On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests

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. Code coverage and mutation score measure how thoroughly tests exercise programs and how effective they are, respectively. The objective is to provide empirical evidence on the impact of pair programming on both, thoroughness and effectiveness of test suites, as pair programming is considered one of the practices that can make testing more rigorous, thorough and effective. A large experiment with MSc students working solo and in pairs was conducted. The subjects were asked to write unit tests using JUnit, and to follow test-driven development approach, as suggested by eXtreme Programming methodology. It appeared that branch coverage, as well as mutation score indicator (the lower bound on mutation score), was not significantly affected by using pair programming, instead of solo programming. However, slight but insignificant positive impact of pair programming on mutations score indicator was noticeable. The results do not support the positive impact of pair programming on testing to make it more effective and thorough. The generalization of the results is limited due to the fact that MSc students participated in the study. It is possible that the benefits of pair programming will exceed the results obtained in this experiment for larger, more complex and longer projects.

1 Introduction

Pair programming (PP) [1] is key software development practice of eXtreme Programming (XP) methodology [2] which has recently gained a lot of attention. Pair programming is a practice in which two distinct roles, called a driver and a navigator, are distinguished. They contribute to a 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 is typing at the keyboard and focusing 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 is looking for tactical and strategic defects or alternatives [5]. The rule is that all production code is written by two people sitting at one machine [2]. In the case of solo programming, all activities are performed by one programmer.

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

Pair programming is supposed to be software development practice that can influence unit testing to make it more rigorous, thorough, and effective. The question is whether the impact of pair programming is significant or not.

2 Measures

Programmers who write unit tests should have a set of guidelines indicating whether their software has been thoroughly and effectively tested.

2.1 Code Coverage

Measuring code coverage is one of such guidelines which can be applied, as code coverage tools measure how thoroughly tests exercise programs [8]. However, it remains a controversial issue whether code coverage is a good indicator for fault detection capability of test cases [9]. Marick [8] shows that code coverage may be misused, but code coverage tools are still helpful if they are used to enhance thought, and not to replace it. Cai and Lyu [10] found that code coverage was a good estimator for fault detection of exceptional test cases, but a poor one for test cases in normal operations.

Kaner [9] lists 101 coverage measures. The important question is which code coverage measure should be used. Useful insights concerning 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, decision coverage, also known as branch coverage, has been devised. Decision coverage is a measure based on whether decision points, such as `if` and `while` statements, evaluate to both true and false during test execution, thus exercising both execution paths. Decision coverage includes statement coverage, since exercising every branch must lead to exercising every statement. However, a shortcoming of this measure is that it ignores branches within boolean expressions which occur due to short-circuit operators. For example, it can preclude calls to some methods. Unfortunately, the most powerful measures as Modified Condition/Decision Coverage (MC/DC), created at Boeing and 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 available code coverage measures. This measure is offered by several tools e.g. Clover, JCoverage, Cobertura. A detailed analysis revealed that Clover, JCoverage and Cobertura calculate branch coverage in slightly different ways. Therefore, to validate the results obtained by Clover, which has the market leader status, branch coverage results were collected by JCoverage and Cobertura as well. Finally, 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, and the actual testing of the software is a side effect [14]. The effectiveness of test suites for fault localization is estimated on the 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 tests quality [15]. The total number of non-equivalent mutants is 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, we are ready to accept the lower bound on mutation score (named mutation score indicator). Accepting it results in cost-effective application of a mutation analysis and still provides meaningful information about fault-finding 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. Fowler [19] found mutation testing tool support useful in practice.

Although mutation testing is powerful, it is not meant as a replacement for code coverage, only as a complementary approach useful to find 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, mutation tools for Java, proposed so far, have several limitations that prevent practitioners from using them. They are too slow to be used in large software projects (e.g. Jester [20]), modify source code of the software components and may break the code making operation risky (e.g. Jester). They do not work with JUnit [21] tests, the most widely used unit testing framework (e.g. MuJava [22,23]), do not support the execution of mutants and are not freely available for download (e.g. JAVAMUT [24]). Therefore, a new mutation tool, called Judy, has been developed, using aspect-oriented approach to speed up mutation testing [25]. Judy has a build-in support of JUnit unit tests, and is under active development to offer a wide range of mutations. Mutations set, used in the experiment, consists of 14 mutations, see Table 1.

