Preprint of an article: Lech Madeyski and Marcin Kawalerowicz, "Continuous Test-Driven Development: A Preliminary Empirical Evaluation using Agile Experimentation in Industrial Settings", vol. 733 of *Studies in Computational Intelligence*, pp. 105–118, Springer, 2018.
DOI: 10.1007/978-3-319-65208-5_8 [BibTeX] Draft:
http://madeyski.e-informatyka.pl/download/MadeyskiKawalerowicz18.pdf

Continuous Test-Driven Development: A Preliminary Empirical Evaluation using Agile Experimentation in Industrial Settings

Lech Madeyski and Marcin Kawalerowicz

Abstract Test-Driven Development (TDD) is an agile software development and design practice popularized by the eXtreme Programming methodology. Continuous Test-Driven Development (CTDD), proposed by the authors, is the recent enhancement of the TDD practice and combines TDD with the continuous testing (CT) practice that recommends background testing. Thus CTDD eliminates the need to manually execute the tests by a developer. This paper uses CTDD research to test out the idea of Agile Experimentation. It is a refined approach performing disciplined scientific research in an industrial setting. The objective of this paper is to evaluate the new CTDD practice vs. the well-established TDD practice via a Single Case empirical study involving a professional developer in a real, industrial software development project employing Microsoft .NET. We found that there was a slight (4 minutes) drop in the mean red-to-green time (i.e., time from the moment when any of the tests fails or the project does not build to the time when the project compiles and all the tests are passing), while the size of the CTDD vs. TDD effect was non-zero but small ($d-index = 0.22$). The recorded results are not conclusive but are in accordance with the intuition. By eliminating the mundane need to execute the tests we have made the developer slightly faster. If the developers that use TDD embrace CTDD it can lead to small improvements in their coding performance that, taking into account a number of developers using TDD, could lead to serious savings in the entire company or industry itself.

Lech Madeyski
Wroclaw University of Science and Technology, Faculty of Computer Science and Management, Wyb.Wyspianskiego 27, 50-370 Wroclaw, POLAND,
e-mail: Lech.Madeyski@pwr.edu.pl

Marcin Kawalerowicz
Opole University of Technology, Faculty of Electrical Engineering, Automatic Control and Informatics, ul. Sosnkowskiego 31, 45-272 Opole, POLAND, and
CODEFUSION Sp. z o.o., ul. Armii Krajowej 16/2, 45-071 Opole, POLAND,
e-mail: marcin@kawalerowicz.net

1 Introduction

Recent surveys, case studies and white papers show wide adoption of agile methods and practices in the software industry [24, 9]. Test-Driven Development (TDD) introduced by Beck [5] is, alongside pair programming, one of the main practices in Extreme Programming (XP) which in turn is one of the main Agile software development methods. Beck in his influential book [4] shows (see Figure 4 in Chapter 11) that both TDD and pair programming are the most interconnected practices in XP, while recent surveys and papers show increasing adoption of TDD among professional developers [14, 23, 1, 22, 6, 18].

Continuous compilation is a practice, used in modern IDEs (Integrated Development Environment), that provides source code compilation in the background that gives the developer immediate feedback about the potential compilation errors while he edits the source code. It is practically a standard in all modern IDEs like Microsoft VisualStudio (since 2010), Eclipse, IntelliJ IDEA and so on. A recent extension of continuous compilation that adds a background testing to the compilation hit the mainstream of IDEs. It is called continuous testing (CT) [20], [21]. Using CT the developer gets not only background compilation but background testing as well. CT provides an immediate test feedback on top of compilation feedback. Microsoft introduced CT in the two highest and most expensive versions of Visual Studio 2012 (Premium and Ultimate).

