Information and Software Technology 53 (2011) 557–573

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Impact of test-driven development on productivity, code and tests: A controlled experiment

Matjaž Pančur *, Mojca Ciglarič

Faculty of Computer and Information Science, University of Ljubljana, Tržaška 25, Ljubljana, Slovenia

ARTICLE INFO

Article history:
Received 24 July 2010
Received in revised form 7 February 2011
Accepted 7 February 2011
Available online 13 February 2011

Keywords:
Empirical software engineering
Controlled experiment
Test-driven development
Iterative test-last development

ABSTRACT

Context: Test-driven development is an approach to software development, where automated tests are written before production code in highly iterative cycles. Test-driven development attracts attention as well as followers in professional environment; however empirical evidence of its superiority regarding its effect on productivity, code and tests compared to test-last development is still fairly limited. Moreover, it is not clear if the supposed benefits come from writing tests before code or maybe from high iterativity/short development cycles.

Objective: This paper describes a family of controlled experiments comparing test-driven development to micro iterative test-last development with emphasis on productivity, code properties (external quality and complexity) and tests (code coverage and fault-finding capabilities).

Method: Subjects were randomly assigned to test-driven and test-last groups. Controlled experiments were conducted for two years, in an academic environment and in different developer contexts (pair programming and individual programming contexts). Number of successfully implemented stories, percentage of successful acceptance tests, McCabe's code complexity, code coverage and mutation score indicator were measured.

Results: Experimental results and their selective meta-analysis show no statistically significant differences between test-driven development and iterative test-last development regarding productivity ($\chi^2(6) = 4.799$, $p = 1.0$, $r = .107$, 95% CI (confidence interval): −.149 to .349), code complexity ($\chi^2(6) = 8.094$, $p = .46$, $r = .048$, 95% CI: −.254 to .341), branch coverage ($\chi^2(6) = 13.996$, $p = .059$, $r = .182$, 95% CI: −.081 to .421), percentage of acceptance tests passed (one experiment, Mann–Whitney $U = 125.0$, $p = .98$, $r = .066$) and mutation score indicator ($\chi^2(4) = 3.807$, $p = .87$, $r = .128$, 95% CI: −.162 to .398).

Conclusion: According to our findings, the benefits of test-driven development compared to iterative test-last development are small and thus in practice relatively unimportant, although effects are positive. There is an indication of test-driven development endorsing better branch coverage, but effect size is considered small.

1. Introduction

Test-driven development (TDD) [1], sometimes also called test-first (TF) programming, is a programming technique adopted in extreme programming (XP). The main idea behind TDD concept is that the programmer writes unit tests before writing any new code. When writing the test code, programmer in fact specifies how exactly the program would need to work in order to pass the test. The test necessarily fails, since there is no code yet. After that, the programmer writes the code needed to pass the test. When the test passes (and other previous tests pass too), the programmer looks at the code and if needed, improves its design. This is called refactoring. After that, a cycle is closed and the programmer proceeds with the next small piece of functionality. Test first, of course. As we can see, TDD is not a testing but rather a programming and designing technique. It is very iterative by nature, since everything happens in short cycles. According to Jeffries et al. [2], with TDD, developers organically develop a test suite while building their applications, thus providing a safety net for the whole system, offering reasonable confidence that no part of the code is broken.

TDD is believed to have several favorable properties [3]: tests provide instant feedback, writing tests encourages the programmer to decompose the problem into manageable tasks (high granularity), which also positively affects low level design, and frequent test runs ensure a certain level of quality. However there are only a handful of controlled experiments on TDD, as we will see in Section 2, and the results are often inconclusive or conflicting. It seems that

* Corresponding author. Tel.: +386 1 4768 277.
E-mail addresses: matjaz.pancur@fri.uni-lj.si (M. Pančur), mojca.ciglaric@fri.uni-lj.si (M. Ciglarič).

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.02.002

the only thing the researchers agree on is that more research is needed. Intuitively, we can see the positive effect of interleaved coding and testing. However it is not clear whether the advantage of TDD really comes from writing tests before writing code and not from high granularity of TDD process. What if most of the supposed benefits of TDD come from short development cycles, not from writing tests before code?

The purpose of this paper is to present the results of the controlled experiment, comparing test-driven development with iterative test-last development regarding programmer productivity, program code quality, test thoroughness (code coverage) and fault-finding capabilities of tests. By this, we hope either to confirm some of the previous research comparing TDD with coarse-grained test-last development, or to offer a challenge to the SE research community about the benefits being rooted on short development cycles instead of writing tests before code.

The rest of the paper is organized as follows. Section 2 reviews related research results. Section 3 describes the conceptual model of our study. In Section 4 we explain the experiment. The analysis of the experiment is conducted in Section 5. Threats to validity are described in Section 6. Section 7 concludes with results and discussion.

2. Related work

This paper builds on our preliminary work [7], describing the short pilot study and preliminary data analysis. Although brief and not deep enough, the study was among the first controlled experiments on the effects of TDD. Research studies and reports about effects of TDD have begun appearing at first as case studies [17,25], experience reports [10] and surveys [2,9,10]. Although TDD has already gained wider recognition and acceptance in the industry as part of Extreme programming methodology [1,11,12], first controlled experiments about effects of TDD were to our knowledge published in 2002 by Mueller and Hagner [5]. Since then, several controlled and quasi-controlled experiments were published, but results regarding the effect of TDD are often conflicting, inconclusive and rarely statistically significant. What is still needed is systematical building of knowledge about TDD through a family of controlled experiments in different contexts and in different execution environments (replications of experiments) [13–15]. In the rest of this section we present a summary of the most relevant results focused on controlled and quasi-controlled experiments, conducted in academic as well as industry environments. Where not explicitly noted, the presented results were not statistically significant.

Mueller and Hagner [5] investigate TDD effectiveness, code reliability and understanding of the program compared to traditional test-last. Experiment was conducted in academic environment with 19 graduate students. In this case, TDD group spent more time in coding, but had significantly lower code reliability ($p = 0.03$) in acceptance testing. However, TDD group achieved better understanding of the program, measured as proper reuse of existing methods. George and Williams [8] compare TDD to a kind of waterfall development with 24 professional pair programmers as test subjects. Results show TDD developers have written better code (18% more functional tests passed) in 16% more time (that means 16% lower productivity). Geras et al. [16] explore productivity and quality in 14 professional developers using TDD and TL. No significant differences were found. Due to some weaknesses, for example sampling policy (volunteering), authors see this as a pilot study or quasi experiment. Erdogmus et al. [3] describe a controlled experiment with 24 undergraduate students, comparing TDD and iterative TL (story-level granularity, compare with Figs. 1 and 2). They found a linear dependency between productivity (measured by the number of implemented stories) and the number of tests, while TDD group wrote 52% more tests than control group ($p = .09$). Thus TDD group achieved higher productivity. Statistical power is low and not explicitly published; mortality rate is high. Positive effects of TDD on quality are reported in two case studies from professional environment (6–8 subjects) by Bhat and Nagappan [17]: reduced defect density (2.6–4.2 times) was found, while development time increased for 15–35%. Janzen and Saiedian [18] compare TDD and test-last development, conducting a family of quasi-experiments in academic and industrial context. The number of participants was low, and they were working in teams. The results showed that in TDD group, the subjects were more likely to write more and smaller units of code that are less complex. In some of the cases, results were statistically significant. Vu et al. [19] evaluate TDD with undergraduate students in a yearlong software engineering course. Their results differ from most of previous research and indicate the teams with test-last approach are more productive and produce more test cases than those using test-driven approach. Turhan et al. [20] describe an exploratory study, comparing code in test-first and test-last development. Their results indicate no significant superiority of test-first approach; moreover, test-last approach had significantly simpler code in terms of cyclomatic complexity. A comparative case study from Siniaalto and Abrahamsson [21] shows inconclusive effect of TDD on program design, while TDD test coverage was significantly superior over the control group.

Huang and Holcombe [4] compare the effectiveness of test-first programming with that of test-last (traditional) in terms of effort spent on testing and coding, productivity, external quality, and possible correlations between them. Test subjects were undergraduate students (39 students in 10 teams, working for 4 external clients). Their findings were that test-first group spend higher percentage of time on testing ($p < 0.1$) and lower percentage on coding, external quality increased with the percentage of time spent on testing in both groups, and a strong linear correlation was found between the amount of effort spent on testing and coding in test-first teams, while this phenomenon was not observed in test-last teams. Canfora et al. [60] describe a controlled experiment in the industry environment. They found that the development process is slower with TDD (statistically significant, $p = 0.02$), while there are no indications that TDD might bring more accurate unit tests.

Madeyski [6] studies the impact of TDD on branch coverage and mutation score indicator of unit tests (test quality) on 22 students. He found no statistically significant difference between TDD and TL group. Further, Madeyski [59] studies impact of TDD on external code quality in pair programming and solo programming. In both cases, external quality in TDD was significantly lower ($p < 0.05$), while when comparing pair to solo programming, there were no significant differences. In [58] he investigates the impact of different development methods (classic solo, classic pairs, TDD solo and TDD pairs) on package level design quality indicators, however no significant impact was found.

Gupta and Jalote [57] evaluate the impact of TDD through a controlled experiment with two groups of students. External code quality is improved in one of the two programs, and it is affected by the testing efforts applied. Also overall development effort is reduced.

Besides that, surveys and case studies from industry and academic environment shed light on the areas interesting for further research [23–26]. A comprehensive survey and discussion on the experiments and other related work in industrial as well as academic environments can be found in the book from Madeyski [56] together with a thorough discussion of three experiments and different statistical methods used. Table 1 presents a quick overview of perceived effects on the parameters, which are also

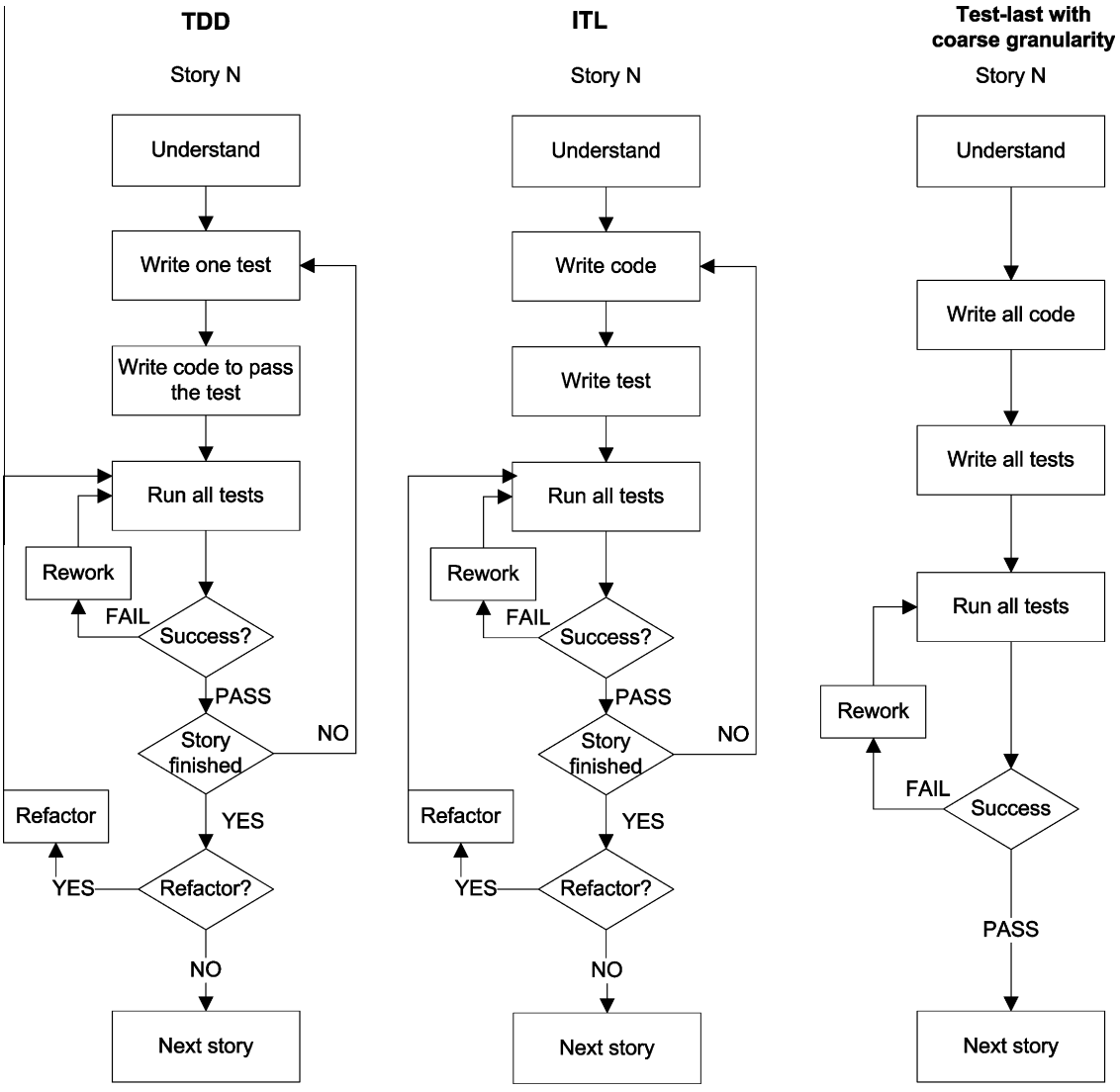

Fig. 1. Comparison of the basic development cycle in TDD, iterative test-last (ITL) and coarse-grained (traditional) test-last development.

subject of our research, as described in related studies. The sign less (<) means the effect of TDD on the parameter in question was in negative direction, for example productivity was lower in TDD group than in control group. The sign greater (>) means the effect of TDD on the parameter in question was in positive direction, for example productivity was higher in TDD group than in control group. Sign "o" means there was no important difference between groups. Study type is represented by controlled experiment (CE), quasi-controlled experiment (QCE) and case study (CS).