Table 1. Judy Mutation Operators

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 assignment replacement
EAM – Accessor method change	EMM – Modifier method change

The first five operators (ABS, AOR, LCR, ROR, UOI) were taken from Offutt et al.'s research [26] on identifying 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. Recently, Ammann and Offutt [27] 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 thought of as good mutation operators for OO programs. Finally, EAM and EMM mutation operators were added, as there is still no determined set of selective mutation operators for class mutation operators. Thus, there is no strong reason to exclude these operators [28].

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

3 Related Work

Researchers and practitioners have reported numerous, sometimes anecdotal and favourable studies of pair programming. Beck and Andres wrote that a pair is even more than twice as effective as the same two people programming solo [2]. However, empirical evidence concerning pair programming practice effort overhead and speedup ratio often points to, more or less, the opposite, see Table 2. The results of empirical studies suggest that the effort overhead is probably somewhere between 15% and 60%, and speedup ratio is between 20% and over 40%.

Table 2. Empirical evidence on pair programming practice effort overhead
Approaches: S(Solo), P(Pair), SbS(side-by-side)

Study	Environment	Subjects	Effort overhead and speedup ratio associated with pair programming
[29]	Industry	15(5P/5S)	42% overhead, 29% speed up
[3]	Academic	41(14P/13S)	15%–60% overhead, 20%–42.5% speed up
[30]	Academic	21(5P/5+6S)	60% overhead, 20% speed up
[31]	Academic	25(5P/5SbS/5S)	50% overhead (but only 20% overhead in the case of SbS programming i.e. everyone has their own PC)
[32]	Acad./Ind.	4 case projects (4/5.5/4/4-6)	Neither P nor S had consistently higher productivity.
[5]	Industry	295(98P/99S)	P in general did not reduce the time required to solve the tasks correctly.

Another important question concerning pair programming practice is whether it improves the quality of software products. Empirical results concerning the impact of pair programming practice on quality of software products are summarized in Table 3. The results of empirical studies suggest that the the positive impact of pair programming on software quality is questionable.

Table 3. Empirical evidence on the impact of pair programming practice on software quality
Approaches: S(Solo), P(Pair), T(TDD), C(Classic, test-last)

Study	Environment	Subjects	Impact on software quality
[33]	Academic	37(10P/17S)	P did not produce more reliable code than S whose code was reviewed.
[34,35]	Academic	188 (28CS/28TS/ 31CP/35TP)	There was no difference in NATP(Number of Acceptance Tests Passed) between S and P. Package dependencies were not significantly affected by P.
[32]	Acad./Ind.	4 case projects (4/5.5/4/4-6)	Lower level of defect density in the case of P was not supported.
[5]	Industry	295(98P/99S)	P in general did not increase the proportion of correct solutions.

To the author's knowledge, there is no empirical evidence concerning the impact of pair programming on thoroughness and fault-finding effectiveness of unit tests. Therefore, the aim of this paper is to fill in this gap.

4 Experiment Description

The definition, design, as well as operation of the experiment are described in this section.

4.1 Experiment Definition

The following definition determines the foundation for the experiment [36]:
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 software development products.
Quality focus. The quality focus is thoroughness and fault-finding effectiveness of unit test suites, measured by code coverage and mutation score indicator, respectively.
Perspective. The perspective is from the researcher's point of view.
Context. The experiment is run using MSc students as subjects involved in the development of finance accounting system in Java.