In our paper from 2013 [16] a combination of those two practices TDD and CT, along with the continuous testing AutoTest.NET4CTDD open source plugin, was proposed and the preliminary feedback, via a survey inspired by Technology Acceptance Model (TAM), from developers in an industrial setting was collected. The new practice, combining CT and TDD was called Continuous Test-Driven Development or CTDD (the practice is described in Section 2). The findings of that paper were that CTDD could gain acceptance among TDD practitioners. Our initial speculations were that the benefits regarding development time could be small for a single developer, but could turn to be large due to the size of the software industry. There was no other scientific research published comparing TDD and CTDD. Hence, we decided to assert our initial speculations that the new practice could provide some time-related benefits in comparison to the usage of TDD. Our research goal and hypothesis are refined in Sections 3.1 and 3.2, respectively.

In our previous paper [17], we argued that software engineering needs agile experimentation and formulated Agile Experimentation Manifesto with the aim to support lightweight, agile experimentation in industrial software development setting, where fully fledged experiments are often not feasible as it is not easy to involve professional software developers in a large-scale experimental research. Because of this natural limitation, in accordance with the first postulate of our Agile Experimentation Manifesto (Use small-n and single case experiments rather than large scale experiments to cut costs and

enable experimentation), we decided to use the single-case/small-n experiment design (described in detail in Section 3.3). We use single developer to compare the TDD and CTDD practices in a baseline-intervention experiment design, where we will treat TDD as a baseline and CTDD as an intervention.

2 Background

As it was proposed by the authors [16], CTDD is a software development practice that combines TDD and Continuous Testing. If a developer uses TDD he needs to first write a test and verify if the tests fails (or the build fails because the functionality that is supposed to be tested was not written yet), then writes the functionality to quickly satisfy the test and executes it to check if it is the case, then he refactors the code regularly performing the tests to check if he did not break anything. We proposed to add the notion of continuous testing to TDD. The idea is that a developer that uses CTDD is not forced to perform the tests by himself. In continuous testing, the code is being compiled and tested automatically after the developer writes it and saves the file. So the need to manually trigger the testing was removed, potentially adding value to the process (via more frequent and earlier feedback from the amended code to the developer). In our earlier paper [16], we conducted a quick TAM inspired survey among the professional software developers that encouraged us to proceed with the new CTDD practice research (and supporting tool development), as it seems to be an improvement over the baseline TDD practice which could be widely accepted and adopted by practitioners. We make a conjecture that if the developers that use TDD embrace CTDD it can lead to small improvement in their coding performance that, taking into consideration a number of developers using TDD, could lead to noticeable savings within a company adopting the practice, not to mention the entire software industry.

AutoTest.NET4CTDD tool [16] that we co-developed makes it possible to use CTDD practice and can gather, in real time, various statistics during software development (the feature that the IDEs with built in CT, e.g., Visual Studio, are lacking). The tool was designed to execute the tests that are related to the changes the developer has made in the project. It detects what tests to run regardless of their type – unit, integration, system. As long as the tests are in the project and are related to the change they are run. If those tests are used as regression tests they will also be targeted by the AutoTest.NET4CTDD tool.

To allow empirical comparison of CTDD with the baseline TDD practice, we needed another tool for gathering the same data while using TDD. We did not found any suitable tool for the project setting we had access to. Hence, we had to develop our tool called NActivitySensor described in the appendix to our recent paper [17]. Having both tools, we were able to gather

data needed to perform the empirical evaluation of the investigated software practices, TDD and CTDD. We were interested in the time that elapses from the moment when any of the tests fails, or the project does not build to the time where the project compiles, and all the tests are passing. We called this the red-to-green time (RTG time). We focused on the RTG time as it is where the advantage of the CTDD practice over the TDD practice may appear and be easily measured using the aforementioned tool set we developed.

3 Experiment Planning

3.1 Experiment Goal

The objective of the empirical study is to determine the differences in individual performance of a professional developer using CTDD vs. TDD. Our quasi-experiment (as without random assignment of participants to different groups) is motivated by a need to understand the differences in individual performance while using the established TDD practice and the new CTDD practice introduced earlier by the authors [16].

The object of the study is the participant of a real software project. He is a professional software developer, computer science graduate with the MSc degree and two years experience (at the time of the experiment) in commercial software development.