We also put special attention on the fact that in TDD, highly iterative development (so called micro iterations), might also be a covert variable in the study of effects of the development process. Hence, we eliminated this threat by employing a similar micro iterative process (ITL) in our control group.

3. Conceptual model

In this section, we present the conceptual model of our study: the independent variable (development process) and the related two treatments (TDD and ITL), the dependent variables (productivity, code and test properties indicators) and the covariates – the variables that can influence the result, but are not part of the independent variable.

3.1. Comparison of TDD and ITL

In this paper, we compare TDD with test-last development. In test-driven development, the tests are the driving force of the whole process: not even a line of production code is supposed to be written without a test code, covering its functionality. In contrast with test-first discipline, in test-last the program code is written first and unit tests are created afterwards. There are several variations of test-last development. There is classical waterfall-like development, where almost all program code is written before any test code. Software engineering literature often compares TDD with test-last development, where larger chunks of code, either the whole program or the whole story, are developed and tested in one cycle [3–6]. In our study, we consider iterative test-last development (ITL), where the pieces of functionality developed first and tested afterwards are smaller, comparable to pieces developed in one cycle of TDD (micro iterations). To our knowledge no other study pays special attention to the granularity of the development process in the control group. Figs. 1 and 2 graphically compare the basic cycle in TDD, iterative test-last (ITL) and traditional coarse-grained test-last development. Please note that we have omitted refactoring phase in the traditional test-last diagram, however this does not mean there should not be one. It simply means it

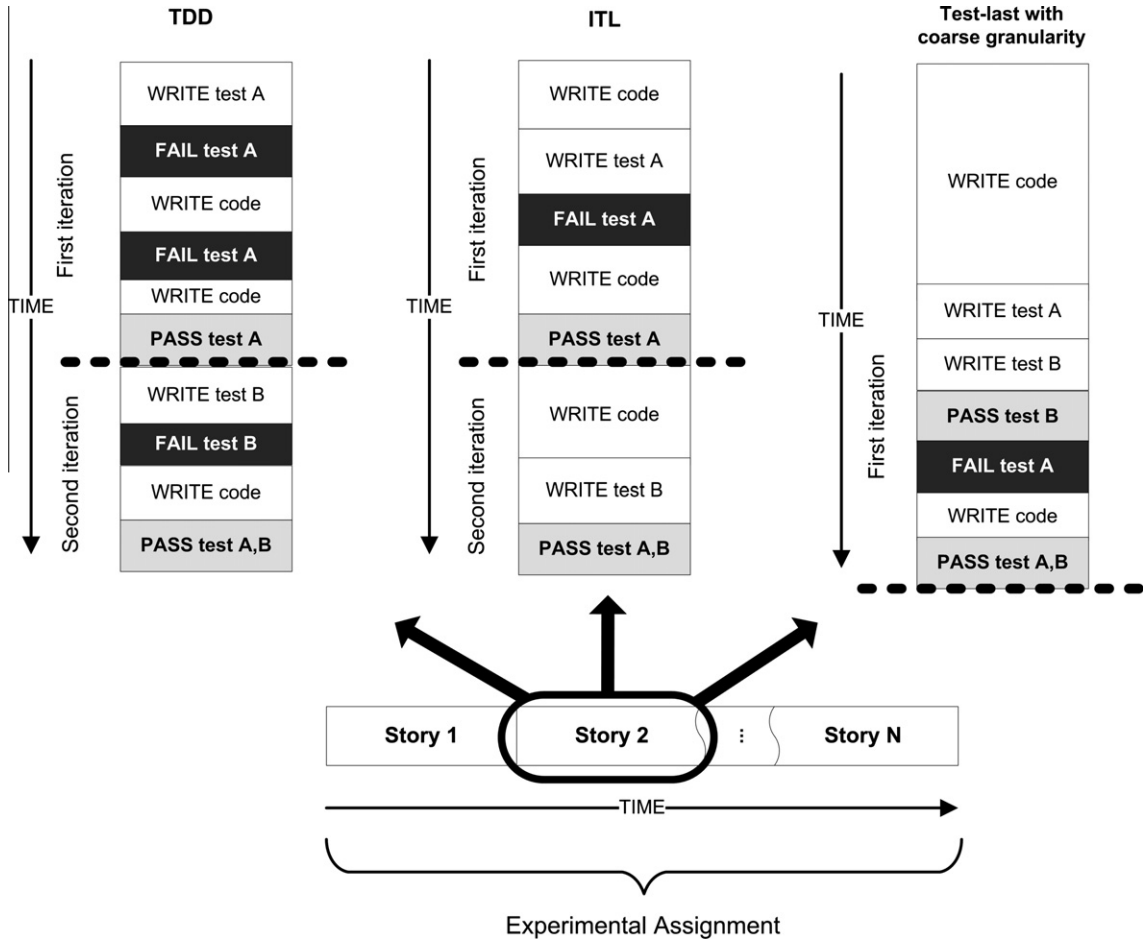

Fig. 2. Granularity of interleaved coding and testing – comparison of TDD, ITL and coarse-grained test-last development. Refactoring steps are omitted for clarity.

Table 1
Overview of percieved effects of TDD on productivity, external quality, complexity, code coverage and MSI in some of the existing studies.

	Study type	Productivity	External quality	Complexity	Code coverage	MSI
Madeyski [56]	CE	<	<		>	>
George and Williams [8]	QCE	<	>			
Bhat and Nagappan [17]	CS	<			>	
Erdogmus [3]	CE	>	o			
Flor and Schneider [32]	QCE	>			<	
Gupta and Jalote [57]	CE	>	>			
Huang and Holcombe [4]	CE	>	o			
Janzen and Saiedian [55]	QCE,CS			>		
Janzen and Saiedian [24]	CE		>			
Crispin [54]	CS		>			
Geras et al. [16]	QCE		o			
Pančur et al. [7]	CE		<		<	
Siniaalto [21]	CS				>	
Mueller and Hagner [5]	QCE				<	
Madeyski [58]	CE		<			

is not suggested by the process, rather it is up to the programmer if and when to refactor.

Pair programming is a software development practice, known in extreme programming, where programmers work in pairs and exchange the roles of active developer – driver and observer – navigator. While the active developer usually focuses on code details, the observer focuses on the big picture and suggests further programming steps. In individual development process, the single programmer plays both roles. Pair programming is often used together with TDD practice, since both are recommended in XP methodology. However due to lack of resources pair programming is not always feasible and in real life it is easy to find TDD practiced individually. Our study therefore investigates TDD in both contexts – in pair programming (PP) as well as individual development (ID).

3.2. Productivity, code properties and test properties

Productivity refers to the amount of work, accomplished in a certain amount of time. Since LoC metrics (number of lines of code) depends on the programmer and is not directly related to the fin-

ished program, we rather focus on the functionality or stories to be implemented: the more stories implemented, the higher amount of work is accomplished.

Code properties are observed from two different viewpoints: from customer's viewpoint (external quality) we are interested in the correct program functionality and from programmer's viewpoint we are interested in program complexity.

Program functionality is assessed via acceptance testing, and the number or percent of successful acceptance tests (NATP or PATP) has been suggested by several researchers as an indicator of external quality, as outlined in [56]. We will use PATP: user requirements are checked by the detailed acceptance tests and the higher is percentage of successful acceptance tests, the more requirements are met. Eventually, every subject can meet all the requirements, however under time pressure the developers tend to omit some functionality. PATP can be used also as an interesting measure of productivity, if it is normalized per time of development.

It is agreed that high code complexity has several undesirable drawbacks: it may result in poor understandability, more errors, defects, exceptions, leading to software requiring more time to be developed, tested and maintained. Complexity and other internal quality related properties are measured with several different metrics, which are overviewed and discussed as early indicators of external quality in [56]. The drawbacks of such metrics are ignorance of organizational context and the purpose of the code, limitation to syntactic aspect, ignorance of essential quality issues, such as the usage of appropriate data structures and meaningful documentation etc., and are discussed elsewhere [45,46]. However despite of that, subjectively several researchers admit that more linear code is easier to understand and indeed correlations can be found between such subjective impressions, code complexity measurements and other quality-related software attributes [46–48,56]. With this in mind, we will use McCabe's cyclomatic complexity [50], which is an indirect metrics calculated from direct metrics such as the number of branches in code, and has been often used when assessing software quality or design quality, and recall that it only measures the number of independent paths through code while its relationship with code readability, understandability and maintainability is unclear.

Not only code properties are important in test-driven development. Among essential properties of tests are their thoroughness and their fault-finding capability. Thorough tests cover more code – more statements, branches or methods. Code coverage is a well-established concept for evaluating tests thoroughness [49,6]. There are several code coverage measures: branch coverage, statement coverage, method coverage, while TPC (Total Coverage Percentage) is a compound measure, derived from method coverage, statement coverage and branch coverage [51]. We use branch coverage as the most appropriate measure of test thoroughness [49], since it measures to what extent all possible paths through code are covered (branch conditions evaluated to both *true* and *false*). Although one could understand branch coverage as a measure for tests quality, we need to be aware it only measures one aspect of test quality, namely the thoroughness of unit tests. Since TDD methodology is designed in such a way that it should get better coverage by definition, we will not measure test properties only with branch coverage in order to avoid favoring TDD already by choice of metrics.

For fault detecting effectiveness of unit tests, we will use mutation testing, suggested already in the eighties, but only recently practically available through automated tools, for example Judy [61,63]. When creating mutants, errors are deliberately injected into program code and tests are expected to fail as a consequence. If they do not fail, they obviously did not test the part of code where the error was injected. Mutation score indicator is an indicator of the lower bound on the ratio of the number of killed mutants to the total number of non-equivalent mutants [6,56] and serves as a complement to code coverage in evaluating test thoroughness and effectiveness. MSI has only recently been suggested for empirical evaluation of fault-finding capabilities of tests, since no efficient tools for capturing MSI were available before.

4. Experiment description

In our experiment design and description we have followed the suggestions by Wohlin et al. [38] and Lott and Rombach [27]. Our primary goal was to find out what are the effects (if any) of test-driven development on agile development process, the resulting code and tests. The parameters of interest are productivity, measured as number of implemented user stories per hour (NIUSPH), percentage of acceptance tests passed (PATP), code complexity, test thoroughness (branch coverage) and fault-finding capabilities of tests (measured as mutation score indicator – MSI).

Therefore, the objects of study are development process, production code and tests. The purpose of the research is evaluation of the impact of TDD on productivity, code and tests. Our quality focus is represented by programmers' productivity (number of stories implemented in a certain time), external code quality (percentage of acceptance tests passed), code complexity (McCabe's code complexity), tests thoroughness (code coverage) and fault-finding capability of tests (mutation score indicator). The experiment is performed and described from the researcher's perspective. The context of experiment was a course on distributed systems in the fourth year of computer science study; the subjects were students.

4.1. Context

The experiment was performed in academic environment, within a course on distributed systems. Java was our programming language of choice since students were already familiar with it from courses they have attended in previous years (programming, algorithms and data structures, computer networking, etc.). Further, Eclipse IDE was used due to its excellent refactoring support, tight integration with JUnit testing framework and TDD support in general, ability to include external plug-ins and overall performance during development. Acceptance tests were written using FIT framework, complemented by Socket Acceptance Tester [34].