4.2 Context Selection

The context of the experiment was the Programming in Java course, and hence the experiment was run off-line (not industrial software development) [36]. Java was a programming language and Eclipse was an Integrated Development Environment (IDE). All the subjects had prior experience, at least in C and C++ programming (using object-oriented approach). The course consisted of seven lectures and fifteen laboratory sessions (90 minutes each), and introduced Java programming language, using pair programming and test-driven development as the key XP practices. The subjects' practical skills in programming in Java, using pair programming and test-driven development, were evaluated during the first seven laboratory sessions. The experiment took place during the last eight laboratory sessions. The problem, development of the finance accounting system, was as close to a real one, as it is possible in academic environment. The requirements specification consisted of 27 user stories. The subjects participating in the study were mainly second and third-year (and few fourth and fifth-year) computer science MSc students. MSc programme of Wroclaw University of Technology is a 5-year programme after high school. The experiment was part of a research, conducted at Wroclaw University of Technology, with the aim of obtaining empirical evidence on the impact of pair programming and test-driven development on different aspects of software products and processes [34,35,37]. The experiment analysis was run with subjects involved in 63 projects conducted, using test-driven development approach, by 28 solo programmers (denoted as S) and 35 pairs (denoted as P).

4.3 Variables Selection

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

4.4 Hypotheses Formulation

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

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

4.5 Selection of Subjects

The subjects are chosen 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 a picture of the students' background. It appeared 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.

4.6 Design of the Experiment

The design is one factor (the software development method), with two treatments (S and P). The assignment to pair programming 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 the real world practice. Thus this is a quasi-experiment [38]. In the case of two solo projects questionnaires were not filled in. In the case of one solo project, tests were not written and checked-in properly. These projects were not included in the analysis. The design resulted in an unbalanced design, with 28 solo programmers and 35 pairs.

4.7 Instrumentation and Measurement

The instruments [36] and materials for the experiment were prepared in advance, and consisted of requirements specification (user stories), pre-test and post-test questionnaires, Eclipse project framework, a detailed description of software development approaches (S and P), duties of subjects, and instructions how to use the experiment infrastructure (e.g. CVS version control system). Branch coverage and mutation score indicator values were collected using Clover [39] and Judy [25] tools, respectively.

4.8 Validity Evaluation

When conducting the experiment, there is always a set of threats to the validity of the results. Cook and Campbell [40] defined *statistical conclusion*, *internal*, *construct*, and *external validity* threats. To enable an analysis of the validity of the current study, the possible threats are discussed, based on Wohlin et al. [36].

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 care taken in the implementation and measurement of the experiment [36]. 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. Measures and treatment implementation are 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. The basic principle is that when one measures a phenomenon twice, the outcome should be the same. The measures used in the experiment are considered reliable, because they can be repeated with the same outcomes.

The *internal* validity of the experiment concerns the question 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. solo programming) 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 so 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 were 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 [36]. 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. It appeared that subjects slightly favoured a pair programming approach. Thus, there seems to be no apparent contradiction between qualitative and quantitative results. 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 [41]. The process conformance threat was handled by attempting to keep deviations from occurring, with the help of assistant lecturers. They controlled how development methods were carried out and forced subjects to follow the prescribed techniques. Moreover, the subjects were informed of the importance of following proper development methods.

Threats to *external* validity are the conditions that limit our ability to generalize the results of our experiment to industrial practice. The largest threat is that the subjects were students, who had short experience in pair programming. However, Kitchenham et al.[42] 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. [43]. Moreover, Tichy argues why it is acceptable to use students as subjects [44]. The threads to external validity were reduced by making the experimental environment as realistic as possible (e.g. requirements specification came from an external client).

4.9 Experiment Operation

The experiment was run at Wroclaw University of Technology and consisted of a preparation phase and an execution phase. The preparation phase of the experiment included lectures and training exercises, given directly before the experiment, in order to improve skills and practice in the areas of pair programming, test-driven development, 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 asked of them. They had to not be overwhelmed by the complexity of, or unfamiliarity with the tasks [44]. Therefore, it took seven laboratory sessions (90 minutes each) 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. In the experiment up-to-date development environment composed of Java Development Kit, Eclipse development environment, JUnit testing framework and also CVS repository were used. 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 (tool developed at the Wroclaw University of Technology).

5 Analysis of the Experiment

The experiment data are analysed with descriptive analysis and statistical tests.

5.1 Descriptive Statistics

Descriptive statistics of gathered measures are summarized in Table 4. Columns "Mean", "Std.Deviation", "Std.Error", "Max", "Median" and "Min" state for each measure and development method the mean value, standard deviation, standard error, maximum, median and minimum, respectively.

