of unit tests. The validity of the results must be considered within the context of the limitations of the study e.g. it is possible that the benefits of pair programming will appear in longer, more complex projects.

Key words: Pair Programming; Unit Testing; Tests Quality; Fault-Finding Effectiveness; Mutation Score; Code Coverage

1 Introduction

Pair programming (PP) [1] has recently gained a lot of attention e.g. as one of the flag-ship practices of eXtreme Programming (XP) methodology [2]. PP is a software development practice in which two distinct roles are identified, i.e. the role of a driver and a navigator. They contribute to the synergy of the individuals in a pair working together at one computer and collaborating on the same development tasks (e.g. design, test, code). The driver types at the keyboard and focuses on the details of the production code or tests. The navigator observes the work of the driver, reviews the code, proposes test cases, considers the strategic implications [3,4] and looks for tactical and strategic defects or alternatives [5]. In the case of solo programming, both activities are performed by a single programmer. Convincing all programmers and managers to accept a pair programming work culture may be a tough task. Managers

sometimes feel as if they just got two people to do the same task, and waste valuable "resources". Better code or tests quality might be a good counter-argument in such situations.

Test-driven development (TDD) [6], also known as test-first programming [2], is another key and the well known software development practice of XP, supposed to be used with PP. TDD constitutes a practice, which is based on specifying a piece of functionality as a test (usually a low-level unit test), then writing production code, and implementing the functionality, so that the test passes, and finally refactoring, and iterating the process. The tests are run frequently, while writing production code and drive the development process, hence the name. Kobayashi et al. [7] suggested that PP, TDD and refactoring (which is the inherent part of the TDD development cycle) had a very good synergy. Therefore, it seems reasonable to evaluate PP practice in the context of TDD.

PP is supposed to be a software development practice that can influence unit testing to make it more rigorous, thorough, and effective. The question is whether the impact of PP is significant or not. Therefore, the research objective is to evaluate the impact of PP practice on unit tests by means of an experiment. The definition, design, and operation of the experiment are described in Section 4.

2 Measures of Unit Tests

Programmers who write unit tests should have a set of guidelines indicating whether their software has been thoroughly and effectively tested. Accordingly, in this section we discuss measures that can serve as such guidelines.

2.1 Code Coverage

Measuring code coverage is one guideline which can be applied, since code coverage tools measure how thoroughly tests exercise programs [8]. However, it remains a controversial issue as to whether code coverage is a good indicator for fault detection capability of test cases [9]. Marick [8] points out that code coverage may be misused, but code coverage tools are still helpful – especially if they are used to enhance thought, and not to replace it.

Kaner [9] presents more than a hundred coverage measures. The crucial question concerns the proper choice of a code coverage measure to be used. Useful insights into this question are given by Cornett [11]. Statement coverage, also known as line coverage, reports whether each executable statement is encountered. The main disadvantage of statement coverage is that it is insensitive to some control structures. To avoid this problem, branch coverage, also known as decision coverage, has been devised. Branch coverage is a measure based on whether decision points, such as `if` and `while` statements, evaluate both true and false during test execution, thus exercising both execution paths. Branch coverage includes statement coverage since exercising every branch must lead to exercising every statement. However, a shortcoming of the branch coverage measure is that it ignores branches within boolean expressions which occur due to short-circuit operators (e.g. `||` and `&&` operators in Java). The second argument of these operators is only executed or evaluated if the first argument does

not suffice to determine the value of the expression. Thus, branch coverage measure could consider control structures completely exercised even without calls to all methods. Unfortunately, the most powerful measures such as Modified Condition/Decision Coverage, required for aviation software, or Condition/Decision Coverage are not available for Java software. Therefore, branch coverage measure was used in our analysis, as the best of the available code coverage measures.

Branch coverage is offered by several tools e.g. Clover, JCoverage, and Cobertura. It appeared that the branch coverage results obtained by JCoverage and Cobertura were in line with the results obtained by Clover, and therefore only Clover results were included in further analysis.

2.2 Mutation Score

A way to measure the effectiveness of test suites is a fault-based technique, called mutation testing, originally proposed by DeMillo et al. [12] and Hamlet [13]. Mutation analysis is a way to measure the quality of the test cases, so the actual testing of the software is a side effect [14]. The effectiveness of test suites for fault localization is estimated on seeded faults. The faults are introduced into the program by creating a collection of faulty versions, called mutants. These mutants are created from the original program by applying mutation operators which describe syntactic changes to the programming language. The tests are used to execute these mutants with the goal of causing each mutant to produce incorrect output.

Mutation score (or mutation adequacy), defined as a ratio of the number of killed mutants to the total number of non-equivalent mutants, is a kind of quantitative measurement of the tests quality [15]. The total number of non-equivalent mutants results from a difference between total number of mutants and the number of equivalent mutants. Equivalent mutants always produce the same output as the original program, so they cannot be killed. Unfortunately, determining which mutant programs are equivalent to the original program is a very tedious and error-prone activity, so even ignoring equivalent mutants is sometimes suggested [14]. Ignoring equivalent mutants means that we are ready to accept the lower bound on mutation score (named mutation score indicator). Accepting it results in the cost-effective application of a mutation analysis and still provides meaningful information about the fault detection effectiveness of test suites.

Empirical studies have supported the effectiveness of mutation testing. Walsh [16] found empirically that mutation testing is more powerful than statement and branch coverage. Frankl et al. [17] and Offutt et al. [18] found that mutation testing was more effective at finding faults than data-flow. Moreover, Fowler [19] found mutation the testing tool support useful in practice.