The subjects are undergraduate students in the fourth – last year of computer science study. Within the course, all the subjects were introduced to testing in general as well as TDD and ITL development. In the laboratory part of the course, the subjects became familiar with all the tools and their assigned development method (TDD or ITL). The total course duration was 15 weeks, each week with two hours' lecture and two hours of laboratory work. The first ten weeks of laboratory part were dedicated to process training and the following five weeks to the described experiment.

The experiment was carried out in two different contexts, covering most likely practical use of TDD:

- The context of pair programming – PP (within XP, TDD is usually used this way)
- The context of individual development – ID (TDD within other agile methodologies, as well as in traditional development, where developers do not have acceptance tests available to guarantee acceptable external quality).

With different contexts we want to embrace different work conditions of developers: fixed external quality or work under time pressure as well as pair and individual programming. We believe that measurements in different contexts enable us to reach more general conclusions on effects of TDD.

4.2. Experimental variables

The only independent experimental variable is the development process: test-first (TDD) or test-last (ITL) development. We focused attention on eliminating the possibility of covert independent variables that could affect the results: our subject attended the same courses, received the same assignments, same tools, have worked in equal conditions; they all have used micro iterative approach. In the control group, we made sure that the process granularity of control group (ITL) was comparable to TDD group, thus eliminating the effect of iteration fragment size being tested. This way, we believe we have isolated the effects of test-first from other covariates (e.g. from granularity of tasks in ITL and TDD, etc.).

The dependent variables are productivity (NIUSPH), percentage of acceptance tests passed (PATP), code complexity (McCabe), branch coverage (BC) and mutation score indicator (MSI).

4.3. Hypotheses formulation

All our hypotheses are bidirectional: we expect that the productivity of TDD developers will be different from the productivity of ITL developers (hypothesis PROD). Due to continuous testing, we further expect the external quality of program code, measured by percentage of acceptance tests passed, will be different in TDD than in ITL (hypothesis QUALE). We expect that due to interwoven phases of testing and coding as well as frequent refactoring, TDD code might be less complex than ITL code. Our assumption is supported by the findings of Janzen and Saiedian [55], however due to the lack of evidence our third hypothesis (COMPL) is bidirectional as well. By test thoroughness we mean the ability to identify most of the situations where something could possibly go wrong. Thorough tests cover more code and our fourth hypothesis (COVER) states that branch coverage is TDD is different from branch coverage in ITL. Madeyski additionally proposes measuring fault-finding capability of tests by means of mutation score indicator [6]. The program code is being systematically changed (the so-called mutations are produced) and the tests are consequently expected to fail – to find the potential faults. However in his study, he found the impact of TDD on mutation score indicator minor. Therefore, our last hypothesis (MSI) will test if there is a difference in TDD and ITL tests regarding mutation score indicator.

Table 2 overviews the formalization of the hypotheses together with corresponding null hypotheses, required for validity testing.

4.4. Subjects selection

Our subjects were selected based on convenience: they were senior students in the fourth – last year (8th semester) of Computer and information science study at University of Ljubljana (4th year of our old undergraduate program is roughly equivalent to our new Bologna reformed first year Masters degree graduate students program). Just before the end of their study, programming knowledge of the students is comparable to the knowledge of less experienced professional developers. All of them have previously attended all fundamental courses: Java programming course, algorithms and data structures, computer networks, etc. We have also dedicated a significant amount of time (10 weeks) to training of the development process.

When checking the equivalence of experimental groups, we took into account average grades (past academic performance), general programming knowledge, experience with OO programming, months of experience in Java programming and eventual experience in industry. There have been no statistically significant differences between groups regarding the above parameters (t-test, $\alpha = 0.05$, $p = 0.10$ to 0.83).

Table 2
Overview of hypotheses formalization.

Parameter	Hypothesis name	Null hypothesis H_0	Alternative hypotheses
Productivity	PROD	$PROD_{TDD} = PROD_{ITL}$	$PROD_{TDD} < PROD_{ITL}$ $PROD_{TDD} > PROD_{ITL}$
Code	QUALE	$QUALE_{TDD} = QUALE_{ITL}$	$QUALE_{TDD} < QUALE_{ITL}$ $QUALE_{TDD} > QUALE_{ITL}$
	COMPL	$COMPL_{TDD} = COMPL_{ITL}$	$COMPL_{TDD} < COMPL_{ITL}$ $COMPL_{TDD} > COMPL_{ITL}$
Tests	COVER	$COVER_{TDD} = COVER_{ITL}$	$COVER_{TDD} < COVER_{ITL}$ $COVER_{TDD} > COVER_{ITL}$
	MSI	$MSI_{TDD} = MSI_{ITL}$	$MSI_{TDD} < MSI_{ITL}$ $MSI_{TDD} > MSI_{ITL}$

4.5. Experiment design

Our experiment type is standard one factor (development approach) and two treatments – TDD or ITL (single-factor, posttest-only, inter-subject design) [28]. We deliberately refrained from multi factor experiment type since we believe switching from test-first to test-last practice would confuse the subjects and the context switch would introduce high risk for results distortion. The assignment of subjects to treatments was randomized. Each subject took part in two projects under the same treatment – the first in pair programming context and the second in individual programming context. With different contexts we want to embrace different work conditions of developers: fixed external quality or work under time pressure as well as pair and individual programming. We believe that measurements in different contexts enable us to reach more general conclusions and make results more robust.

Conformance of subjects with the designated development approach (TDD or ITL) was made possible by analyzing the data collected by ProcessLog Eclipse plug-in. The subjects who did not submit their projects or where process conformance was weak were excluded from the study, resulting in an unbalanced design of experiment in both series and both contexts (individual and pair programming). The exact numbers are presented and explained in Section 4.7.

4.6. Instrumentation and measurement

All the materials needed for the experiment were prepared in advance: pre-test questionnaire (on preexisting knowledge and experiences with programming), post-test questionnaire (qualitative observations and subjective experiences with the development process and process conformance), instructions on TDD and ITL use, coding standards, programming specifications and user stories.

The subjects were using a properly configured "Local history" in Eclipse IDE. Essentially, this is an integrated local source code versioning system, which captured the code whenever the subjects saved the code or ran the tests. This data were essential when analyzing the course of the development process, while it also greatly reduced the possibility of plagiarism, since the subjects knew any unusual episodes could be detected. This has also strengthened development process adherence.

Development process properties including testing and coding times were measured by means of Eclipse plug-in ProcessLog that we have developed ourselves. ProcessLog, enables time logging of the development process metadata: times spent in coding, testing and refactoring as well as who was the active developer in each time slice; further it times when the tests are run and the number of successful and failed tests. ProcessLog is fully integrated in

Eclipse and its log files, together with source code versions history, enabled collection of detailed data and consequent analysis of development process for all subjects.

Code coverage, McCabe code complexity and mutation score indicator were measured by means of Clover [51], Metrics [62] and Judy [61,63] tools, respectively, while percentage of acceptance tests passed was captured directly from JUnit.

4.7. Experiment operation

The experiment consisted of two phases: a thorough preparation followed by the actual coding. Both experiment repetitions (series) have followed the same procedure.

At the beginning, the students were given a questionnaire asking about Java knowledge, general programming knowledge, OO principles knowledge, practical experience in real world projects etc. We have also included the average study grade (past academic performance), which was not self-reported. All these parameters (covariates) were used to check the equivalence of experimental groups within each series for each experiment and no statistically significant differences were found (t-test, $\alpha = 0.05$, $p > 0.05$). Due to mortality, we have repeated the significance testing at the end of experiment, before data analysis. The differences regarding above covariates were still not statistically significant.

Main goal of the extensive preparation phase was to make sure all test subjects are familiar with the development process and have enough hands-on experience to use it smoothly. Studies researching the use of agile methodologies in university environment indicate the students need some time to understand and use them properly [29,30,33]. Further, although both development processes are in a way similar, it has been reported several times, that mastering TDD requires a certain amount of training [29–32], before a subject is considered TDD conformant. Our subjects were presented with theoretical foundations in form of lectures as well as with several practical examples on unit testing, JUnit, refactoring, agile design, frequent mistakes, etc. Afterwards, they have developed three short assignments themselves, which were then carefully examined by the instructor and each subject was provided with individualized comments and warnings. After that, all of the subjects were sufficiently prepared for further work in a way that they could focus on a problem, not on the development process. The preparation phase took ten weeks.

We have performed two series of experiments with different groups of students in two consecutive study years, each consisting of two controlled experiments in two different contexts (pair programming and individual developers programming). The students were randomly assigned to treatments; however each subject took part in two experiments. The first was pair programming project, lasting for 5 weeks. The students were allowed to work in the laboratory and at home, as long as they were following the required process and other recommendations.

The second part of the experiment was development of the individual project, which was performed under supervision in the laboratory as a part of final exam and lasted for four hours.

The number of subjects in each experiment by series is given in Table 3. In PP columns, the numbers denote number of pairs, not individual subjects. Series 1 and Series 2 denote the first and the second repetition of the experiment, respectively. Similarly, later in the paper when referring to the first repetition of experiment, we use ID1 for individual developers and PP1 for pair programming, and for the second repetition of experiment we use ID2 and PP2. Due to mortality, final numbers are lower than initial.

The subjects were required to use a coding standard based on Sun's Java Coding Standard in order to minimize productivity differences due to different coding standards. Automated compliance checking was used (with Eclipse plug-in). Both groups were given same lectures and exercises, have used same tools and the only difference, i.e. independent variable, was the development process, namely writing tests before or after writing code.

Table 3
Number of test subjects in test groups.

	Initial numbers			Final numbers (due to mortality)			
	Total	TDD	ITL	PP-TDD	PP-ITL	ID-TDD	ID-ITL
Series1	38	20	18	10	9	14	9
Series2	34	16	18	7	9	14	18

We have explained and emphasized the importance of process adherence to students (also when working at home) and made sure they understood. We have also explained the importance of following the process strictly; following the process was more important than project quality for final grading.

The process phases (coding, testing) and developer roles (who is typing and who is watching) were logged and conformance with the process was checked later in the analysis phase: we subjectively assessed the interleaving of coding and testing phases and checked the developer roles on the basis of ProcessLog data. When in doubt, we additionally compared the intermediate code versions. At last, we applied objective criteria: we excluded all the projects with branch coverage less than 50% and also those with ratio test code to production code (NCLOC) less than 50% (if testing is properly applied, the number of tests lines of code is supposed to be at least half as much as the number of production code lines).

The PP programming assignments were in the area of distributed systems and network programming: in PP1, it was implementation of a simple distributed database server with built-in data replication mechanism. The assignment consisted of ten stories, which needed to be implemented in the predefined order. Detailed acceptance tests consisted of more than 700 asserts. In PP2, the assignment was implementation of a chat server, consisting of 13 stories and accompanying acceptance tests with more than 1100 asserts. In both cases, the subjects were given acceptance tests in advance, since external quality was fixed and they were required to submit their code only after they reached 100% of acceptance tests passed.

The assignments with independent developers were limited to 4 h and therefore simpler, consisting of only two stories, the second representing a refinement of the first one (incremental development). In ID1, the subjects were given a minimal set of acceptance tests, similar to black box tests a customer would provide. However the projects were assessed later in detail with a set of very thorough acceptance tests, similar to white box tests a quality assurance department would provide. The minimal set of tests represented roughly 50% of detailed acceptance tests.

However, the availability of minimal acceptance tests was not a good idea since four developers in the ITL group have used these tests instead of writing their own tests. Since iterative development without writing tests does not adhere to our definition of ITL process, we had to exclude these cases from our research. We can see the developers under time pressure tend to omit writing tests, especially in ITL process. Therefore in the second experiment replication (ID2), the subjects were not given minimal acceptance tests. The projects were assessed later with detailed acceptance tests, again employing 50% success as the lowest acceptable threshold for a successful story implementation. The same criteria were used by Erdogmus et al. [3]. It is worth mentioning that we did not expect students to achieve 100% marks in detailed acceptance testing since tests were really thorough and indeed only one subject (ID1, TDD group) managed to obtain 100% mark, although many were higher than 90%.

Table 4 summarizes the essential properties of experiments.

Table 4
Comparison of experiments.

Independent developers (ID1, ID2)	Pair Programming (PP1, PP2)
Limited time (4 h)	Unlimited time (average spent: 26 h)
Variable amount of work (as many stories as possible)	Fixed amount of work (implement all stories with fixed external quality)
Incremental development	Incremental development
Short assignment (two stories)	Longer, more complex assignment (10 stories in PP1, 13 stories in PP2)
ID1 – minimal acceptance tests available	Detailed acceptance tests available
ID2 – no acceptance tests available	

At the end, the subjects were given another questionnaire, this time anonymous, asking about qualitative observations and subjective experiences with the development process and process conformance. The results were used later as an additional means of assessing process conformance, while they also enabled deeper insight into subjective experiences with TDD and ITL.