Table 4. 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	S	.38	.22	.042	.90	.39	.00
(BC)	P	.39	.21	.036	.83	.32	.09
Mutation Score	S	.39	.22	.042	.72	.43	.04
Indicator (MSI)	P	.47	.29	.049	.98	.44	.09

The first impression is that developers working in pairs (denoted as P), and developers working solo (S) performed similarly. However, it appears that pair programming seems to have some positive impact on mutation score indicator, as there is over 20% increase in the mean value of MSI (.47 vs. .39). This difference is supported by differences in minimum and maximum values of MSI but not the median. However, it is worthwhile to mention that the mean is resistant to sampling variation, whilst the median is more likely to differ across samples. This is important, as we want to infer something about the entire population. 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 [45], we can conclude that pairs achieved slightly higher mutation score indicator ($M = .47$, $SD = .29$) than solo programmers ($M = .39$, $SD = .22$), whilst branch coverage for pairs ($M = .39$, $SD = .21$) was similar to solo programmers ($M = .38$, $SD = .22$). It is worth noting that

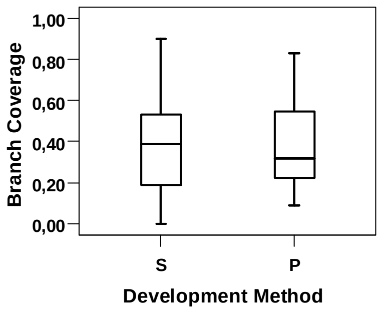

Fig. 1. Branch Coverage Boxplots

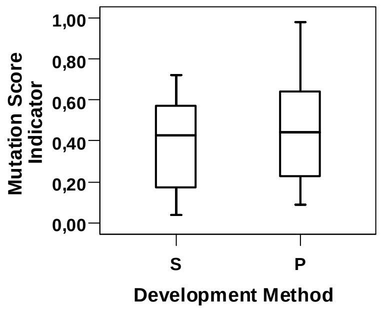

Fig. 2. Mutation Score Indicator Boxplots

mutation analysis required about 30000 mutants to be created for 63 projects. To answer the question whether the impact of pair programming on mutation score indicator and branch coverage is significant, or not, statistical tests must be performed.

5.2 Hypotheses Testing

We start from exploratory analyses on the collected data to check whether they follow the assumptions of the parametric tests (i.e. normal distribution, interval or ratio scale, homogeneity of variance). The first 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. We find that the data are not normally distributed, see Table 5.

Table 5. Tests of Normality

	Development	Kolmogorov-Smirnov[1]			Shapiro-Wilk		
	Method	Statistic	df[2]	Significance	Statistic	df[2]	Significance
Branch Coverage	S	.121	28	.200[3]	.964	28	.423
(*BC*)	P	.147	35	.053	.936	35	.043
Mutation Score	S	.113	28	.200[3]	.933	28	.072
Indicator (*MSI*)	P	.125	35	.179	.922	35	.016

[1] Lilliefors Significance Correction.
[2] Degrees of freedom.
[3] This is a lower bound of the true significance.

For the branch coverage data the distribution for pairs appears to be non-normal ($p < .05$), whereas that for solos is normal according to the Shapiro-Wilk test. It is worth noting that the Shapiro-Wilk test yields exact significance values and is thus more accurate (though less widely used) than the Kolmogorov-Smirnov test. For the mutation score indicator data results are similar. The Shapiro-Wilk test is in fact significant for pairs but not for solos. This finding alerts us to the fact that a non-parametric test should be used. Therefore the hypotheses from section 4.4 are evaluated using 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 6. Mann-Whitney Test Statistics (grouping variable: Development Method)

	Branch Coverage (BC)	Mutation Score Indicator (MSI)
Mann-Whitney U	471.500	423.000
Wilcoxon W	877.500	829.000
Z	-.256	-.927
Asymp. Sig. (1-tailed)	.399	.177

Table 6 shows test statistics and significances. It appeared that branch coverage was not significantly affected by pair programming approach (the Mann-Whitney test statistics: $U = 471.5$, non-significant, $z = -.26$). Mutation score indicator was not significantly affected by pair programming approach (the Mann-Whitney test statistics: $U = 423.0$, non-significant, $z = -.93$), either. An effect size ($r = \frac{Z}{\sqrt{N}}$ where Z is the z-score in Table 6, 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 rather small, for mutation score indicator ($r = -.12$). The later result may suggest the need for further experimentation.

