This is a preprint of an article: Lech Madeyski, "The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design — An Experiment", in Product Focused Software Process Improvement, ser. Lecture Notes in Computer Science, J. Münch and M. Vierimaa, Eds., LNCS 4034. Springer, 2006, pp. 278-289
http://dx.doi.org/10.1007/11767718_24

The Impact of Pair Programming and Test-Driven Development on Package Dependencies in Object-Oriented Design — An Experiment

Lech Madeyski

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

Abstract. Background: Test-driven development (TDD) and pair programming are software development practices popularized by eXtreme Programming methodology. The aim of the practices is to improve software quality.
Objective: Provide an empirical evidence of the impact of both practices on package dependencies playing a role of package level design quality indicators.
Method: An experiment with a hundred and eighty eight MSc students from Wroclaw University of Technology, who developed finance-accounting system in different ways (CS — classic solo, TS — TDD solo, CP — classic pairs, TP — TDD pairs).
Results: It appeared that package level design quality indicators (namely package dependencies in an object-oriented design) were not significantly affected by development method.
Limitations: Generalization of the results is limited due to the fact that MSc students participated in the study.
Conclusions: Previous research revealed that using test-driven development instead of classic (test-last) testing approach had statistically significant positive impact on some class level software quality indicators (namely CBO and RFC metrics) in case of solo programmers as well as pairs. Combined results suggest that the positive impact of test-driven development on software quality may be limited to class level.

1 Introduction

Test-driven development (TDD) [1] and pair programming (PP) [2] have recently gained a lot of attention as the key software development practices of eXtreme Programming (XP) methodology [3]. The main idea of test-driven development is that programmers write tests before production code. Pair programming is software development practice where two programmers work together, collaborating on the same development tasks. The basic aim of both practices, described

in section 3.5, is to improve software quality. The question is whether both practices (used separately or together) really improve software quality.

Researchers and practitioners have reported numerous, often anecdotal and favourable studies of XP practices and methodology. Empirical studies on pair programming often concern productivity [4,5,6,7,8]. A few studies have focused on pair programming or test-driven development as practices to remove defects [5,6,9,10], influence external code quality (measured by the number of functional, blackbox test cases passed) [11,12,13] or reliability of programs (a fraction of the number of passed tests divided by the number of all tests) [14,15,16]. Janzen [17] has pointed out that there was no research on the broader efficacy of test-driven development, nor on its effects on internal design quality outside a small pilot study [18]. Recently, Madeyski [19] pointed out that using test-driven development instead of classic (test-last) development had significant positive impact on two Chidamber and Kemerer (CK) [20] class level software quality indicators — Response For a Class (RFC) and Coupling Between Object classes (CBO). Obtained results did not support similar, positive impact of pair programming practice [19]. Hulkko and Abrahamsson [21] also suggested that pair programming might not necessarily provide as extensive quality benefits as suggested in literature. The key findings from empirical studies concerning software quality are summarized below in table 1.

Table 1. Pair programming and test-driven development literature review.
Abbreviations: S(Solo programmers), P(Pairs), x4(groups of four), T(TDD), C(Classic)

Study	Environment	Subjects	Key findings
PP studies:			
[5,6]	Academic	41(14P/13S)	P had 15% less code defects than S
[15,16]	Academic	37(10P/17S)	P did not produce more reliable code
[21]	Acad./Ind.	4x(4–6)	P did not provide extensive quality benefits
TDD studies:			
[14]	Academic	19(9CS/10TS)	T did not produce more reliable code
[11,12]	Industrial	24(6CP/6TP)	TP products passed 18% more tests than CP
[9,10]	Industrial	13(5CS/9TS)	Minimal/no difference in *LOC* per person-month T reduced defect rate by 40–50%
[18]	Academic	8(1Cx4/1Tx4)	No meaningful differences in package dependencies between T and C project
Combined study:			
[13,19]	Academic	188 (28CS/28TS/ 31CP/35TP)	TS passed significantly less acc. tests than CS TP passed significantly less acc. tests than CP No difference between CS and CP as well as TS and TP in $NATP$ (Number of Acc.Tests Passed) T had significant positive impact on RFC and CBO CK metrics in case of S and P

In spite of a wide range of empirical studies there is still limited evidence concerning the impact of pair programming and test-driven development on quality

of an object-oriented design in terms of dependencies between packages (collections of related classes), which in turn may have impact on external qualities e.g. fault-proneness or maintainability. The aim of this paper is to fill in this gap.