Although mutation testing is powerful, it is not meant as a replacement for code coverage, but only as a complementary approach useful in finding code that is executed by running tests, but not actually tested. Moreover, it is time-consuming, and impractical to use without a reliable, fast and automated tool that generates mutants, runs the mutants against a suite of tests, and reports the mutation score of the test suite. Unfortunately, to date, proposed mutation tools for Java have several limitations that prevent practitioners from using

them. Jester [20] was probably the first and most widely known tool dedicated to Java language. However, Jester is too slow to be used in large software projects. It modifies the source code of the software components and may break the code making operation risky. Offutt [21] even stated that Jester turned out to be "a jest" of mutation, rather than a mutation testing tool. MuJava [22] is the most interesting tool that offers a large set of mutation operators dedicated to the Java language. Unfortunately, MuJava has some limitations that may prevent practitioners from using this tool in everyday development. For example, it does not work with the JUnit, the most widely used unit testing framework. However, it is worth mentioning that the recently released MuClipse plugin [23] seems to be an interesting and promising solution to overcome this limitation. Unfortunately, MuClipse exhibits some limitations concerning automation of mutants generation. For example, it is not possible to adjust test classes naming convention of MuClipse to the naming convention used by the system under test, and to easily select whole project to test. Also there are limitations with the testing process (e.g. it is not possible to select more than one class simultaneously, it is necessary to select additional tests for each class explicitly). As a result, mutation testing by means of MuClipse is far from being easy to run in a production environment. Moreover, lack of integration with a build tool (Ant or Maven) is still an issue. Chevalley and Thévenod-Fosse [24] proposed another mutation tool that unfortunately does not support the execution of mutants and is not freely available for download. Therefore, a new mutation tool, called Judy, has been developed, using an aspect-oriented approach to speed up mutation testing [25]. Judy has a build-in support of JUnit and Ant.

The mutations set consists of 16 mutations: ABS (Absolute value insertion), AOR(Arithmetic operator replacement), LCR (Logical connector replacement), ROR (Relational operator replacement), UOI (Unary operator insertion), UOD(Unary operator deletion), SOR (Shift operator replacement), LOR (Logical operator replacement), COR (Conditional operator replacement), ASR (Assignment operator replacement), EOA (Reference and content assignment replacement), EOC (Reference and content comparison replacement), EAM (Accessor method change), EMM (Modifier method change), JTD (`this` keyword deletion), JTI (`this` keyword insertion). The first five operators (ABS, AOR, LCR, ROR, UOI) were taken from the research of Offutt et al.'s research [26] to identify a set of sufficient mutation operators. The idea of sufficient mutation operators is to minimize the number of mutation operators, whilst getting as much testing strength as possible. Ammann and Offut [27] have recently presented these five mutation operators along with UOD, SOR, LOR, COR, ASR as program-level mutation operators dedicated to Java language. Ma et al. [28] found that EOA and EOC mutation operators can model object-oriented (OO) faults that are difficult to detect and therefore, can be considered to be good mutation operators for OO programs. Finally, EAM, EMM, JTD and JTI mutation operators were added, as there was still no determined set of selective mutation operators for class mutation operators. Thus, there is no reason to exclude these operators [28].

Branch coverage and mutation score indicator were used as measures to determine thoroughness and fault detection effectiveness of test suites.

3 Related Work

Researchers and practitioners have reported numerous, sometimes anecdotal, and favourable studies of PP. Beck and Andres wrote that a pair is even more than twice as effective as the same two people programming solo [2]. However, empirical studies concerning the effort overhead and speedup ratio of the PP practice are inconclusive [29,3,30,31,32,33,5]. According to the empirical results, the effort overhead is probably somewhere between 7%(13% excluding rework) [33] and 84% (obtained in a large one day experiment in 29 international consultancy companies) [5]. Speed-up ratio associated with PP is between 8% [5] and almost 47% [33]. Results obtained by other researchers [29,3,30,31,32] are somewhere between the above extremes.

Another important question concerning PP practice is whether it improves the quality of software products. Müller showed that in an academic setting a pair of developers did not produce more reliable code than a single developer whose code was reviewed [34,35]. Reliability was defined as the number of passed tests divided by the total number of tests. Madeyski conducted a large experiment with 188 students and concluded that there was no difference in the number of acceptance tests passed when PP was used instead of solo programming [36]. Moreover, package level design quality indicators (Martin's metrics [37]) were not significantly affected by using the PP practice [38]. Hulkko and Abrahamsson imply that PP may not necessarily provide as extensive quality benefits as suggested in the literature, and on the other hand, does not result in consistently superior productivity when compared to solo programming [32]. Arisholm et al. come to the conclusion that PP in general does not increase the proportion of correct solutions, as there was only a 7% increase in the proportion of correct solutions of the pairs compared with the individuals [5].

PP may have a positive impact on unit tests, e.g. code coverage as suggested by Langr [39], because of a synergy of the individuals in a pair working together on the same unit tests. However, to the author's knowledge, there is no empirical evidence concerning the impact of PP on the thoroughness and fault detection effectiveness of unit tests, with exception of the results of the experiment presented in Section 4. Therefore, the aim of this paper is to fill this gap.

4 Experiment Description

The definition, design, as well as operation of the experiment is described in this section. The following definition determines the foundation for the experiment [40]:

Object of study. The objects of study are software development products (developed code).

Purpose. The purpose is to evaluate the impact of pair programming practice on unit tests.

Quality focus. The quality focus is the thoroughness and fault detection effectiveness of unit test suites, measured by branch coverage and mutation score indicator, respectively.

Perspective. The perspective is from the researcher's point of view.

Context. The experiment is run with the help of MSc students as subjects involved in the development of a finance accounting system in Java.

4.1 Context Selection

The context of the experiment (described in Table 1) was the Programming in Java course, and hence the experiment was run off-line (not industrial software development) [40]. Java was the programming language and Eclipse was the Integrated Development Environment (IDE). All the subjects had prior experience in C and C++ programming (using OO approach) as a minimum. The course consisted of seven lectures and fifteen laboratory sessions (90 minutes each), and introduced Java programming language, using PP and TDD. The subjects' practical skills in programming in Java using PP and TDD were developed through training and evaluated during the first seven laboratory sessions (10.5 hours). The experiment took place during the last eight laboratory sessions (12 hours). The problem, i.e. the development of the finance accounting system, was as close to a real one, as it is possible in an academic environment. The system the students developed was aimed to support a small internet provider company in collecting data about clients and requested services, as well as the monitoring of accounts, and the history of all financial requests. The subjects participating in the study were mainly second and third-year (and few fourth and fifth-year) computer science MSc students. The MSc programme of Wroclaw University of Technology (WUT) is a 5-year programme after high school.