Why did not pair programming result in a significant increase of testing thoroughness or fault-finding effectiveness, measured by branch coverage and mutation score indicator, respectively? The plausible explanation is that when software project is not big enough, and the requirements are decomposed into small features (user stories), the impact of pair programming 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.

Another possible explanation is that when the scope of the project is limited, the impact of pair programming practice on branch coverage and mutation score indicator may be insignificant, because of the limited number of tests.

6 Summary and Conclusions

The unique aspect of an experiment conducted at Wroclaw University of Technology was that it included the first ever assessment of the impact of pair programming on thoroughness and fault-finding 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. It appeared that the pair programming practice used by the subjects, instead of solo programming, did not significantly affect branch coverage ($U = 471.5$, non-significant, $r = -.03$), or mutation score indicator ($U = 423.0$, non-significant, $r = -.12$). It means that the impact of pair programming on thoroughness and fault-finding effectiveness of unit test suites was not confirmed. 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 a larger project, while securing a sample of large enough size to guarantee a high-power design. Further experimentation in other contexts (e.g. in industry, on larger projects) is needed to establish evidence-based recommendations for the impact of pair programming practice on thoroughness and effectiveness of test suites.

7 Acknowledgements

The author expresses his gratitude to the students and lecturers participating in the experiment, the members of the e-Informatyka team and Norbert Radyk 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 (2002)
2. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change. 2nd edn. Addison-Wesley (2004)
3. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the Case for Pair Programming. IEEE Software **17**(4) (2000) 19–25
4. Williams, L.A., Kessler, R.R.: All I really need to know about pair programming I learned in kindergarten. Communications of the ACM **43**(5) (2000) 108–114
5. Arisholm, E., Gallis, H., Dybå, T., Sjøberg, D.I.K.: Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. IEEE Transactions on Software Engineering **33**(2) (2007) 65–86
6. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
7. Kobayashi, O., Kawabata, M., Sakai, M., Parkinson, E.: Analysis of the Interaction between Practices for Introducing XP Effectively. In: ICSE '06: Proceeding of the 28th International Conference on Software Engineering, New York, NY, USA, ACM Press (2006) 544–550
8. Marick, B.: How to Misuse Code Coverage. In: Proceedings of the 16th International Conference on Testing Computer Software. (1999) http://www.testing.com/writings/coverage.pdf.
9. Kaner, C.: Software Negligence and Testing Coverage. In: STAR 96: Proceedings the 5th International Conference, Software Testing, Analysis and Review. (1996) 299–327