An experiment, performed in 2004 at Wroclaw University of Technology, was aiming to investigate the impact of test-driven development and pair programming practices on different aspects of software development. One of the interesting results of the experiment is that using test-driven development instead of classic testing approach has statistically significant positive impact on class level software quality indicators (RFC and CBO) in case of solo as well as pair programming [19]. The interesting research question, investigated in this paper, is whether the positive impact of test-driven development on software quality is limited to the class level. It is important question because test-driven development practice (also known by names such as, test-first design and test driven design) is considered not only one of the core programming practices of XP but also one that we use instead of writing detailed design specifications [22]. Practitioners emphasize that test-driven development is primarily a method of designing software, not just a method of testing [23] and that pair programming tend to come up with higher quality designs [24].

The quality of an object-oriented design is strongly influenced by a system's package relationships. Loosely coupled and highly cohesive packages are qualities of good design. Therefore to investigate the impact of test-driven development and pair programming on object-oriented design we used Martin's package level dependency metrics [25,26] that can be used to measure the quality of an object-oriented design in terms of the interdependences between the packages of that design. Designs which are highly interdependent tend to be rigid, unreusable and hard to maintain [25].

Martin's metrics, investigated in this study and measured by our tool [27], are defined as follows [25]:

- Ca (Afferent Couplings) — The number of classes outside the package that depend upon classes within the package.
- Ce (Efferent Couplings) — The number of classes inside the package that depend upon classes outside the package.
- I (Instability) — The ratio $(Ce/(Ca+Ce))$ of efferent coupling (Ce) to total coupling $(Ce+Ca)$. This metric is an indicator of the package's resilience to change and has the range $[0, 1]$. $I = 0$ indicates a maximally stable package. $I = 1$ indicates a maximally instable package.
- A (Abstractness) — The ratio of the number of abstract classes to the total number of classes in package. This metric range is $[0, 1]$. 0 means concrete package and 1 means completely abstract package.
- Dn (Normalized Distance from Main Sequence) — This is the normalized perpendicular distance of the package from the idealized line $A + I = 1$. This metric is an indicator of the package's balance between abstractness and stability. Dn metric's results are within a range of $[0, 1]$. A value of zero indicates perfect package design.

Underlying theory about a relationship between the object-oriented metrics and fault-proneness as well as maintainability due to the effect on cognitive complexity has been provided in [28] and [29].

2 Problem Statement

The following definition determines a foundation for the experiment [30]:
Object of study. The objects of study are software development products — developed code.
Purpose. The purpose is to evaluate the impact of test-driven development and pair programming practices on software development products.
Quality focus. The quality focus is the object-oriented design quality in terms of the interdependences between packages of that design.
Perspective. The perspective is from the researcher's point of view.
Context. The experiment is run using MSc students as subjects involved in finance-accounting system development.

Summary: The analysis of *the developed code* for the purpose of *evaluation of the test-driven development and pair programming practices impact on the developed code* with respect to *interdependences between packages* from the point of view of *the researcher* in the context of *finance-accounting system development performed by MSc students.*

3 Experiment Planning

The planning phase of the experiment can be divided into seven steps [30]: context selection, hypotheses formulation, variables selection, selection of subjects, experiment design, instrumentation and validity evaluation.

3.1 Context Selection

The context of the experiment was the Programming in Java (PIJ) course, and hence the experiment was run off-line [30]. Java was the programming language, Eclipse 3.0 was the IDE (Integrated Development Environment). All subjects had prior experience at least in C and C++ programming (using object-oriented approach). The PIJ course consisted of seven lectures (90 minutes per each) and fifteen laboratory sessions (also 90 minutes per each). The course introduced Java programming language using test-driven development and pair programming 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 close to the real one (not toy-size). 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 of Wroclaw University of Technology. In total 188 students were involved

in the experiment, see table 2. A few people were involved in the experiment planning, operation and analysis.