Table 1. The context of the experiment

Mean programming experience of subjects [years]	3.79
Duration of training tasks (7 sessions) [hours]	10.50
Duration of the experiment task (8 sessions) [hours]	12
Number of user stories used to describe requirements	27
Mean number of acceptance tests passed	13.13
Mean number of non-commented lines of code	895.98
Mean number of unit tests	20.37
Mean number of executed asserts	42.19

The study constituted part of an empirical research with the aim of obtaining empirical evidence on the impact of PP and TDD on different aspects of software products and processes [36,38,41,42,43,44].

4.2 Variables Selection

The independent variable is the software development method – solo (SP) or pair programming (PP) – used by the experiment groups. The dependent variables are mean values of branch coverage (denoted as BC) and mutation score indicator (denoted as MSI), described in Section 2.

4.3 Hypotheses Formulation

The crucial aspect of the experiment is to find out and formally state what is intended and to evaluate it. The following null hypotheses are to be tested:

- $H_{0\ BC,\ SP/PP}$ — There is no difference in the mean value of branch coverage (BC) between the solo programmers and pairs (SP and PP).
- $H_{0\ MSI,\ SP/PP}$ — There is no difference in the mean value of mutation score indicator (MSI) between the solo programmers and pairs (SP and PP).

4.4 Selection of Subjects

The choice of the subjects is based on convenience. They are students taking the Programming in Java course. Prior to the experiment, the students filled in a pre-test questionnaire. The aim of the questionnaire was to get the picture of the students' background. It turned out that the mean value of programming experience in calendar years was 3.7 for solos and 3.9 for pairs. The ability to generalize from this context is further elaborated, when discussing threats to the experiment.

4.5 Design of the Experiment

The design is one factor (the software development method), with the two treatments (SP and PP). The assignment to PP teams took into account the subjects' preferences (i.e. they were allowed to suggest partners), as it seemed to be more natural and close to real world practice. Thus this is a quasi-experiment [45]. All of the analysed teams were supposed to follow the rules of TDD practice. In the case of two of the solo projects, pre-test questionnaires were not filled in. In the case of one solo project, tests were not written and submitted properly. These projects were not included in the analysis. The resulting design was unbalanced, with 28 solo programmers and 35 pairs. Analysis of these 63 projects is conducted in Section 5, whilst further analysis of selected projects is performed in Sections 6.1 and 6.2.

4.6 Instrumentation and Measurement

The instruments [40] and materials for the experiment were prepared in advance, and consisted of requirements artifacts (user stories), pre-test and post-test questionnaires, Eclipse project framework, a detailed description of software development approaches (S and P), subjects' duties, instructions how to use the experiment infrastructure (e.g. CVS version control system) etc. BC and MSI were collected by means of Clover [46] and Judy [25] tools, respectively.

4.7 Validity Evaluation

When conducting the experiment, there is always a set of threats to the validity of the results. To enable an analysis of the validity of the current study, the possible threats are discussed, based on Wohlin et al. [40].

Threats to the *statistical conclusion* validity are concerned with the issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of the experiment, e.g. choice of statistical tests, tools and samples sizes, and the care taken in the implementation and measurement of the experiment [40]. Threats to the *statistical conclusion* validity are considered to be under control. Robust statistical techniques, tools (e.g. SPSS) and large sample sizes to increase statistical power are used. Non-parametric tests are used which do not require a certain underlying distribution of the data. Treatment implementation is considered reliable. However, the risk in the treatment implementation is that the experiment was spread across laboratory sessions. To minimize the risk, access to the Concurrent Versions System (CVS) repository was restricted to specific laboratory sessions (access hours and IP addresses). The validity of the experiment is highly dependent on the reliability of the measures which can be divided into two classes: objective measures, and subjective measures [40]. Measures used in the study were selected to be objective rather than subjective e.g. do not involve human judgement [40].

The *internal* validity of the experiment concerns the question of whether the effect is caused by independent variables, or by other factors. Concerning the *internal* validity, the risk of compensatory rivalry, or demoralization of subjects receiving less desirable treatments must be considered. The group using the classical method (i.e. SP) may do their very best to show that the old method is competitive. On the other hand, subjects receiving less desirable treatments may perform not as well as they generally do. However, the subjects were informed that the goal of the experiment was to measure different development methods, not the subjects' skills. A possible diffusion or imitation of treatments was under control of the assistant lecturers. The threat of selection was also under control, as the experiment was a mandatory part of the course. It was also checked that mean programming experience (in calendar years) was similar in each group (S and P). Moreover, according to questionnaires, mean programming experience of the subjects who took part in the experiment, and three solo subjects who were excluded from the analysis, were almost the same.

Construct validity concerns the ability to generalize from the experiment result to the concept behind the experiment. Some threats relate to the design of the experiment, and others to social factors [40]. Threats to the *construct* validity are considered not very harmful. The mono-operation bias is a threat, as the experiment was conducted on a single software development project. Using a single type of measure is a mono-method bias threat. To reduce mono-method threats, the post-test questionnaire was added, to enable qualitative validation of the results. There seems to be no apparent contradiction between qualitative and quantitative results. For example, it turned out that, in the case of solo programmers, 8 subjects (28.6%) strongly disagreed, while 4 subjects (14.3%) disagreed with the statement "I am more sure of software developed by means of pair programming approach than solo programming". On the other hand 11 subjects (39.3%) agreed, while 5 subjects (17.9%) strongly agreed with the statement. In the case of programmers working in pairs, 1 subject (1.4%) strongly disagreed, while 9 subjects (12.9%) disagreed with the statement. Furthermore, 36 subjects (51.4%) agreed, while 24 subjects (34.3%) strongly agreed with the statement. Interaction of different treatments is limited due to the fact that the

subjects were involved in one study only. Other threats to construct validity are social threats (e.g. hypothesis guessing and experimenter expectancies). As neither the subjects nor the experimenters have any interest in favour of one technique or another, we do not expect it to be a large threat.

