Experimental Evaluation of Test-Driven Development With Interns Working on a Real Industrial Project

Bartosz Papis, Konrad Grochowski, Kamil Subzda, and Kamil Sijko

Abstract—Context: There is still little evidence on differences between Test-Driven Development and Test-Last Development, especially for real-world projects, so their impact on code/test quality is an ongoing research trend. An empirical comparison is presented, with 19 participants working on an industrial project developed for an energy market software company, implementing real-world requirements for one of the company's customers. *Objective:* Examine the impact of TDD and TLD on quality of the code and the tests. The aim is to evaluate if there is a significant difference in external code quality and test quality between these techniques. *Method:* The experiment is based on a randomized within-subjects block design, with participants working for three months on the same requirements using different techniques, changed from week to week, within three different competence blocks: Intermediate, Novice and Mixed. The resulting code was verified for process conformance. The participants developed only business logic and were separated from infrastructural concerns. A separate group of code repositories was used to work without unit tests, to verify that the requirements were not too easy for the participants. Also, it was analysed if there is any difference between the code created by shared efforts of developers with different competences and the code created by participants isolated in the competence blocks. The resulting implementations had LOC order of magnitude of 10k. *Results:* Statistically significant advantage of TDD in terms of external code quality (1.8 fewer bugs) and test quality (5 percentage points higher) than TLD. Additionally, TDD narrows the gap in code coverage between developers from different competence blocks. At the same time, TDD proved to have a considerable entry barrier and was hard to follow strictly, especially by Novices. Still, no significant difference w.r.t. code coverage has been observed between the Intermediate and the Novice developers - as opposed to TLD, which was easier to follow. Lastly, isolating the Intermediate developers from the Novices had significant impact on the code quality. *Conclusion:*TDD is a recommended technique for software projects with a long horizon or when it is critical to minimize the number of bugs and achieve high code coverage.

Index Terms—Empirical software engineering, iterative test last development, Test driven development

1 Introduction

Unit testing is the most fundamental approach to software testing [1], in the sense that it verifies the desired part of the implementation directly, by simply executing it. Among many possible approaches to unit testing, two main development techniques are chosen for comparison: Test-Driven Development (TDD) and Test-Last Development (TLD). The studies concerning TDD are still reporting inconsistent results about the merits of TDD [2], [3], [4], [5], [6] so it is important to keep gathering more data on the subject. There are two existing research trends concerning measuring this impact: simulation modelling approach and experimental approach. In this paper, we focus on the latter.

Although there are many existing experimental assessments of unit testing techniques, they are usually constrained by their academic or commercial nature (as in [7], [8], [9]). The purpose of this work is to achieve experimental conditions that are very rarely met in the studies on the subject [10]. We have worked with *Transition Technologies S.A.*, which, at the time, was employing more than 800 employees and was a provider of software solutions for the gas and power industry. The company was successfully convinced to perform the same real, commercial project for the gas industry several times simultaneously under our control. It is worth noting, that this is not a common occurrence [4].

This study is focused on external code quality and test quality, referred to as code quality and code coverage respectively. The reasons for focusing on these are that external code quality is the most important characteristic of a software to its end users [11] and code coverage is one of the most important characteristic of internal code quality [12]. The main goal of this study is to answer two research questions: (i) is there a significant difference between the external code quality of code created using Test-Driven Development and Test-Last Development, (ii) is there a significant difference between the code coverage with unit tests created using Test-Driven Development and Test-Last Development. Additionally, we address the following additional hypothesis: (iii) is there a significant difference in

- *Bartosz Papis is with the Google Inc., 00-113 Warsaw, Poland. E-mail: bartoszkp@gmail.com.*
- *Konrad Grochowski is with the Warsaw University of Technology, 00-661 Warszawa, Poland. E-mail: Konrad.Grochowski@pw.edu.pl.*
- *Kamil Subzda is with the Nexwell Engineering in Wroclaw, 54-440 Wroclaw, Poland. E-mail: kamil.subzda@gmail.com.*
- *Kamil Sijko is with the Transition Technologies S.A., Warsaw, Poland. E-mail: kamil@sijko.pl.*

Manuscript received 5 Dec. 2019; revised 3 Sept. 2020; accepted 24 Sept. 2020. Date of publication 28 Sept. 2020; date of current version 16 May 2022. (Corresponding author: Konrad Grochowski.) Recommended for acceptance by C. B. Seaman. Digital Object Identifier no. 10.1109/TSE.2020.3027522

code quality between teams consisting of members with significant difference in their competence level and teams consisting of developers with similar competence level. The experiment design we propose for these research questions has the following key features:

The first feature is measuring process conformance at the end of the experiment. Because we want to compare TDD with TLD, it is necessary to be sure that the participants actually use these two development techniques correctly. Due to our validation procedure it is possible to define how well the work of each participant represents each technique. There are many existing approaches to validating constraints of software development techniques [13], [5], [14], [15], [16] and most of them are based on the Hackystat tool [17]. In this work, we propose to use a special committing scheme and use a different validation method than proposed in the aforementioned papers.

The second important characteristic of this experiment is separating the participants of the experiment from infrastructural concerns. A programming concern is regarded as infrastructural if it is not solely related to the business logic needed by the end customer. Such concerns include network communication, database modelling and architectural decisions. Most of them require either specific technical knowledge or good documentation searching/reading skills, which are completely orthogonal to unit testing. Consequently, a shell solution with all these aspects implemented is provided for the experiment. All functions designed to implement business logic, i.e., the essence of what the application is doing, are left empty. The participants seek to implement the missing, business-logic-related parts of the solution only. The idea of separating participants from infrastructural concerns is also presented in [10]. This is important for two reasons: i) it reduces the impact of confounding variables related to subjects' experience with any 3rd-party library used in the project ii) it focuses the study on the part of the software that is most important to its end users: the business logic [11].

The third aspect of our experiment is random repository switching, which is an implementation of within-subjects design. Participants contribute to a group of shared code repositories, with different programming techniques. Each repository has a different programming technique assigned and represents an alternative development timeline of the same project. In the design proposed in this paper, the participant-repository assignment for each week is randomised (with some constraints), so most of the time they work with a different technique each week. This helps in making the results independent of participants' individual traits. An additional benefit of such an approach is that this forces the participants to work with other people's code, which is much more realistic than confining oneself to one's own implementation.

The last feature of the proposed experiment is employing two manipulation checks [18]: (i) verifying that the requirements to implement are not too easy (ii) verifying if it makes sense to use a block design for the experiment, where participants form blocks based on their competence level. For the first check, participants are also working without writing any tests at all. Additional code repositories without unit tests serve as an informal baseline, to check if unit tests, in one form or another, have any impact on the quality of the implementation of the tasks. The second check consists of creating three competence blocks: with intermediate developers only, with novice developers only, and a mixed block with some intermediate and some novice developers. The outcomes of the experiment for the last block, representing a design without competence blocks, are compared with the two former blocks - a significant difference would give some insight into whether the block design is the right choice.

Beside the aforementioned design decisions, the contributions of this paper are as follows: (i) the real-world requirements of an industrial project, implemented by students. Experiments with students and professionals are both considered valuable in current research [19] and quite common. Little evidence exists on TDD impact for industrial projects [10], (ii) the final solution has around 10k LOC, which classifies it as a mid-size project. Most other papers either do not report sizes of their code base [9], [20], or are focused on small projects [4], [21], [22].

To present the composition of this experiment and report the results, this paper is structured as follows. The next Section 2 briefly summarizes the current research in the field of experimental evaluation of TDD. Section 3 presents the definitions of TDD, TLD and No Unit Tests Development (NUT) as introduced to the participants and describes the core ideas of this work, i.e., the proposed design of an experiment for addressing the two main hypotheses concerning comparison of TDD and TLD, and the additional hypothesis for comparing different compositions of groups of developers with regard to their competence level. Section 4 describes how the experiment is carried out. Section 5 follows, presenting the results of the experiment and their statistical analysis. The interpretation and discussion are presented in Sections 6 and 7 concludes the paper.

2 Related Work

2.1 Overview

In recent years, many studies concerning TDD effectiveness have been conducted. A couple of factors that make them significantly different may be distinguished, affecting the strength and scope of their conclusions about the TDD effectiveness as such. These are:

- *project requirements* – a study may be conducted 'in the industry', on a commercial project with real-world requirements (e.g., [7]), or 'in the lab', on an artificial project with requirements created solely for the purpose of the experiment (e.g., [20]),
- *conditions controllability* – some studies are designed before the coding work, having therefore an impact on its organization and being able to collect specific data crucial to the experiment during its process (e.g., [9]); while others are in the form of analysis after the fact, discussing the known sources about the course of the already executed project (e.g., [8]),
- *results comparability* – an experiment may concern only a single project or a group of unique projects, the features of which may vary, making them hard to compare (e.g., [7]); on the other hand, it may refer to a project repeated several times, each time in

TABLE 1
Sample Sizes in Experiments Comparing TDD and TLD Effectiveness

Study	Year	Subjects	Number of subjects	Duration	Real world tasks	LOC	Controlled	Comparable
[27]	2001	Students	12	Eight weeks	N	?	N	Y
[26]	2002	Students	19	One month	N	?	Y	Y
[21]	2003	Professionals	24	?	N	200	Y	Y
[22]	2003	Professionals	48	One week	N	200	Y	Y
[25]	2004	Inexperienced professionals	?	?	Y	?	?	?
[23]	2005	Students	24	Few days	N	300[1]	Y	Y
[7] (case A)	2006	Professionals	6	Four months	Y	10k	N	N
[7] (case B)	2006	Professionals	5-8	Half a year	Y	50k	N	N
[30]	2006	Simulated contributors	ca. 350	Not applicable	N	200k	Y	Y
[4]	2008	Students/Professionals	27	Four months	Y/N[2]	5k	N	Y
[20]	2008	Professionals	28	One day	?	?	Y	Y
[8] (case A)	2008	Professionals	9	One year	Y	70k	N	N
[8] (case B)	2008	Professionals	7	Three months	Y	200k	N	N
[9]	2011	Students	23	Five weeks	N	?	Y	Y
[29]	2014	Professionals	13	90 minutes	N	300	Y	Y
[10]	2016	Professionals	24	Three days	N	?	Y	Y
[5]	2017	Professionals	39	20 days	N	1k	Y	Y

[1]*Assuming 300 LOC for "Bowling Scorekeeper Kata" exercises, by analogy from other studies using this exercise.*
[2]*Both real-world and academic tasks.*

similar circumstances and controllable, experiment-relevant conditions, including the possibility of control groups (e.g., [20]),

- *developers' experience* – experiments involve programmers with a different level of experience and job status – in extreme cases, they deal with the work of professional and advanced developers qualified in the validated development technique (e.g., [20]) or with the work of beginner students completely unfamiliar with the technique (e.g., [23]),
- *work scale* – a set of measurable or qualitative factors specifying the size and difficulty of the project: its duration, number of engaged developers, number of produced lines of code, specificity of the domain and implementation complexity (see comparison in Table 1 below).

Typically, these factors tend to have an impact on each other. Studies conducted on commercial, industry projects have a more realistic and interesting work scale [24], [25], while academic experiments are generally shorter and simplified, being detached from real-world concerns, such as client demands, management priorities, need of innovative solutions, critical business deadlines, and the overall R&D context of the company. The 'laboratory' environment gives a greater possibility to control the requirements and conditions of the experiment [9], as opposed to commercial projects, which are only intended to be financially effective. For this reason, industrial projects are usually unique and the repetition of the experiment is impracticable [4]. On the other hand, academic experiments are by their nature designed to be repeatable [26], [23]. They are also often conducted on a group of students [27], [28], which is another reason for their simplicity, but sometimes they happen to be executed with the help of professional developers [5], [21], [29]. Industrial experiments involve the permanently employed company workers [20], but still possibly also interns. Studies of industrial cases are sometimes realized in the form of a historical analysis and the results of the projects are executed with no scientific inquiry in mind [10], [7], [8]. This makes them more realistic, but, at the same time, more obscure for investigation. On a very different side of the research spectrum, there are also experiments based on a modelling technique called software process simulation (SPS), where no real participants are involved [30]. Table 1 summarizes properties in other experiments, similar to this study with respect to the aforementioned factors. In this table, "Controlled" refers to *conditions controllability* and "Comparable" to *results comparability* defined in the list above.