4.8. Power estimation

An important part of hypothesis testing in a controlled environment is evaluation of statistical power – probability that a test will correctly reject the null hypothesis. If the statistical power of the experiment is weak, the probability of finding significant effect is small. It is agreed that in the area of empirical software engineering there is inadequate reporting of statistical power analysis [36,37].

Statistical power is related to three other criteria:

(a) Statistical significance α is a probability that we wrongly reject a null hypothesis (Type I error). We set α to 0.05, which is common procedure in software engineering experiments.
(b) Sample size N. With higher sample size, at any value of α, the precision increases and statistical power is higher. However we had no choice regarding sample size since we employed all the students enrolled in the chosen course. The sample sizes across series and experiments are shown in Table 3.
(c) Effect size (Cohen's d [37]) refers to the real difference between null and alternative hypotheses. Cohen's d can be calculated from averages μ and standard deviations σ obtained from two independent samples.

Based on literature and our own experience, we decided to focus on large effects (Cohen's $d = 0.8$). Large effects, beside the fact we can detect them more reliably, are also practically more important. Let us consider a productivity related effect: if there was a statistically significant effect of small size, it could mean 0.05% higher productivity, which is completely irrelevant for practical application.

Our preliminary estimation of statistical power (a priori) was based on two-tailed test and the expected number of students (40), with $\alpha = 0.05$ and $d = 0.8$ and the result for ID context was 0.69. In PP context, the number of subjects (pairs) is halved; therefore also a priori statistical power was lower (0.4).

In the course of experiment, a few students chose not to take part and also later we had some dropouts, mainly due to violated process conformance, so post hoc statistical power is lower than our a priori estimated values. The results are shown in Table 5.

The value of 0.38 means that we have only 38% chance to find the effect, assuming one exists. We have tried to compensate for lowered statistical power with experiment replication. We also explicitly quote the numbers, which enables the readers to create their own interpretation of our results and their reliability. Further, when taking into account also the reported effect sizes, the interpretation becomes more resilient to errors originating from small sample size. We include meta-analysis of our results in the Analysis section, while the results are also precise enough to include our study in eventual future meta-analyses. It is also to be mentioned that in similar research papers, statistical power is often not reported or is even lower ([32] reports 0.17 with $\alpha = 0.05$, [5] reports 0.645 with $\alpha = 0.1$, [40] reports 0.4 with $\alpha = 0.05$, [6] reports 0.66 and 0.44 with $\alpha = 0.05$, etc.). Dyba et al. [37] discovered in their meta-analysis that an average power of software engineering experiments is two thirds for large effect sizes.

Table 5
Statistical power of experiments (post hoc values, $\alpha = 0.05$, effect size $d = 0.8$).

	PP1	PP2	ID1	ID2
Post-hoc power	0.38	0.32	0.43	0.58

5. Analysis

This section presents analysis of the experimental data.

Pre-test questionnaires were evaluated to make sure there are no significant differences between TDD and ITL groups in each series. No statistically significant differences between TDD and ITL groups were found (t-test, $\alpha = 0.05$, $p > 0.05$ for all tested parameters).

After that, we employed classical hypothesis testing, where hypotheses are about the differences between two independent means (two groups of results or measurements). We have tested normality of data distribution with Kolmogorov–Smirnov (with Lilliefors significance correction) and Shapiro–Wilk tests at $\alpha = 0.05$. Since it turned out the data are not normally distributed (the significances were ranging from 0.000 to 0.937 in Kolmogorov–Smirnov and from 0.000 to 0.0 = .927 with Shapiro–Wilk), we have used non-parametric Mann–Whitney U exact test [35] in later analysis, which is also well suited for use on smaller samples.

The results describe four experiments (ID1, ID2, PP1 and PP2) and five dependent variables: number of implemented user stories per hour (NIUSPH), percentage of acceptance tests passed (PATP), McCabe's code complexity (McCabe), branch coverage (BC) and mutation score indicator (MSI).

In PP1 and PP2 external quality was fixed, so all the projects needed to achieve 100% PATP before students could turn them in. Therefore, we do not include results on PATP in PP1 and PP2.

In PP1 and PP2 we had problems obtaining MSI without considerable alterations to experiment subject's code and/or tests. Typical problems and errors reported during running Judy were socket binding errors (multithreaded socket connection tests), memory leakage problems (e.g. out of memory errors) and too much processing time allocated to garbage collection. To avoid additional level of experiment mortality, we decided to omit MSI measurement in PP1 in PP2. ID projects were simpler and we had no problems obtaining MSI for those. Therefore we only include results for MSI in ID1 and ID2.

5.1. Descriptive statistics

For each experiment and metrics, we present average values (mean), standard deviation (st. dev.), standard error, maximal value, median value, minimal value as well as lower and upper bound of 95% confidence interval in Table 6.

The boxplot graphs, comparing the values graphically, can be seen in Fig. 3 (ID1), Fig. 4 (ID2), Fig. 5 (PP1) and Fig. 6 (PP2).

Table 6
Descriptive statistics of the whole experiment data.

Measure	Experiment, treatment	Mean	St. Dev.	St. Error	Max	Median	Min	95% CI lower bound	95% CI upper bound
NIUSPH	PP1 TDD	0.41	0.19	0.06	0.90	0.37	0.21	0.27	0.55
	PP1 ITL	0.44	0.27	0.09	1.16	0.36	0.29	0.23	0.65
	ID1 TDD	0.23	0.19	0.05	0.63	0.24	0.00	0.12	0.34
	ID1 ITL	0.31	0.12	0.04	0.53	0.25	0.24	0.22	0.40
	PP2 TDD	0.64	0.26	0.10	1.11	0.59	0.29	0.40	0.88
	PP2 ITL	0.61	0.26	0.09	1.16	0.60	0.24	0.41	0.82
	ID2 TDD	0.35	0.14	0.04	0.53	0.26	0.22	0.28	0.43
	ID2 ITL	0.30	0.14	0.03	0.56	0.26	0.00	0.23	0.36
BC	PP1 TDD	87.54	6.26	1.98	97.00	88.30	79.00	83.06	92.02
	PP1 ITL	88.86	5.08	1.69	98.00	88.90	83.00	84.95	92.76
	ID1 TDD	93.39	6.74	1.80	100.00	94.25	75.00	89.50	97.28
	ID1 ITL	95.66	5.88	1.96	100.00	97.40	82.00	91.14	100.17
	PP2 TDD	89.42	6.10	2.30	99.00	89.36	83.00	83.78	95.06
	PP2 ITL	84.61	8.90	2.97	99.00	80.90	74.00	77.77	91.46
	ID2 TDD	87.44	11.46	3.06	100.00	88.75	63.00	80.82	94.06
	ID2 ITL	80.78	11.89	2.80	100.00	82.70	55.00	74.87	86.70
McCabe	PP1 TDD	3.32	0.93	0.30	4.87	3.28	1.48	2.65	3.98
	PP1 ITL	4.27	3.09	1.03	12.36	3.00	2.73	1.89	6.64
	ID1 TDD	5.93	3.84	1.03	18.00	5.17	2.67	3.71	8.15
	ID1 ITL	6.59	3.26	1.09	12.33	5.60	3.00	4.09	9.10
	PP2 TDD	3.44	0.66	0.25	4.58	3.34	2.58	2.83	4.05
	PP2 ITL	19.73	44.48	14.83	138.00	3.86	2.26	−14.46	53.92
	ID2 TDD	2.63	0.97	0.26	5.23	2.29	1.53	2.07	3.19
	ID2 ITL	2.34	0.51	0.12	3.57	2.15	1.80	2.09	2.60
PATP	ID1 TDD	74.81	16.38	4.38	100.00	71.05	50.00	65.36	84.27
	ID1 ITL	86.84	5.42	1.81	95.00	89.47	76.00	82.67	91.01
	ID2 TDD	80.15	10.44	2.79	96.00	80.33	64.00	74.13	86.18
	ID2 ITL	78.23	18.16	4.28	98.00	82.38	27.00	69.20	87.26
MSI	ID1 TDD	55.79	17.46	4.67	80.00	61.00	22.00	45.70	65.87
	ID1 ITL	71.78	11.58	3.86	88.00	73.00	48.00	62.87	80.68
	ID2 TDD	59.64	8.10	2.17	70.00	60.00	45.00	54.97	64.32
	ID2 ITL	56.50	12.75	3.01	72.00	57.00	20.00	50.16	62.84

Looking at the boxplots, we can see a few outliers, however when we analyzed it case by case, we found no reasons to exclude them. We can see some differences between TDD and ITL from the diagrams and the values in the table, however it is hard to say more without deeper statistical analysis, which follows in the next section.

5.2. Hypothesis testing

After we found that the data is not distributed normally, we did not test further assumptions of parametric tests. We used non-parametric Mann–Whitney U Exact test for hypotheses testing ($\alpha = 0.05$): if two-tailed $p < 0.05$, we can reject the null hypothesis on equality of means and accept the alternative hypothesis.

The results of our analysis are presented in Table 7. Beside Mann–Whitney test statistics, Wilcoxon W, Z and exact significance p for two-tailed tests, we include effect sizes detected in our experiments. For convenience, we include Cohen's size of effect d, since this metrics is used in most of the software engineering experiments. For further calculations, we also include another measure of effect size, Pearson product-moment correlation coefficient r, which is calculated from d and sample sizes, as outlined in [56].

We can see that most of the tests are statistically insignificant. In our experiments, significant differences between groups are found only in ID1 regarding MSI, where the effect is in negative direction. However ID1 results are questionable due to high mortality in both groups; we suspect the results are biased. Our doubts are explained more in detail in Section 6.1 – internal validity. Here let us point out that none of the medium or large effects found in ID1 is confirmed in any of the remaining three experiments.

The rest of p-values are above 0.05, however due to weak statistical power it is possible that effects exist and we were unable to detect them.

Effect sizes are more robust to small sample sizes than p-values. Cohen's d is an indication of the standardized difference between two means, where values around 0.2 are considered to describe small effects, 0.5 medium and 0.8 large effects. Madeyski [56] suggests using the intervals, defined by Kampenes et al. in order to characterize the real size of effects more reliably: values of r lower than 0.193 denote small effects, values from 0.193 to 0.456 medium effects and values from 0.456 and higher denote large effect. Negative effect values mean that control group (ITL) performed better than test group (TDD).

In our experiments, we found one large effect – again in ID 1 and regarding MSI. Additionally, we found several medium-size effects: in ID1 regarding NIUSPH and PATP, both in negative direction; in ID2 regarding NIUSPH and BC, both in positive direction; in PP1 regarding McCabe complexity in positive direction; in PP2 regarding McCabe complexity again in positive direction and in PP2 regarding BC in positive direction. The rest of effects are small. Overview of medium and large effects is presented in Table 8. Only large (L) and medium (M) effects are shown; effects in negative direction are indicated by minus sign (−) and positive effects with plus (+) sign. ID1 results are under suspicion of mortality bias. The rest of the effects are small.

