Effectiveness of Test-Driven Development and Continuous Integration

A Case Study

Chintan Amrit
University of Twente

Yoni Meijberg
Topicus

In a case study where a Dutch small-to-medium enterprise (SME) implemented test-driven development and continuous integration, researchers observed that the SME discovered a higher number of defects compared to a baseline case study, and that there was an increase in the focus on quality and test applications.

Although many older development methodologies include a separate lengthy testing phase at the end of development, test-driven development (TDD) comes with a new paradigm: always test before coding. Continuous integration (CI), on the other hand, involves running integration builds and automated tests on the code committed by the developers. CI combines development and testing by enabling unit and functional tests while profiling the application code.[1] CI was designed to improve the number of testing cycles and the resulting application quality while decreasing the amount of time it takes to find problems and reducing the cost to fix them.[1]

Both TDD and CI, however, cost the development company in terms of time and money.[2,3] Hence, well-grounded empirical evidence is required to determine the applicability of these practices in actual industrial settings. In response to this, some empirical studies were conducted on the effectiveness of TDD implementations in academic as well as industrial settings (see the sidebar for a literature review). The outcomes of these empirical studies seem to indicate an increase in code quality along with an inconclusive effect on developer productivity (see the table in the sidebar). The extra effort required when writing tests in advance was given as a reason for the decrease in developer productivity. On the other hand, one can argue that TDD could improve internal and external code quality, leading to lower defect generation and faster fixes, thereby improving productivity. Yahya Rafique and Vojislav B. Misic[4] mention in their extensive literature review that some experiments with a more detailed design obtained better results on TDD implementation. This was verified in a recent project where TDD was implemented successfully.[5] However, this claim has not been adequately or quantitatively tested in an industrial setting, nor has it been discussed widely.[5] Furthermore, most papers do not provide a detailed description of the TDD implementation.

In this article, we describe a case study of both TDD and CI in a Dutch small-to-medium enterprise (SME). The contributions of this article are multi-fold. While previous research dealt with few metrics for accessing the cost and quality, we use multiple metrics. Along with an evaluation of the technical quality (which was also done in previous studies), we propose a quantitative evaluation of the impact of TDD and CI implementation and provide a detailed account of the case setting. To aid future research, we list a set of adherence metrics to measure the extent to which TDD/CI principles have been applied. Our article contributes to the growing body of literature on TDD and CI implementation, as well as its evaluation in an industrial setting. Similar to other studies, we cannot isolate the effect of the sole application of TDD from the effect of CI.[5]

THE CASE CONTEXT

The data for this research originates from a two-case comparison of software implementation at a Dutch SME. The two cases were consecutive software development projects at the software company. The first case followed a development process without any methodology alterations nor any involvement by the researchers, while the second project followed a process that was refined by TDD and CI principles. The second author of this article was present as both an information analyst and chief methodology implementer of the second project. We use the first project as a reference case to compare the potential differences in the outcome of the two projects.

Both software development projects dealt with healthcare claim handling. Healthcare claims are mostly digital transactions (involving the transfer of messages, digital invoices, or money) between healthcare providers and insurers. As soon as an individual receives some form of healthcare from a provider (for instance, surgery), the provider usually makes several transactions that need to be appraised by the insurer. On average, each individual in the Netherlands is involved in about 10 transactions per year. This implies that managing these digital transactions is a high-volume industry. As these transactions are highly standardized by the Dutch government, they are well suited for automation. The Dutch SME in this case study makes software applications that provide such automation. The software project described in this article is based on one such software application. For the sake of clarity, we denote the software development project in which TDD and CI testing was implemented as case study 2 (CS2), and the previous version where it was not implemented as case study 1 (CS1).

CS1 served mainly as an entry portal for incoming healthcare transactions from healthcare providers at an insurance intermediary. Its main goal was to structure, check, and help in the appraisal of incoming transactions. After this process, the transaction was entered into the financial database.

CS2's software application was installed at an invoicing intermediary, which takes care of everything around healthcare transactions for a set of healthcare providers. This is done so that the healthcare providers can concentrate on providing proper care instead of getting bogged down with the many financial transactions that are required. However, in terms of functionality, CS1 and CS2 were very similar.