2.2 Basic Conclusions in the Current Research

The general results of various experiments in current research are coherent and indicate mostly that TDD improves the quality of the output code in some way [6]. However, some researchers do not report such conclusions and a few claim the contrary. In [25], the authors state briefly that TDD improved the team performance, and [30] conclude that it yields better results in terms of code quality. [21] and [8] note higher quality code when TDD is used, but increased development time, while [23] states that students using TDD write more tests, and that writing more tests increases productivity. [26] do not notice any acceleration in development process nor change in the quality of code, however it seems to the authors that writing tests first supports better understanding of created programs. The authors of [5] conclude that the reported quality and productivity improvements are associated with code granularity and uniformity, rather than with the order of test and production code writing. In [9], [29], and [10] no statistically significant differences are reported between most important metrics used. What is worth noticing, from this short comparison of results, is that the actual meaning of the conclusion depends significantly on the conditions and metrics of the experiment.

In [22] the authors report conducting a set of trials involving 48 professional programmers working in pairs, dividing them into six pairs working with the TDD method and the other six serving as a control group, with a waterfall-like approach. They state that TDD teams developed code that passes approximately 18 percent more test cases than the code of the control groups, but needed approximately 16 percent more time to complete the task. The programmers vary from novices to experts, but the project itself is short (200 lines of code) and artificial.

The authors of [7] decide not to conduct an experiment, but analyse an existing, industrial projects in retrospect. Additionally, they choose two very different projects (the second one larger and involving more experienced personnel), and try to compare the results between the team working using TDD and a similar team from the same company that is not using it. Their results also indicate positive impact of TDD on code quality. In the case of the first project, the density of defects is stated to be 2.6 times less for the TDD team, but the time taken for the task is 25-35 percent longer. For the second project, these measurements are, respectively 4.2 and 15 percent. As the researchers report, the TDD and non-TDD projects, being executed as a part of an uncontrolled study, might not be on the same level of development difficulty, and therefore their comparison might be inadequate.

Janzen and Saiedian in [4] design their experiment to evaluate the "test-first" factor of the TDD method, striving to minimize other differences between the TDD group and the control group. Then, they compare the TDD method with its reversed, test-last analogue, asking all participants to work in a rapid test-code or code-test iterations manner. The teams worked on two real commercial projects of similar size. The authors conducted pre-experiment surveys to ensure that there is no significant differences between participants' experience. Having no opportunity to repeat the development tasks, they decide to split the work in the projects into two phases, sharing it between two investigated methods. The researchers conclude that there is a possible tendency for the developers using the TDD method to write smaller and simpler blocks of code, such as classes and methods. However, they admit also that they cannot substantiate any claim concerning improvement in class cohesion. The authors also indicate that the limited conclusions are a consequence of a small number of developers involved, non-random participant selection, and the inability to simultaneously control the experiment variables and sustain the commercial character of the projects.

Broader reviews of current research on the topic of TDD effectiveness may also be found in a book by Madeyski [6], which in general focuses on the analysis of impact of programming practices like test-first programming, test-last programming, pair programming, solo programming and other closely related approaches. Also, further discussion can be found in systematic reviews such as [31] or [32]. Munir *et al.* [3] present a systematic review which additionally classifies presented studies according to rigor and relevance.

3 Experimental Design

3.1 Software Development Techniques

Our experiment design concerns three development techniques, namely Test-Driven Development (TDD), Test-Last Development (TLD) and No Unit Tests (NUT). They vary in strictness: from completely unstructured and almost arbitrary NUT, to more clearly defined and methodical TLD, to almost formally defined and rigid TDD. In the following sections we describe how those methods are defined for the purpose of this study. The details of the definitions might be arguable, but the main objective was to compare the impact of the moment of creation of unit tests. Even though design is sometimes deemed as the most important aspect of TDD [4], end users care much more about the software working according to specifications [11] – and the end users are the primary beneficiaries of business use software. Thus, in this experiment, we focus on the development aspect of TDD and TLD, and we do not address the impact of software development techniques on software design [2].

3.1.1 No Unit Tests (NUT)

The name of this technique is introduced for the purpose of this experiment. It represents an "old school" technique, where developers wish to create production code as quickly and as well as possible. The only restriction of this method is that testing, if any, is performed only manually at the application level.

3.1.2 Test-Last Development (TLD)

Probably the most well-known approach to automated testing: after developing some part of functionality. Usually it is the developer's responsibility to decide when the testing should start, or, in other words, when functionality is sufficiently complete to be the subject of tests. In this experiment, each task is tightly connected to a complete function of the system. The word "last" in TLD might suggest writing a unit test after complete implementation of the whole application, but that is uncommon. Applications are usually developed in a sequence of iterations, and "last" is interpreted in this experiment as "the last part of a single iteration", where iteration represents adding a new function, implementing a single task. In this experiment, we aim to define small enough tasks to write unit tests after the complete function has been implemented. If needed, fixes or refactoring can follow. Enforcing the moment of writing unit tests is introduced to minimize the influence of personal testing skills and traits of the participants.

3.1.3 Test-Driven Development (TDD)

TDD [33] is defined in the most formal way, by providing The Three Rules of TDD[1]: *(i) one is not allowed to write any production code unless it is to make a failing unit test pass, (ii) one is not allowed to write any more of a unit test than is sufficient to fail – and compilation failures are failures, (iii) one is not allowed to write any more production code than is sufficient to pass the one failing unit test*. TDD consists of the following cycle: (i) create *minimal* failing test code, (ii) write *minimal* production code to make tests pass, (iii) refactor code and repeat, until the task may be considered done. Such a cycle should be repeated multiple times during the implementation of each task and each cycle should be complete in a matter of minutes.

1. Robert C. Martin, *The Three Rules of TDD*: http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

3.2 Research Objectives

The main foci of this study are the following research questions concerning external code quality (code quality) and test quality (code coverage):

Is there a significant difference between the code quality of code created using Test-Driven Development and Test-Last Development?

Is there a significant difference between the coverage of code created using Test-Driven Development and Test-Last Development?

Additionally, as a form of manipulation check [18], we evaluate if indeed there are grounds for splitting the participants into competence-level blocks, which is formulated as the following additional hypotheses (tested separately for both TDD and TLD):

Is there a significant difference between the code quality created by groups of developers with significant differences in their competence level and groups consisting of developers with similar competence level?

Is there a significant difference between the coverage of code created by groups of developers with significant differences in their competence level and groups consisting of developers with similar competence level?

3.3 Overview

Our experiment is organised into weekly *iterations* and consists of *participants* working with code *repositories*. Each repository is assigned to one of the three development *techniques*: TDD, TLD and NUT, as presented in Section 3.1. In each iteration, participants get assigned to a different repository, to work using different techniques. The participants are grouped into three *blocks*: Intermediate, consisting only of the non-beginner developers; Novice, consisting only of the most inexperienced developers; and Mixed, consisting of the advanced developers working together with the inexperienced ones.

Similarly to [23], to compare the three development techniques we use the black-box acceptance tests approach. In order to validate requirements of the project, we have first created a reference implementation - solutions for all experiment's tasks, along with a reference unit-test suite to validate the business logic against the requirements. These reference unit tests were then used as a base to create 8320 automated acceptance tests (15k LOC). The number of failing acceptance tests represent the external code quality. When working with TDD or TLD, the participants create code and unit tests. To measure the quality of the tests created by the participants, we calculate code coverage from their tests at the end of each iteration.

3.4 Variables

The independent variables are: (i) the type of technique used by a participant each iteration: $technique$ and (ii) the name of the competence block a participant is assigned to: $competence$.

The dependent variables are the number of failing acceptance tests: $bugs\_end$ or Q for "quality" ($Q = bugs\_end$) and the percentage of sequence points from participant's code executed when running participant's unit tests: $coverage\_end$ or C for "coverage" ($C = coverage\_end$). In addition to the independent and dependent variables defined for the sake of formulating the hypotheses, the following covariates can be identified: the number of the iteration: $iteration$, process conformance ($conformance$) and the state of the repository at the beginning of an iteration: number of bugs ($bugs\_start$) and code coverage ($coverage\_start$). The last two are the same quantities as $bugs\_end$ and $coverage\_end$ resp., but measured at the beginning of an iteration, not at the end. Process conformance is an important variable since it allows to quantify how well the code in each repository represents each technique. The outcomes of each iteration, represented by the values of the dependent variables, depend on the sequence of iterations, which is tracked by the iteration number, since different tasks were implemented during each iteration. The variables representing the state of the code in each repository at the beginning of an iteration also impact the outcomes of each iteration: (i) the more bugs are already present in the repository when the participant starts their work at the beginning of the iteration, the more work they need to devote before they can address the tasks assigned to the current iteration (ii) the better the code coverage is, the less is the chance of breaking the existing code.

Additionally, we include $crashes\_start$ to represent the number of failing acceptance tests that result in an unhandled exception, which results in collapsing the application. Such bugs, commonly referred to as "crashes", are more severe - the application is not only yielding incorrect results, but becomes completely unusable. Since "crash" is never an expected outcome of any business scenario used in acceptance tests $crashes\_start \leq bugs\_start$. This variable is introduced under the assumption that crashes are more meaningful than non-crashing bugs and thus are more important to avoid. The results of the experiment show that this distinction is of no importance in this paper; however, we keep it for the sake of accuracy and completeness of this report.

The confounding variables in this experiment are related to personal traits of the participants. They are discussed in Section 6.1.

3.5 Hypotheses

We have two main hypotheses, namely H_Q and H_C, that concern the code quality ($Q = bugs\_end$) and code coverage ($C = coverage\_end$), resp. and additional hypotheses H_{BQ} and H_{BC} concerning competence level groups - as a form of manipulation check. Denoting with an additional index 0 the null hypothesis, and with 1 the alternative hypothesis, our hypotheses can be formulated as follows:

$$\begin{aligned} H_{Q0} &: Q(\mathrm{TDD}) = Q(\mathrm{TLD}) \\ H_{Q1} &: Q(\mathrm{TDD}) \neq Q(\mathrm{TLD}) \\ H_{C0} &: C(\mathrm{TDD}) = C(\mathrm{TLD}) \\ H_{C1} &: C(\mathrm{TDD}) \neq C(\mathrm{TLD}) \end{aligned}$$

$$\begin{aligned} H_{BQ0} &: Q(\mathrm{Intermediate}) = Q(\mathrm{Mixed}) \wedge Q(\mathrm{Novice}) = Q(\mathrm{Mixed}) \\ H_{BQ1} &: Q(\mathrm{Intermediate}) \neq Q(\mathrm{Mixed}) \vee Q(\mathrm{Novice}) \neq Q(\mathrm{Mixed}) \\ H_{BC0} &: C(\mathrm{Intermediate}) = C(\mathrm{Mixed}) \wedge C(\mathrm{Novice}) = C(\mathrm{Mixed}) \\ H_{BC1} &: C(\mathrm{Intermediate}) \neq C(\mathrm{Mixed}) \vee C(\mathrm{Novice}) \neq C(\mathrm{Mixed}). \end{aligned}$$

The H_{Q1} represents our expectation that TDD influences code quality, i.e., developers using TDD will not make the same number of mistakes as developers working with TLD. H_{C1} expresses our second expectation, i.e., that TDD has significant impact of the quality of tests created. H_{BQ1} presents the claim that mixing different developer competence levels has significant impact on the code quality and H_{BC1} represents the same for code coverage. H_{BQ} and H_{BC} hypotheses are formulated using conjunctions and alternatives. The consequence of this formulation is that if the data supports rejecting the null hypothesis, we will not know if it was because of the difference between Intermediate and Mixed developers or because of the difference between Novice and Mixed developers. This is however enough to validate the design decision of creating competence blocks and simplifies the analysis.