As with most empirical studies in software engineering, an important threat is the process conformance represented by the level of conformance of the subjects to the prescribed techniques. Process conformance is a threat to statistical conclusion validity, through the variance in the way the processes are actually carried out, and also to construct validity, through possible discrepancies between the processes as prescribed, and the processes as carried out [48]. The process conformance threat was handled to some extent by attempting to avoid deviations, with help from assistant lecturers. They controlled how development methods were carried out and induced subjects to follow the prescribed techniques. Moreover, the subjects were informed of the importance of following proper development methods. However, we can not exclude deviations from the prescribed techniques so the issue of process conformance may not be neglected. The ratio of lines of test code to lines of production code is rather low (16998:39538). Therefore, in more selective analysis in Section 6.1, we decided to remove projects with seriously limited number of JUnit assertions executed by unit test suites so as to sample from the population of projects that might follow TDD guidelines more strictly.

Threats to *external* validity are the conditions that limit our ability to generalize the results of our experiment to industrial practice. The largest threat stems from the fact that the subjects were students, who had short-time experience in PP. However, Kitchenham et al. [49] states that students are the next generation of software professionals and thus, are relatively close to the population of interest. Some indications on the similarities between student subjects and professionals are also given by Höst et al. [50]. Moreover, Tichy provides several arguments why students are acceptable as subjects [51]. The threats to external validity were reduced by making the experimental environment as realistic as possible (e.g. requirements specification came from an external client).

4.8 Experiment Operation

The experiment was run at WUT and consisted of a preparation phase and an execution phase. The preparation phase of the experiment embraced lectures and training exercises, given directly before the experiment in order to improve skills and to give practice in the areas of PP, TDD, and unit testing using JUnit. Lectures and exercises were given by the author, as well as by assistant lecturers. The goal of this preparation phase was to train student subjects sufficiently well to perform the tasks required of them. They must not be overwhelmed by the complexity of, or unfamiliarity with, the tasks [51]. Therefore, it took seven laboratory sessions (over 10 hours) to achieve the goal. Then, the subjects were given an introductory presentation of a finance accounting system and were asked to implement it during eight laboratory sessions of the execution phase. Both, the preparation phase and the execution phase, were conducted in classroom settings under continuous supervision of assistant lecturers. The subjects were divided into S and P groups. Up-to-date development environment composed of Java

Development Kit, Eclipse IDE, JUnit testing framework and CVS repository were used in the experiment. Additionally, the subjects filled in pre-test and post-test questionnaires, to evaluate their experience and opinions, as well as to enable qualitative validation of the results. The subjects were not aware of the actual hypotheses stated. The data were collected automatically by tools such as Clover and Judy (the tool developed at WUT).

5 Analysis of the Experiment

The experiment data are analysed with descriptive analysis and statistical tests. The preliminary analysis of all 63 projects is conducted in this section, whilst further, more selective analysis is conducted in Section 6.1, and analysis of confounding variables is performed in Section 6.2.

5.1 Descriptive Statistics

Descriptive statistics of gathered measures are summarized in Table 2. Columns "Mean", "Std.Deviation", "Std.Error", "Max", "Median" and "Min" state the mean value, standard deviation, standard error, maximum, median and minimum for each measure and development method, respectively.

Table 2. Descriptive statistics for branch coverage (BC) and mutation score indicator (MSI)

Measure	Development Method	Mean (M)	Std.Deviation (SD)	Std.Error (SE)	Max	Median (Mdn)	Min
Branch Coverage (BC)	SP	.38	.22	.042	.90	.39	.00
	PP	.39	.21	.036	.83	.32	.09
Mutation Score Indicator (MSI)	SP	.27	.14	.027	.50	.29	.00
	PP	.24	.14	.023	.56	.24	.00

The first impression is that developers working in pairs (denoted as P), and developers working solo (S) performed similarly. The accuracy of the mean as a model of the data can be assessed by the standard deviation which, unfortunately, is rather large (compared to the mean). The standard deviation, as well as boxplots in Figures 1 and 2 tell us more about the shape of the distribution of the results.

Summarizing descriptive statistics in correct APA (American Psychological Association) format [52], we can conclude that branch coverage for pairs ($M = .39$, $SD = .21$) and solo programmers ($M = .38$, $SD = .22$) are similar. Mutation score indicator for pairs ($M = .27$, $SD = .14$) and solo programmers ($M = .24$, $SD = .14$) are similar as well.

To answer the question whether the impact of PP on mutation score indicator and branch coverage is significant, or not, statistical tests must be performed.

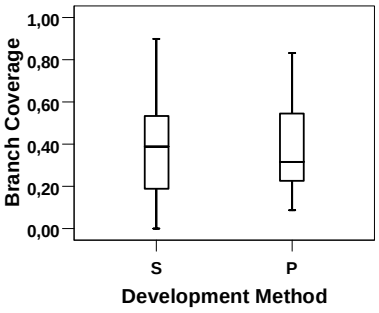

Fig. 1. Branch coverage boxplots

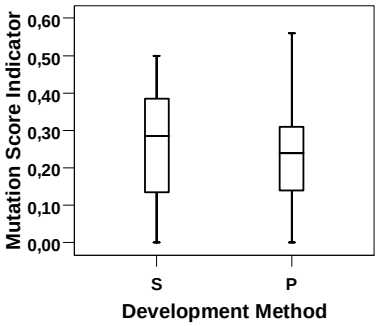

Fig. 2. Mutation score indicator boxplots

5.2 Hypotheses Testing

We start with exploratory analysis on the collected data to check whether they follow the assumptions of the parametric tests. That is, observations should be independent. The dependent variable should be measured on at least an interval scale. The variances in each experimental condition should be fairly similar, and data should come from a normally distributed population.

The first assumption, namely that observations should be independent, and the second assumption, that the collected data must be measured at an interval or ratio level are met.

The third assumption (homogeneity of variance) means that the variances should be the same throughout the data (i.e. between groups). Levene's tests, used to check this assumption, are non-significant (because significances p are greater than .05), indicating that the variances in the different experimental groups are roughly equal (i.e. not significantly different).