5.3. Meta-analysis

Since our results were mainly inconclusive regarding the significance of results and the power of our experimens is somehow low, we combined the results of our experiments to achieve higher

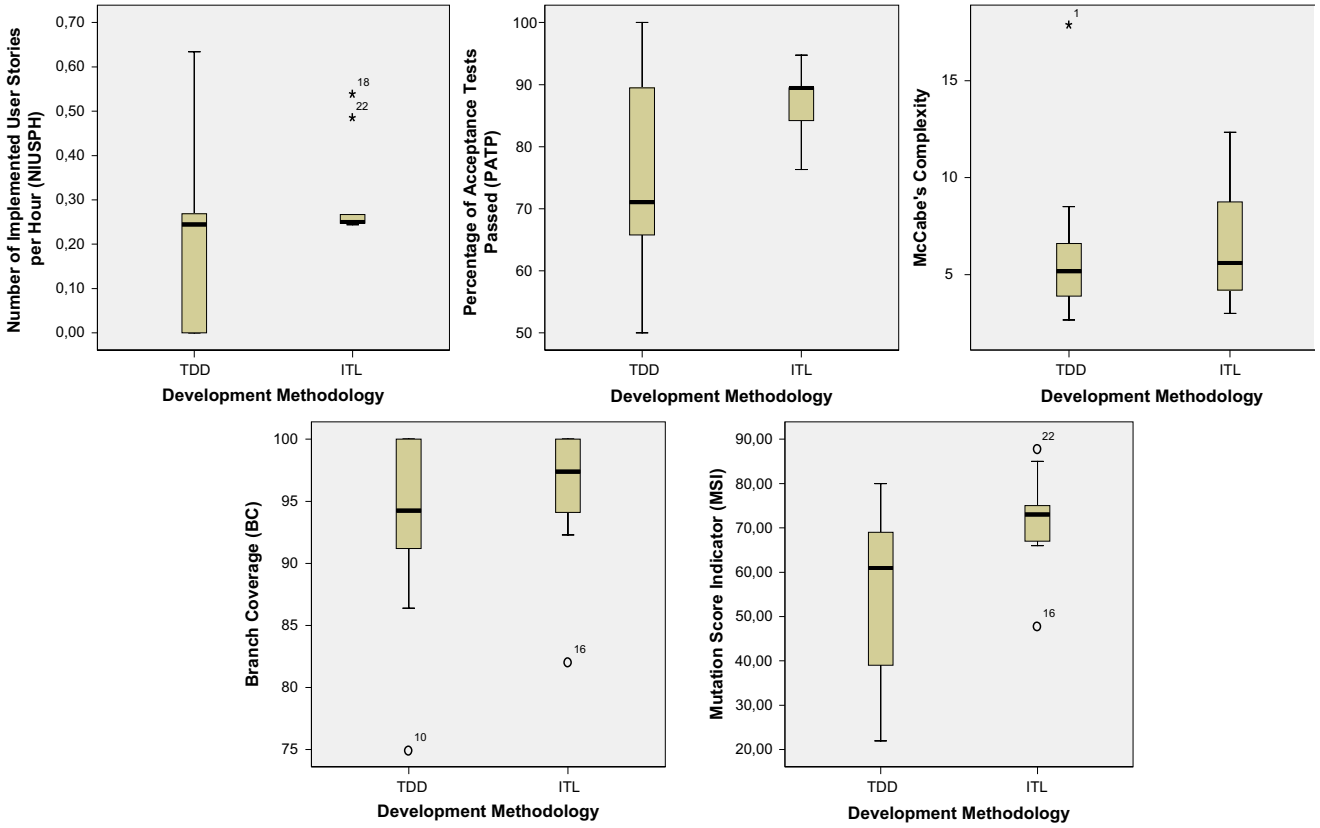

Fig. 3. Boxplot graphs for ID1.

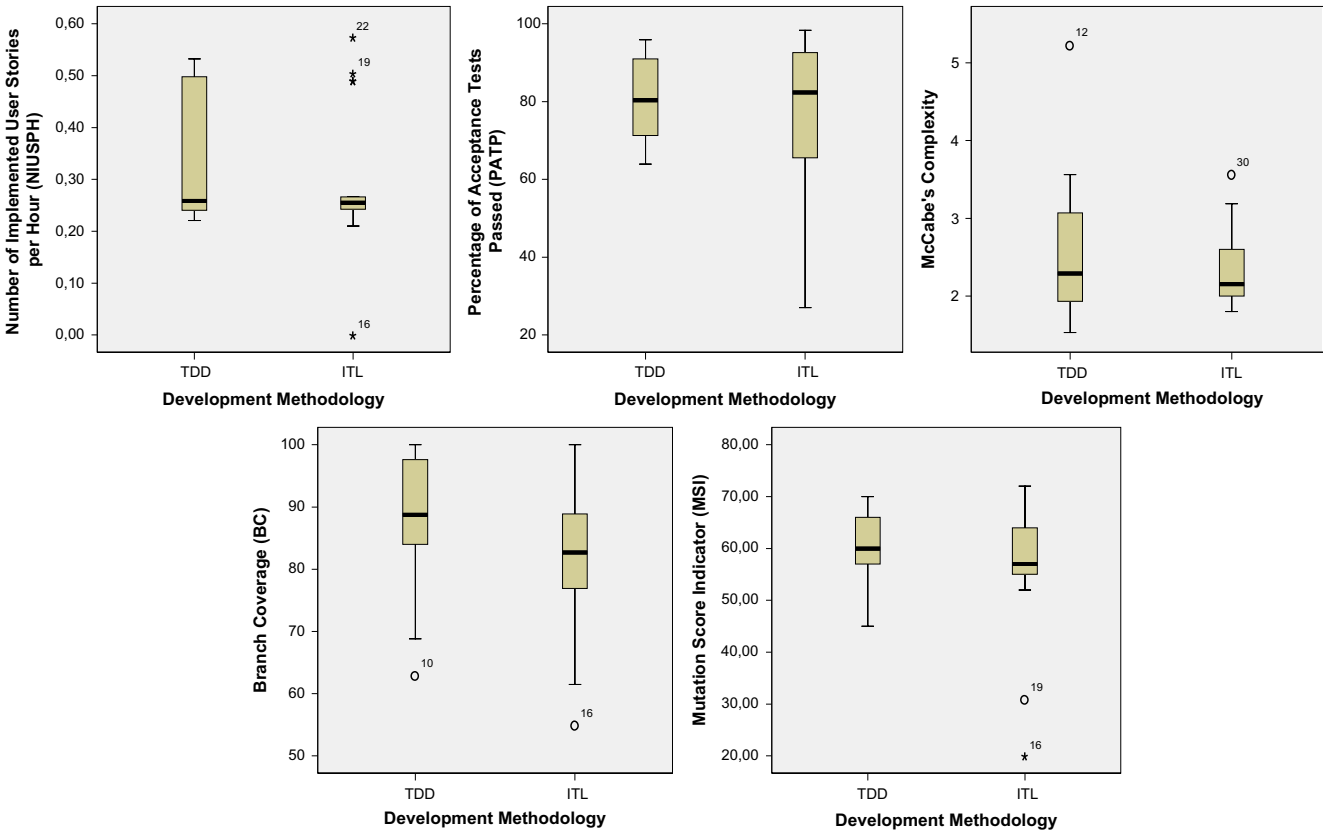

Fig. 4. Boxplot graphs for ID2.

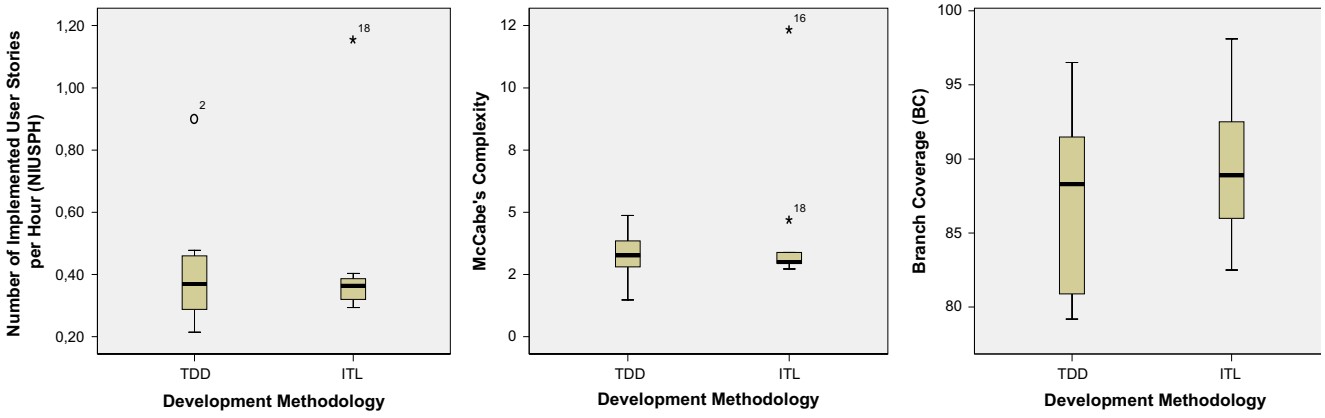

Fig. 5. Boxplot graphs for PP1.

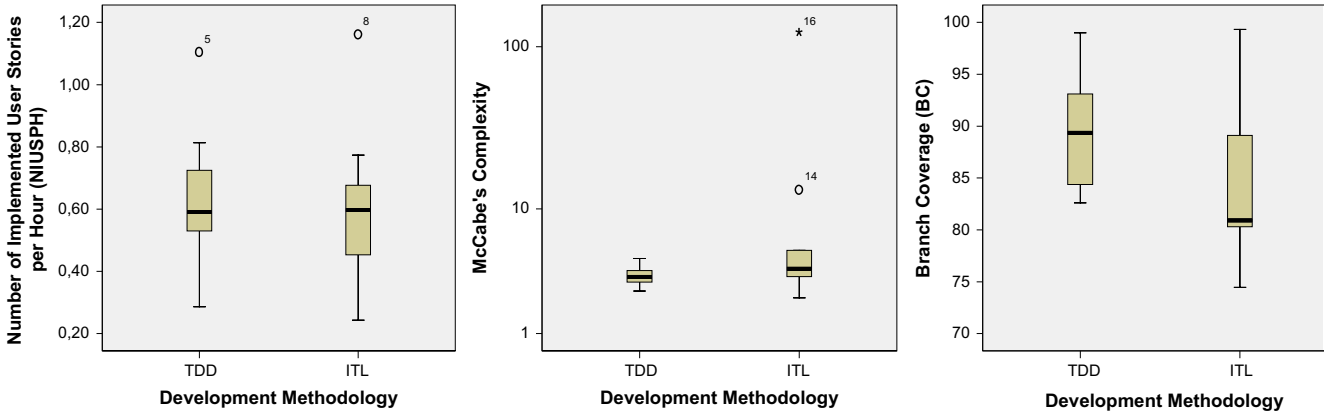

Fig. 6. Boxplot graphs for PP2.

Table 7
Mann–Whitney test statistics (TDD vs. ITL) and effect sizes.

		NIUSPH	PATP	McCabe	BC	MSI
ID1	Mann–Whitney U	46.000	38.500	49.500	47.000	25.000
	Wilcoxon W	151.000	143.500	154.500	152.000	130.000
	Z	−1.074	−1.558	−0.853	−1.030	−2.395
	Exact Sig. (2-tailed)	0.296	0.125	0.410	0.317	**0.015**
	Effect size *d* (Cohen)	−0.534	−0.982	0.187	−0.346	−1.079
	Effect size r	−0.263	−0.448	0.095	−0.174	−0.483
ID2	Mann–Whitney U	117.000	125.000	116.000	82.000	104.500
	Wilcoxon W	288.000	230.000	287.000	253.000	275.500
	Z	−0.342	−0.038	−0.380	−1.673	−0.819
	Exact Sig. (2-tailed)	0.750	0.978	0.722	0.097	0.424
	Effect size *d* (Cohen)	0.430	0.130	−0.375	0.570	0.294
	Effect size r	0.215	0.066	−0.189	0.280	0.149
PP1	Mann–Whitney U	44.000		43.500	39.000	
	Wilcoxon W	89.000		88.500	94.000	
	Z	−0.082		−0.123	−0.490	
	Exact Sig. (2-tailed)	0.968		0.922	0.661	
	Effect size *d* (Cohen)	−0.106		0.416	−0.231	
	Effect size r	−0.056		0.215	−0.121	
PP2	Mann–Whitney U	31.000		19.000	16.000	
	Wilcoxon W	76.000		47.000	61.000	
	Z	−0.053		−1.323	−1.641	
	Exact Sig. (2-tailed)	1.000		0.210	0.114	
	Effect size *d* (Cohen)	0.113		0.518	0.643	
	Effect size r	0.060		0.265	0.323	

Table 8
Overview of perceived effect sizes.

	NIUSPH	PATP	MCCABE	BC	MSI
ID1	−M	−M			−L
ID2	+M			+M	
PP1			+M		
PP2			+M	+M	

power and maybe obtain statistically significant results. The analysis was performed in SPSS with additional meta-analysis macros from Marta Garcia Graniero. The theory of meta-analysis and its use in software engineering are presented in depth in [56] and will not be repeated here.

Combination of *p*-values accross a set of experiments or studies can be performed by means of several methods. Madeyski [56] suggests using Fisher's method, combining disparate *p*-values into a new test statistic with χ^2 distribution, since it can offer more power than other methods. The method can be used when the samples are independent, in one-tailed tests all *p*-values need to refer to the same tail, while in two-tailed tests we combine one-tailed *p* values and afterwards convert combined one-tailed *p* into a 2-tailed combined *p* simply with multiplication by 2. In our case, samples were independent and all tests are two-tailed, so we may use Fisher's method. For comparison we also include results of Mudholkar and George method, based on transforming *p*-valuse into logits, and Stouffer's method, based on *Z*-values, (unweighted – UW, weighted by weighting factor – WWF and weighted by square roots of weighting factors – WWS).