3.6 Design

The design of the experiment is based on a set of code repositories, with a different technique assigned to every one of them. Depending on the repository, every participant is working either using the TDD, TLD or NUT technique. Each repository contains a complete solution, developed by different participants using the assigned technique, starting from the same shell solution.

The NUT technique serves as a way of validating the complexity of the tasks given to the participants - the first proposed manipulation check [18]. This way we can evaluate if the unit tests, no matter how created, have any influence on the quality of the code created during the experiment. If the code created using the NUT technique is free of bugs, it would suggest that the tasks in our experiment are too easy for the participants. For a task created without any mistake, there is no room for improvement in terms of external code quality. So, for the software development techniques, for which one of the purposes is to minimize the number of bugs, any comparisons in such context are not meaningful. However, the existence of any bugs would show that the tasks are not trivial, at least at the level of proficiency of the participants and measuring how TDD and TLD improve external quality makes sense.

We employ a randomised within-subjects block design, with elements of a counterbalanced design [34]. The blocks consist of participants with a similar competence level, and all participants use all development techniques in a different, random order. The aim for creating blocks is to minimize the impact of the competence level on quality. We propose to use three blocks: Novice, Intermediate, and Mixed. The last block is used to address the additional hypotheses H_{BQ1} and H_{BC1}, to implement the second manipulation check, verifying that using competence blocks is justified. If the results for Mixed block are not different than for other blocks, it would suggest that the blocks are not a necessary part of the design of the experiment.

The experiment is divided into iterations. Each iteration lasts one week and has a predefined list of tasks assigned that are expected to be completed within the iteration. During each iteration, each participant is working with a different repository than in the previous iteration, using a different technique.

Switching techniques/repositories from week to week serves two purposes: i) to minimize the impact of differences in personal skills between participants on code/test quality in the given repository, ii) to make the experiment closer to real-world conditions, where developers need to become acquainted with someone else's code before starting to work on assigned tasks. On the other hand, switching techniques while working on the same code base with similar tasks does not seem like a natural way of working. However, it is a part of the design of the experiment aimed to fully utilise the available sample size – each participant contributes to each evaluated technique.

3.7 Subjects

The participation in the experiment requires at least basic programming skills. The knowledge about TDD and TLD is not required, since the experiment is preceded by a training, aimed to present uniform definitions of TDD and TLD to all participants. The proposed recruitment consists of solving a set of four design and programming tasks: 1) Propose a data structure to hold information about daily gas flow for every hour in local time. 2) Assuming all required data for the structure designed in task 1 is stored in a text file, propose a method of verifying its correctness. The participants assume the availability of a method `GetNumberOfHoursInDay` which returns an integer and accepts a single `DateTime` parameter. 3) Write unit tests for the validation of the method from task 2. 4) Implement the method from task 2 using TDD.

The resulting code is to be rated anonymously, for example using a scale 1–5. Even though the tasks do not seem easy for novice developers, the rating is mainly focused on the following points: 1) Understanding that day can have a variable number of hours, instead of using a fixed 24-hour data structure. 2) Understanding what a unit test should look like. 3) Understanding the basics of TDD. 4) General programming skills.

3.8 Objects

3.8.1 Context

The objective for the participants is to implement a backend provider of web services supporting gas market and underground gas storage facility operations. The company has more than 10 years of expertise in this area, so the project has well defined requirements, based on the real-world needs of the company's clients. These services are focused on processing of time series, called data signals. Each signal can have one of multiple data types (ranging from simple integer numbers to character strings) and time granularity (i.e., a discretization unit, ranging from second to year). The server must support simple operations, such as storing time series data in a database, and more complex ones concerning the conversion of signal's granularity and filling missing data in time series using various algorithms. The project was created in .NET, using the C# language.

The participants ara given a shell implementation: code with all infrastructure related to web services, data transfer objects to domain objects mapping, and integration with a relational database through an ORM engine. The shell solution consists of 59 classes and 28 interfaces counting total 3226 LOC. Only the parts containing domain specific logic are left empty, to be filled in by the participants. Separating participants from infrastructural concerns in such a way should significantly reduce the influence of participants' personal traits based on their technical knowledge and previous experience with particular libraries and frameworks (e.g., database setup, DAL/nHibernate intricacies, WCF configuration etc.). This is especially important for a participant with no professional experience, but can be useful also in the context of professional developers, since even professional developers are not always necessarily familiar with all possible tools and frameworks. Additionally, the web service layer constitutes a common, well defined API, which allows the running of acceptance tests in the same way for each implementation.

Acceptance tests are used to evaluate implementations created by the participants during each iteration. They are constructed similarly to unit tests, but are executed in a complete environment via web services and with a working database. This makes them fully separated from the details of each implementation.

The shell implementation was extracted from the complete working implementation of the project with all requirements, created by the authors beforehand to validate the feasibility of the requirements. This implementation was also used to check correctness of the acceptance tests. Participants are not aware of the implementation nor of the acceptance tests' existence.

3.8.2 Tasks

During the experiment, the participants are required to work on two types of tasks. The tasks of the first type are defined before the experiment, based on the requirements given by the real-world customer needs. All participants receive the same tasks of the first type at the same time and they do not know the full list of tasks until the end of the experiment. This way, the experiment resembles normal day-to-day work, when developers receive new requirements. To enforce the need of modification of existing code even further, some tasks are designed as a "change in specification". This also allows participants to implement simpler algorithms first and become accustomed to the domain.

The second type of tasks are bugs detected at the end of each iteration. For each repository, each bug detected in iteration n results in a task assigned to the iteration $n+1$. This task is then included in the overall list of tasks, together with the predefined tasks of the first type.

To minimize potential misunderstandings and misinterpretations, all tasks are described in a language known well to all participants', and accompanied by small code examples, demonstrating how the desired feature is to be used (i.e., how the web service function call looks, and what are the expected results). Participants are asked to be frank, when in any doubt regarding the interpretation of task descriptions or how the tasks should fit into the existing implementation. If a question is general enough, it should be answered in an e-mail addressed to all the participants. All that effort is aimed at reducing the impact of participants' personal understanding of the tasks on experiment results.

All participants are instructed to prioritize their work as follows:

1) Fix compilation problems and failing unit tests.
2) Fix reported bugs (tasks of second type).
3) Finish any unfinished work from the previous iteration.
4) Implement tasks assigned to the current iteration.

3.9 Instrumentation

The technological infrastructure needed for implementation of the tasks should be prepared in a standardized way before the experiment: same machines with the same operating system and coding environment. Participants should work on commit-based repositories.

Before each iteration, an automated process sets the access rights to the repositories, so the participants can only work with a repository assigned to them in the current iteration.

Code quality should be measured by executing acceptance tests using web services for each repository at the end of each iteration. Similarly, code coverage should be measured by executing participants' unit tests at the end of each iteration.

3.10 Data Collection Procedure

The procedure of collecting data consists of running the set of acceptance tests and participants' unit tests at the end of each iteration. The reports from the tests' execution are used for the following purposes:

- measuring code quality of each implementation, in the form of a list of failing acceptance tests,
- measuring code coverage of each implementation, in the form of code coverage information,
- creating bug reports, which constituted tasks of the second type (see Section 3.8.2).

Another set of data collected from the experiment is needed to assess how well the participants conform to the rules of the techniques assigned to each repository. Knowing whether participants actually use the techniques we want to evaluate is essential for drawing meaningful conclusions from the experiment. The existing approaches [13], [5], [14], [15], [16], based on the Hackystat tool [17] were not directly applicable for this experiment because at the time it was only compatible with Visual Studio 2008. The minimum version for the project used in this experiment was Visual Studio 2010. Instead, we propose the following, simpler approach.

To enable the evaluation of process conformance for each development technique, participants are asked to document their work using a special committing scheme – creating separate commits for tests and implementation. It is a minor inconvenience, as committing this way corresponds to the rules of TDD and TLD. Also, small, frequent commits are well supported by Git. For example, working in TDD requires the participants to proceed in the following way:

1) write test code,
2) *commit #1*,
3) write production code,
4) *commit #2*,
5) refactor code,
6) *commit #3*,
7) repeat.

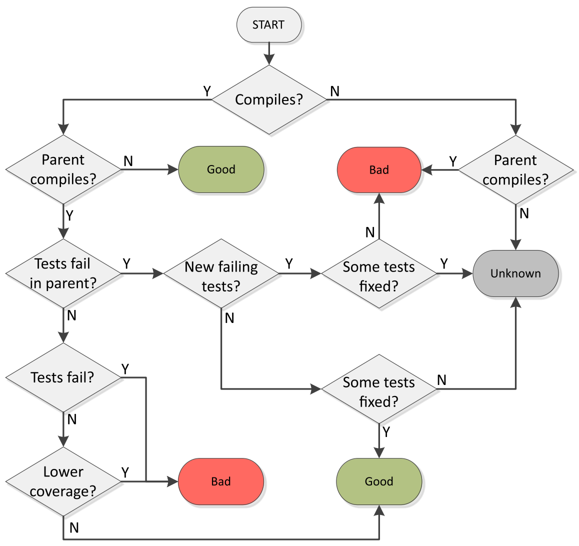

Fig. 1. Decision diagram for TDD process conformance, for an implementation commit (applied sequentially to each commit, taking its parent commit into account). Note that the two decision blocks with the same name, namely “Some tests fixed?” address two different situations, depending on the parent decision: “New failing tests?”. It is allowed to fix tests in small steps – i.e., some tests are still failing, but some were fixed, but it should not be allowed to break any new tests. The situation when some tests were fixed, but some previously passing tests are now failing is considered ambiguous.

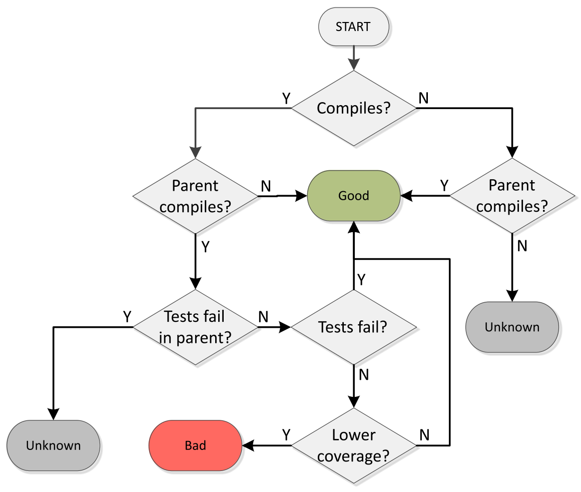

Fig. 3. Decision diagram for TDD process conformance for a test commit (applied sequentially to each commit, taking its parent commit into account).

(an analogous committing scheme is required for TLD, but with the production code step going first, before the test code step).

Due to this, calculating process conformance for a repository or a participant can be performed automatically. The algorithm can be as follows: first, divide commits into three categories: i) implementation commits, ii) test commits, iii) mixed commits. The third group is instantly discarded, as being too ambiguous. Such commits can be either invalid steps or, for example, refactorings that change a name of a function or a parameter, which should be allowed. Next, verify if implementation commits and test commits follow the rules of a technique assigned to this repository. The overview of the procedure used for implementation commits is presented in Figs. 1 and 2. Test commits should be processed using procedures shown in Figs. 3 and 4. The algorithm categorizes each commit separately, in the context of the preceding commit, as *good* – if it conforms to the presented rules, *bad* if it does not, and *unknown* when it is difficult to decide. An example for the last case is presented in Fig. 2 when after an implementation commit, which usually should increase the amount of untested code and thus decrease code coverage, we actually observe code coverage increasing. This is inherently not a bad thing, and can occur during refactoring, so one cannot classify this as a *bad* commit. But one also can not be sure whether it is a *good* commit, hence as a compromise, the *unknown* category is assigned. In a typical case, coverage either decreases or stays the same, so the commit can be classified as *good*.

Given such a procedure it needs to be decided how to treat *unknown* commits. For the sake of simplicity, we propose to treat *unknown* commits in the same way as *bad* commits. In principle they are not the same, but we prefer type II errors in this case. Thanks to this, the analysis is robust in dealing with participants who do not follow the rules. As a result, for each repository, the percentage of *good* commits can be calculated.