The purpose of the quasi-experiment is to evaluate the individual performance when CTDD and TDD are employed. The experiment provides insight into what can be expected regarding individual performance when using the CTDD practice instead of TDD.

The perspective is from the point of view of the researchers, i.e. the researchers would like to know if there is any systematic difference in the CTDD vs. TDD performance.

The main effect studied in the experiment is the individual performance measured by the RTG time introduced in Section 2. As mentioned earlier, the RTG time elapses from the moment the project is rendered not compiling or any of the tests is not passing, until the time the project builds, and all the tests are passing. The shorter the time, the more time the developer is spending doing actual work, i.e., development of new features. Because the task of executing the tests and waiting for their result is common and constantly recurring, we can assume that if we can reduce the time spent handling the tests we can reduce waste and make developers more productive.

The experiment is run within the context of the real software project in which a civil engineering software for calculation of concrete constructions was developed. The investigated developer developed modules for data exchange between this software and other data formats.

The summary of scoping of the study made according to the goal template by Basili et al. [3] is as follows: *Analyze* the CTDD practice *for the purpose of* evaluation *with respect to* its effectiveness *from the point of view of the* researcher *in the context of* a professional software developer in a real-world (industrial) software development project.

3.2 Hypothesis Formulation

An important aspect of experiments is to know and to formally state precisely what is going to be evaluated.

Null hypothesis, H0: There is no difference in the developer coding velocity, measured as the RTG time (T_{RTG}) introduced in Section 2, between the CTDD and TDD observation occasions, i.e., H0: $T_{RTG}(CTDD) = T_{RTG}(TDD)$

Alternative hypothesis, H1: $T_{RTG}(CTDD) < T_{RTG}(TDD)$

As CTDD reduce waste mentioned in Section 3.1, we may assume a directed hypothesis.

3.3 Experiment Design

We decided to perform the experiment in a real professional software development environment and on a real project to increase the external validity of the obtained results to the level hard to achieve with computer science students at a university lab. It would help us to generalize the results of our experiment to industrial practice. However, we need to take into account that the industrial resources willing to spend their precious time on research investigation instead of on commercial projects are scarce. Furthermore, we are aware that the goals of the different parties regarding experimentation are not necessary converging. For example, researchers are interested in performing controlled experiments in real projects resulting in reliable conclusions that lead to industry process improvement. The project owner is mainly interested in return of investment (ROI) and only secondarily in incorporating the results of experimentation provided they have a positive impact on the project itself. Furthermore, professional developers are expensive and busy people, while projects they are working on are seldom available for scientific research. Up front, we gave up the idea to perform a large scale experiment in an industrial setting, and we had no possibility to perform the project or even its part twice, once with the traditional (TDD) approach and once with the new one (CTDD), as it would require a lot of money. However, we have access to a small software development company based in Poland where one of the customers had a scientific background and was kind enough to allow

some experimentation on a small staffed project. So we had a single object (professional software project) and single subject (developer) available for experimentation.

A good experimental design removes threats to internal validity, i.e., eliminate alternative explanations. A powerful tool in achieving internal validity is randomization. In classic large-n experimental designs, treatments (interventions) are randomly assigned to subjects (participants). Unfortunately, it is unattainable in single-case studies, but it does not mean that it is impossible to randomly assign treatments to observation occasions. If we make a series of observations on a single case, or on a few cases, we can think of each as an observation occasion. As we randomly assign treatments to participants in large-n experimental designs, we may randomly assign treatments to observation occasions in single-case or small-n experimental designs. It is worth mentioning that classic parametric tests can not be used to analyze data coming from single-case or small-n experimental designs, even if we do randomly allocate treatments to observation occasions. This is because parametric tests assume (apart from other assumptions) that the observations are independent, which is obviously not suited to the situation when we collect a series of measurements on a single case. Fortunately, there is a kind of tests, namely randomization tests, that fit great to the scenario. First of all, they do not require that the observations are independent. Secondly, they do not rely on the unrealistic assumption of random sampling from a population which is often not true in large-n experiments. Concluding, single-case/small-n experimental design combined with randomization tests provide us with a convenient design and analysis framework for agile experimentation in an industrial setting. Such research methods were used until now mainly in social psychology, medicine, education, rehabilitation, and social work [11]. Although they not entirely new in software engineering [10, 25].