Beside combined *p*-values we are also interested in the combined effect sizes. There are two models for calculating a combined effect size: fixed effects model is used when we assume all the effect sizes to be combined estimate the same population mean. When we know the populations are different or when the homogeneity analysis indicates heterogeneity, we need to use random effects model. In our case, only MSI and McCabe effects indicated homogenity, so we needed to use random effects model. Beside that, we are aware of differences in ID and PP experiments, namely using individual or pair programming. This can be a substantive source of heterogeneity, so even when the homogeneity test does not show heterogeneity, we are safer with random effects model. We use Wolf's method for calculation of fixed effects and Dersimonian–Laird's method for calculation of random effects.

Meta-analysis found two statistically significant differences between groups (Table 9). Branch coverage in TDD is significally higher while MSI is significantly lower in TDD. In both cases, the combined effect sizes are small. The finding is somewhat confusing, since BC and MSI are both tightly related to test quality. When test code covers more branches, one should expect it also catches more faults in the code. Moreover, one would have expected TDD would affect them both in the positive direction. To explain this effect, we raise the question of validity of ID1 results again. In this experiment, in ITL group the students with lower academic grades dropped out, while in TDD group the drop-out was at the top with academic grades (excluded becaudse of poor process conformance). The post-mortem differences were not statistically significant though (see Section 6.1 – mortality bias). If the groups were out of balance and ITL group would have better performing students than TDD group, this could explain the anomalies in the results.

To be on the safe side and to ensure against wrong interpretation of the results, we decided to run a selective meta-analysis. We excluded ID1 and rerun the meta-analysis on the remaining three experiments (ID2, PP1, PP2) for dependent variables NIUSPH, McCabe and BC.

For MSI and PATP, we ran out of data for meta-analysis after we excluded ID1. Since MSI is a new measure which still has to find its place in software engineering research, we tried to find a comparison for our findings in the existing literature. The only published study that we know of considering MSI as an indicator of tests fault-finding capabilities from Madeyski [6,56], found the effects of TDD on MSI minor although positive, which is consistent with our findings from ID2. Since the experiment is similar to ours and the MSI metric is still relatively new, we decided to perform meta-analysis of ID2 and experiment from Madeyski to asses combined *p*-value and effect size. However, please note that possible threat to the validity of MSI meta-analysis result is also coarser granularity of the control group process in Madeyski's experiment. The results of selective meta-analysis can be found in Table 10.

Branch coverage is significantly higher in TDD than in ITL according to Mudholkar and George method and all three versions of Stouffer's method, while according to Fisher's method, *p*-value is just above 0.05 and the differences are therefore insignificant. The effect size is almost medium and it is in the anticipated direction. Although these results strongly indicate branch coverage is higher in TDD, we are left with inconclusive results and only further research could provide the definitive answer.

Differences regarding the number of implemented user stories per hour and McCabe's code complexity are insignificant and both effects are considered small.

It turns out that we cannot reject any of our null hypotheses PROD, QUALE, COMPL, COVER and MSI even after meta-analysis; however there is an indication that branch coverage might be higher in TDD.

6. Threats to experiment validity

In this section, we review and discuss any possible threats to validity of our experiment. Experiment validity can be viewed from four viewpoints according to [28]:

Table 9
Meta-analysis of *p*-values (2-tailed combined *p*-values are given) and effect sizes (correlation coefficients *r* and their 95% confidence intervals are given).

		NIUSPH	PATP	McCabe	BC	MSI
	Experiments combined:	ID1, ID2, PP1, PP2	ID1, ID2	ID1,ID2, PP1, PP2	ID1,ID2, PP1, PP2	ID1, ID2
Meta-analysis of *p*-values	Fisher's: 2-tailed *p*	.7506	.2743	.3744	**.0475**	**.0237**
	Fisher's method: χ^2 (2 K)	8.6204	6.9760	11.2635	17.6801	12.8880
	Mudholkar&George: 2-tailed *p*	.5096	.2659	.2356	**.0274**	**.0206**
	Stouffer's UW: 2-tailed *p*	.4827	.2695	.2057	**.0193**	**.0223**
	Stouffer's WWF: 2-tailed *p*	.4525	.3587	.2620	**.0185**	**.0386**
	Stouffer's WWS: 2-tailed *p*	.4613	.3110	.2293	**.0181**	**.0291**
Meta-analysis of effect size	Fixed effects: *r*	.011	−.156	.043	.093	−.125
	Fixed effects: 95% CI	[−.208, .228]	[−.412, .122]	[−.177, .259]	[−.128, .305]	[−.385, .153]
	Random effects: *r*	.011	−.192	.043	.086	−.175
	Random effects: 95% CI	[−.208, .228]	[−.624, .330]	[−.177, .259]	[−.173, .334]	[−.686, .451]

Table 10
Selective meta-analysis of *p*-values and effect sizes on ID2, PP1 and PP2 for NIUSPH, McCabe complexity and branch coverage, and on ID2 and experiment SUBMISSION from Madeyski [6,56] for MSI.

		NIUSPH	McCabe	BC	MSI
	Experiments combined (K)	ID2, PP1, PP2	ID1, PP1, PP2	ID2, P1, PP2	ID2, SUBM
Meta-analysis of *p*-values	Fisher's method: 2-tailed p	1.000[a]	.4626	.0594	.8655
	Fisher's method: χ^2 (2 K)	4.7993	8.0941	13.9961	3.8071
	Mudholkar&George: 2-tailed *p*	.8487	.3462	**.0418**	.8520
	Stouffer's UW: 2-tailed *p*	.8359	.3243	**.0337**	.8506
	Stouffer's WWF: 2-tailed *p*	.7868	.4110	**.0323**	.6799
	Stouffer's WWS: 2-tailed *p*	.8091	.3625	**.0313**	.7581
Meta-analysis of effect size	Fixed effects: *r*	.107	.026	,183	.128
	Fixed effects: 95% CI	[−.149, .349]	[−.228, .276]	[−.072, .416]	[−.162,.398]
	Random effects: *r*	.107	.048	.182	.128
	Random effects: 95% CI	[−.149, .349]	[−.254, .341]	[−.081, .421]	[−.162, .398]

[a] 1-tailed $p = 0.5698$.

1. Internal validity: are there any issues that may indicate a causal relationship, although there is none?
2. Construct validity: do the measurements really measure what they were intended to measure?
3. Conclusion validity: there are no logical mistakes in conclusions analysis.
4. External validity: can the results be generalized over other subjects, settings etc.?

In the following subsections, we review the threats to all four kinds of validity and discuss the options to deal with them.

6.1. Internal validity

Internal experiment validity is good when we haven't introduced any **covert variables** into the experiment. Covert variables represent factors that may affect dependent variables without our knowledge. Our main concern regarding covert variables was to isolate test-first or test-last development from highly iterative (fine grained) or more traditional (coarse grained) development process. The development process in the control group is often described only superficially in the available literature so that it is impossible to tell whether iterative test-last or traditional test-last development was used. With this paper it is our intention to attract more attention to the control group development process. As already stated earlier, we have dismissed this threat by employing the same degree of iterativity in both groups, TDD and ITL. Thus any differences found would have to be accounted only to writing tests before or after writing code and not to the granularity of development process.

Although in ID and PP experiments the subjects did not have same kind of acceptance tests available (minimal or no tests vs. detailed tests), with different availability of acceptance tests we believe we covered more real-life situations and enabled the result to be more robust and generalizable. However this also introduces an additional threat to validity of results, since we cannot know if the results are due to independent variables or maybe due to different availability of acceptance tests. Since no important differences are found between PP and ID results, we believe that the benefit of more generalizable conclusions outweighs the possible introduction of additional threat to validity.

Further eventual threats to internal validity are not known only in the software engineering experiments. In our case, most of them are dismissed by random assignment of subjects to treatments.

- **Hawthorne effect:** Some people work harder and perform better when they know they are being watched, while others might react oppositely. In our case we investigate relative differences between balanced groups, so it is likely that the effect, although affecting both groups, will not affect the difference. Also, later researches suggest that the original results regarding Hawthorne effect (reporting as much as 10% increase in productivity) have been overstated and the effect is weak at best [39].
- Other **social threats** (hypothesis guessing and experimenter expectancies – the researcher and his/her beliefs and expectations affect the subjects and thus also the result) are considered to be low since neither the experimenters nor the subject have any interest in favoring one treatment over the other.
- Lack of objectivity due to **mortality** (mortality bias, attrition): it occurs when an experimental sample is not representative any more due to excessive subject dropouts. Mortality problem appears in almost every experiment in empirical software engineering. In our case, the problem arose in experiment ID1, where ITL group experienced 50% mortality (9 subjects remained out of 18). This is in sharp contrast with TDD group with only 30% mortality (one subject opted out, 5 more were excluded due to poor TDD process conformance; the initial number was 20). That left us not only with unbalanced groups: 14 TDD vs. 9 ITL, but also with a strong suspicions of mortality bias, because in ITL group, all students with low skill ratings and low academic grades had dropped out (e.g. all in the lowest 25%!). In TDD group, on the other hand, three out of the five dropouts had high academic grades. Although post-mortem analysis still did not show significant differences between ITL and TDD group regarding mean grades at $\alpha = 0.05$ (*t-test*, $t(21) = -0.8$, $p = 0.43$, $r = -0.180$) the sample was small and the power of *t*-test is low (only 30% probability to detect a medium sized ($d = 0.5$) difference among groups, if one exists, at $\alpha = 0.05$). We think ID1 should be excluded from the experiment, however for the sake of completeness we include the results of ID1 in the paper and include our doubts in the discussion so that the readers are able to make their own interpretation.
- **Control group bias** (compensatory rivalry): the threat resembles Hawthorne effect. When the control group subjects know they belong to control group, they might feel less important and act differently as they would otherwise. And, if TDD subjects knew that we think they may produce better code, they might work harder. We have dismissed this threat by not communicating our hypotheses to subjects and by paying the same attention to subjects from both groups. They were only told we are comparing two development processes and that there is no evidence yet on which one is better.
- **Diffusion (imitation) of treatments** was prevented by periodically reminding the subjects that there is no need to imitate the other group. Additionally, we have checked the code history after the experiment to make sure that tests were written and run before code in TDD and vice versa in ITL group.

6.2. External validity

External validity of the study is good when the conclusions can be generalized to cover the whole population or other populations and contexts of use. In our research, an ideal experiment would last a few years, and two identical groups of professional developers in the role of test subjects would be solving the same problem simultaneously. However such experiment would be highly impractical and overly expensive. On the other hand, several studies suggest that results of experiments with student developers can be extrapolated to professional developers, especially with senior or graduate students ([41] and [42]). Tichy et al. [43] identify situations where it is acceptable to have students as experimental subjects: when students are properly trained, when the purpose of the study is assessment of trends (for example, comparison of two methods – if one of them is better in student population, it will probably be better in professional population too), when the purpose of the study is elimination of alternative hypotheses, or when the results are used to attract professional developers to further study. In our experiment design, we control this threat by following all the recommendations from Tichy. The students are in the fourth (last) year of study, they have been carefully prepared and trained for the experiment, as described earlier in the paper, and the goal of study is comparison of two methods.

Carver et al. [44] point out that similarly as students are sometimes not fully representative of the professional population, also professionals in one environment may not be representative of professionals in other environments, even within the same application domain. However the results may still be generalized to a certain extent after the external validity issues are considered. The authors provide a list of nine requirements, helping to achieve highest research and pedagogical value from the experiment. Although our experiment was already finished when their paper was published, we have carefully examined the list and realized that we met all but one requirement, which was met partly.

Like in other similar studies, our weakness regarding external validity is relative simplicity of experimental assignments in comparison to typical industry-level projects. However it is in our favor that both processes, TDD and ITL, are highly iterative: only a small piece of functionality is implemented and tested at a time. From this point of view, our results may be extrapolated to longer lasting iterative projects, where the number of stories is higher, but the programmer still implements a single piece of a story at a time, either preceded or followed by the development of corresponding test code. Since professionals have more practical skills, the effect sizes might differ though.

6.3. Construct validity

Discussions on threats to construct validity examine the reasons why inferences about the construct that characterize study operations might be incorrect or, in other words, how do we know we are measuring what we think we are measuring [45]?

In IEEE standard 1061 on software quality metrics methodology, direct measurements have a status of inherent validity, while derived metrics are validated with respect to direct measurements. However, many attributes we study do not have either direct or derived generally agreed methods of measurement. Instead, we often use alternate measures, presumed to be related to the actual attributes. In the rest of this section, we explain our choice of metrics and their relation to the attributes that they are supposed to measure.