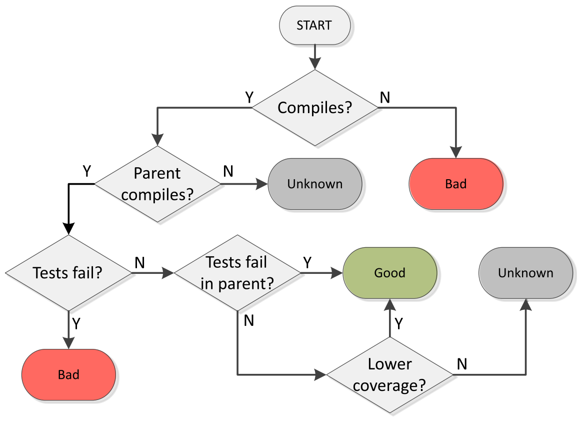

Fig. 2. Decision diagram for TLD process conformance, for an implementation commit (applied sequentially to each commit, taking its parent commit into account).

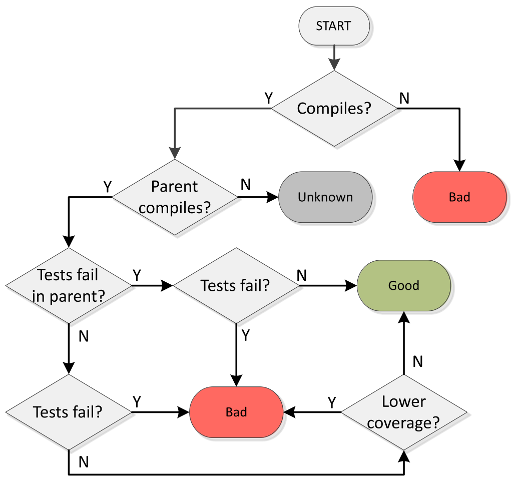

Fig. 4. Decision diagram for TLD process conformance for a test commit (applied sequentially to each commit, taking its parent commit into account).

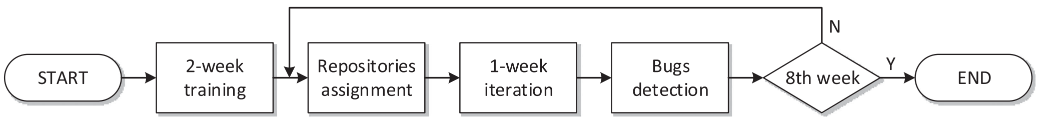

Fig. 5. Experiment flow overview.

4 Execution

The experiment lasted three months (July, August and September 2016) with 19 participants in three cities (13 in the Lublin office, four in the Warsaw office and two in the Białystok office). Fig. 5 shows the general course of the experiment. After two weeks of training (including presentations and exercises) the participants started to work in iterations. Each iteration started on Monday and finished on Friday. Before each iteration, the assignment of repositories to the participants was changed in such a way that on each Monday each participant would start working with a different programming technique than in the previous iteration. Tasks were revealed to participants at the beginning of the week, together with a new assignment of the repositories. The procedure was repeated for each week of the experiment. During each iteration (from Monday to Friday) the participants worked as they pleased, but with constraints imposed by the priorities presented in Section 3.8.2. Within the tasks of the same priority they could choose the order of implementation. On Friday, access to the repositories was revoked and the automatic acceptance test suite was run. The detected bugs were added to the task lists of the appropriate repositories. This made the experiment correspond to a real-world scenario, in which a developer is often faced with the task of fixing bugs in a code written by someone else. If a participant did not manage to implement all of the tasks assigned for an iteration, the unfinished tasks remained in the repository's task list, waiting for the next participant. Note that the process of switching repositories (i.e., reassignment before each iteration) always occurred on weekends, to help participants switch to a different programming technique smoothly. Also, during the weekends the code in each repository was made anonymous by removing user name information from commits, so participants were not aware who worked previously with their current repositories.

4.1 Sample

The participants of the experiment were students. The main reason for this decision was that hiring students instead of professionals solely for the purpose of an experiment significantly reduced costs for the company. Since hiring students could be one of the workarounds for workforce shortages in the IT market, the conclusions from this experiment could still be relevant for IT companies – especially that students were shown to be good representatives for developers in general in software engineering experiments [35]. An additional benefit was that students seemed to be less likely to have developed any personal preconceptions or preferences regarding software development techniques than professionals. The participants were chosen during a few months long recruitment process. The accepted candidates were employed as paid summer interns in the company, with the prospect of potential future ordinary employment. Passing the internship was also required by their universities; so as a result, the participants should have been motivated to work as they would work in normal circumstances.

There were 60 candidates that enrolled for the experiment. The solutions for the design and implementation tasks from the recruitment process explained in Section 3.7 were blindly graded by the authors of this paper. Afterwards, by inspecting the average grades and the solutions, we have agreed that participants with initial grades of at least three out of five show sufficient programming proficiency to qualify for the experiment. Thus, we have chosen a threshold of three for the candidates to qualify. Out of 60 candidates, only 26 received an average grade of three or more. Unfortunately, seven of them resigned from the experiment before it had started, leaving 19 participants. The distribution of the initial grades was as follows: no one received a grade of five, six participants received a grade of four, 20 participants received a grade of three, 29 participants received a grade of two and one participant received a grade of one. Four participants did not show up for the initial assessment. When grading the candidates, the average standard deviation between the authors of the experiment was 0.42. After the recruitment, the 19 chosen participants were asked to fill out a survey about their programming experience. The survey consisted of questions about their subjective assessment of their programming skill in various programming languages, their programming language preferences and their experience in TDD and TLD. The initial grades for the recruitment design and implementation tasks and grades from the survey were combined with different weights to form *final grades* assigned to the participants. The average standard deviation of the grades given by the authors individually was 0.38. The final grade was the average. Ten participants received a final grade of four, eight received a final a grade of three, and one participant received a final grade of two. These final grades were used to split participants into two categories: "A" (average grade of four or higher in 1–5 scale) and "B" (grade of three or lower). The categories were then used to form blocks for the experiment (see Section 3.6). Five randomly chosen participants have formed the Mixed block (two with category "A", and three with category "B", identified as M1–M5). The remaining participants with category "A" formed the Intermediate block (seven participants, identified as I1–I7), and the remaining ones categorized as "B" were used to form the Novice block (identified as N1–N7).

Having 19 participants, five in the Mixed block, seven in the Intermediate block and seven in the Novice block, 19 separate code repositories were created and split across the blocks to match the number of participants in each block.

Development techniques were assigned to repositories – each repository contained code created using a single technique (TDD, TLD, or NUT). As comparing TDD with TLD was considered more important than comparing any of them with NUT, only three NUT repositories were created

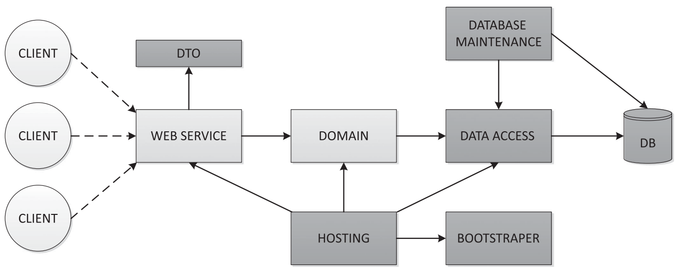

Fig. 6. Project's architecture overview.

(NUT1-NUT3). The remaining 16 repositories were split equally between TDD and TLD (TDD1-TDD8, TLD1-TLD8). Repositories were assigned to each block as follows: 1) Intermediate: TDD1-TDD3, TLD1-TLD3, NUT1; 2) Novice: TDD4-TDD6, TLD4-TLD6, NUT2; 3) Mixed: TDD7-TDD8, TLD7-TLD8, NUT3.

4.2 Preparation

The tasks where implemented on machines running the Windows 10 operating system with Visual Studio 2017 and the Firefox web browser. For communication purposes, Skype and Microsoft Outlook mail client were available. Repositories were provided to the participants via a local GitLab instance.

Before the experiment a list of tasks of the first type was defined. The tasks had different levels of complexity, which could be estimated by comparing the number of acceptance tests needed to verify each task. There were 18 tasks, which were covered by 462 acceptance tests on average (SD 519). The simplest tasks, like "Deleting a signal" required only three acceptance tests, while others like "Implement first-order interpolation for signal data" were covered by 1197, up to 1750 for the most complex tasks. The cyclomatic complexity of the reference implementation for these tasks ranged from 2 for "Deleting a signal" to 13 for "Implement gap-filling for signal data using another signal", with an average of 3.5 (SD 3.43). The tasks were assigned to iterations, so all participants had to implement the same requirements in each iteration. Then, all the tasks were implemented by the authors of the experiment, resulting in a reference implementation of a full server application. The reference implementation was developed using .NET environment (C#) in the Domain-Driven Design paradigm [36], as may be seen in Fig. 6.[2] By removing the business logic from the domain layer, shell code was created, which could serve as a starting point for the participants. The components from the reference implementation, which were part of the shell code are shown as dark grey in Fig. 6. Participants were not allowed to modify these projects.

The reference implementation served multiple purposes: 1) was used to validate the definitions of tasks for the participants (their completeness and complexity), 2) enabled development of a shell of the project, given to the participants in order to separate them from infrastructural concerns, 3) was used for creating black-box acceptance tests for the participants' code, 4) is now being used by the company in its products.

2. The reference implementation is publicly available at GitHub: https://github.com/bartoszkp/TDDEvaluation in the master branch.

The reference implementation had approx. 9k LOC (Lines-Of-Code), approx. 200 reference unit tests and a few integration tests. These tests were created directly from the business requirements. Each business requirement described a desired behavior of the system, related to one of the web service methods of the server. Each reference unit test represented an aspect of or a whole business requirement related to a single web service method. After creating the reference implementation, these reference unit tests were used as a base for creating the black-box acceptance tests, used to measure code quality of each repository at the end of each iteration during the experiment.

To execute the repository switching procedure, a plan of assigning participants to repositories was also defined beforehand. The order of repositories for each participant was random, but with two constraints imposed: (i) no participant should work two weeks in a row using the same technique and (ii) each participant should always work with a repository they did not work with before. It was impossible to meet these constraints fully, because of the relation between the number of developers and the number of techniques (number of repositories) and the number of iterations, but they were satisfied for most iterations. For example, the second requirement could not be fulfilled for the Mixed block, and in the last iteration, participants did revisit some of previously encountered repositories (as they have already worked with all of them at this point). An example sequence of repositories over the course of the experiment is: TLD2, TDD1, TLD1, TDD3, TLD3, TDD2, NUT1 for iterations 1–7. The complete assignment plan can be found in the replication package.[3]

The first two weeks of the experiment were devoted to training. During first day of the training, our proposed definitions of TDD and TLD were presented (see Section 3.1) using a short slide deck, created for the purpose of this experiment. The next part was organised in a form of a workshop, consisting of demonstrating the working environment (Visual Studio 2010, Git basics, the structure of the shell code) and presenting the required committing scheme (see Section 3.10). Then, the participants were asked to implement two easy, introductory tasks – one for each training week, using both TDD and TLD, in a common repository with the shell code. This way, the participants could practice both techniques used in the experiment and get accustomed to the shell code and to the required committing scheme. All questions asked by the participants were occasionally compiled into a single FAQ-like e-mail, sent to all the participants.