We have access to a small project with two professional developers of which one will be working using TDD. Specific characteristics (constraints) related to our industrial software project environment are presented in Table 1.

Table 1 Software project under investigation

No. of programmers	2
No. of programmers in experiment	1
No. of testers	1
Project time per month	160 hours
Project duration	12 months
No. of modules	37
No. of TDD modules	approx. 4–8
No. of TDD classes	approx. 40–80

The software project characteristics impose some constraints on the design of the experiment. Studying experimental designs discussed by Dugard et

al. [8] we found a *single-case randomized blocks design with two conditions* to be the most appropriate because:

1. We have only one participant.
2. We have two conditions to compare: TDD (A) and CTDD (B).
3. We can arrange the two conditions in blocks.
4. It is possible to assign conditions to observation occasions in blocks at random - how we do that is described in Section 4.

We will have one participant (developer), and we expect minimum 40 classes (in 4 modules) to be relevant for our research. Having approximately 40 classes to observe development on and two treatments (TDD and CTDD) we would have 20 blocks[1] giving us 2^{20}, i.e., a little over a million (1048576 to be exact), possible arrangements, which should give us good statistical power to detect difference between TDD and CTDD. We have used the Excel macro by Dugard et al. [8] to confirm our calculations.

4 Execution of experiment

The experiment was conducted on an industry grade software project. The software that was being created is a construction engineering software for analysis and design of concrete constructions. Under investigation were all the classes from one module used specifically to calculate the units of measurement in the software. The customer from the United States for which the software was build agreed to perform experimentation on the project, but he issued one condition: the impact in terms of time and cost on the project should be minimal. It was agreed that the software engineers conducting the project would spend the minimal amount of time on the tasks not increasing the immediate ROI for the customer. The tools developed for this research proved to be sufficient for this task. The developers used the NActivitiSensor and AutoTest.NET4CTDD – the use of those tools does require no or minimal developer attention. The only manual thing required from developers to do was to use another tool for randomization. As mentioned in Section 3.3 we decided to use *single-case randomized blocks design with two conditions*. To introduce randomization, we implemented a small tool. This tool was used by the developers to randomly choose weather TDD or CTDD should be used for a given source code part. The tool is presented in Figure 1. It randomly chooses weather the next class that a developer adds to the project should or should not to be decorated with a comment //AUTOTEST_IGNORE. If the comment is present the usage of AutoTest.NET4CTDD is disabled for this class, and the developer needs to execute the test manually. Classes without this comment are processed continuously tested. The act of "generating" the

[1] There are always two possible arrangements for every block: first A then B or first B then A.

next random phase A (TDD) or B (CTDD) and possibly decoration the class with a comment is the only thing the developers needed to do to allow us to perform the experiment.

The data was gathered in the relational databases of NActivitiSensor and AutoTest.NET4CTDD. Post-processing of the data was done using a simple C# script that calculated the time from of the raw timestamped event data saved in the database. It calculated red-to-green (RTG) duration. RTG is the time in minutes that elapses either from the moment the project is not building to the moment the project is building properly, and all tests are passing or from the moment a test or tests are not passing to the moment when the project is building properly, and all tests are passing.

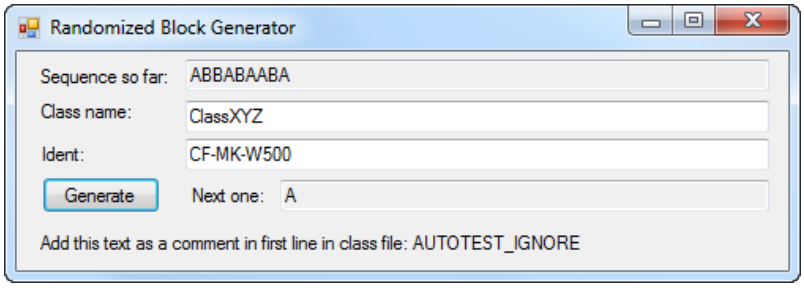

Fig. 1 Random block generator tool

5 Analysis

For the statistical analysis of the data, we used the R environment with the SSDforR package. The former is a language and software environment for statistical computing [19], while the latter is the R package for analyzing single-subject data [2].