The fourth assumption of parametric tests is that our data have come from a population that has normal distribution. Objective tests of the distribution are Kolmogorov-Smirnov and Shapiro-Wilk tests. For the mutation score indicator data the distribution for pairs, as well as solos appears to be normal ($p > .05$). For the branch coverage data the distribution for solos is normal, whereas that for pairs seems to be non-normal ($p < .05$) according to the Shapiro-Wilk test. This finding suggests we should use a non-parametric test. Therefore, the hypotheses from Section 4.3 are evaluated by means of the Mann-Whitney one way analysis of variance by ranks. The Mann-Whitney non-parametric tests are used for testing differences between the two experimental groups (S and P), when different subjects are used in each group.

Table 3 shows test statistics and significances. An effect size ($r = \frac{Z}{\sqrt{N}}$ where Z is the z-score in Table 3, and N is the size of the study i.e. 63) is an objective and standardized measure of the magnitude of observed effect. The effect size is extremely small for branch coverage ($r = -.03$) and a bit higher, but still small, for mutation score indicator ($r = -.12$). So, we could report that branch coverage was not significantly affected by pair programming approach (the Mann-Whitney

Table 3. Mann-Whitney test statistics (grouping variable: Development Method)

	Branch Coverage (BC)	Mutation Score Indicator (MSI)
Mann-Whitney U	471.500	422.000
Wilcoxon W	877.500	1052.000
Z	-.256	-.941
Exact Sig. (2-tailed)	.802	.351

test statistics: $U = 471.5$, non-significant, $r = -.03$). Moreover, mutation score indicator was not significantly affected by the pair programming approach (the Mann-Whitney test statistics: $U = 422.0$, non-significant, $r = -.12$), either.

6 Discussion of the Results and Further Analysis

Why did PP not result in a significant increase of testing thoroughness or fault detection effectiveness, measured by branch coverage and mutation score indicator, respectively? A plausible explanation is that when a software project is not big enough, and the requirements are decomposed into small features (user stories), the impact of PP practice on branch coverage and mutation score indicator may be insignificant, because development skill may, to a certain extent, compensate for the lack of a second pair of eyes. Furthermore, when the scope of the project is limited, the impact of PP practice on branch coverage and mutation score indicator may be insignificant, because of the limited number of tests. Therefore, in further analysis in Section 6.1 we decided to remove projects with limited number of assertions executed by unit tests suites.

An alternative explanation is that development method may not be the best, or the only one explanatory variable. Therefore, we decided to conduct analysis of covariance (ANCOVA) in Section 6.2 to reduce pre-existing differences among subjects that already have influence over our dependent variables, as well as to get a more sensitive measure of our experiment effect.

6.1 Selective Analysis

In the further, more selective analysis we decided to remove projects with a seriously limited number of JUnit assertions executed by unit tests suites as we intended to sample from the population of projects that might take advantage of unit tests more seriously. The threshold of 30 assertions executed by unit tests suites was determined as a compromise between number of testing resources required to take advantage of unit testing and number of projects needed to conduct inferential statistics. As a result, further analysis was run with subjects involved in the 36 projects conducted, using TDD approach, by 17 solo programmers (denoted as S) and 19 pairs (denoted as P).

Descriptive Statistics. Descriptive statistics of gathered measures are summarized in Table 4, whilst the shape of the distribution of the results is shown by boxplots in Figures 3 and 4.

Table 4. Descriptive statistics for branch coverage (BC) and mutation score indicator (MSI) of selected 36 projects

Measure	Development Method	Mean (M)	Std.Deviation (SD)	Std.Error (SE)	Max	Median (Mdn)	Min
Branch Coverage (BC)	SP	.47	.17	.040	.90	.46	.14
	PP	.47	.20	.047	.83	.40	.19
Mutation Score Indicator (MSI)	SP	.34	.11	.026	.50	.37	.10
	PP	.30	.14	.031	.56	.29	.09

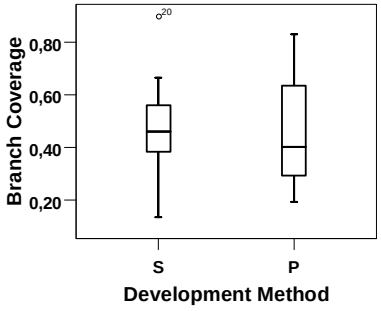

Fig. 3. Branch coverage boxplots of 36 selected projects

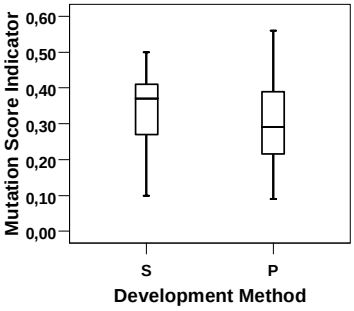

Fig. 4. Mutation score indicator boxplots of 36 selected projects

The decision to remove projects with a limited number of testing resources resulted in higher values of BC, as well as for MSI means, medians, and minimums. Summarizing descriptive statistics in APA format, we can conclude that BC for pairs ($M = .47$, $SD = .20$) and solo programmers ($M = .47$, $SD = .17$) are similar. MSI for pairs ($M = .30$, $SD = .14$) and solo programmers ($M = .34$, $SD = .11$) differ slightly. To answer the question whether the impact of PP on mutation score indicator and branch coverage is significant or not, statistical tests must be performed.

Hypotheses Testing. Levene's tests of homogeneity of variances are non-significant indicating that the variances in the different experimental groups are not significantly different. Kolmogorov-Smirnov and Shapiro-Wilk tests of normality are non-significant ($p > .05$) indicating that the data are normally distributed, therefore, we decided to use parametric statistics. We may use the independent t-test in situations in which there are only two levels of the independent variable (i.e. two experimental groups), and different participants will be used in each group. Table 5 shows the output from an independent t-test.