4.3 Data Collection Performed

As planned, at the end of each iteration the acceptance tests and participants' unit tests were executed in an automated manner. Code quality was measured using a set of Python scripts that executed acceptance tests using web services for each repository and produced an extensive report. In particular, the report included the list of failing acceptance tests. The reports were used to create tasks of the second type,

3. Replication Package can be found at https://github.com/bartoszkp/TDDEvaluation in the "Replication package" folder of the "master" branch

TABLE 2
Process Conformance Per Participant (Represented by Their Identifier) for Each Technique (Percent of *Good* Participant's Commits in TDD and TLD Repositories Separately) and Per Participant Overall (Percent of *Good* Participant's Commits in all TDD/TLD Repositories)

Id	Category	Block	Technique TDD [%]	Technique TLD [%]	Overall [%]	Ambiguous [%]
P1	A	Intermediate	57	71	63	18
P2	A	Mixed	45	76	52	43
P3	A	Intermediate	54	70	57	33
P4	A	Intermediate	51	71	58	32
P8	A	Mixed	36	79	52	32
P10	A	Mixed	36	83	57	26
P14	A	Intermediate	65	86	69	27
P15	A	Intermediate	64	97	71	28
P16	A	Intermediate	56	85	61	29
P17	A	Intermediate	70	87	73	21
P5	B	Novice	33	66	42	8
P6	B	Novice	30	77	49	18
P7	B	Novice	20	63	27	17
P9	B	Novice	39	82	47	33
P11	B	Novice	31	66	39	29
P12	B	Novice	40	71	49	23
P13	B	Novice	27	63	41	12
P18	B	Mixed	34	67	41	40
P19	B	Mixed	27	75	30	56

The last column shows the percentage of unknown *commits, which are treated in the same way as* bad *in this experiment, but in fact are ambiguous.*

related to bugs introduced by the participants. Additionally, we have performed a manual inspection of the implementations, which led us to extending the suite of acceptance tests to cover additional corner cases. To cover all possible execution paths when testing unknown code, it was necessary to include tests covering all possible combinations of possible values of all function arguments (to some extent, i.e., when an argument is an integer, testing for all possible integers is obviously infeasible). As a result, more than 8000 acceptance tests were used for the experiment. Participants were not aware of the existence of the acceptance tests, nor did they ever see their code. Using automatically generated reports of correctness for each repository, manually prepared bug reports were assigned to appropriate repositories, usually revealing a single failing test case. This approach was also justified by our efforts to simulate a common life cycle of software, where customers usually find one scenario demonstrating an application's incorrect behavior.

Code coverage was measured by a similar set of Python scripts that executed participants' unit tests with the NCover[4] coverage-measuring tool enabled to gather information about test coverage at the end of each iteration for each participant and for each repository.

Also, a Python script was designed for the purpose of executing the process conformance calculation algorithm outlined in Section 3.10.

4. https://www.ncover.com/

TABLE 3
Descriptive Statistics for Process Conformance Within the Categories

Category	TDD conformance Mean [%]	TDD conformance St. Dev. [%]	TLD conformance Mean [%]	TLD conformance St. Dev. [%]
A	53	12	80	9
B	31	6	70	7

5 Analysis

5.1 Overview

The set of repositories, with their fully reproducible history of project changes, is the direct source for all measurements, including their varying over time through the whole period of the project's progress.[5,6] The analysis is based on these measurements.

We start with presenting the analysis of process conformance among the participants in Section 5.2. Results from this analysis allow us to test the main hypotheses under investigation in the context of how well the participants performed w.r.t. to process conformance. The next section, Section 5.3, describes the statistical methods used for analysis and presents details for each of the main hypotheses, namely:

- H_{Q1}: TDD has impact on code quality (Section 5.3.2),
- H_{C1}: TDD has impact on test quality (Section 5.3.3).

In Section 5.4 we present an analysis of the additional hypotheses H_{BQ1} and H_{BC1} (impact of mixing competence levels on code quality and code coverage) for TDD and TLD separately. Finally, in Section 5.5 we show a brief validation of the level of complexity of the tasks in the experiment, comparing TDD and TLD outcomes to working without any unit testing technique.

5.2 Process Conformance

Table 2 presents process conformance for each participant, calculated as a percentage of valid repository commits (*good* commits, see Section 3.10), where their validity depends on the correspondence to the rules of each technique. Table 3 shows this data aggregated for categories "A" and "B". Using the Shapiro-Wilk test, we can retain the assumption that the samples for TDD and TLD for both categories are normal (p-values for TDD "A", "B" and TLD "A", "B" respectively: 0.53, 0.76, 0.41, 0.17). Then, using Welch's unequal variances two-tailed T-test we compare the mean process conformance with TDD and TLD techniques between category "A" participants and category "B" participants. As can be seen in Tables 3 and 4, the results show that the process conformance is significantly higher among the intermediate programmers than among the novices. The reported p-values (0.0002 and 0.0178 for TDD and TLD resp., have been adjusted using the Bonferroni correction [37]). Cohen's d effect sizes for TDD and TLD are 0.5 and 0.14 resp. As suggested in [38] the effect size for TDD can be

5. The repositories are publicly available at GitHub: https://github.com/bartoszkp/TDDEvaluation in appropriate branches

6. The replication package for the statistical analysis is available in the same repository in the "Replication package" folder of the "master" branch

TABLE 4
Two-tailed Welch's Unequal Variances T-Test Results and Effect Sizes for Comparing Participant's Process Conformance Between "A" and "B" Categories

	TDD	TLD
T-test	0.0002	0.0178
Cohen's d effect size	0.5000	0.1386
Common language effect size [39] [%]	96	83
95% confidence interval for $(\overline{A} - \overline{B})$	22.18 ± 9.21	10.5 ± 7.59

classified as *medium* and for TLD as *very small*. However, the common language effect size [39] is 96 and 83 for TDD and TLD resp., which means there is a very high probability (0.96 and 0.83 resp.) of observing higher process conformance for an intermediate programmer than for a novice.

There is a statistically significant difference (Two-tailed Welch's T-test for samples, p < 0.05) in the process conformance between the participants who received a grade of four or higher (category "A" – the Intermediate/Mixed group) and the rest of them (category "B" – the Novice/Mixed group) in the initial, anonymous evaluation performed by the authors. This strengthens the rationale using the competence blocks and that they were created in an objective and appropriate way. The range of the results for TDD technique is wide: between 27 and 70 percent, unlike the range for TLD (between 62 and 97 percent).

The process conformance variance within each block is still considerable. Because of that, in addition to competence blocks, we include average conformance level for each technique and each participant in the statistical models used to analyse the outcomes of the experiment in the following section.

5.3 Main results

5.3.1 Statistical Methods Used

We speculate that software development technique (TDD versus TLD) has impact on code quality and test quality. These two features are represented by the number of bugs and the code coverage at the end of each iteration, measured (weekly) for each participant. For this reason we decide to take iteration as our unit of analysis (row in a data frame). There are 7 iterations overall, with 19 participants, which gives 133 possible combinations. The data is analysed using R language for statistical computing,[7] with `lme4`[8] and `ggplot2`[9] packages.

The data is correlated because the number of bugs and code coverage are repeatedly observed for the same participants and for the same tasks. Linear Mixed-Effects Models (LMM) are selected as an appropriate tool for the analysis, since they allowed dealing with problems frequently present in longitudinal studies [40], such as:

7. R: A Language and Environment for Statistical Computing, http://www.R-project.org/

8. Bates D, Mächler M, Bolker B, Walker S (2015). "Fitting Linear Mixed-Effects Models Using lme4." Journal of Statistical Software, 67 (1), 1–48. doi: 10.18637/jss.v067.i01.

9. Wickham H (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. ISBN 978-3-319-24277-4, https://ggplot2.tidyverse.org.

1) heterogeneity of participants and tasks, caused by individual differences,
2) correlated measurement errors,
3) missing observations,
4) coexisting time-invariant and time-related covariates.

To test our hypotheses we first prepare a reference model which predicts the outcome (code quality or code coverage) with the help of fixed effects (contextual variables, full list below) and random effects associated with participants and tasks. Then we build a model that represents our hypotheses by including information about the software development technique used (TDD versus TLD) with process conformance as an additional variable in the regression model. We use Likelihood Ratio Test (LRT)[41] to verify if the estimate for an additional variable is not zero. If it is, following the guidelines from [41], for $p - value < 0.05$ we assume it is justified to use the more complex model that includes the additional variable.

The following variables are used in the models:

- $techniqueTDD$ – a dummy variable that indicates the cases when the technique used in a repository for an iteration is TDD. For such cases, the values of dependent variables are relative to the reference level given by the results for TLD,
- $techniqueTLD$ – a dummy variable that indicates the cases when the technique used in a repository for an iteration is TLD. Used for cases when all three techniques are compared - then, for both $techniqueTDD$ and $techniqueTLD$, the values of dependent variables are relative to the reference level given by the results for NUT,
- $conformance$ – mean conformance level for a given participant and given technique (TDD or TLD) across the course of the study, expressed as percentage (0-100) of commits adhering to the rules of the technique used (see Section 3.10),
- $iteration$ – the iteration number,
- $bugs\_start$ – the number of failing acceptance tests at the beginning of the iteration,
- $crashes\_start$ – the number of bugs at the beginning of the iteration, which result from an unhandled exception,
- $coverage\_start$ – code coverage at the beginning of the iteration,
- $competencenovice$ – a dummy variable that indicates the participants from the Novice block. The values of the dependent variables are relative to the reference level given by results for the Intermediate block.
- $competencemixed$ – a dummy variable that indicates the participants from the Mixed block. The values of the dependent variables are relative to the reference level given by results for the Intermediate block.
- $competenceintermediate$ – a dummy variable that the indicates the participants from the Intermediate block. If present with one of the two former variables, their values are relative to the reference level given by results for the third, remaining block.

To verify each of the main hypotheses we create three LMM models:

TABLE 5
Models That Inform Hypothesis H_{Q1} (Code Quality)

	bugs_end		
	(1)	(2)	(3)
techniqueTDD		0.378 (0.419)	−1.814 (0.883)
conformance			−0.067 (0.024)
iteration	0.722 (0.321)	0.761 (0.327)	0.767 (0.314)
bugs_start	−0.021 (0.145)	−0.046 (0.147)	−0.067 (0.144)
coverage_start	−0.022 (0.017)	−0.025 (0.017)	−0.025 (0.017)
crashes_start	0.0003 (0.0004)	0.0004 (0.0004)	0.0004 (0.0004)
competencemixed	0.199 (0.592)	0.195 (0.591)	−0.764 (0.615)
competencenovice	1.950 (0.567)	1.947 (0.566)	0.639 (0.687)
Constant	1.594 (1.185)	1.544 (1.200)	7.399 (2.366)

Columns (1), (2) and (3) Present Reference Model, Technique Model and Conformance Model Respectively. Each row presents estimates with their standard errors. The values for techniqueTDD are relative to the reference level given by results for TLD technique. The values for competencenovice and competencemixed are relative to the reference level given by the results for the Intermediate block.

1) Reference Model:
This model serves as a baseline for comparison. It assumes nested characteristics of the dataset (different tasks and different participants) and fixed effects for important contextual variables: iteration number (or the number of weeks since the beginning of the experiment), competence block, number of bugs, number of crashes, and code coverage at the beginning of the iteration.
2) Technique Model:
This model includes information about the development technique used in an iteration. This tests hypotheses H_{Q1} and H_{C1} directly. The development technique is one-hot encoded with TLD as a reference group. A statistically significant parameter for the TDD level would mean that groups differ in quality / coverage, hence affirming the alternative hypotheses.
3) Conformance Model:
In this model, we include information about the average level of process conformance for each participant and for each technique. The actual results for process conformance give a fine-grained insight into how well the participants adhered to the rules of each development technique.

In each regression model, we include a random intercept for both nesting effects (participant and task).

5.3.2 Code Quality

The results for H_{Q1} for all models are presented in Table 5.

We compare the models with LRT to verify if including $techniqueTDD$ and $conformance$ variables is statistically significant. While the results for Technique Model in Table 6 suggest that including $techniqueTDD$ variable in the analysis is not justified ($p = 0.38$), the additional complexity of Conformance Model including $techniqueTDD$ and $conformance$ variables is justified ($p < 0.05$), as shown in Table 7.

The results for Conformance Model in Table 5 show that higher process conformance has a minimal effect on reducing the number of bugs (-0.067) however, following TDD gives 1.8 fewer bugs per iteration, than when using TLD.

5.3.3 Code Coverage

For testing the code coverage hypothesis, H_{C1}, we use the same strategy as for code quality. The results for H_{C1} for all models are presented in Table 8.