5.1 Descriptive statistics

A typical way to begin comparing the baseline (TDD) and the intervention (CTDD) phases is by calculating descriptive statistics including measures of central tendency (e.g., mean, median, trimmed mean), as well as variation in both phases. It is often recommended to look for outliers and observations beyond two SDs are often checked, considered outliers and excluded from further analysis. We have followed this recommendation and found that the RTG times including the midnight (when the project with some failing test(s)

was left until the next working day) were removed. Investigating further, we found that some developers want to know where to start in the next working day and that is why they leave a failing test.

Table 2 shows the descriptive statistics of the data collected during our empirical study. The RTG times are further visualized in Figure 2, as well as using boxplot in Figure 2 comparing the RTG time of the developer using TDD and CTDD. Visual examination of the boxplot indicates a slight drop of the RTG in the CTDD phase of the experiment.

Table 2 Descriptive statistics

Measurement	**A (TDD)**	**B (CTDD)**
number of observations	85	55
median (Md)	1.768	3.018
mean (M)	12.372	8.388
10% trimmed mean (tM)	7.713	5.67
standard deviation (SD)	20.724	12.459
minimum	0.275	0.295
maximum	116.746	60.286
IQR	13.439	9.804
0% quantile	0.275	0.295
25% quantile	0.584	0.758
75% quantile	14.023	10.562
100% quantile	116.746	60.286

The mean is a measure of central tendency helpful in describing the typical observation. Smaller RTG time mean in the intervention phase (CTDD) than in the baseline phase (TDD) suggests the positive impact of the CTDD practice ($M_{TDD} = 12.372$ vs. $M_{CTDD} = 8.388$). Unfortunately, the mean can be strongly influenced by outliers, especially when the number of observations is small.

The median, i.e., the value for which a half of the observations fall above and half below, is the only measure of central tendency that shows the advantage of TDD, as suggests that the typical value of RTG time in TDD is smaller than in CTDD ($Md_{TDD} = 1.768$ vs. $Md_{CTDD} = 3.018$).

However, the recommended measure of central tendency in empirical software engineering are trimmed means. They which can support reliable tests of the differences between the central location of samples [12]. Trimming means removing a fixed proportion (e.g., 10 or 20%) of the smallest and largest values in the data set. The obvious advantage of the trimmed mean is that it can be used in the presence of outliers. 10% trimmed mean again suggests the positive impact of the CTDD practice versus the TDD baseline and reduction of the RTG time by over 2 minutes (10% $tM_{TDD} = 7.713$ vs. 10% $tM_{CTDD} = 5.67$).

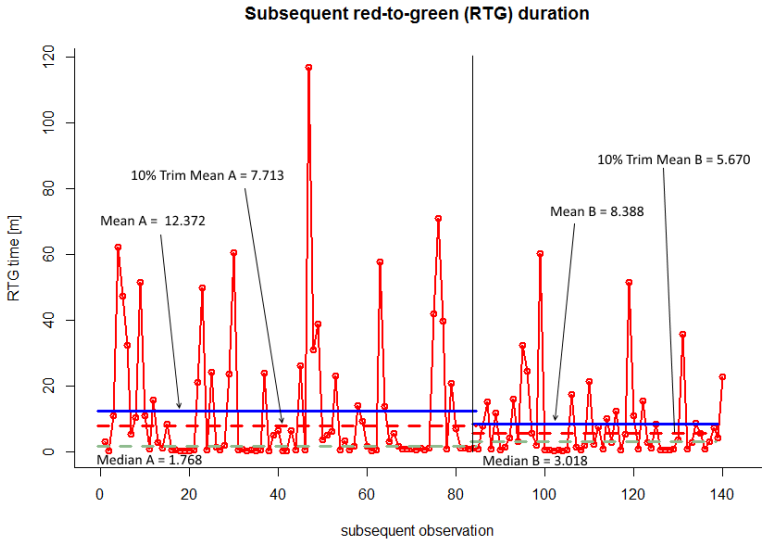

Fig. 2 Subsequent RTG durations in phases A (TDD) and B (CTDD) with the mean and median imposed

