

Controlled Experiments on Pair Programming: Making Sense of Heterogeneous Results

Reidar Conradi^a, Muhammad Ali Babar^b

^a Norwegian University of Science and Technology, Trondheim, Norway

^b IT University of Copenhagen, Denmark

reidar.conradi@idi.ntnu.no¹, malibaba@itu.dk²

Summary ¹

Recently, several controlled experiments on adoption and use of pair programming (PP) have been reported. Most of these experiments have been conducted by constructing small-scale programs of a few hundred lines of code, using correctness, duration, and effort metrics. However, none of these experiments is a replication of a previous one; and there are significant differences in contextual factors such as local goals, tasks (“treatments”), subject selection and pairing, defined metrics, and collected measures. Hence, it is very difficult to compare, assess, and generalize the results from such experiments. We illustrate this situation by comparing metrics and measures from two well-known PP experiments. We also discuss a published, formal meta-analysis of 18 PP experiments, including the two compared in this study, and which found PP effects of 10-15 % on key factors like program correctness. We then show how the author of the second PP experiment failed in applying large-scale, commercial defect rates and costs on the small-scale software systems from the author’s PP experiment. We finally argue that: 1) there should be more cooperation and standardization between experimental researchers that investigate the quantitative effects of PP e.g. on program correctness; 2) researchers should to a larger degree apply qualitative methods to study the social and cognitive impact of PP on teamwork, and how professional teams acquire and share knowledge to develop and maintain quality software. Such investigations will require longitudinal case studies in commercial settings, which cannot be achieved through experiments of short duration in academic environments.

1. Introduction

Software Engineering applies a spectrum of research methods with varying rigor and relevance – from controlled experiments to multi-case studies. There has been a substantial industrial uptake of agile approaches and methods during the last decade [4]. A rather popular one is Pair

¹ This paper was presented at the NIK-2010 conference. For more information, see [//www.nik.no](http://www.nik.no).

Programming (PP), being one of the twelve eXtreme Programming (XP) practices. There has similarly been an increase in research efforts (e.g., [26] and [1]) that investigate the effects of PP, typically by controlled experiments with both professionals and students. Most of such *primary* studies of PP involve small tasks to develop or maintain miniature software systems. The *dependent* variables are typically productivity (code size, effort, duration), quality (defect density), or aspects of social behaviour (team spirit, personal motivation, and dissemination of skills). The *independent* variables are typically the given software tasks (requirements from a treatment specification), paired vs. single-person execution, pair/person selection and allocation, maximum duration, and participants' background (age, education, affiliation, knowledge, and skills). *Contextual* or stable background variables are the actual programming environment (language and tools), and work environment (work hours, communication channels, and social organization).

Recent studies (see ons 2-4) reveal that PP typically contributes 10-15% to improved software correctness and increased development speed, but causes a similar decrease in productivity (number of written lines-of-code per person-hour – called LOC per p-h) or in total development effort (spent person-hours, p-hs). However, the different local aims, metrics, measures and logistics of these studies lead to rather heterogeneous results and severe methodological problems in comparing and interpreting the collected data across reported studies, see for instance [19-20]. Hence, it is not clear if we can come up with a contextual cost/benefit model of PP's influence on software development. Such trade-off models use aggregated evidence that can help practitioners to compose effective teams for industrial-scale PP.

There have been some efforts to perform *secondary* studies (with informal comparisons or formal meta-analysis) of primary studies of agile methods, including PP [8, 12]. However, the coverage and quality of the published primary studies of agile approaches still need considerable improvement [8]. Hence, the current PP experiments may prove insufficient to synthesize a reliable body of evidence for such meta-analysis. Given the Goal Question Metrics principle of lean metrics in empirical work [3], it should not come as a surprise that we cannot come *after* the completion of a study and demand different or possibly supplementary data. However, we can try to agree upon a minimal common metrics for a few key variables (see Section 5).

Furthermore, PP involves interesting social and cognitive aspects of teamwork – such as work satisfaction, team spirit, knowledge dissemination, and experience-driven learning. Investigations of such issues will need longitudinal and multi-case studies in industrial settings, not “one-shot” experiments with miniature systems in academia.