3.2 Quantifiable Hypotheses Formulation

The crucial aspect of the experiment is to know and formally state what we intend to evaluate in the experiment. This leads us to the formulation of the following quantifiable hypotheses to be tested:

- $H_{0\ X,\ CS/TS/CP/TP}$ — There is no difference in the mean value of X metric (where X is Ca, Ce, I, A or Dn) between the software development projects using any combination of classic (test-last) / TDD (test-first) testing approach and solo / pair programming development method (CS, TS, CP and TP are used to denote development methods).
- $H_{A\ X,\ CS/TS/CP/TP}$ — There is a difference in the mean value of X metric between the software development projects using any combination of classic (test-last) / TDD (test-first) testing approach and solo / pair programming development method.

If we reject null hypotheses $H_{0\ X,\ CS/TS/CP/TP}$ (where X is Ca, Ce, I, A or Dn) we can try to investigate more specific hypotheses concerning differences between development methods (CS vs. TS, CP vs. TP, CS vs. CP, and TS vs. TP).

3.3 Variables Selection

The independent variable is the software development method used (CS, TS, CP or TP). The dependent (response) variables are mean values of Ca, Ce, I, A and Dn (denoted as M_X where X is Ca, Ce, I, A or Dn).

3.4 Selection of Subjects

The subjects are chosen based on convenience — the subjects are students taking the PIJ course. Prior to the experiment, the students filled in a pre-test questionnaire. The aim of the questionnaire was to get a description of the students' background, see table 2 for sample results. The ability to generalize from this context is further elaborated when discussing threats to the experiment.

3.5 Design of the Experiment

The design is one factor (the software development method) with four treatments (alternatives):

- Solo programming using classic testing approach — tests after implementation (CS).
- Solo programming using test-driven development (TS).

Table 2. The context of the experiment

Context factor	ALL	CS	TS	CP	TP
Number of MSc students:	188	28	28	62	70
– on the 2nd year	108	13	16	40	39
– on the 3rd year	68	12	11	18	27
– on the 4th year	10	3	0	3	4
– on the 5th year	2	0	1	1	0
– with industry experience	33	4	6	8	15
Mean value of:					
– Programming experience in years	3.8	4.1	3.7	3.6	3.9
– Java experience in months	3.9	7.1	2.8	3.4	3.5
– Another OO language experience in months	20.5	21.8	20.9	19.2	21.1

- Pair programming using classic testing approach — tests after implementation (CP).
- Pair programming using test-driven development (TP).

Pair programming is a practice in which two programmers (called the driver and navigator) work together at one computer, collaborating on the same development tasks (e.g. design, test, code). The driver, is typing at the computer or writing down a design. The navigator observes the work of the driver, reviews the code, proposes test cases and considers the implementations strategic implications [5,31]. In case of solo programming all activities are performed by one programmer.

Test-driven development is a practice based on specifying piece of functionality as a low level test before writing production code, implementing the functionality so that the test passes, refactoring (e.g. removing duplication) and iterating the process. Tests are run frequently, while writing production code. In case of classic (test-last) development tests are specified after writing production code and less frequently [32].

The assignment of subjects to groups was performed first by stratifying the subjects with respect to their skill level, measured by graders, and then assigning them randomly to test-driven development or classic testing approach treatment groups. However the assignment to solo or pair programming teams took into account the people preferences (as it seemed to be more natural and close to agile software development practice).

Students who did not complete the experiment were removed from the analysis. Sixteen teams dropped out, did not check in the final version of their program or did not fill in questionnaires. Therefore, we retained data from 122 teams. The design resulted in an unbalanced design, with 28 solo programmers and 31 pairs using classic testing approach, 28 solo programmers and 35 pairs using test-driven development practice.