The exact two-tailed significance values of t are greater than .05, so we would have to conclude that there was no significant difference between the means of our two groups. To discover the effect size we can use the equation $r = \sqrt{\frac{t^2}{t^2+df}}$. We know the value of t (t-test statistic) and df (degrees of freedom calculated by adding the two groups sizes and then subtracting the number of groups). The effect size is extremely small for branch coverage ($r = .01$) and a bit higher,

Table 5. Independent samples t-Tests

	t-test for Equality of Means				
	t	df	Sig.(2-tailed)	Mean Diff.	Std.Error Diff.
Branch Coverage (BC)	.077	34	.939	.005	.063
Mutation Score Ind. (MSI)	.842	34	.406	.035	.041

but still small, for mutation score indicator ($r = .14$). The difference in BC and MSI between solo programmers and pairs was not significant in the case of branch coverage ($t(34) = .077$, $p > .05$, $r = .01$), or in mutation score indicator ($t(34) = .842$, $p > .05$, $r = .14$). Thus, the results of the selective analysis conducted in this section are in line with the results of the preliminary analysis performed in Section 5.

6.2 Analysis of Covariance

Up to now the only explanatory variable we have used is the one that denotes group membership (i.e. development method SP or PP). However, in many situations, such a group-membership variable accounts for a relatively small proportion of the total variance in the dependent variables. In fact, in this section we will discover that this is the case indeed. This should not be surprising as pre-existing differences among subjects are at least as important a predictor of their scores on the dependent variables as any treatment variable [53]. Therefore, we apply analysis of covariance (ANCOVA) and make use of information concerning the individual differences among the subjects that were present at the beginning of the study.

Moreover, the ability of ANCOVA to compensate to some extent for pre-existing differences among groups justifies the fact that ANCOVA is also recommended as a means of addressing the threats to the internal validity that arise in studies with selection differences between groups [47].

Tables 6 and 7 show the ANCOVA tables for BC and MSI when covariates are not included. According to the significance values ($p > .05$), there are no differences in BC, or MSI between the two groups, i.e. solos and pairs. Therefore, PP practice seems to have no significant effect on BC and MSI. It is obviously in line with the results obtained in Section 5 and 6.1.

What is interesting in ANCOVA results (see Tables 6 and 7) is that development method variable accounts for a very small proportion of the total variance in the dependent variables (BC and MSI). In branch coverage results (see Table 6) the total amount of variation to be explained (SS_T) was 1.193, of which the experimental manipulation (development method) accounted for .000 units (SS_M), whilst 1.193 were unexplained (SS_R). In mutation score indicator results (see Table 7) the total amount of variation to be explained (SS_T) was .535, of which the experimental manipulation (development method) accounted for .011 units (SS_M), whilst .524 were unexplained (SS_R).

One of the findings is that in future experiments we should take into account other variables (covariates), that are not part of the main experimental manipulation but have an influence on the dependent variable. In fact, it can be

Table 6. Test of between-subjects effects without covariates (BC)

Source	Type III Sum of Squares	df	Mean Square	F	Significance	Partial Eta Squared
Corrected Model	.000[1]	1	.000	.006	.939	.000
Intercept	7.947	1	7.947	226.531	.000	.869
Development Method	.000	1	.000	.006	.939	.000
Error	1.193	34	.035			
Total	9.160	36				
Corrected Total	1.193	35				

[1] R Squared = .000

Table 7. Test of between-subjects effects without covariates (MSI)

Source	Type III Sum of Squares	df	Mean Square	F	Significance	Partial Eta Squared
Corrected Model	.011[1]	1	.011	.709	.406	.020
Intercept	3.653	1	3.653	236.986	.000	.875
Development Method	.011	1	.011	.709	.406	.020
Error	0.524	34	.015			
Total	4.177	36				
Corrected Total	.535	35				

[1] R Squared = .020

seen as a convenient way to deal with context variables e.g. education, experience, personality, communication skills etc. To make the move in this direction and present the useful statistical framework right now, we decided to take into account additional information about subjects i.e. grades obtained by subjects after the preparation phase of the experiment (see Section 4.8). A traditional four-point system was used; the grades are: 2.0 (fail), 3.0 (pass), 3.5, 4.0, 4.5, 5.0 (very good) along with additional, non-standard 5.5 (the highest grade accepted at WUT). Categorical covariates can only be used in ANCOVA analysis if you dummy code them into binary variables. However, all the subjects got grades within the range 4.0 and 5.5. Therefore, only three dummy variables (Grade4.5, Grade5.0 and Grade5.5) for the sake of covariance analysis were created. In ANCOVA one less dummy variable is always needed than the number of groups (e.g. subjects that got grade 4.0 have a code of 0 for all three dummy variables). The so called pretest is often used as a covariate, because how the subjects score before treatments generally correlates with how they score after treatments [54].

Table 8 shows the results of Levene's test of equality of error variances. We test the null hypotheses that the error variances of the dependent variables (BC and MSI) are equal across groups. Fortunately, Levene's tests are non-significant ($p > .05$) and this assumption of ANCOVA has not been violated.

Tables 9 and 10 are the ANCOVA tables where covariates are included in the model. Looking at the significance values in Table 9 and 10, neither development method nor covariates significantly predicts the dependent variables

Table 8. Levene's test of equality of error variances (design: Intercept+Grade4.5+Grade5.0+Grade5.5+Development Method)

Dependent Variable	Levene Statistic	Significance
Branch Coverage (BC)	1.183	.284
Mutation Score Indicator (MSI)	.144	.707

(BC and MSI). However, in both cases development method accounts for a very small portion of variance accounted for by the models. The amount of variation accounted for by the branch coverage model has increased from .000 to .115 units (*corrected model*) of which the development method accounts for .001 units only, while the covariates Grade4.5, Grade5.0 and Grade5.5 account for .001, .030 and .024 units, respectively. The amount of variation accounted for by the mutation score indicator model has increased from .011 to .063 units (*corrected model*) of which the development method accounts for .012 units, while the covariates Grade4.5, Grade5.0 and Grade5.5 account for .005, .023 and .023 units, respectively.

Table 9. Test of between-subjects effects with covariates (BC)

Source	Type III Sum of Squares	df	Mean Square	F	Significance	Partial Eta Squared
Corrected Model	.115[1]	4	.029	.827	.518	.096
Intercept	.101	1	.101	2.893	.099	.085
Grade4.5	.001	1	.001	.040	.843	.001
Grade5.0	.030	1	.030	.874	.357	.027
Grade5.5	.024	1	.024	.687	.414	.022
Development Method	.001	1	.001	.036	.852	.001
Error	1.078	31	.035			
Total	9.160	36				
Corrected Total	1.193	35				

[1] R Squared = .096

This example illustrates how ANCOVA can help us to exert stricter experimental control by taking confounding variables into account to give us a purer measure of the effect of the experimental manipulation. Without taking confounding variables into account we would not have taken account of the context of the experiment e.g. pre-existing differences among subjects.