The metrics used in this experiment are:

- For productivity: number of successfully implemented user stories per hour.
- For external quality: percentage of successful acceptance tests.
- For code complexity: McCabe's code complexity [50].
- For tests thoroughness: branch coverage [51].
- For fault-finding capability of tests: mutation score indicator [6].

Productivity refers to the amount of work, accomplished in a certain amount of time. We use it to evaluate subjects' performance. We know how to measure time and also have a means to log the actual values (our ProcessLog plug-in). Productivity is observed in two different contexts: in PP context, the amount of work is fixed (all the stories need to be implemented), while in ID context, time is limited however it is not fixed. The amount of programming work could be measured directly in lines of code, but one could argue about what is a line and what is code and how do they relate to the finished software product. We control this threat by focusing on the functionality or stories to be implemented: the more user stories implemented, the higher is amount of work. From this point of view, also the number of stories is considered a direct measurement and our measure of productivity is number of user stories implemented per hour.

Possible threats to productivity measurement are inconsistent use of ProcessLog and inequality among the amount of work required to accomplish different stories. To dismiss the first threat, we trained the students and made sure they knew how to use the ProcessLog correctly and understood the importance of the measurements. We also asked them to report eventual mistakes, and after the experiment, we have reviewed the numbers carefully, looking for eventual obvious discrepancies. To dismiss the second threat, we took every effort to define balanced stories and additionally we required the stories to be implemented in the predefined order, thus making the XP planning game irrelevant. This way it cannot happen that one subject implemented three shortest stories and the other subject implemented three longest ones and we wouldn't notice the difference in productivity.

Possible threats regarding measurement of percentage of acceptance tests passed are inconsistent use of Process Log (discussed above) and unbalanced coverage of requirements with the acceptance tests. If for example a piece of functionality was not covered by acceptance tests, two software products might be considered of equal quality although only one of them would implement this functionality. We controlled this threat by writing precise acceptance tests and double-checked them with the assignment requirements so that no requirements were left untested.

The drawback of McCabe's cyclomatic complexity is its limitation to syntactic aspect and unclear relation to overall code quality. We controlled this threat by not overemphasizing the results and combining the interpretation of McCabe's complexity with the results of PATP as they both evaluate code properties.

Branch coverage is a well-established concept for evaluating tests thoroughness [49,6,56]. It is true that branch coverage measure indirectly benefits TDD since in TDD the programmers are not supposed to write any code that does not have tests implemented yet. As a threat to using branch coverage let us stress that it only measures how well are all the logical decisions covered by tests, and nothing about tests correctness. However correct tests are able to detect faults in the code and the ability to detect faults is measured by means of mutation score indicator. So again to control this threat, joint interpretation of BC and MSI is needed as they both evaluate test properties.

6.3.1. Mono-operation bias

When the experiment is performed only once, the conclusions drawn may refer to the particular experiment execution, not to the actual construct. We have controlled mono-operation bias by repeating each experiment with different experimental assign-

ments and different subjects in two consecutive years. Additionally, each experiment was performed in two different contexts (pair programming and individual programming), thus further reducing the possibility of mono-operation bias. The results were combined by means of meta-analysis.

6.3.2. Bias due to inconsistencies with the development process

If the subjects did not adhere to the agreed development process for their group, the results would likely be biased (threat, related to the reliability of treatment implementation). To eliminate this threat, we have trained the students well and made sure they understood the importance of the process. We gave them appropriate theoretical foundations and practical training, as described earlier in the paper. Further, we have analyzed the resulting code, including detailed code history, and applied objective measures (described in the section 0 – Experiment operation) on the basis of which we excluded the subjects who did not adhere to the predefined development process in the group they have belonged to.

6.4. Conclusion validity

Conclusion validity is concerned with the possibility to draw correct conclusions regarding the relationship between treatment and outcome of the experiment.

The reliability of measurements was controlled by choice of objective measures, independent of researcher's subjective estimations.

The main issue is statistical power of the tests, which is discussed in section 0 - Statistical power. The values of around 0.8 are considered sufficient to show real effects, however several experiments with considerably lower power are reported in the available software engineering literature. Since we were limited in the number of subjects with the number of students enrolled in the course, our experiment has limited statistical power: it is possible there is a significant effect however we were unable to detect it, so the results need to be interpreted correspondingly. Moreover, we found the data non-normally distributed and we needed to use non-parametric statistical test, which is less sensitive compared to parametric tests in a sense that there may be a difference but the test may not discover it. On the other hand, we combined the results of the experiments by means of meta-analysis and thus achieved somewhat higher statistical power.

But also meta-analysis has a few important threats. Inadequate conceptualization of the problem relates to the fact one can perform meta-analysis on experiments having fundamental conceptual differences (regarding subjects, measurements, design, etc.). In such cases the results may be doubtful. In our meta-analysis, conceptualization is under control because we conducted all the experiments under similar conditions. However the problem with inadequate conceptualization may appear in meta-analysis of our results on MSI with results reported by Madeyski, since in our control group micro-iterative test-last development is used, while in Madeyski's experiment, the granularity of control group process is coarser.

Similarly, inadequate assessment of study quality could result in overestimation of the suspicious results in the combined result. We have carefully assessed experiments quality and exposed possible problems with ID1. To keep the threat under control, we have also performed selective meta-analysis with ID1 excluded and interpreted the results correspondingly so that each reader can create his or her own opinion.

Otherwise, the threats to the statistical conclusion validity are considered to be under control, since robust statistical methods and reliable tools (SPSS) are used.

7. Discussion and conclusions

In this section we present a short summary of the results of the experiments, give interpretation of the results, compare them with previous experiments and also provide joint interpretation.

The research findings come from an experiment, conducted in academic environment in two contexts (individual and pair programming) and repeated in two consecutive years, so that it yielded four sets of results: ID1, ID2, PP1 and PP2. Due to high mortality we think ID1 results are biased (although no statistically significant differences could be detected in the post hoc analysis regarding mean grades (t-test, $t(21) = -0.8$, $p = 0.43$, $r = -0.180$)) and are thus not included in this summary (see Section 6.1 Internal validity).

No statistically significant differences could be found in any context and in any series regarding the dependent variables NIUSPH (number of implemented user stories per hour), PATP (percent of acceptance tests passed), McCabe (McCabe's code complexity), BC (branch coverage) or MSI (mutation score indicator). However due to lower statistical power of our experiments (0.58 for ID2, 0.38 for PP1 and 0.32 for PP2) it is possible effect exists while we were unable to detect them. We found medium effect sizes in ID2 (positive effect on NIUSPH, $r = 0.215$), PP1 and PP2 (positive effect on McCabe, $r = 0.215$ and $r = 0.265$, respectively), ID2 and PP2 (positive effect on BC, $r = 0.28$ and $r = 0.323$, respectively), although none of these were statistically significant.

To increase our chances to find effects if they exist, we performed meta-analysis of our results. Selective meta-analysis found statistically significant differences in branch coverage using four out of the five used methods (Mudholkar&George: $t(5\,k + 4) = t(19) = 2.1834$, $p = 0.0418$; Stouffer's unweighted: $p = 0.0337$, $r = 0.2595$; Stouffer's WWF: $p = 0.0323$, $r = 0{,}2616$; Stouffer's WWS $p = 0.0313$, $r = 0.2630$), while Fisher's method returned p-value just above 0.05 (($\chi^2(2\,k) = \chi^2(6) = 13.996$, $p = .0594$). This means there are strong indications that TDD might positively affect branch coverage.

A possible explanation for better branch coverage is that by its nature, TDD requires writing tests first and only coding the functionality that is already covered by tests. However using TDD does not automatically mean complete branch coverage. This explanation is also supported by our observation that the developers under time pressure (in both ID experiments) tend to omit writing tests in ITL process, while this was not the case in TDD process. Since our experiment was not designed to prove this effect, we recommend the topic as possible material for further research.

We have also performed a meta-analysis of our ID2 results and Madeyski's results [6] regarding the effects of TDD on MSI. We found the combined effect is positive and small ($r = 0.128$), while the difference is still statistically insignificant ($\chi^2(4) = 3.807$, $p = .87$).

According to our basic analysis and selective meta-analysis results, regarding perceived effect sizes our findings are as follows:

- The effect of TDD on number of implemented user stories per hour (NIUSPH) is small and in positive direction ($r = .107$, 95% CI: $-.149$ to .349).
- The effect of TDD on McCabe's code complexity is small and in positive direction ($r = .048$, 95% CI: $-.254$ to .341).
- The effect of TDD on percentage of acceptance tests passed (PATP) is small and in positive direction (based only on ID2, $r = .066$).
- The effect of TDD on branch coverage is almost medium and in positive direction ($r = .182$, 95% CI: $-.081$ to .421).
- The effect of TDD on mutation score indicator (MSI) is small and in positive direction (based either only on ID2, $r = 0.149$, or on selective meta-analysis, ($r = .128$, 95% CI: $-.162$ to .398)).

None of the above effects is statistically significant at $\alpha = 0.05$.

To summarize, we did not detect any statistically significant differences between TDD and ITL regarding the observed five dependent variables, although there is an indication that branch coverage might be higher in TDD. All perceived effect sizes are small, although the effect on branch coverage is on the verge of being medium.

Although we have conducted the experiments in two different contexts, no important differences were found among results of ID and PP context. Since ID1 is under suspicion of mortality bias, we exclude its results from this discussion. Regarding productivity, there was a medium positive effect on number of implemented user stories per hour in ID2, however it was far from significant ($p = 0.75$). In PP1 and PP2, effects on productivity were small and again insignificant. Regarding McCabe code complexity, although the results were insignificant, in both PP experiments medium sized effects in positive direction were detected. This could mean that, providing there is enough time for development (time was not limited in PP experiments), developers will refactor code more thoroughly and thus achieve lower complexity than in ITL process. However under time pressure (in ID2), this effect dissolves. Regarding branch coverage, medium effects are found in ID2 and PP2, both in positive direction, which is consistent with the inherent property of TDD that it seems to foster higher code coverage than ITL.

Since our results are not statistically significant, we can agree with Madeyski [56] that pre-existing differences among subjects are more important predictors of outcome than development technique. In our case, beside individual differences among subjects there are also differences between ID and PP context: the result can also be affected by individual or pair development, by fixed external quality or by time pressure. Since our experiment was not designed to prove these differences, this should be taken into account when interpreting the results.

A plausible explanation form small effects found is that in post-test questionnaire and in informal contacts with subjects it came up that they found TDD hard to learn with a steep learning curve, which was not the case with ITL. So even after ten weeks of training it is possible that TDD group did not have enough practice to develop its full potential to exhibit eventual bigger benefits behind TDD. We believe that when used by more experienced subjects, the positive effects of TDD could be larger.

Let us compare our results with other researchers' findings.

TDD proponents claim that tight interleaving of testing and coding phases have beneficial effects on productivity. In an overview from Jeffries [2], from 19 studies on TDD, ten report on increased effort, one on increased cost, two on improved productivity, two on no effect, while four results are inconclusive or unavailable. Negative effect of TDD on productivity was found by George and Williams [8], Bhat and Nagappan [17], Madeyski [56], which contradicts with our finding that the effect of TDD is small although positive. However positive effect is found by Erdogmus et al. [3], Flor and Schneider [32], Gupta and Jalote [57] and Huang and Holcombe [4]. Although productivity is measured in different ways and the findings are rarely statistically significant, while TDD is compared to either test-last or traditional or even generic non-TDD approach, we believe the effect on productivity might be related to the amount of testing: the more test code is written, less time remains for writing production code. It would be interesting to know if there is any substantial correlation among productivity and the ratio of tests and code.

Production code properties were measured by means of PATP and McCabe code complexity. Janzen and Saiedian [55] found that code complexity is significantly higher in test-last group in two out of the six analyzed cases. This is consistent with our finding, where the effect of TDD on complexity is positive, although very small.

Several studies imply external quality could be higher in TDD in terms of more acceptance tests passed [24,8,54], while other found no differences [3,16] or negative effect in TDD [7,3,56]. These results seem inconsistent again and while different metrics are used, also the results are difficult to compare. Our results show that the effect on PATP is statistically insignificant and very small, although positive, which is consistent with [24,8,54] in a sense of direction and with [3,16,56] in a sense that the effect of TDD is insignificant and small.