Similarly to the code quality analysis, we compare the models with LRT to verify if including $techniqueTDD$ and $conformance$ variables is statistically significant. The results for Technique Model in Table 9 justify including $techniqueTDD$ variable in the analysis ($p < 0.05$). In the same way as presented in Table 10, the analysis justifies

TABLE 6
LRT Comparison of the Reference Model and Technique Model (Containing $techniqueTDD$ Additional Variable) for Code Quality

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	10	516.26	543.44	−248.13	496.26			
Technique Model	11	517.48	547.38	−247.74	495.48	0.78	1	0.3771

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 7
LRT Comparison of the Reference Model and Conformance Model (Containing $techniqueTDD$ and $conformance$ Additional Variables) for Code Quality

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	10	516.26	543.44	−248.13	496.26			
Technique Model	12	511.34	543.97	−243.67	487.34	8.9122	2	0.01161

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 8
Models that Inform Hypothesis H_{C1} (Code Coverage)

	coverage_end		
	(1)	(2)	(3)
techniqueTDD		5.353 (2.085)	6.913 (4.981)
conformance			0.048 (0.140)
iteration	−0.657 (1.834)	0.338 (1.520)	0.365 (1.508)
bugs_start	0.045 (0.745)	−0.419 (0.738)	−0.419 (0.741)
coverage_start	0.326 (0.089)	0.237 (0.083)	0.234 (0.083)
crashes_start	−0.001 (0.002)	−0.0003 (0.002)	−0.0003 (0.002)
competencemixed	−1.290 (3.413)	−1.330 (3.361)	−0.649 (3.912)
competencenovice	−5.989 (3.240)	−6.065 (3.190)	−5.128 (4.231)
Constant	58.421 (6.811)	59.221 (5.688)	55.178 (13.413)

Columns (1), (2) and (3) present Reference Model, Technique Model and Conformance Model respectively. Each row presents estimates with their standard errors. The values for $techniqueTDD$ are relative to the reference level given by results for TLD technique. The values for $competencenovice$ and $competencemixed$ are relative to the reference level given by the results for the Intermediate block.

including both $techniqueTDD$ and $conformance$ comparing to the Reference Model ($p < 0.05$). As presented in Table 11, comparing Technique Model and Conformance Model suggests that it is better to stick to the simpler of these two ($p = 0.57$ for additionally including $conformance$ in the analysis).

The result for Technique Model in Table 8 show that following TDD gives 5 percentage point (pp) more code coverage per iteration, than when using TLD.

5.4 Competence Groups

The results for testing the additional hypothesis H_{BQ1} are shown in Table 12. We choose a similar approach as for the main hypotheses. For TDD and TLD separately, we start with the Reference Model which takes into account the following variables: iteration number (or number of weeks since the beginning of the experiment), number of bugs, number of crashes and code coverage at the beginning of the iteration. The alternative model, Competence Model takes into account the competence block for each participant. This results in four models, two for TDD and two for TLD which are presented in subsequent columns of Table 12.

To decide if the $competenceintermediate$ and $competencenovice$ variables have significant impact on the results we again use the LRT approach. The results are presented in Table 13 for TDD and in Table 14. The results for both techniques suggest that the competence block makes a significant difference and needs to be taken into account ($p - value < 0.05$ for both TDD and TLD). Table 12 shows that more skilled developers make fewer bugs (-1.15 per iteration) when working without the less proficient with TDD. For TLD, the effect seems negligible - fewer than one bug per iteration. On the other hand, novices tend to make more mistakes when working on code that is not shared with more proficient developers (about 3 bugs per iteration more for TDD). Again, the effect is much smaller for TLD.

TABLE 9
LRT Comparison of the Reference Model and Technique Model (Containing $techniqueTDD$ Additional Variable) for Code Coverage

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	10	885.29	912.48	−432.65	865.29			
Technique Model	11	879.70	909.60	−428.85	857.70	7.5934	1	0.005858

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 10
LRT Comparison of the Reference Model and Conformance Model (Containing $techniqueTDD$ and $conformance$ Additional Variables) for Code Coverage

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	10	885.29	912.48	−432.65	865.29			
Technique Model	12	881.38	914.00	−428.69	857.38	7.9157	2	0.0191

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 11
LRT Comparison of the Technique Model (Containing $techniqueTDD$ Additional Variable) and Conformance Model (Containing $techniqueTDD$ and $conformance$ Additional Variables) for Code Coverage

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	11	879.70	909.6	−428.85	857.70			
Technique Model	12	881.38	914.0	−428.69	857.38	0.3223	1	0.5702

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 12
Models That Inform Hypothesis H_{BQ1}: Impact of Mixing Competence Levels on Code Quality

	bugs_end			
	(1)	(2)	(3)	(4)
iteration	0.650 (0.394)	0.699 (0.380)	0.745 (0.422)	0.746 (0.367)
bugs_start	0.222 (0.159)	0.077 (0.164)	0.208 (0.156)	−0.073 (0.153)
coverage_start	−0.029 (0.021)	−0.030 (0.020)	−0.026 (0.025)	−0.010 (0.022)
competenceintermediate		−1.153 (0.652)		0.829 (0.765)
competencenovice		0.271 (0.630)		3.699 (0.892)
Constant	2.730 (1.417)	3.314 (1.446)	1.978 (1.478)	−0.049 (1.401)

Columns Show Results for Two Pairs of Models (1, 2) and (3, 4) for TDD and TLD resp. Each pair consists of a Reference Model (columns 1 and 3 for TDD and TLD resp.) and Competence Model (columns 2 and 4 for TDD and TLD resp.). Each row presents estimates with their standard errors. The values for $competencenovice$ and $competenceintemediate$ are relative to the reference level given by the results for Mixed block.

TABLE 15
Models that Inform Hypothesis H_{BC1}: Impact of Mixing Competence Levels on Code Coverage

	coverage_end			
	(1)	(2)	(3)	(4)
week	0.786 (1.591)	0.799 (1.597)	2.539 (1.100)	2.414 (1.118)
bugs_start	−0.800 (0.822)	−0.697 (0.891)	−1.415 (0.654)	−0.333 (0.705)
coverage_start	0.202 (0.096)	0.198 (0.097)	0.075 (0.078)	0.009 (0.078)
levelintermediate		0.585 (3.754)		1.671 (4.062)
levelnovice		−0.740 (3.636)		−12.020 (4.625)
Constant	63.749 (5.709)	63.787 (6.309)	61.870 (4.084)	67.623 (4.868)

Columns show results for two pairs of models (1, 2) and (3, 4) for TDD and TLD resp. Each pair consists of a Reference Model (columns 1 and 3 for TDD and TLD resp.) and Competence Model (columns 2 and 4 for TDD and TLD resp.). Each row presents estimates with their standard errors. The values for $competencenovice$ and $competenceintermediate$ are relative to the reference level given by the results for Mixed block.

The same four models are constructed for hypothesis H_{BC1} and are presented in Table 15.

The results for LRT for TDD are shown in Table 16 and for TLD in Table 17. While for TLD the competence level is significant ($p-value < 0.05$) it is not the case for TDD ($p-value = 0.93$). The results in Table 15 for TLD show that proficient developers produce better code coverage by only about $1pp$ when working alone than when sharing code with the less proficient. On the other hand, the effect is more pronounced when comparing novices working alone to when they have a proficient member on their team: $-12pp$ worse code coverage. For TDD, there is no significant difference between the Mixed block and the uniform (Intermediate and Novice) blocks.

5.5 Unit testing

For verifying if unit testing makes any sense in the context of the tasks we designed for the participants, we compare the overall results from all iterations between TDD, TLD and NUT. We create a model taking into account the NUT technique. The results are presented in Table 18. The rows with dummy variables $techniqueTDD$ and $techniqueTLD$ indicate when TDD or TLD were used resp., and show values relative to the reference level given by results for the NUT technique. While the biggest effect comes from the competence level, both unit-testing techniques result in 1.1 bugs fewer for TDD and 1.4 bugs fewer for TLD on average per iteration than without using unit testing at all.

6 Interpretation

The results of the analysis of the process conformance show that it is much easier to follow rules of TLD than TDD. This effect is especially present in the Novice block, and since the difference between category "A" participants and category "B" participants is statistically significant, we conclude that the block design with blocks representing competence

TABLE 13
LRT Comparison of the Reference Model and Competence Model (Containing $competencenovice$ and $competenceintermediate$ Additional Variables) for Code Quality With TDD

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	7	252.85	267.03	−119.43	238.85			
Technique Model	9	250.48	268.71	−116.24	232.48	6.3735	2	0.04131

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 14
LRT Comparison of the Reference Model and Competence Model (Containing $competencenovice$ and $competenceintermediate$ Additional Variables) for Code Quality With TLD

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	7	281.49	295.67	−133.74	267.49			
Technique Model	9	267.78	286.01	−124.89	249.78	17.704	2	0.0001431

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 16
LRT Comparison of the Reference Model and Competence Model (Containing $competencenovice$ and $competenceintermediate$ Additional Variables) for Code Coverage With TDD

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	7	437.32	451.5	−211.66	423.32			
Technique Model	9	441.17	459.4	−211.59	423.17	0.1539	2	0.9259

The bottom-right value shows the p-value for the justification for using the more complex model.

TABLE 17
LRT Comparison of the Reference Model and Competence Model (Containing $competencenovice$ and $competenceintermediate$ Additional Variables) for Code Coverage With TLD

	Df	AIC	BIC	logLik	deviance	Chisq Chi	Df	Pr(> Chisq)
Reference Model	7	451.12	465.30	−218.56	437.12			
Technique Model	9	445.50	463.73	−213.75	427.50	9.6142	2	0.008172

The bottom-right value shows the p-value for the justification for using the more complex model.

groups was an appropriate choice for this experiment. This also indicates that the TDD technique has some considerable entry barrier, and was hard to follow for the novices, with at most 40 percent process conformance among the novices for TDD. The results for the intermediate developers also suggest that TDD is much harder than TLD: $51\% - 70\%$ range for TDD versus $70\% - 97\%$ for TLD.

The results and the analysis of the main outcomes of the experiment suggest a significant impact of the software development technique used on code quality and code coverage. Using TDD gives on average 1.8 fewer bugs per week and on average $5pp$ more code coverage per week, than when following TLD. This implies the following conclusion for the main hypotheses of this paper:

- H_{Q0} hypothesis is rejected in favor of H_{Q1}: there is a significant difference between the code quality of code created using TDD and TLD, in favor of TDD,
- H_{C0} hypothesis is rejected in favor of H_{C1}: there is a significant difference between the coverage of code created using TDD and TLD, in favor of TDD.

TABLE 18
Model for Number of Bugs at the End of the Iteration for TDD and TLD With the Reference Level Given by the NUT Technique

	bugs_end
techniqueTDD	−1.056 (0.627)
techniqueTLD	−1.360 (0.622)
bugs_start	−0.124 (0.146)
crashcount_start	0.001 (0.0004)
competencemixed	−0.049 (0.612)
competencenovice	2.074 (0.595)
Constant	3.741 (0.979)

The effect for code quality is 1.8 which is relatively small for short projects. Given the considerable entry barrier for TDD, it may not be worth pursuing for teams not familiar with this technique, just for avoiding at most eight bugs at the end of a month-long project. This of course depends on the criticality of the software and the severity of the bugs - for systems dealing with business domains such as industry process control, aviation or medicine, even a single bug is unacceptable. For projects lasting longer than a month, 1.8 bugs per week piles up to a significant backlog, which may be impossible to address when the team is focused on delivering subsequent features.

The effect for code coverage also seems small - only $5pp$; however, the code coverage at the end of an iteration comes also from the code coverage at the beginning of the iteration ($0.24pp$ with 0.083 standard error). So, every percent point of code coverage early in the project is easy to maintain and hard to reclaim. The reason is that most features developed by the participants are independent, and there is no requirement in the experiment to improve the code coverage first (as opposed to the requirement to fix all remaining bugs first).

As a result, TDD seems to be a good practice to recommend for long-term projects, where 1.8 bugs per week costs more than an extensive TDD training for the developers.