Based on the reasons behind Evidence-Based Software Engineering (EBSE) [17] and by looking through available Structured Literature Review papers (meta-studies) dealing with PP [8, 12], we have formulated the following two *research questions* and one *research challenge* for our modest meta-study:

- **RQ1.** How to assess and compare the effects of different primary studies of PP? This involves both PP-specific comparisons between actual results (Section 2), and *horizontal* analysis to put the small-scale PP results in a broader lifecycle context (Section 4).
- **RQ2.** What common metrics could possibly be applied in the primary studies on PP (Section 3 and 5).
- **RChall.** How to study and promote the social-cognitive aspects of PP – beyond that PP is “fun”?

2. Comparing Two PP Experiments: the Simula and Utah ones

We have looked through recently published literature reviews [8, 12] on agile approaches to find suitable studies for PP-specific comparisons. We have selected *two of the most well-known PP experiments*. The first one was conducted by Arisholm et al. [1-2] at the Simula Research Laboratory in Oslo, and the second one by Williams and her team then at The University of Utah in Salt Lake City [26]. In Section 4 we will also discuss the horizontal dimension, by the attempt of Erdogmus [10] to add commercial defect rates and costs onto the Williams' data. Both PP studies intended to measure the effect of paired vs. solo teams by varying several factors, as shown in Table 1 through Table 4.

Table 1. General information.

General issues	Simula Research Lab.	Univ. of Utah
<i>Hypotheses</i>	Assess PP effect on software maintenance wrt. duration, effort, and correctness (defined as number of similar teams with zero remaining defects) – by varying task complexity, programming expertise, and team size (pair / soloist).	Same, plus code size (LOC) but not task complexity. And only development, no maintenance.
<i>Treatment</i>	T0: Briefing, Questionnaire, and Java Programming Environment try-outs, but no practical PP training (“pair yelling”). T1: A pre-test to modify ATM software, with 7 classes and 358 LOC. T2-T4: 3 incremental changes on same evolving source in two architectural variants: simple / complex. T5: a final task; not considered.	T0: Briefing, a bit practical PP training. T1-T4: Develop 4 (non-explained) independent tasks from scratch. 4 th task had incomplete data.
<i>Subjects</i>	295 extra-paid professionals in 29 companies in 3 countries in phase2; 99 professionals in phase1.	41 senior (4 th year) students, many with industrial experience; as part of a sw.eng. course.
<i>Test suite of test cases etc.</i>	Pre-prepared by the researchers.	Same

Table 2. Independent variables.

Independent Variables	Simula Research Lab.	Univ. of Utah
<i>Given objects: Software artefacts</i>	Two variants of Java program code: Simple / Complex	(from scratch – no initial C program)
<i>Executing team</i>	Paired / Soloist – randomized	Same
<i>Task specification</i>	Extend coffee vending	Four different and unrelated

	machine functions, in 3 steps (T2-T4).	tasks (not revealed in paper).
<i>Assumed individual programming expertise (not PP-related)</i>	Junior / Intermediate / Senior; assessed by job manager.	Same, but according to previous exam marks.
<i>Pre-test of actual individual programming skills</i>	Observed duration to complete a correct T1 pre-test.	(nothing)

Table 3. Dependent variables.

Dependent Variables	Simula Research Lab.	Univ. of Utah
<i>Final objects: Software artefacts and logs.</i>	Modified Java code, number (i.e., percentage) of teams w/ all tests passed.	Developed C code, percentage of tests passed.
<i>Code size</i>	Neither size of baseline program (200-300? LOC), nor number or size of code increments are revealed in paper – but known by the researchers.	Ca. 100-150 LOC per program (not revealed in original paper).
<i>Correctness</i>	Number of teams (0..10, i.e. a relative number!) with all three tests (T2, T3, T4) finally passed and after an extra censor's final code inspection (binary score: Pass/Fail). Total number of committed defects not known!	Percentage of tests being finally passed – i.e., can deduce remaining defects. Total no. of committed defects not known!
<i>Duration</i>	Maximum 5-8 hours in same day.	4 sessions of maximum 5 hours during 6 weeks.
<i>Effort</i>	In p-hs, including defect fixing (but excluding corrective work that does not lead to more passing of tests!).	In p-hs, including defect fixing.
<i>Statistical data for comparing team performance (Code size, Correctness, Duration, Effort) vs. team composition and specified tasks.</i>	Formal hypotheses testing, no code size; see paper.	Similar but less combinations; see paper.