ANCOVA has the same basic assumptions as all of the parametric tests, but it has an additional one as well: that is, the assumption of homogeneity of regression slopes. For example, if the relationship between the dependent variable and covariate differs across the groups then this additional assumption of ANCOVA is violated and the overall model is inaccurate. This assumption of ANCOVA has not been violated in this case.

Analysis of covariance presents a more sensitive measure of our experiment effect and reveals the fact that the development method (SP or PP) accounts

Table 10. Test of between-subjects effects with covariates (MSI)

Source	Type III Sum of Squares	df	Mean Square	F	Significance	Partial Eta Squared
Corrected Model	.063[1]	4	.016	1.032	.406	.118
Intercept	.031	1	.031	2.036	.164	.062
Grade4.5	.005	1	.005	.332	.569	.011
Grade5.0	.023	1	.023	1.497	.230	.046
Grade5.5	.023	1	.023	1.514	.228	.047
Development Method	.012	1	.012	.772	.386	.024
Error	.472	31	.015			
Total	4.177	36				
Corrected Total	.535	35				

[1] R Squared = .118

for an extremely small proportion of the total variance in the branch coverage and mutation score indicator. As a result, we may conclude that after adjusting for pre-intervention grades, branch coverage, and mutation score indicator were not significantly affected by PP approach ($F(1, 31) = .04$, $p > .05$, partial eta squared $< .01$ and $F(1, 31) = .77$, $p > .05$, partial eta squared $= .02$, respectively). Effect sizes, as indicated by the corresponding partial eta squared values, are small and indicate how much of the variance in the dependent variables is explained by the independent variable (i.e. the development method). The covariates were not significantly related to the dependent variables, either.

7 Summary and Conclusions

The unique aspect of the experiment conducted at WUT was that it included the first ever assessment of the impact of pair programming on thoroughness and fault detection effectiveness of unit tests. Branch coverage and mutation score indicator were examined to find how thoroughly tests exercise programs, and how effective they are, respectively. The main result of this study is that the pair programming practice used by the subjects, instead of solo programming, did not significantly affect branch coverage or mutation score indicator. It means that the impact of pair programming on thoroughness and the fault detection effectiveness of unit test suites was not confirmed.

In further, more selective analysis we removed projects with a limited number of testing resources as we intended to sample from the population of projects that might take advantage of unit tests. The results of the selective analysis were in line with our previous findings that pair programming did not significantly affect aforementioned unit tests qualities. Moreover, carrying out the analysis of covariance in Section 6.2 we have attempted to reduce pre-existing differences among subjects that already have influence over our dependent variables, and got a more sensitive measure of our experiment effect. It turned out that development method (SP or PP) as an explanatory variable accounts for an extremely small proportion of the total variance in the dependent variables (i.e. BC and MSI). Therefore, in future experiments we should take account of other context

variables (e.g. experience, communication skills), that are not part of the main experimental manipulation but have an influence on the dependent variable. The presented analysis of covariance can be seen as a convenient way to deal with context variables.

The validity of the results must be considered within the context of the limitations discussed in the validity evaluation section. The study can benefit from several improvements before replication is attempted. The most significant one is conducting larger projects, while securing a sample of large enough size to guarantee a high-power design. Further experimentation in different contexts (e.g. within industry) is needed to establish evidence-based recommendations for the impact of pair programming practice on thoroughness and fault detection effectiveness of unit test suites.

8 Acknowledgements

The author expresses his gratitude to the students and lecturers participating in the experiment, the members of the e-Informatyka team (Wojciech Gdela, Tomasz Poradowski, Jacek Owocki, Grzegorz Makosa, Mariusz Sadal and Michał Stochmiałek) and especially Norbert Radyk (the lead developer of Judy) for their help in preparing the measurement infrastructure and collecting data. This work has been financially supported by the Ministry of Science and Higher Education, as a research grant 3 T11C 061 30 (years 2006-2007).

References

1. Williams L, Kessler R. *Pair Programming Illuminated*. Addison-Wesley: Boston, 2002.
2. Beck K, Andres C. *Extreme Programming Explained: Embrace Change*. 2nd edn. Addison-Wesley: Boston, 2004.
3. Williams L, Kessler RR, Cunningham W, Jeffries R. Strengthening the Case for Pair Programming. *IEEE Software* **2000; 17**(4): 19–25.
4. Williams LA, Kessler RR All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM* **2000; 43**(5): 108–114.
5. Arisholm E, Gallis H, Dybå T, Sjøberg, DIK. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions on Software Engineering* **2007; 33**(2): 65–86.
6. Beck K. *Test Driven Development: By Example*. Addison-Wesley: Boston, 2002.
7. Kobayashi O, Kawabata M, Sakai M, Parkinson E. Analysis of the Interaction between Practices for Introducing XP Effectively. In: *ICSE'06: 28th International Conference on Software Engineering*, ACM Press: New York, 2006, 544–550.
8. Marick B. How to Misuse Code Coverage. In: *16th International Conference on Testing Computer Software*, Washington, 1999, (Retrieved 2007) http://www.testing.com/writings/coverage.pdf.
9. Kaner C. Software Negligence and Testing Coverage. In: *STAR 96: 5th International Conference, Software Testing, Analysis and Review*, Orlando, 1996, 299–327.
10. Cai X, Lyu MR. The Effect of Code Coverage on Fault Detection under Different Testing Profiles. *SIGSOFT Software Engineering Notes* **2005; 30**(4): 1–7.
11. Cornett S. Code Coverage Analysis (Retrieved 2007) http://www.bullseye.com/coverage.html.