Including the average conformance into the statistical analysis is essential for observing the impact of TDD on code quality. This is because there is considerable variance in the conformance level between participants from the same competence block. Thus, taking into account the actual measure of the degree of conformance with the development technique's rules produce better differentiation between the actual proficiency levels of the developers. Figs. 7 and 8 present how the process conformance influences the outcome of the experiment. While TDD is much harder to follow (the levels of process conformance are considerably lower than for TLD), it improves code quality much earlier on the conformance scale than TLD. Adhering to TDD rules 60 percent of the time seems to be as good as working according to TLD rules for $80 - 90\%$ of the time. Similarly, much lower conformance levels ($40 - 70\%$) give better code coverage than even the biggest conformance levels for TLD (exceeding 90 percent). The reason might be that

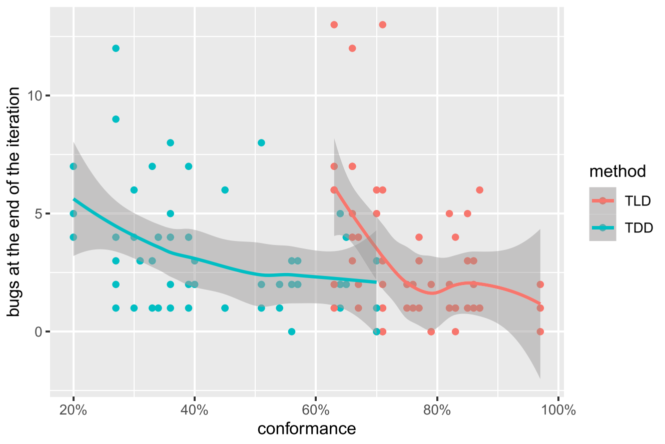

Fig. 7. Diagram of the influence of process conformance on the number of bugs for TDD and TLD. All the data points have conformance in the range $[20\%, 97\%]$ and the number of bugs in the range $[0, 17]$ - graph axes are clamped to these ranges for better graph readability.

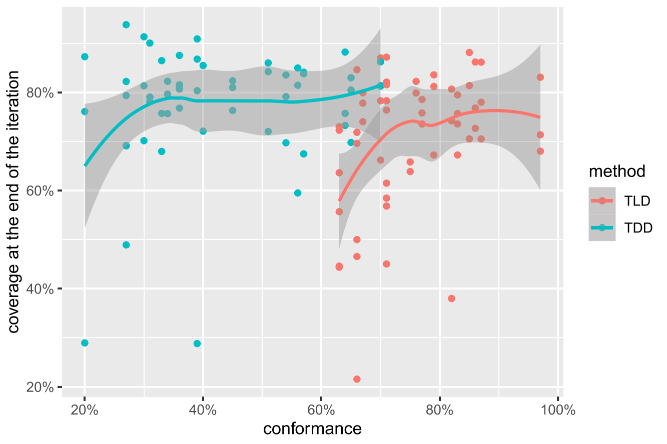

Fig. 8. Diagram of the influence of process conformance on the code coverage for TDD and TLD. All the data points have conformance in the range $[20\%, 97\%]$ and the coverage in the range $[22\%, 94\%]$ - graph axes are clamped to these ranges for better graph readability.

TDD's rules are more strict and give more structure to the way a developer works. As a result, even following these rules loosely, gives more rigor to the software development process than a simpler framework, such as TLD.

This also shows that process conformance is an essential metric to take into account in experiments such as this one. It is impossible to compare two techniques fairly when one is followed in a considerably stricter manner than the other.

The results of the last part of the analysis additionally validate the split between the Intermediate and the Novice developers. The competence level is a significant factor for code quality with TDD and TLD, with the biggest impact on the Novices creating 3.6 bugs per iteration more when working without a support of proficient developers. At the same time, Intermediates make fewer mistakes (1.2 bugs per iteration) when working without novices on the team.

Another interesting outcome of the analysis can be seen for code coverage. On the one hand, the competence level is a significant factor for TLD, showing that Novices lose $12pp$ of code coverage per iteration when working alone, and Intermediates gain $1.6pp$ of code coverage per iteration when working without the influence of the Novices. On the other hand, the competence level is not significant for TDD when comes to code coverage. The reason for this might be that TDD, being a more strict process than TLD, narrows the gap between the Intermediate and the Novice developers when it comes to code coverage. Overall, we conclude that H_{BQ0} and H_{BC0} hypotheses should be rejected in favor of H_{BQ1} and H_{BC1}: mixing competence levels has significant impact on code quality (for TDD and TLD) and code coverage (for TLD).

These two findings related to Novices working alone or together with Intermediates imply two important conclusions:

- for the industry: novice developers have a significant negative impact on the work done by their more experienced teammates in terms of external code quality. This effect can be partially mitigated by using TDD, as the difference becomes insignificant in terms of code coverage,
- for research: isolating intermediate and novice developers has a significant impact on the outcomes of an experiment.

Finally, the short validation at the end of the analysis section suggests that the tasks chosen for the experiment are of appropriate complexity, making it easier not to make mistakes if unit testing is employed. Thus, applying unit testing techniques for these tasks is justified.

6.1 Threats to Validity

The validity of the proposed experiment design is analysed with respect to four types of validity: conclusion, construct, internal, and external [42]. The conclusion validity threats impact the quality of the statistical analysis of the results, construct validity threats impact the relevance of the results to the studied phenomenon, internal validity threats influence the independent variables without the researcher's knowledge and external validity threats influence the possible generalization of the results to the industrial practice.

The following threats to validity are identified:

- Conclusion validity: Sample size, Selective and rigorous process conformance analysis, Different task types
- Internal validity: Selection of the participants, Task experience carry-over, Technique switching carry-over, Fatigue of the participants, Maturation of the participants, Enforced committing scheme
- External validity: Selection of the participants, Task experience carry-over, Technique switching carry-over, Separation from infrastructural concerns, Enforced committing scheme
- Construct validity: Selection of the participants, Selective and rigorous process conformance analysis, Task experience carry-over, Technique switching carry-over, Enforced committing scheme, Separation from infrastructural concerns

We analyse them in the following paragraphs, with the last subsection containing the conclusion of the validity threat analysis.

Sample Size. Given the complexity and the diversity of the subject, 19 participants working on an over month-long project with six iterations does not appear to be a large sample. However, in the context of related work, 19 subjects are not uncommon (see Table 1 in Section 2.1). Also, with about

3040 person-hours, this experiment is classified as *large* using the scale proposed in [31].

Selective and Rigorous Process Conformance Analysis. The process conformance analysis proposed in this paper is rigorous in the sense that commits classified as *unknown* are not ignored, but treated in the same way as *bad*. Such commits may contain a valid step, perfectly acceptable within the rules of a technique – however, our classification algorithm is unable to detect that. While this impacts the values of the $conformance$ variable, and thus, the results of the statistical tests, it makes our conclusions valid for a very strict definition of the analysed techniques. On average, there are only 28 percent of *unknown* commits among the participants (SD 11 percent), so the vast majority of their work is classified properly.

Another aspect of rigorous process conformance is that very few participants appear to have followed the rules of techniques as they should have. It can be argued that it is hard to make any conclusions about TDD when even the more experienced participants have not been working according to the rules of TDD roughly half of the time. However, our process conformance validation has very high sensitivity and low specificity to ensure not counting any actions not evidently valid as representative for a given technique. Thus, we argue that the differences among the participants are more important than the absolute values of the results of the process conformance validation procedure and the statistical differences between the competence blocks are significant.

Additionally, while the process conformance analysis is rigorous when comes to the committing scheme it does not cover ensuring that no tests were written when working with the NUT technique, and that the development steps when working with TDD and TLD are of appropriate sizes. For example, even though TLD is defined as creating tests after first draft of a complete task has been completed, there is no way of verifying that the participants did so. This is a very hard problem in general, that can be solved only by screen recording or personal supervision of each participant. *Selection of the Participants*. Personal experience has a great impact on the effectiveness of using development techniques. A comparison of TDD and TLD, as equally well defined methods, should be organized with the participation of developers equally acquainted with them. This is theoretically possible, but difficult to achieve. What makes it especially hard is the status of the TDD method – still considered “new” and not universally adopted, both in academic education and in the industry. This is mitigated by the following measures:

- the same training attended by the same participants, followed up by one warm-up iteration,
- dividing participants into competence blocks,
- switching of techniques, so every participant is contributing to each technique equally,
- commit strategy, which provides a uniform, structured process of working for the participants,
- monitoring of process conformance, which allows excluding commits not matching rules of the required technique and is accounted for in statistical analysis.

Another aspect of selection of the participants validity threat is participants’ personal attitudes. Any comparison between software development techniques cannot be blind, the participants need to know what method they are using and thus cannot escape any personal attitude towards them. In the case of the TDD method, there still seem to exist some strong convictions in the industry that make it look novel, counter-intuitive, challenging, and weird. All these features can have a psychological impact on a programmer’s approach, making them more or less willing and more or less at ease with the work.

Finally, the participants were inexperienced developers and the experiment aims to measure aspects of techniques employed by professionals. This is not necessarily a bad thing: the more experienced a developer the more personal convictions toward specific techniques, including having *favorite* or *hated* technique. While it is not uncommon in the industry to employ students, that group may seem to be not representative for all developers. Students do however represent the group of people that will, in a year or two, become *junior developers*. Thus, the conclusions of this paper are mainly relevant to junior programmers, however, when considering a new programming technique, students have been shown to be as representative as professionals in software engineering research [35].

Maturation of the Participants. The experience of the programmers can significantly increase as the project develops and change the quality of their work through time. Neither can we exclude a dependency resulting from communication between the participants during each iteration. The participants were asked not to talk to each other about their tasks/repositories, but we are unable to judge what their mutual influence was. This aspect is hard to avoid – especially novice programmers can be expected to advance quickly in their fluency with an exercised technique.

Task Experience Carry-Over. The carry-over problem arises when a task is left unfinished in a repository (i.e., some acceptance tests for this task are failing) at the end of an iteration. During the next iteration, this task is solved by another participant. The solution is however influenced by the fact that it is this participant’s second attempt, because they have already been working on the same task in a different repository, during the previous iteration. Implementing the same task for the same project twice in a row does not seem like something that happens often in the real world. The impact of such a scenario however does not seem completely undesirable – the failing acceptance tests and unit test coverage from the issue’s original iteration are correctly accounted for and the accumulation of bugs is something that happens in the real world. The disturbance is that the next developer has less time to work on issues assigned to their current iteration because bugs need to be fixed first. But this again, can be seen as a consequence of the technique assigned to that particular repository.

Technique Switching Carry-Over. The authors of [9] argue that switching between development techniques can significantly distort the results of an experiment. Switching from TLD to TDD seems not to be an issue, since the qualifications for the first one are also essential to the latter. Switching back from TDD to TLD may be problematic. We cannot rule out the possibility that an implementation committed before a unit test was actually created with a unit test or at least a rough idea for a unit test in mind. We have tried to minimize this possibility with two counter-measures: (i) every participant attended exactly the same two week training, where TDD and TLD were presented as two very similar approaches with some vital differences in the ordering of the activities involved in creating implementation and tests, (ii) the repository switching procedure was always executed through the weekend, so, each participant finished their work on Friday, and started working with a different technique on Monday, after a two day rest. Additionally, TDD and TLD iterations were separated with NUT iterations whenever possible.

Also, we cannot rule out a possibility that for any technique the order of commits was fake and did not reflect the true sequence of implementation steps. Our only countermeasure for this problem was that the participants were not aware of the existence of process conformance calculation procedure and they knew that repositories were anonymized after each iteration, so their performance would not be used for any personal evaluation. Additionally, they were asked for honesty, and meeting each of them personally, we had no suspicion of bad intentions (but this of course cannot be verified).

Fatigue of the Participants. The schedule during the experiment was relaxed – the subjects were allowed to decide how many hours they are willing to work each day, as performance and time optimization were not under investigation in the experiment. The working time was not controlled or monitored, however excessive overtime was impossible due to offices being closed in the afternoon. The subjects had no other responsibilities, so fatigue could only come from the participation in the experiment itself, which is unavoidable.