Table 4. Contextual variables.

Context variables	Simula Research Lab.	Univ. of Utah
<i>Programming language</i>	Java	C (assumed but not stated).
<i>Programming environment / tools</i>	Text editor, e.g. JDK w/ Java-compiler (cf. T0).	Emacs, C compiler etc.
<i>Site</i>	Distributed work places and "offices".	Central university lab.

The data presented in Table 1 through Table 4 reveal that the two PP experiments, in spite of rather similar overall objectives, vary from each other on a number of characteristics. Let us just consider the *testing set-ups* in these two studies:

- All test cases and test suites are pre-made (same for both).
- Assuming only coding defects; no requirements, design, or other kinds of defects (same).
- Only correcting pre-release defects, not post-release ones with extra reproduction costs (same) – but see Section 4.
- No version control or systematic regression testing (same).
- May correct the program several times until it passes the given test(s)? Note, that for Simula, unsuccessful correction efforts are ignored.
- Correctness metrics is very different; – *at Simula*: Percentage of similar teams (between 0 and 10) with all tests passed; – *at Utah*: percentage of tests actually passed; – *commonly elsewhere*: number of corrected pre-release defects or corresponding defect rate (number of such defects divided by LOC).

These observations on variations between two studies raise an important point: do we really need so much diversity? Neither *code size*, *correctness (software quality)*, nor *effort* have comparable metrics, only *duration*. Indeed, we have found *no* repeated and published PP experiment, except that the Simula and Utah ones went in several treatment rounds internally. Hence, it is not possible to aggregate generalized evidence from these two - very nice - experiments to support decision-making for industrial PP adoption.

3. A Formal Meta-Analysis to Compare 18 PP Experiments

Let us extend the scope of comparison to 18 PP-experiments, including the Simula and Utah ones, by looking at the statistical meta-analysis done by Hannay et al. [12]. This meta-study has chosen to study the PP-induced relations between three “competing”, dependent variables – Correctness, Duration and Effort – see Figure 1. It aims to show how these three variables are statistically related, and which studies contribute most to such relations. Overall results are visualized as “forest” plots, not shown here.

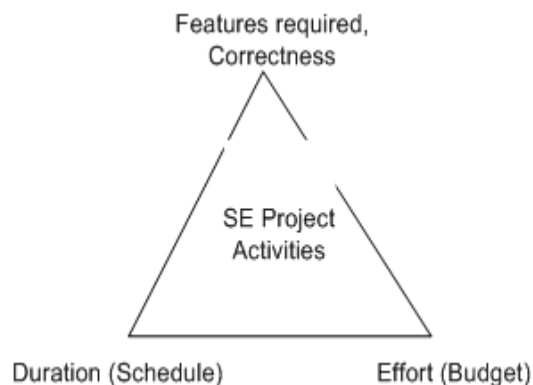


Figure 1 The classic Software Engineering (SE) Project Triangle with three “competing” corners.

We will not repeat the presentation of the meta-analysis results here, but convey some concerns about the validity of combining “apples and bananas”:

- *Programming task*: 18 different variants.
- *Correctness* is defined in twelve different ways: by OO quality metrics, by quality rating of changed UML designs, by a mixture of OO metrics and questions to the programmers, by test coverage as share of branches executed, by quality of requirements for later phases using PP vs. inspections, by defining a threshold value for share of test cases passed, by just recording the actual share of test cases passed (four cases including Utah), by a mixture of test cases passed and correct programmer (?) answers about two programs, by counting the share of similar teams with all tests passed plus final approval by external censor (Simula), by grading program performance using an external censor, by normal student grades, or simply missing (four cases). Maybe just *success factor* had been a better term?
- *Duration (in person-hours, with four variants)*: clock time from start till the team itself decided to stop, till a pre-set time limit expired, till a threshold test level was achieved, or till the team had passed all tests and a censor had approved the work (Simula).
- *Effort (person-hours, with two variants)*: Duration multiplied by one for soloists or by two for pairs – for all. But not including correction effort that does not improve quality (in Simula, but this rule is hard to practise). Many primary papers also use *time* ambiguously for Duration and Effort.