10. Cai, X., Lyu, M.R.: The Effect of Code Coverage on Fault Detection under Different Testing Profiles. SIGSOFT Softw. Eng. Notes **30**(4) (2005) 1–7
11. Cornett, S.: Code Coverage Analysis (Retrieved 2006) http://www.bullseye.com/coverage.html.
12. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer **11**(4) (1978) 34–41
13. Hamlet, R.G.: Testing Programs with the Aid of a Compiler. IEEE Transactions on Software Engineering **3**(4) (1977) 279–290
14. Offutt, A.J., Untch, R.H.: Mutation 2000: Uniting the Orthogonal. In: Mutation testing for the new century. Kluwer Academic Publishers, Norwell, MA, USA (2001) 34–44
15. Zhu, H., Hall, P.A.V., May, J.H.R.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys **29**(4) (1997) 366–427
16. Walsh, P.J.: A Measure of Test Case Completeness. PhD thesis, Univ. New York (1985)
17. Frankl, P.G., Weiss, S.N., Hu, C.: All-Uses vs Mutation Testing: An Experimental Comparison of Effectiveness. Journal of Systems and Software **38**(3) (1997) 235–253
18. Offutt, A.J., Pan, J., Tewary, K., Zhang, T.: An Experimental Evaluation of Data Flow and Mutation Testing. Software Practice and Experience **26**(2) (1996) 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 Marchesi, M., Succi, G., eds.: XP 2001: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering. (2001) 84–87
21. Gamma, E., Beck, K.: JUnit Project Home Page (Retrieved 2006) http://www.junit.org/.
22. Offutt, J., Ma, Y.S., Kwon, Y.R.: An Experimental Mutation System for Java. SIGSOFT Software Engineering Notes **29**(5) (2004) 1–4
23. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: A Mutation System for Java. In: ICSE '06: Proceeding of the 28th International Conference on Software Engineering, New York, NY, USA, ACM Press (2006) 827–830
24. Chevalley, P., Thévenod-Fosse, P.: A mutation analysis tool for Java programs. International Journal on Software Tools for Technology Transfer (STTT) **5**(1) (2003) 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, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An Experimental Determination of Sufficient Mutant Operators. ACM Transactions on Software Engineering and Methodology **5**(2) (1996) 99–118
27. Ammann, P., Offutt, J.: Introduction to Software Testing. (2008) In progress.
28. Ma, Y.S., Harrold, M.J., Kwon, Y.R.: Evaluation of Mutation Testing for Object-Oriented Programs. In: ICSE '06: Proceeding of the 28th International Conference on Software Engineering, New York, NY, USA, ACM Press (2006) 869–872
29. Nosek, J.T.: The Case for Collaborative Programming. Communications of the ACM **41**(3) (1998) 105–108
30. Nawrocki, J.R., Wojciechowski, A.: Experimental Evaluation of Pair Programming. In: ESCOM '01: European Software Control and Metrics. (2001) 269–276

31. Nawrocki, J.R., Jasiński, M., Olek, L., Lange, B.: Pair Programming vs. Side-by-Side Programming. In Richardson, I., Abrahamsson, P., Messnarz, R., eds.: EuroSPI. Volume 3792 of Lecture Notes in Computer Science., Springer (2005) 28–38
32. Hulkko, H., Abrahamsson, P.: A Multiple Case Study on the Impact of Pair Programming on Product Quality. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, New York, NY, USA, ACM Press (2005) 495–504
33. Müller, M.M.: Are Reviews an Alternative to Pair Programming? Empirical Software Engineering **9**(4) (2004) 335–351
34. Madeyski, L.: Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality. In Zieliński, K., Szmuc, T., eds.: Software Engineering: Evolution and Emerging Technologies. Volume 130 of Frontiers in Artificial Intelligence and Applications. IOS Press (2005) 113–123
35. Madeyski, L.: The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design - An Experiment. In Münch, J., Vierimaa, M., eds.: Product Focused Software Process Improvement. Volume 4034 of Lecture Notes in Computer Science., Springer (2006) 278–289
36. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA (2000)
37. Madeyski, L.: Is External Code Quality Correlated with Programming Experience or Feelgood Factor? In Abrahamsson, P., Marchesi, M., Succi, G., eds.: Extreme Programming and Agile Processes in Software Engineering. Volume 4044 of Lecture Notes in Computer Science., Springer (2006) 65–74
38. Shadish, W.R., Cook, T.D., Campbell, D.T.: Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Houghton Mifflin (2002)
39. Cenqua Pty Ltd: Clover project (Retrieved 2006) http://www.cenqua.com/clover/.
40. Cook, T.D., Campbell, D.T.: Quasi-Experimentation: Design and Analysis Issues. Houghton Mifflin Company (1979)
41. Sørumgård, L.S.: Verification of Process Conformance in Empirical Studies of Software Development. PhD thesis, The Norwegian University of Science and Technology (1997)
42. Kitchenham, B., Pfleeger, S.L., Pickard, L., Jones, P., Hoaglin, D.C., Emam, K.E., Rosenberg, J.: Preliminary Guidelines for Empirical Research in Software Engineering. IEEE Transactions on Software Engineering **28**(8) (2002) 721–734
43. 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 **5**(3) (2000) 201–214
44. Tichy, W.F.: Hints for Reviewing Empirical Work in Software Engineering. Empirical Software Engineering **5**(4) (2000) 309–312
45. American Psychological Association: Publication manual of the American Psychological Association. (2001)