3.6 Instrumentation

The instrumentation of the experiment consisted of requirements specification (user stories), pre-test and post-test questionnaires, Eclipse project framework, detailed description of software development methods (CS, TS, CP, TP) and duties of subjects, instructions how to use the experiment infrastructure (e.g. CVS Version Management System) and examples (e.g. sample source code of applications developed using TDD approach and JUnit tests). Martin's metrics were collected using aopmetrics tool [27] developed and supported by members of e-Informatyka development team at Wroclaw University of Technology.

3.7 Validity Evaluation

The fundamental question concerning results of each experiment is how valid the results are. When conducting the experiment, there is always a set of threats to the validity of the results. Shadish, Cook and Campbell [33] defined four types of threats: *statistical conclusion*, *internal*, *construct* and *external validity*.

Threats to the *statistical conclusion* validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of the experiment. Threats to the *statistical conclusion* validity are considered to be under control. Robust statistical techniques, tools (e.g. Statistica) and large sample sizes to increase statistical power are used. Measures and treatment implementation are considered reliable. However, the risk in the treatment implementation is that the experiment was spread across laboratory sessions. To avoid the risk, access to the CVS repository was restricted to the specific laboratory sessions (access hours and IP addresses). Validity of the experiment is highly dependent on the reliability of the measures. The basic principle is that when you measure 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.

Threats to the *internal* validity are influences that can affect the independent variable with respect to causality, without the researcher's knowledge. Concerning the *internal* validity, the risk of rivalry between groups must be considered. The group using the traditional method may do their very best to show that the old method is competitive. On the other hand, subjects receiving less desirable treatments may not perform 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. Possible diffusion or imitation of treatments were under control of the graders.

Construct validity concerns generalizing the results of the experiment to the concepts behind the experiment. Threats to the *construct* validity are not considered very harmful. Inadequate explication of constructs does not seem to be the threat as the constructs were defined, before they were translated into measures or treatments. The mono-operation bias is a threat as the experiment was conducted on a single software development project; however, the size of

the project was not a toy-size. Using a single type of measure would be a mono-method bias threat; however, different measures were used in the experiment.

Threats to *external* validity are conditions that limit our ability to generalize the results of our experiment to industrial practice. The largest threat is that students (who had short experience in pair programming and test-driven development) were used as subjects. However, Kitchenham et al. [34] state that students are the next generation of software professionals, so, are relatively close to the population of interest. In summary, the threats are not regarded as being critical.

4 Experiment Operation

The experiment was run at Wroclaw University of Technology in 2004 during eight laboratory sessions. The data was primarily collected by automated experiment infrastructure. Additionally, the subjects filled in pre-test and post-test questionnaires, primarily to evaluate their experience. The package for the experiment was prepared in advance and is described in section 3.6.

5 Analysis of the Experiment

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

5.1 Descriptive Statistics

Descriptive statistics of gathered Martin's metrics are summarized in table 3. Columns "Mean", "StdDev", "Max", "Median" and "Min" state for each metric and development method ("DevMeth") the mean value, standard deviation, maximum, median, minimum, respectively.

The first impression is that development methods performed similarly. Results shown in table 3 also indicate imperfect package design (e.g. values of normalized distance from main sequence are close to 1), no matter which development method was used.

5.2 Hypotheses Testing

Experimental data are analysed using models that relate the dependent variable to the factor under consideration. The use of these models involves making assumptions concerning the data that need to be validated. Therefore we run some exploratory analysis on the collected data to check whether they follow the assumptions of the parametric tests:

- Normal distribution — the collected data come from a population that has a normal distribution.
- Interval or ratio scale — the collected data must be measured at an interval or ratio level (since parametric tests work on the arithmetic mean).