However, only 14 of the 18 PP studies have Correctness data, including 4 studies with only Correctness data. 11 studies have Duration data, and another set of 11 studies have Effort data. Further, only six studies have data for all three variables, 8 studies have data on Correctness and Duration, another 8 studies have data on Correctness and Effort, and 10 studies have data on Duration and Effort (a rather un-interesting combination).

Another, less ambitious meta-paper by Stephens and Rosenberg [23] concludes that "pair programming is not uniformly beneficial or effective" ... "you cannot expect faster and better and cheaper". The obvious solution is to introduce experience-based cost/benefit models and estimates. So if quality is paramount, then duration and/or effort may have to suffer, but a compromise is often negotiable. Likewise, we may compose programmer pairs according to expected needs now, or for training novices to later maintenance. There are nevertheless some unexplained paradoxes, such as novice teams behaving like "the blind leading the blind", cited by Voas [24], or capable of a "remarkable 149% increase" in correctness for more complex tasks when compared with "novice soloists", cited by Arisholm et al. for the Simula study [1].

Lastly, although the researchers behind this impressive meta-analysis are perfectly open about their acquired, heterogeneous and incomplete data, we simply cannot accept this as a permanent status of empirical PP research. On the other hand, the forest plots look credible, even seductive.

4. Extending the Utah Experiment with Commercial Defect Rates and Costs

Erdogmus and Williams [10] extended the PP-experiment data from [26] with commercial defect rates and costs to obtain a horizontal, lifecycle perspective. All this was put into a *Net Product Value (NPV)* model, using common interest rates for investments (effort spent) vs. delayed payback. The goal was to check whether moderate quality or productivity differences around 15 % might in fact translate into significant savings or losses. Their extra analysis might thus convert local *effectiveness* data into more general *cost efficiency* data.

The actual PP data showed that average *code productivity* for “soloists” was 25 LOC² per p-h and 21.7 for pairs. However, productivity rates for miniature programs are expectedly 3-5 times higher than the ca. 5 LOC per p-h found for large commercial programs. The productivity rate is simply computed by dividing the *total* code size by *all* development costs. It expresses that coding represents only 20 % of the total development effort, according to the common “4+2+4” rule, see e.g. in Sommerville’s textbook [22].

The average share of passed pre-release tests (provided for free by the researchers) was 75 % for soloists, and 85 % for pairs (corrected down from an initial figure of 90 %). We assume that each test targets one unique defect, not combinations. However, it is not reported how many such tests were available, e.g., 75 % could mean passing 3 out of 4 tests, or 15 out of 20.

The following baseline measures stem from Capers Jones [15] and come from a sample of 150 large commercial programs with average code size of 2.15 MLOC (megaLOC) and a median of 0.28 MLOC. From this sample, Capers Jones predicts that ca. 5 defects per Function Point (FP) will result from the coding and previous phases; and thus be subject to pre-release testing. Erdogmus and Williams followed Jones' assumption that there are 128 LOC-C (lines of C code) per FP, or ca. 60 Java code lines per FP. This again translates into **39 post-coding-phase defects per 1000 LOC-C**. According to Jones, about 15 % of these, or 6 defects, will escape pre-release testing. The delivered code from pairs will then have a mean of 6 (15 % of 39) post-release defects per 1000 LOC-C, while soloist code will have 10 such defects (25 % of 39).

Comment 1 – Normalization of defect densities: To retain an average of 6 post-release defects per 1000 LOC-C for soloists, we must “normalize” our number of post-release defects to 6 for soloists and to 4 for pairs (Erdogmus and Williams used, respectively, 6 and 3 here).

Comment 2 – Three times too many coding defects: The 5 defects per FP (for the years 1986-1996, p. 117 in [15]) include not only coding defects caught by testing (1.75 defects), but also requirements defects caught by inspections (1.00), design defects caught by other kinds of inspections (1.25), defects from faulty user documents (0.60), plus secondary defects from bad fixes and wrong test cases (0.40). So the relevant number of “true” coding defects in the PP-developed software is expected to be only 1.75 per FP, not 5.00 – i.e., only 35 % of initially envisaged.