Our result on better code coverage in TDD group is consistent with [17,21,56], however other studies, for example our pilot study [7,32] do not agree. Better code coverage can be explained with an XP rule stating that the programmer should only write the code that is absolutely necessary to pass the test. Strictly viewed, this rule should guarantee 100% code coverage. It is easier to leave a method or a branch uncovered (untested) in ITL since production code is written first and the programmer might simply forget or skip writing a test for a little piece of functionality, especially if the code "seems" to work.

However also ITL programmers can be thorough and ITL process does not prevent writing detailed tests in any way. Hence the difference in research results (not all of the studies finding better coverage in TDD groups) is understandable: the thoroughness of ITL groups may vary as well as the amount of attention the experimenter pays to the control group process.

Since also our controlled experiments presented in this paper found no significant differences regarding external quality (PATP), productivity and tests properties between TDD and ITL, we are challenged with the assumption that the few studies showing positive effects of TDD on productivity and PATP might be affected by the granularity of their control group process. If we speculate that all of them employed coarse grained test-last development in the control group, the benefits found in TDD group might be caused by short development cycles instead of writing tests before code. And the rest of the research field would seem more consistent, stating there is little or no real evidence proving superiority of TDD compared to fine grained test (iterative) test last.

One of the possible explanations is already exposed earlier in discussion, namely the tendency to omit detailed test writing in ITL. Because of writing tests in advance, TDD "forces" developers to think more about testing and also how to write code that is easier to test. In other words, in ITL it is easier for developers to be sloppy and somehow forget about rigorous post-code testing. When using coarse-grained process in control group, it is even easier to forget about including all the details in the test suite since more time has passed since writing code. The result is lower branch coverage in control group and possibly also lower external quality as a result of eventual undetected errors in the uncovered branches. With higher iterativity, the phases of coding and testing are more tightly interleaved and programmers only focus on one piece of functionality at a time. Consequently, tests are more thorough, more branches are covered, more errors are found and removed and final code quality is higher.

However these explanations are only hypothetical and further research is needed to explore their viability. Based on our experiences, future work to further strengthen or dispute our findings would be the following: replication and controlled experiment in different contexts and with bigger sample sizes, automated process adherence testing [52], longer and more industrial type of assignments or assignments with professional developers. More empirical research on isolating high iterativity from writing tests first is also needed.

References

[1] K. Beck, Test-Driven Development: By Example, Addison-Wesley (2003).

[2] R. Jeffries, G. Melnik, Guest Editors' Introduction, TDD – The Art of Fearless Programming, IEEE Software 24 (3) 24–30.
[3] H. Erdogmus, M. Morisio, M. Torchiano, On the effectiveness of the test-first approach to programming, IEEE Transactions on Software Engineering 31 (3) (2005) 226–237.
[4] L. Huang, M. Holcombe, Empirical investigation towards the effectiveness of test first programming, Information and Software Technology 51 (2009) 182–194.
[5] M.M. Mueller, O. Hagner, Experiment about test-first programming, IEE Proceedings – Software 149 (5) (2002) 131–136.
[6] L. Madeyski, The impact of test-first programming on branch coverage and mutation score indicator of unit tests: an experiment, Information and Software Technology 52 (2) (2010) 169–184.
[7] M. Pančur, M. Ciglarič, M. Trampuš, T. Vidmar, Towards empirical evaluation of test-driven development in a university environment, in: EUROCON', Proceedings of the International Conference on Computer as a Tool, 2003, pp. 83–86.
[8] B. George, L. Williams, A structured experiment on test-driven development, Information and Software Technology 46 (2004) 337–342.
[9] M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams, M. Zelkowitz, Empirical findings in agile methods, in: Proceedings of Extreme Programming and Agile Methods – XP/Agile Universe 2002, Lecture Notes in Computer Science, vol. 2418, Springer, 2002, pp. 197–207.
[10] E.M. Maximillien, L. Williams, Assessing test-driven development at IBM, in: Proceedings of the International Conference on Software Engineering (ICSE 2003), 2003, pp. 564–569.
[11] M.M. Müller, F. Padberg, W. Tichy, Ist XP etwas für mich? Empirische Studien zur Einschätung von XP, Software Engineering 2005, Lecture Notes in Informatics LNI, Gesellschaft für Informatik, Essen, 2005.
[12] G. Succi, M. Marchesi (Eds.), Extreme Programming Examined, Addison-Wesley, 2001.
[13] W.F. Tichy, Hints for reviewing empirical work in software engineering, Empirical Software Engineering 5 (2000) 309–312.
[14] V. Basili, F. Shull, F. Lanubile, Building knowledge through families of experiments, IEEE Transactions on Software Engineering 25 (4) (1999) 456–473.
[15] F.J. Shull, J.C. Carver, S. Vegas, N. Juristo, The role of replications in empirical software engineering, Empirical Software Engineering 13 (2) (2008) 211–218.
[16] A. Geras, M.R. Smith, J. Miller, A prototype empirical evaluation of test driven development, in: IEEE METRICS'2004: Proceedings of the 10th IEEE International Software Metrics Symposium, IEEE Computer Society, 2004, pp. 405–416.
[17] T. Bhat, N. Nagappan, Evaluating the efficacy of test-driven development: industrial case studies, in: Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering (Rio de Janeiro, Brazil, September 21–22, 2006), ISESE '06, ACM, New York, NY, pp. 356–363.
[18] D. Janzen, H. Saiedian, Test-driven learning in early programming courses, in: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, Portland, USA, March 12–15, 2008, pp. 532–536.
[19] J.H. Vu, N. Frojd, C. Shenkel-Therolf, D. Janzen, Evaluating test-driven development in an industry-sponsored capstone project, in: Proceedings of Sixth International Conference on Information Technology: New Generations, 2009, pp. 229–234.
[20] B. Turhan, A. Bener, P. Kuvaya, M. Oivo, A quantitative comparison of test-first and test-last code in an industrial project, in: A. Silliti et al. (Eds.), XP 2010, LNBIP 48, Springer-Verlag, Berlin, 2010, pp. 232–237.
[21] M. Siniaalto, P. Abrahamsson, A comparative case study on the impact of test-driven development on program design and test coverage, in: ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, 2007, pp. 275–284.
[23] J.H. Hayes, A. Dekhtyar, D. Janzen, Towards traceable test-driven development, in: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering (2009), International Conference on Software Engineering, IEEE Computer Society, Washington, DC, 2009, pp. 26–30.
[24] D. Janzen, H. Saiedian, On the influence of test-driven development on software design, in: CSEET, 19th Conference on Software Engineering Education & Training (CSEET'06), 2006, pp. 141–148.
[25] J.C. Sanchez, L. Williams, E.M. Maximilien, A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry Proc. Agile 2007 Conf., IEEE CS Press, 2007.
[26] C. Desai, D. Janzen, K. Savage, A survey of evidence for test-driven development in academia, SIGCSE Bulletin 40 (2) (2008) 97–101.
[27] C.M. Lott, H.D. Rombach, Repeatable software engineering experiments for comparing defect-detection techniques, Journal of Empirical Software Engineering 1 (1996) 241–277.
[28] C. Wohlin, M. Höst, Special section: controlled experiments in software engineering, Information and Software Technology 43 (2001) 921–924.
[29] M.M. Müller, W.F. Tichy, Case study: extreme programming in a university environment, in: Proceedings of the International Conference on Software Engineering (ICSE 2001), Toronto, Canada, 2001.
[30] R. Mugridge, Challenges in teaching test driven development, in: Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, Italy, 2003, pp. 410–413.
[31] R. Mugridge et al., Five challenges in teaching XP, in: Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, Italy, 2003.
[32] T. Flohr, T. Schneider, Lessons learned from an XP experiment with students: test-first needs more teachings, in J. Münch, M. Vierimaa (Eds.), PROFES 2006, LNCS 4034, pp. 305–318.
[33] G. Melnik, F. Maurer, Perceptions of agile practices: a student survey, in: Proceedings of Extreme Programming and Agile Methods – XP/Agile Universe 2002, Lecture Notes in Computer Science, vol. 2418, Springer, 2002, pp. 241–250.
[34] Fit: Framework for Integrated Test, http://fit.c2.com, with SAT framework: Socket Acceptance Test (accesed 2010).
[35] S. Glantz, Primer of Biostatistics, Fifth ed., McGraw Hill, 2002.
[36] J. Miller, J. Daly, M. Wood, M. Roper, A. Brooks, Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research, Information and Software Technology 39 (4) (1997) 285–295.
[37] T. Dyba, V.B. Kampenes, D.I.K. Sjoberg, A systematic review of statistical power in software engineering experiments, Information and Software Technology 48 (8) (2006) 745–755.
[38] C. Wohlin et al., Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, USA, 2000.
[39] S.D. Levitt, J.A. List, Was There Really a Hawthorne Effect at the Hawthorne Plant? An Analysis of the Original Illumination Experiments (May 2009). NBER Working Paper Series, vol. W15016, 2009.
[40] M.M. Müller, A. Höfer, The effect of experience on the test-driven development process, Empirical Software Engineering 12 (2007) 593–615.
[41] A. Porter, L. Votta, Comparing detection methods for software requirements inspection: a replication using professional subjects, Empirical Software Engineering (3) (1995) 355–380.
[42] M. Höst, B. Regnell, C. Wohlin, Using students as subjects – a comparative study of students and professionals in lead-time impact assessment, Empirical Software Engineering 5 (3) (2000) 201–214.
[43] W.F. Tichy, Should computer scientist experiment more?, IEEE Computer 31 (5) (1998) 32–40
[44] J.C. Carver, L. Jaccheri, S. Morasca, F. Shull, A checklist for integrating student empirical studies with research and teaching goals, Empirical Software Engineering 15 (2010) 35–59.
[45] C. Kaner, W.P. Bond, Software engineering metrics: What do they measure and how do we know?, in: METRICS 2004, IEEE CS Press, 2004.
[46] M. Broy, F. Deissenboeck, M. Pizka, Demystifying maintainability, in Proceedings of the 2006 International Workshop on Software Quality (Shanghai, China, May 21–21, 2006), WoSQ '06, ACM, New York, NY, pp. 21–26.
[47] K. Stroggylos, D. Spinellis, Refactoring – does it improve software quality?, in Proceedings of 5th International Workshop on Software Quality (WoSQ'07: ICSE Workshops), 2007, pp. 10–16.
[48] Y. Jiang, B. Cuki, T. Menzies, N. Bartlow, Comparing design and code metrics for software quality prediction, in: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering (2008), PROMISE '08, ACM, New York, NY, pp. 11–18.
[49] S. Cornett, Code Coverage Analysis <http://www.bullseye.com/coverage.html> (accessed 2010).
[50] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (4) (1976) 308–320.
[51] Clover project. Atlassian Pty Ltd., <http://www.atlassian.com/software/clover> (accessed 2010).
[52] H. Kou, P.H. Johnson, H. Erdogmus, Operational definition and automated inference of test-driven development with Zorro, Automated Software Engineering 17 (1) (2010) 57–85.
[54] L. Crispin, Driving software quality: how test-driven development impacts software quality, IEEE Software 23 (6) (2006) 70–71.
[55] D.S. Janzen, H. Saiedian, Does Test-driven development really improve software design quality?, IEEE Software 25 (2) (2008) 77–84
[56] L. Madeyski, Test-Driven Development: An Empirical Evaluation of Agile Practice, Springer, London, UK, 2010.
[57] A. Gupta, P. Jalote, An experimental evaluation of the effectiveness and efficiency of the test driven development, in: ESEM'07: International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, Washington, DC, USA, 2007, pp. 285–294.
[58] L. Madeyski, Preliminary analysis of the effects of pair programming and test-driven development on the external code quality, in: K. Zielinski, T. Szmuc (Eds.), Software Engineering: Evolution and Emerging Technologies, IOS Press, 2005.
[59] L. Madeyski, The impact of pair programming and test-driven development on package dependencies in object-oriented design – an experiment, in: J. Muench, M. Vierimaa (Eds.), PROFES'06: Product Focused Software Process Improvement, Lecture Notes in Computer Science, vol. 4034, Springer, Berlin, Heidelberg, 2006, pp. 278–289.
[60] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, C.A. Visaggio, Evaluating advantages of test driven development: a controlled experiment with professionals, in: ISESE'06: ACM/IEEE International Symposium on Empirical Software Engineering, ACM Press, New York, NY, USA, 2006, pp. 364–371.
[61] Judy – a mutation testing tool for Java. <http://www.e-informatyka.pl/sens/Wiki.jsp?page=Projects.Judy>.
[62] Metrics – Eclipse plug-in for software metrics. <http://eclipse-metrics.sourceforge.net/>.
[63] L. Madeyski, N. Radyk, Judy – A mutation testing tool for Java. IET Software 4(1) (2010) 32–42.