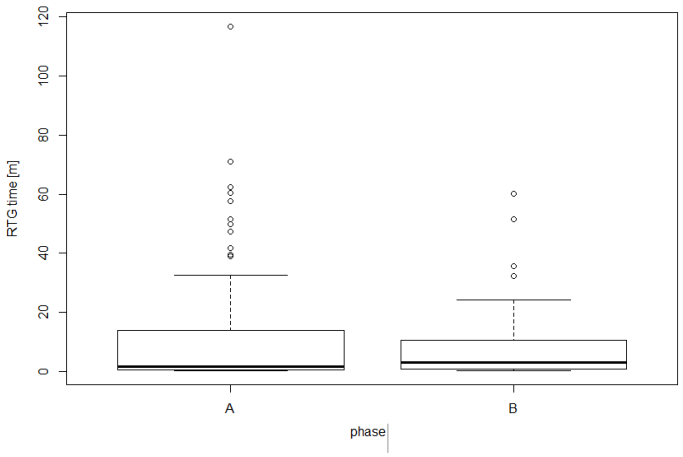

Fig. 3 Boxplot for RTG duration in A (TDD) and B (CTDD)

5.2 Measures of variations

As we reported a range of measures of central tendency, we need to describe the degree to which scores deviate from typical values. The simple measure

of variation is the difference between the highest and the lowest observed values. However, a bit more valuable form of this measure is the interquartile range (IQR), i.e., the difference between the third (or 75%) quantile and the first (or 25%) quantile. Figure 2 shows a great deal of variation. We have calculated the interquartile range (IQR) and got 0.584 for the 1st and 14.023 for the 3rd quartile in phase A, i.e., $IQR_{TDD} = 13.439$, and respectively 0.76 and 10.56 for the phase B, i.e., $IQR_{CTDD} = 9.804$. Hence, the variation in the middle 50% of the data is substantial.

The most common measure of variation frequently used together with the mean is the standard deviation (SD), which is the average distance between the scores and the mean. If the scores would be normally distributed then 95% of them would fall between 2 SDs below and above the mean, while typical scores fall between 1 SD below and above the mean. The standard deviation while using the CTDD practice is about half of the standard deviation of the baseline TDD practice ($SD_{TDD} = 20.724$ vs. $SD_{CTDD} = 12.459$) which is a desirable effect of CTDD. It is also easy to explain as the aim of the CTDD practice is to provide a fast feedback, to a developer, that tests do not pass.

5.3 Effect size

Effect size is a name given to indicators that measure the magnitude of a treatment effect (CTDD vs. TDD in our case). Effect size measures are very useful, as they provide an objective measure of the importance of the experimental effect, regardless of the statistical significance of the test statistic. Also, effect sizes are much less affected by sample size than statistical significance and, as a result, are better indicators of practical significance [12, 15]. In the context of the performed empirical study, effect size quantifies the amount of change between the TDD and CTDD phases.

The SSD for R package calculates different kinds of effect size including ES and $d - index$. ES is defined as the difference between the intervention (CTDD) and baseline (TDD) means divided by the standard deviation of the baseline. ES may show deterioration, lack of change or improvement due to the intervention.

$$ES = \frac{M_{intervention(CTDD)} - M_{baseline(TDD)}}{SD_{baseline(TDD)}} \tag{1}$$

In contrary to ES, $d - index$ uses a pooled standard deviation (i.e., a weighted average of standard deviations for two groups) which improves accuracy and is more appropriate when the variation between the phases differs, which is also the case in our study (see Table 2). However, it is worth mentioning that $d - index$ does not show the direction of the effect.