The above **39** general defects per 1000 LOC-C must therefore be adjusted to $39 \cdot 0.35 = \mathbf{13.7}$ which includes only coding defects. The revised number of post-release coding defects per 1000 LOC-C then becomes **1.4** ($4 \cdot 0.35$) for pairs, i.e. that **12.3** (13.7-1.4) coding defects are committed, discovered and corrected before release. Similarly, the number of post-release coding defects becomes **2.1** ($6 \cdot 0.35$) for soloists, i.e. that **11.6** (13.7-2.1) such coding defects are corrected before release. The average effort to correct a pre-release coding defect then seems to be less than one p-h for unit-test defects, and 3-5 p-hs for system-test defects, given around 40 p-hs of total code development.

Comment 3: 3-4 times too high correction effort for each post-release coding defect: Jones ([15], p. 129) reports that the cost to correct an average defect is \$1500 or 34 p-hs, given a total hourly salary of \$44.2. This means that a soloist first will spend a working week of 40 p-hs to produce 1000 LOC-C of software code, plus some 160 more p-hs (a factor of 4 more) to make corresponding requirements, design, user manuals, and test cases. This total amount of software will have 6 post-release defects of *any kind*, totally costing $6 \cdot 34 = 204$ p-hs to correct. Other industrial studies indicate, that a post-release defect will cost 33-88 p-hs to correct, see Humphrey [14]. However, an average, post-release defect-correction cost of such magnitude is

² For simplicity we do not distinguish between LOC (all source lines) and NSLOC (non-commenting and non-blank source lines).

simply *not credible* in the miniature systems from the given PP experiments. It merely illustrates that data from large commercial systems must be adopted with great care into other contexts. For instance, the mentioned "4+2+4" rule predicts that testing is roughly twice as expensive as coding, but that covers a lot of test-related activities not included in the given experiments.

If we, for instance, apply the previous 35 % reduction to provide a revised effort estimate for correcting post-release coding defects, we get $34 \times 0.35 = 12$ p-hs per corrected defect. In a recent Norwegian study we found that a typical correction effort was 8 p-hs in one company with a 3 MLOC system, and 11 p-hs in a company with systems of some hundred KLOC, see Li et al. [18]. We can merely say, that *10 p-hs to correct a post-release coding defect* may serve as an initial quality cost estimate. Note anyhow, that the miniature programs found in [26] were *never* intended to have an "after-use", leading to corrective maintenance.

Example A: It now takes a soloist 40 p-hs to write 1000 LOC-C including fixing 11.6 pre-release coding defects, and another $2.1 \times 10 = 21$ p-hs to fix post-release coding defects, totally $40 + 21 = 61$ p-hs. A pair will use 43.4 p-hs for coding including fixing 12.6 pre-release coding defects, and another $1.4 \times 10 = 14$ p-hs for fixing post-release coding defects, totally $43.4 + 14.0 = 57.4$ p-hs. So the *pair "wins" by a margin of 6 % less coding and defect-fixing effort over the soloist*. However, the margin is narrow and the whole experiment so far inconclusive, especially considering the many simplifications and assumptions made.

Example B: A customer may initially offer a premium on correct software or an inverse penalty on buggy software, which can justify higher quality cost during development. Time-to-market pressure may inversely dictate a progressively lower price on future deliveries, so correctness is sacrificed on behalf of shorter duration.

Summing up: Commercial software has programming productivity of around 5 LOC/p-h due to large overheads, and a post-release code-defect correction cost of 30-80 p-hs/defect, see e.g. [14] [15] [22]. In miniature systems, these numbers are closer to 25 LOC/p-h and 5-10 p-hs/defect. So, Erdogmus and Williams adopted from [15] a three times too high coding defect rate, followed by a 3-4 times too high correction effort for post-release coding defects. While Erdogmus and Williams' instrumented NPV model may still be valid, the anticipated defect costs must be reduced by a factor of ten.