12. DeMillo RA, Lipton RJ, Sayward FG. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* **1978; 11**(4): 34–41.
13. Hamlet, RG. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering* **1977; 3**(4): 279–290.
14. Offutt AJ, Untch RH. Mutation 2000: Uniting the Orthogonal. In: *Mutation Testing for the New Century.* Kluwer Academic Publishers: Norwell, 2001, 34–44.
15. Zhu H, Hall PAV, May JHR. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys* **1997; 29**(4): 366–427.
16. Walsh PJ. A Measure of Test Case Completeness. PhD thesis, Univ. New York: 1985.
17. Frankl PG, Weiss SN, Hu C. All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness. *Journal of Systems and Software* **1997; 38**(3): 235–253.
18. Offutt AJ, Pan J, Tewary K, Zhang T. An Experimental Evaluation of Data Flow and Mutation Testing. *Software Practice and Experience* **1996; 26**(2): 165–176.
19. Venners B. Test-Driven Development. A Conversation with Martin Fowler, Part V (Retrieved 2007) http://www.artima.com/intv/testdrivenP.html.
20. Moore I. Jester - a JUnit test tester. In: *XP'01: 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2001, 84–87.
21. Offutt J. An analysis of Jester based on published papers (Retrieved 2006) http://www.ise.gmu.edu/˜ofut/jester-anal.html.
22. Ma YS, Offutt J, Kwon YR. MuJava: A Mutation System for Java. In: *ICSE'06: 28th International Conference on Software Engineering*, ACM Press: New York, 2006, 827–830.
23. Smith BH, Williams L. An Empirical Evaluation of the MuJava Mutation Operators. In: *TAICPART-MUTATION'07: Testing: Academic and Industrial Conference Practice and Research Techniques*, 2007, 193–202.
24. Chevalley P, Thévenod-Fosse P. A mutation analysis tool for Java programs. *International Journal on Software Tools for Technology Transfer* **2003; 5**(1): 90–103.
25. Madeyski L, Radyk N. Judy mutation testing tool project. (Retrieved 2007) http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.Judy
26. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* **1996; 5**(2): 99–118.
27. Ammann P, Offutt J. Introduction to Software Testing. (2007) In preparation. http://ise.gmu.edu/ offutt/softwaretest/
28. Ma YS, Harrold MJ, Kwon YR. Evaluation of Mutation Testing for Object-Oriented Programs. In: *ICSE'06: 28th International Conference on Software Engineering*, ACM Press: New York, 2006, 869–872.
29. Nosek JT. The Case for Collaborative Programming. *Communications of the ACM* **1998; 41**(3): 105–108.
30. Nawrocki JR, Wojciechowski A. Experimental Evaluation of Pair Programming. In: *ESCOM'01: European Software Control and Metrics*, London, 2001, 269–276.
31. Nawrocki JR, Jasiński M, Olek L, Lange B. Pair Programming vs. Side-by-Side Programming. *Lecture Notes in Computer Science* **2005; 3792**: 28–38.
32. Hulkko H, Abrahamsson P. A Multiple Case Study on the Impact of Pair Programming on Product Quality. In: *ICSE'05: 27th International Conference on Software Engineering*, ACM Press: New York, 2005, 495–504.
33. Müller MM. Two controlled experiments concerning the comparison of pair programming to peer review. *Journal of Systems and Software* **2005; 78**(2): 166–179.
34. Müller MM. Are Reviews an Alternative to Pair Programming? In: *EASE'03: Conference on Empirical Assessment In Software Engineering*, 2003, 3–12.

35. Müller MM. Are Reviews an Alternative to Pair Programming? *Empirical Software Engineering* **2004; 9**(4): 335–351.
36. Madeyski L. Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. *Frontiers in Artificial Intelligence and Applications* **2005; 130**: 113–123. http://madeyski.e-informatyka.pl/download/Madeyski05b.pdf
37. Martin RC. OO Design Quality Metrics: An Analysis of Dependencies. (Retrieved 2006) http://www.objectmentor.com/resources/articles/oodmetrc.pdf
38. Madeyski L. The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment. *Lecture Notes in Computer Science* **2006; 4034**: 278–289. DOI: 10.1007/11767718_24
39. Langr J. Pair Programming Observations. (Retrieved 2007) http://www.langrsoft.com/articles/pairing.shtml
40. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers: Norwell, 2000.
41. Madeyski L. Is External Code Quality Correlated with Programming Experience or Feelgood Factor? *Lecture Notes in Computer Science* **2006; 4044**: 65–74. DOI: 10.1007/11774129_7
42. Madeyski L. On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests. *Lecture Notes in Computer Science* **2007; 4589**: 207–221. DOI: 10.1007/978-3-540-73460-4_20
43. Madeyski L, Szała Ł. The Impact of Test-Driven Development on Software Development Productivity - An Empirical Study. *Lecture Notes in Computer Science* **2007; 4764**: 200–211. DOI: 10.1007/978-3-540-75381-0_18
44. Madeyski L, Szała Ł. Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. *IET Software* **2007; 1**(5): 180–187. DOI: 10.1049/iet-sen:20060071
45. Shadish WR, Cook TD, Campbell DT. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin: Boston, 2002.
46. Cenqua Pty Ltd Clover project (Retrieved 2006) http://www.cenqua.com/clover/.
47. Cook TD, Campbell DT. *Quasi-Experimentation: Design and Analysis Issues*. Houghton Mifflin: Boston, 1979.
48. Sørumgård LS. *Verification of Process Conformance in Empirical Studies of Software Development*. PhD thesis, Norwegian University of Science and Technology, 1997.
49. Kitchenham B, Pfleeger SL, Pickard L, Jones P, Hoaglin DC, Emam KE, Rosenberg J. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering* **2002; 28**(8): 721–734.
50. Höst M, Regnell B, Wohlin C. Using Students as Subjects — A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering* **2000; 5**(3): 201–214.
51. Tichy WF. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Software Engineering* **2000; 5**(4): 309–312.
52. *Publication manual of the American Psychological Association*. 5th edn., American Psychological Association: Washington, 2001.
53. Maxwell SE, Delaney HD. *Designing Experiments and Analyzing Data: A Model Comparison Perspective*. 2nd edn. Lawrence Erlbaum: Mahwah, 2004.
54. Stevens JP. *Applied Multivariate Statistics for the Social Sciences*. Lawrence Erlbaum: Mahwah, 2002.