These effect size measures should only be used if there is no trend in the phases [13]. Each phase should be judged on the presence of the trend before

continuing with the further investigation. If observations in any of the phase exhibit a trend, either decreasing or increasing, then measures of central tendency have limited abilities to correctly assess the response. In such case, neither mean nor median should be used [7], and the same applies to effect sizes ES and $d - index$.

A trend may be defined and visualized by the slope of the best fitting line within the phase. The trends of both phases, A and B, were calculated using ordinary least squares (OLS) regression because it was found to be an accurate measure of the trend [7]. The multiple R-squared values were very close to 0 (0.005 for the phase A as well as B), while the p-values for the slopes in both phases were not statistically significant ($p > .05$), $p = 0.519$ for A and $p = 0.608$ for B. Hence, we may conclude that there were no (or were negligible) trends in the data.

As there was no trend in the data, we calculated effect size to measure the amount of change between A (TDD) and B (CTDD). The calculated effect sizes ($ES = -0.192$ and $d - index = 0.222$) can be interpreted as small, albeit non-zero, according to the guidelines provided by Bloom [7].

6 Conclusions and future work

The results of our first quasi-experiment are rather inconclusive. The mean RTG time dropped in our experiment from 12.4 to 8.4 minutes, whereas the median slightly increased from 1.8 to 3 minutes. Both effect sizes ($ES = -0.192$ and $d - index = 0.222$) indicate a small degree of change (reduction) in the red-to-green time needed to satisfy the tests. The calculated effect sizes seem to suggest that there may be a slight impact of the application of the CTDD practice regarding the RTG time, which aligns with to some extent with our hypothesis. However, we need to remember that effect sizes alone, do not prove that the intervention was the cause of the observed change and further investigation is needed to obtain more reliable conclusions.

In our experiment the same developer uses TDD and CTDD which eliminates the threat of experience variability among the subjects but it raises the question whether the results of the experiment would scale across programmers of various backgrounds. That is one of the questions we would like to address in the follow up study we are planning on this topic.

From the informal interviews with the developers taking part in the quasi-experiment, we gathered positive feedback about the ease of use of the tool but noted slight dissatisfaction with the AutoTest.NET4CTDD tool overall performance. That might have an impact on the overall performance of the CTDD practice itself. In the short term, we will address those concerns by vertically scaling developer workplaces. To resolve the issue in the longer term, substantial work on the tool itself will be necessary.

Nevertheless, we were able to put our rules for Agile Experimentation Manifesto in motion while researching TDD vs. CTDD. We noticed that we were able to incorporate state-of-the-art research techniques into a business-driven software project without affecting the project itself, which enable further experimentation. Using tools developed for this research we effectively minimized the time a developer need to spend on tasks not related to his core activities (e.g., related to research). Agile Experimentation applied in practice due to the course of the reported research appeared very promising to bridge the gap between research and industry in general, and developer, researcher and business owner in software engineering project in particular.

References

1. Ambler, S.W.: How agile are you? 2010 survey results (2010). URL http://www.ambysoft.com/surveys/howAgileAreYou2010.html
2. Auerbach, C., Zeitlin, W.: SSDforR: Functions to Analyze Single System Data (2017). R package version 1.4.15
3. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley (1994)
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston, MA, USA (1999)
5. Beck, K.: Test Driven Development: By Example. Addison-Wesley, Boston, MA, USA (2002)
6. Berłowski, J., Chruściel, P., Kasprzyk, M., Konaniec, I., Jureczko, M.: Highly Automated Agile Testing Process: An Industrial Case Study. e-Informatica Software Engineering Journal **10**(1), 69–87 (2016). DOI 10.5277/e-Inf160104. URL http://www.e-informatyka.pl/attach/e-Informatica_-_Volume_10/eInformatica2016Art4.pdf
7. Bloom, M., Fischer, J., Orme, J.: Evaluating Practice: Guidelines for the Accountable Professional. Pearson/Allyn and Bacon (2008)
8. Dugard, P., File, P., Todman, J.: Single-case and Small-n Experimental Designs: A Practical Guide to Randomization Tests, 2nd edn. Routledge (2012)
9. Geracie, G.: The study of product team performance (2014). URL http://www.actuationconsulting.com/wp-content/uploads/studyofproductteamperformance_2014.pdf
10. Harrison, W.: N = 1: An alternative for software engineering research? (1997). URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.2131&rep=rep1&type=pdf. Based upon an editorial of the same title in Volume 2, Number 1 of Empirical Software Engineering (1997)
11. Kazdin, A.E.: Single-case Research Designs: Methods for Clinical and Applied Settings. Oxford University Press (2011)
12. Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., Pohthong, A.: Robust Statistical Methods for Empirical Software Engineering. Empirical Software Engineering **22**(2), 579–630 (2017). DOI 10.1007/s10664-016-9437-5. URL http://link.springer.com/content/pdf/10.1007%2Fs10664-016-9437-5.pdf
13. Kromrey, J.D., Foster-Johnson, L.: Determining the efficacy of intervention: The use of effect sizes for data analysis in single-subject research. The Journal of Experimental Education **65**(1), 73–93 (1996). DOI 10.1080/00220973.1996.9943464