5. Discussion

Regarding **RQ1**, we can say that it is *very hard to assess and compare the effects of different PP studies*, as they vary drastically in terms of contextual factors and studied variables [12]. We may be able to perform only *relative* comparisons within the same context, not *absolute* ones across contexts. That is, each organization is unique with regards to its application domain, software portfolio, technology, skills, people, and company culture. Productivity and defect measures are therefore not likely to be commensurable between organizations. So, we may be able to compare teams vs. soloists in the *same* context, but not teams vs. teams across contexts. Finally, as hinted in Section 4, a reliable *scale factor* (of 3-5?) is needed to translate findings between miniature systems and commercial systems.

Taking a broader perspective, it seems that available (textbook) methods for designing, carrying out and reporting controlled experiments are not systematically practised by many researchers [9, 13, 16]. This makes research cooperation even harder, if each research group go on "re-inventing the wheel" and even in an inferior way. The overview of controlled experiments by Sjøberg et al. [21] lists few replicated experiments, and even worse – not a single PP

experiment. The entire research community has a shared responsibility here. Furthermore, how can a PhD student get sufficient credit for being, e.g., the 73rd replicator of experiment XX and the 27th replicator of case study YY? Lastly, can an organization like ISERN (International Software Engineering Research Network, <http://isern.iese.de>) play a role as an advisor, coordinator, or shared archive here?

Regarding **RQ2** on *how to define a partially standard metrics for software*, we suggest that the elements of a minimal common metrics should include:

- Code size: a volume or size measure for code (in LOC or Function Points), and neutral, automatic and free counting tools to perform size calculations. Note, that “much” code is no goal in itself, only the minimum amount of code to satisfy the specifications. We must also distinguish between new development as in [26] and maintenance as in [1].
- Quality: e.g. defined as Defect rate (measured pre- or post-release).
- Effort (p-hs) – must here account for software reuse.
- Duration (wall-clock hours).
- Personal background.

For our **RChall** on *how to study and promote the social and cognitive aspects of PP?*, we recommend that researchers should rather study the qualitative, socio-technical benefits of PP, since the quantitative effects on defects and costs seem to be slight (+- 15 %). Such aspects should then be appropriately studied by designing and conducting longitudinal case studies in industrial settings – and go deeper than finding PP “fun” to do. We also need more PP studies on learning, either regarding new technologies or insight in developed software vs. customer requirements wrt. future maintenance and support. However, it is not clear if PP is well-suited to make novices learning from experts [25], or if other team structures are more suitable [5].

The paper by Cockburn and Williams enthusiastically summed up the expectations from PP ten years ago [6], cf. also DeMarco’s book on *Teamware* [7]. But few concrete studies of the role of PP in teamware have been reported from the Agile community. On the effect of human personality on PP performance, there is however a recent study by Hannay et al. [11].

6. Conclusion

Based on this comparative study, we can conclude that there are several kinds of problems in aggregating the findings from PP experiments, mostly due to heterogeneous contextual variables and missing data. Hence, approaches like systematic literature reviews may not be a panacea to systematize and learn from experiences, unless we fix the underlying problem of heterogeneity.

Since the overall cost/benefits of adopting PP seem slight (+- 10-15 %), we may rather apply and investigate PP just because of its anticipated benign effects on teamwork.

Acknowledgements: We are grateful to discussions from colleagues and for comments from the anonymous reviewers. This work was partly funded by the Research Council of Norway under grant # 179851/I40.