Table 3. Descriptive statistics of Martin's metrics

Metric	DevMeth	Mean	StdDev	Max	Median	Min
Ca	CS	.46	1.12	4.50	0	0
	TS	.20	.72	2.75	0	0
	CP	.31	.98	3.60	0	0
	TP	.11	.50	2.80	0	0
Ce	CS	.24	.53	1.67	0	0
	TS	.17	.53	2.25	0	0
	CP	.15	.49	2.00	0	0
	TP	.07	.31	1.60	0	0
I	CS	.08	.17	.54	0	0
	TS	.07	.22	1.00	0	0
	CP	.03	.10	.34	0	0
	TP	.03	.12	.50	0	0
A	CS	.00	.02	.08	0	0
	TS	.01	.02	.08	0	0
	CP	.01	.03	.17	0	0
	TP	.00	.02	.09	0	0
Dn	CS	.92	.18	1.00	1.00	.42
	TS	.92	.22	1.00	1.00	0
	CP	.96	.10	1.00	1.00	.66
	TP	.97	.12	1.00	1.00	.50

Table 4. Tests of Normality

Metric	DevMeth	Kolmogorov-Smirnov[1]			Shapiro-Wilk		
		Statistic	df[2]	Significance	Statistic	df[2]	Significance
M_{Ca}	CS	.480	28	.000	.481	28	.000
	TS	.536	28	.000	.287	28	.000
	CP	.529	31	.000	.350	31	.000
	TP	.529	35	.000	.232	35	.000
M_{Ce}	CS	.494	28	.000	.495	28	.000
	TS	.519	28	.000	.370	28	.000
	CP	.527	31	.000	.353	31	.000
	TP	.536	35	.000	.254	35	.000
M_I	CS	.496	28	.000	.494	28	.000
	TS	.517	28	.000	.366	28	.000
	CP	.529	31	.000	.348	31	.000
	TP	.539	35	.000	.251	35	.000
M_A	CS	.509	28	.000	.342	28	.000
	TS	.535	28	.000	.295	28	.000
	CP	.539	31	.000	.176	31	.000
	TP	.539	35	.000	.161	35	.000
M_{Dn}	CS	.466	28	.000	.521	28	.000
	TS	.455	28	.000	.400	28	.000
	CP	.514	31	.000	.409	31	.000
	TP	.518	35	.000	.284	35	.000

– Homogeneity of variance — roughly the same variances between groups or treatments (as we use different subjects).

We find that — according to the Kolmogorov-Smirnov and Shaprio-Wilk statistic (see table 4) — the data are not normally distributed. This finding alerts us to the fact that a nonparametric test should be used.

Hypotheses $H_{0\ X,\ CS/TS/CP/TP}$ (where X is Ca, Ce, I, A or Dn) are evaluated using the Kruskal-Wallis one way analysis of variance by ranks. The Kruskal-Wallis test is used for testing differences between the four experimental groups (CS, TS, CP, TP) when different subjects are used in each group. Table 5 shows test statistics and significances.

Table 5. Kruskal-Wallis Test Statistics — grouping variable: DevMeth

	M_{Ca}	M_{Ce}	M_I	M_A	M_{Dn}
Chi-Square	2.917	2.323	2.402	2.039	2.420
Asymp. Significance	.405	.508	.493	.564	.490

We can conclude that the software development method used by the subjects do not significantly affected interdependencies between the packages.

6 Summary and Conclusions

It appeared that package level design quality indicators (namely package dependencies in an object-oriented design) were not significantly affected by development method. Using test-driven development instead of classic (test-last) testing approach as well as pair programming instead of solo programming had not significant impact on package dependencies. Previous research revealed that using test-driven development instead of classic testing approach had statistically significant positive impact on some class level software quality indicators (namely CBO and RFC) in case of solo as well as pair programming [19]. Combined results suggest that the positive impact of test-driven development on software quality may be limited to class level. Therefore software engineers and academics may benefit from using test-driven development but they should take care of package level design issues. Further research is needed to replicate the study, to evaluate the impact in other contexts (e.g. in industry) as well as on other package level software quality indicators and to establish evidence.

7 Acknowledgments

The author would like to thank the students for participating in the investigation, the graders and the members of the e-Informatyka team (Michał Stochmiałek,

[1] Lilliefors Significance Correction.
[2] Degrees of freedom.