Enforced Committing Scheme. The enforced work scheme, although mitigating some other validity threats, introduces an uncommon work pattern for the participants. Instead of focusing on the technique, they can be only blindly following the commit pattern. Also, this way of working does not represent the way developers work in the real world. Thus, it can be argued that the results of the experiments cannot attributed solely to TDD, but were a result of the rigid committing scheme. To mitigate, during training weeks, the commit scheme was introduced as part of the techniques (as additional steps), not as a driving force of the technique. While there is no way to decide how this pattern could influence the results, it is important to note that the participants were not aware of the results of the conformance validation, so they could not try to influence their results during the experiment. Also, committing scheme is related to the way a source control system is used, not necessarily relevant to the development technique and can be seen only as a minor distraction when writing the code.

Separation From Infrastructure Concerns. In industry, programmers usually work with the complete system and are not restricted to a selected set of objects within it. Even when the system is split into components and different teams are responsible for various items, complete separation from underlying infrastructure is not possible. However, when a new person is joining the already existing large project, it takes weeks or months to comprehend the complete system. During that time the novice programmer treats the unknown parts of the system as black-box, similarly as in this experiment.

Different Task Types. The difference between bugs – leftovers from previous iteration and new features – tasks assigned to the current iteration is not taken into account directly in the analysis, so may confound the results. However, the $bugs\_start$ variable is included in the linear models used for analysis and it tells how many leftovers there were in each iteration. Since the number of new features is predefined and identical among all participants, the values of this variable imply the relation between the amount of bug-fixing work and new-features work.

Conclusion. The results of this experiment are analysed using well known statistical techniques. The chosen variables: number of failing acceptance tests and code coverage are appropriate for representing the quality of produced software and can be measured in an objective way, relying on automatic tests and code coverage tool. The validity threats that seem to be the most impactful are related to the enforced committing scheme and the sample size. Hence, the conclusions from this experiment might not relate to a typical, practical approach to TDD and the experiment should be replicated to increase the sample size.

7 Conclusions and Future Work

This paper presents an experiment intended to compare the effects of software development techniques, namely Test-Driven Development (TDD) and Test-Last Development (TLD) on code quality. The experiment's design combines the following features:

- Using a commercial, industrial project, with real-world requirements;
- Validating how well the participants conform to the rules of the required programming technique;
- Focusing the implementation on the business logic and separating the participants from infrastructural concerns;
- Random repository switching, among 19 repositories with the same requirements implemented;
- Validating tasks' complexity to justify application of unit testing techniques.

The most important findings are as follows:

- TDD results in significantly higher external code quality than TLD (1.8 bugs per week fewer),
- TDD results in significantly higher test quality than TLD ($5pp$ code coverage per week more),
- TDD is a difficult practice to follow, especially for novices, resulting in significantly lower process conformance with the technique's rules than for TLD,
- including process conformance results in analysis is essential to observe TDD's positive influence,
- even though following TDD strictly is much more difficult for novices, it results in no significant difference w.r.t. code coverage between intermediate and novice developers (this is not the case for TLD),
- for external code quality, isolating intermediate developers from the novices has a significant impact

on the outcome of experimental comparison of software development techniques.

All findings are consistent with the majority of current research.

Future work includes continuing the experiment on the same code base, with follow-up requirements (possibly from the company's customers). Data from two months and another 19 developers would greatly improve the quality of the results. Including some professional developers in the experiment with the same code-base and requirements would provide valuable insights into the difference between TDD and TLD. Given the findings of the current experiment, for a similar task complexity and a similar number of participants the following experiment could use all the available resources on comparing TDD and TLD in isolated competence groups, without sacrificing part of the data for the Mixed block and No Unit Tests technique. Also, to help understanding and tuning the proposed process conformance analysis, a baseline could be obtained from someone known to know TDD and TLD well.

Acknowledgments

This work was funded within the GasLux product group, by Transition Technologies S.A. Thanks are due to Dr Katarzyna Bartkiewicz and Dr Piotr Gawkowski for the article review and valuable comments.

References

[1] M. Cohn, *Succeeding With Agile: Software Development Using Scrum*, Reading, MA, USA: Addison-Wesley, 2010.

[2] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, and H. Erdogmus, "What do we know about test-driven development?," *IEEE Softw.*, vol. 27, no. 6, pp. 16–19, Nov./Dec. 2010.

[3] H. Munir, M. Moayyed, and K. Petersen, "Considering rigor and relevance when evaluating test driven development: A systematic review," *Inf. Softw. Technol.*, vol. 56, no. 4, pp. 375–394, 2014.

[4] D. Janzen and H. Saiedian, "Does test-driven development really improve software design quality?," *IEEE Softw.*, vol. 25, no. 2, pp. 77–84, Mar. 2008.

[5] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A dissection of the test-driven development process: Does it really matter to test-first or to test-last?," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 597–614, Jul. 2017.

[6] L. Madeyski, *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Berlin, Germany: Springer, 2010.

[7] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: Industrial case studies," in *Proc. 5th ACM/IEEE Int. Symp. Empir. Softw. Eng.*, 2006, pp. 356–363.

[8] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: Results and experiences of four industrial teams," *Empir. Softw. Eng.*, vol. 13, no. 3, pp. 289–302, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1007/s10664–008-9062-z

[9] M. Pančur and M. Ciglarič, "Impact of test-driven development on productivity, code and tests: A controlled experiment," *Inf. Softw. Technol.*, vol. 53, no. 6, pp. 557–573, Jun. 2011.

[10] A. Tosun *et al.*, "An industry experiment on the effects of test-driven development on external quality and productivity," *Empir. Softw. Eng.*, vol. 22, pp. 1–43, 2016.

[11] A. Ko *et al.*, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, vol. 43, pp. 1–44, 2011.

[12] T. W. Williams, M. R. Mercer, J. P. Mucha, and R. Kapur, "Code coverage, what does it mean in terms of quality?," in *Proc. Int. Symp. Product Quality Integrity Annu. Rel. Maintainability Symp.*, 2001, pp. 420–424.

[13] D. Fucci, B. Turhan, and M. Oivo, "Impact of process conformance on the effects of test-driven development," in *Proc. 8th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2014, Art. no. 10.

[14] K. Becker, B. de Souza Costa Pedroso, M. S. Pimenta, and R. P. Jacobi, "Besouro: A framework for exploring compliance rules in automatic TDD behavior assessment," *Inf. Softw. Technol.*, vol. 57, pp. 494–508, 2015.

[15] Y. Wang and H. Erdogmus, "The role of process measurement in test-driven development," in *Proc. Conf. Extreme Program. Agile Methods*, 2004, pp. 32–42.

[16] H. E. Hongbing Kou, and Philip M. Johnson, "Operational definition and automated inference of test-driven development with zorro," *Automated Softw. Eng.*, vol. 17, no. 1, 2010, Art. no. 57.

[17] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita, "Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from hackystat-UH," in *Proc. Int. Symp. Empir. Softw. Eng.*, 2004, pp. 136–144.

[18] J. Hoewe, *Manipulation Check*. Atlanta, GA, USA: American Cancer Society, 2017, pp. 1–5.

[19] D. Falessi *et al.*, "Empirical software engineering experts on the use of students and professionals in experiments," *Empir. Softw. Eng.*, vol. 23, pp. 452–489, 2017.

[20] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio, "Productivity of test driven development: A controlled experiment with professionals," in *Proc. 7th Int. Conf. Product-Focused Softw. Process Improvement*, 2006, pp. 383–388.

[21] B. George and L. Williams, "An initial investigation of test driven development in industry," in *Proc. ACM Symp. Appl. Comput.*, 2003, pp. 1135–1139.

[22] B. George and L. Williams, "A structured experiment of test-driven development," *Inf. Softw. Technol.*, vol. 46, no. 5, pp. 337–342, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584903002040

[23] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 226–237, Mar. 2005.

[24] L. Williams, E. M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *Proc. 14th Int. Symp. Softw. Rel. Eng.*, 2003, pp. 34–45.

[25] K. M. Lui and K. C. C. Chan, "Test driven development and software process improvement in china," in *Proc. 5th Int. Conf. Extreme Program. Agile Processes Softw. Eng.*, 2004, pp. 219–222.

[26] O. H. M. Müller, "Experiment about test-first programming," *IEEE Proc. Softw.*, vol. 149, no. 5, pp. 131–136, Oct. 2002.

[27] M. M. Müller and W. F. Tichy, "Case study: Extreme programming in a university environment," in *Proc. 23rd Int. Conf. Softw. Eng.*, 2001, pp. 537–544.

[28] D. S. Janzen and H. Saiedian, "On the influence of test-driven development on software design," in *Proc. 19th Conf. Softw. Eng. Educ. Training*, 2006, pp. 141–148.

[29] H. Munir, K. Wnuk, K. Petersen, M. Moayyed, and K. Se, "An experimental evaluation of test driven development vs. test-last development with industry professionals," in *Proc. ACM Int. Conf. Proc. Series*, 2014, pp. 1–10.

[30] M. C. Turnu, "Modeling and simulation of open source development using an agile practice," *J. Syst. Archit.*, vol. 52, no. 11, pp. 610–618, 2006.

[31] B. Turhan, L. Layman, M. Diep, F. Shull, and H. Erdogmus, *How Effective is Test Driven Development*. Sebastopol, CA, USA: O'Reilly Media, 2010, pp. 207–219.

[32] W. Bissi, A. G. S. S. Neto, and M. C. F. P. Emer, "The effects of test driven development on internal quality, external quality and productivity: A systematic review," *Inf. Softw. Technol.*, vol. 74, pp. 45–54, 2016.

[33] K. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley, 2002.

[34] M. L. Mitchell and J. M. Jolley, *Research Design Explained*. Belmont, CA, USA: Wadsworth Publishing, 2012.

[35] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 666–676.

[36] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Reading, MA, USA: Addison-Wesley, 2003.

[37] R. Miller, *Simultaneous Statistical Inference*. New York, NY, USA: Springer, 2012.

[38] S. Sawilowsky, "New effect size rules of thumb," *J. Modern Appl. Statist. Methods*, vol. 8, pp. 597–599, 2009.

[39] K. O. McGraw and S. P. Wong, "A common language effect size statistic," *Psychol. Bull.*, vol. 111, pp. 361–365, 1992.

[40] R. D. Gibbons, D. Hedeker, and S. DuToit, “Advances in analysis of longitudinal data,” *Annu. Rev. Clin. Psychol.*, vol. 6, no. 1, pp. 79–107, Mar. 2010.
[41] H. Baayen, D. Davidson, and D. Bates, “Mixed-effects modeling with crossed random effects for subjects and items,” *J. Memory Lang.*, vol. 59, pp. 390–412, 2008.
[42] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.

Bartosz Papis received the MSc and PhD degrees in computer science from the Warsaw University of Technology, Poland, in 2008 and 2015, respectively. From 2006 to 2018 he was with Transition Technologies S.A. in Warsaw, working as a developer, software architect, and scrum master. From 2018 he was with Sonova working as a software engineer and since 2019 he has been working at Google. His research interests include machine learning and software engineering.

Konrad Grochowski received BSc and MSc degrees in computer science from the Warsaw University of Technology, Poland, in 2008 and 2010, respectively. From 2006 to 2014 he was with Transition Technologies S.A. in Warsaw, working as a Software Architect. From 2014 he is with Warsaw University of Technology working as research assistant. Since 2014 he is also working with N7 Space Sp. z o.o. on development of space on-board software. His research interests include software engineering and systems dependability.

Kamil Subzda received the MSc degree in electronics from the Wrocław University of Science and Technology, Poland, in 2009. From 2008 he is with Nexwell Engineering in Wrocław, working as a developer and embedded software architect.

Kamil Sijko received the MA degree in psychology. He worked at the Warsaw School of Social Sciences, Polish Educational Research Institute, currently working at Transition Technologies S.A. as a lead data scientist. Experienced in social research (surveys, sampling, qualitative and quantitative approach), educational research (IRT, international surveys, complex sampling). Currently working with big-data in medical field. founder and board member of CoderDojo Poland Foundation, fellow mentor at CoderDojo Zambrów.