14. Kurapati, N., Manyam, V., Petersen, K.: Agile software development practice adoption survey. In: C. Wohlin (ed.) Agile Processes in Software Engineering and Extreme Programming, *Lecture Notes in Business Information Processing*, vol. 111, pp. 16–30. Springer Berlin Heidelberg (2012)
15. Madeyski, L.: Test-Driven Development: An Empirical Evaluation of Agile Practice. Springer, (Heidelberg, London, New York) (2010). DOI 10.1007/978-3-642-04288-1
16. Madeyski, L., Kawalerowicz, M.: Continuous Test-Driven Development - A Novel Agile Software Development Practice and Supporting Tool. In: L. Maciaszek, J. Filipe (eds.) ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 260–267 (2013). DOI 10.5220/0004587202600267. URL http://madeyski.e-informatyka.pl/download/Madeyski13ENASE.pdf
17. Madeyski, L., Kawalerowicz, M.: Software Engineering Needs Agile Experimentation: A New Practice and Supporting Tool. In: Software Engineering: Challenges and Solutions, *Advances in Intelligent Systems and Computing*, vol. 504, pp. 149–162. Springer (2017). DOI 10.1007/978-3-319-43606-7_11. URL http://madeyski.e-informatyka.pl/download/MadeyskiKawalerowicz17.pdf
18. Majchrzak, M., Łukasz Stilger: Experience Report: Introducing Kanban Into Automotive Software Project. e-Informatica Software Engineering Journal **11**(1), 41–59 (2017). DOI 10.5277/e-Inf170102. URL http://www.e-informatyka.pl/attach/e-Informatica_-_Volume_11/eInformatica2017Art2.pdf
19. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2016)
20. Saff, D., Ernst, M.D.: Reducing wasted development time via continuous testing. In: Fourteenth International Symposium on Software Reliability Engineering, pp. 281–292. Denver, CO (2003)
21. Saff, D., Ernst, M.D.: An experimental evaluation of continuous testing during development. In: ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis, pp. 76–85. Boston, MA, USA (2004)
22. Sochova, Z.: Agile adoption survey 2009 (2009). URL http://soch.cz/AgileSurvey.pdf
23. West, D., Grant, T.: Agile development: Mainstream adoption has changed agility (2010). URL http://programmedevelopment.com/public/uploads/files/forrester_agile_development_mainstream_adoption_has_changed_agility.pdf
24. West, D., Hammond, J.S.: The forrester wave: Agile development management tools, q2 2010 (2010). URL https://www.forrester.com/The+Forrester+Wave+Agile+Development+Management+Tools+Q2+2010/fulltext/-/E-RES48153
25. Zendler, A., Horn, E., Schwärtzel, H., Plödereder, E.: Demonstrating the usage of single-case designs in experimental software engineering. Information & Software Technology **43**(12), 681–691 (2001). DOI 10.1016/S0950-5849(01)00177-X