References

- [1] Arisholm, E., Gallis, H., Dybå, T., and Sjøberg, D.I.K., Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise, *IEEE Transactions on Software Engineering*, 2007. **33**(2): pp. 65-86.
- [2] Arisholm, E. and Sjøberg, D.I.K., Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software, *IEEE Transactions of Software Engineering*, 2004. **30**(8): pp. 521-534.
- [3] Basili, V.R., Caldiera, G., and Rombach, H.D., The Goal Question Metric Approach, in *Encyclopedia of Software Engineering*, J. Marcinak, Editor. 1994, John Wiley. pp. 528-532.
- [4] Beck, K., et al. Manifesto for Agile Software Development. Last accessed on October 10, 2010, Available from: <http://AgileManifesto.org>.
- [5] Brown, J.S. and Duguid, P., Organizational Learning and Communities-of-Practice: Toward a Unified View of Working, Learning, and Innovation, *Organization Science*, 1991. **2**(1): pp. 40-57.
- [6] Cockburn, A. and Williams, L., The Costs and Benefits of Pair Programming, *First Annual Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2001. pp. 223-247.
- [7] DeMarco, T. and Lister, T., *Peopleware: Productive Projects and Teams*. 2nd ed. 1999: Dorset House Publishing.
- [8] Dybå, T. and Dingsøy, T., Empirical Studies of Agile Software Development: A Systematic Review, *Information and Software Technology*, 2008. **50**(9-10): pp. 833-859.
- [9] Dybå, T., Kampenes, V.B., and Sjøberg, D.I.K., A Systematic Review of Statistical Power in Software Engineering Experiments, *Information and Software Technology*, 2006. **48**(8): pp. 745-755.
- [10] Erdogmus, H. and Williams, L., The Economics of Software Development by Pair Programmers, *Engineering Economist*, 2003. **48**(4): pp. 283-319.
- [11] Hannay, J.E., Arisholm, E., Engvik, H., and Sjøberg, D.I.K., Effects of Personality on Pair Programming, *IEEE Transactions of Software Engineering*, 2010. **36**(1): pp. 61-80.
- [12] Hannay, J.E., Dybå, T., Arisholm, E., and Sjøberg, D.I.K., The Effectiveness of Pair-Programming: A Meta-Analysis, *Information and Software Technology*, 2009. **55**(7): pp. 1110-1122.
- [13] Hannay, J.E., Sjøberg, D.I.K., and Dybå, T., A Systematic Review of Theory Use in Software Engineering Experiments, *IEEE Transactions of Software Engineering*, 2007. **33**(2): pp. 87-107.
- [14] Humphrey, W.S., *A Discipline for Software Engineering*. SEI Series in Software Engineering, ed. J.T. Freeman and J. D.Musa. 1995: Addison Wesley Longman, Inc.
- [15] Jones, Capers, *Software Quality – Analysis and Guidelines for Success*. 1997, Boston, MA: International Thomson Computer Press (a pearl of industrial data).
- [16] Kampenes, V.B., Dybå, T., Hannay, J.E., and Sjøberg, D.I.K., A systematic review of effect size in software engineering experiments., *Information and Software Technology*, 2007. **49**(11-12): pp. 1073-1086.
- [17] Kitchenham, B., Dybå, T., and Jørgensen, M., Evidence-Based Software Engineering, Proc. 26th International Conference on Software Engineering (ICSE'1994), Edinburgh, U.K., 23-28 May 1994, IEEE CS Press. pp. 273-281.
- [18] Li, Y., Stålhane, T., and Conradi, R., Improving Software Defect Tracking Systems to Facilitate Timely and Continuous Software Quality Assessment and Improvement, draft paper, NTNU, Oct. 2010, 11 pp.
- [19] Miller, J., Applying meta-analytical procedures to software engineering experiments, *Journal of Systems and Software*, 2000. **54**(1): pp. 29-39.
- [20] Pickard, L.M., Kitchenham, B.A., and Jones, P., Com-bining empirical results in software engineering, *Information and Software Technology*, 1998. **40**(14): pp. 811-821.
- [21] Sjøberg, D.I.K., Hannay, J.E., Hansen, O., Kampenes, V.B., Karahasanovic, A., Liborg, N.-C., and Rekdal, A., A Survey of Controlled Experiments in Software Engineering, *IEEE Transactions on Software Engineering*, 2005. **31**(9): pp. 733-753.
- [22] Sommerville, I., *Software Engineering*, 9th Ed., Pearson, 2011, ISBN 978-0-13-705346-9, 773 pages.
- [23] Stephens, M. and Rosenberg, D., *Extreme Programming Refactored: The Case against XP*. 2003: Apress.
- [24] Voas, J., *Faster, Better, and Cheaper*, *IEEE Software*, 2001. **18**(3): pp. 96-99.
- [25] Weinberg, G.M., *The Psychology of Computer Programming*, Programming Silver Anniversary Edition. 1998, New York: Dorset House Publishing.
- [26] Williams, L., Kessler, R.R., Cunningham, W., and Jeffries, R., Strengthening The Case for Pair Programming, *IEEE Software*, 2000. **17**(4): pp. 19-25.