Wojciech Gdela, Tomasz Poradowski, Jacek Owocki, Grzegorz Makosa, Mariusz Sadal) for their help during development of the measurement infrastructure (e.g. aopmetrics tool [27]).

References

1. Beck, K.: Test Driven Development: By Example. Addison-Wesley (2002)
2. Williams, L., Kessler, R.: Pair Programming Illuminated. Addison-Wesley (2002)
3. Beck, K.: Extreme Programming Explained: Embrace Change. 2nd edn. Addison-Wesley (2004)
4. Nosek, J.T.: The case for collaborative programming. Communications of the ACM **41**(3) (1998) 105–108
5. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the case for pair programming. IEEE Software **17**(4) (2000) 19–25
6. Williams, L.: The Collaborative Software Process. PhD thesis, University of Utah (2000)
7. Nawrocki, J.R., Wojciechowski, A.: Experimental evaluation of pair programming. In: ESCOM '01: European Software Control and Metrics. (2001) 269–276
8. 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
9. Williams, L., Maximilien, E.M., Vouk, M.: Test-Driven Development as a Defect-Reduction Practice. In: ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering, Washington, DC, USA, IEEE Computer Society (2003) 34–48
10. Maximilien, E.M., Williams, L.A.: Assessing Test-Driven Development at IBM. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society (2003) 564–569
11. George, B., Williams, L.A.: An Initial Investigation of Test Driven Development in Industry. In: SAC '03: Proceedings of the 2003 ACM Symposium on Applied Computing, ACM (2003) 1135–1139
12. George, B., Williams, L.A.: A structured experiment of test-driven development. Information and Software Technology **46**(5) (2004) 337–342
13. 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
14. Müller, M.M., Hagner, O.: Experiment about test-first programming. IEE Proceedings - Software **149**(5) (2002) 131–136
15. Müller, M.M.: Are Reviews an Alternative to Pair Programming? In: EASE '03: Conference on Empirical Assessment In Software Engineering. (2003)
16. Müller, M.M.: Are Reviews an Alternative to Pair Programming? Empirical Software Engineering **9**(4) (2004) 335–351
17. Janzen, D.S.: Software Architecture Improvement through Test-Driven Development. In: OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 222–223

18. Kaufmann, R., Janzen, D.: Implications of Test-Driven Development: A Pilot Study. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 298–299
19. Madeyski, L.: An empirical analysis of the impact of pair programming and test-driven development on CK design complexity metrics. Technical Report PRE I31/05/P-004, Institute of Applied Informatics, Wroclaw University of Technology (2005)
20. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering **20**(6) (1994) 476–493
21. 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
22. Object Mentor, Inc.: Test Driven Development (2005) http://www.objectmentor.com/writeUps/TestDrivenDevelopment.
23. Wikipedia, the free encyclopedia: Test-driven development (2005) http://en.wikipedia.org/wiki/Test_driven_development.
24. Wikipedia, the free encyclopedia: Pair programming (2005) http://en.wikipedia.org/wiki/Pair_programming.
25. Martin, R.C.: OO Design Quality Metrics, An Analysis of Dependencies (1994)
26. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall (2004)
27. Wroclaw University of Technology, e-Informatyka and Tigris developers: aopmetrics project (2005) http://aopmetrics.tigris.org/.
28. Briand, L.C., Wüst, J., Ikonomovski, S.V., Lounis, H.: Investigating quality factors in object-oriented designs: an industrial case study. In: ICSE '99: Proceedings of the 21st International Conference on Software Engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 345–354
29. Emam, K.E., Melo, W.L., Machado, J.C.: The Prediction of Faulty Classes Using Object-Oriented Design Metrics. Journal of Systems and Software **56**(1) (2001) 63–75
30. 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)
31. Williams, L.A., Kessler, R.R.: All I really need to know about pair programming I learned in kindergarten. Commun. ACM **43**(5) (2000) 108–114
32. Erdogmus, H., Morisio, M., Torchiano, M.: On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering **31**(3) (2005) 226–237
33. Shadish, W.R., Cook, T.D., Campbell, D.T.: Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Houghton Mifflin (2002)
34. 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