The applications of the two cases were installed at different points of the healthcare chain, and thus satisfied different stakeholders. They were similar in terms of functionality, as they interacted (indirectly) within the same industry while applying common transaction standards. The CS2 implementation was, in particular, based on the CS1 architecture and most of the CS2 modules were refactored CS1 modules. As 92.5 percent of the modules overlapped between the two architectures, we can safely assume that the number and complexity of the features were similar. Note that this overlap is more than the 75 percent overlap reported by Roberto Latorre.[5]

Table 1 lists the similar project characteristics found in the two cases in terms of context factors and software product measure factors.[6–8] We notice that both kinds of factors, especially the contextual factors, are comparable and rather similar between the two case studies.

Table 1. Description of the context and product measures of case studies 1 and 2.

Characteristics	**Case study 1 (CS1)**	**Case study 2 (CS2)**
Context factors		
Application	Transactional system for technical verification and appraisal, as well as administrative handling and payment of healthcare declarations	Transactional system for technical verification, routing, and appraisal of healthcare declarations
Customer	Authorized insurance intermediary	Invoicing intermediary
Duration (time)	8 months	10 months
Average monthly effort (man days)	78	73
Total effort (man days)	732	731
Team size	<10	<10
Team people overlap	N/A	>4
Experience level (<5 years, 6-10 years, >10 years)	Most members < 5 years' experience	Most members < 5 years' experience
Project manager's expertise	> 5 years' experience	> 5 years' experience
Applied technology	C# ASP.NET MSSQL	C# ASP.NET MSSQL NHibernate
Product measures		
Source LoC	28,049	21,340
Maintainability index average	79.20	85.32
Cyclomatic complexity average	280.53	297.71
Depth of inheritance average	02.62	04.70
Class coupling average	57.59	77.65

MEASUREMENT METRICS

To measure the effectiveness of TDD and CI, we considered three perspectives: defect reduction, defect lead and throughput, and development productivity.

Defect reduction is whether TDD and CI helped in reducing the number of defects; for example, if CS2 has fewer defects compared to CS1. This includes the pre-release defect level (the number of defects detected in the pre-release software per KLOC[9]) and post-release defect level (the number of defects occurring in the post-release software per KLOC).[9,10]

Defect lead and throughput is whether TDD and CI helped in reducing the time to find and fix the defects. This includes defect resolution duration—the duration between the discovery and closure of defects, given by

$$\frac{\sum_D d_{\text{closedate}} - d_{\text{reportdate}}}{|D|}$$

where |D| is the number of defects. This also includes defect pre/post-solvability—the proportion of defects uncovered prior to and after the release given by $\frac{|D^{\text{Pre}}|}{|D|}$ per pre-solvability release and $\frac{|D^{\text{Post}}|}{|D|}$ per post-solvability release.

Development productivity is the development productivity and rework rate in development per release, given by $\frac{\text{KLOC}}{\text{Effort}}$, where the KLOC is the amount of KLOC added per release.

RESULTS

We first analyzed our data using descriptive statistics and then considered inferential statistics. Thirty-nine versions of the CS1 software and 24 versions of the CS2 software were released in the period from the start of the two projects up to when the data for this study was collected. Both projects used the Mantis bug tracker, and developers and customers submitted bug reports. We analyzed the CS1 and CS2 Mantis bug trackers and removed duplicates and issues that were not really bugs. CS1 had 414 viable bug reports for analysis, while CS2 had 474 viable bug reports. The bugs were prioritized in both CS1 and CS2 in terms of defect severity. But this was not done explicitly in both projects. The developers and project leaders knew which bugs to resolve first, based on which part of the code they occurred in, and this knowledge was tacit (not made explicit in the bug tracker). In the case of CS1, the prioritization and resolution were done manually. In both projects, the bugs that caused build failures were fixed first and the rest were fixed in the order of prioritization. In the case of CS2, CI helped in this regard, as the CI-related best practice of "executing all tests with every build and making a single failed test fail the build" was followed.[1]

When we asked the project managers of both projects to compare the list of bugs (in terms of the number of high-priority bugs and the severity of the defect), they agreed that the bugs from both the projects were comparable.

Descriptive Statistics

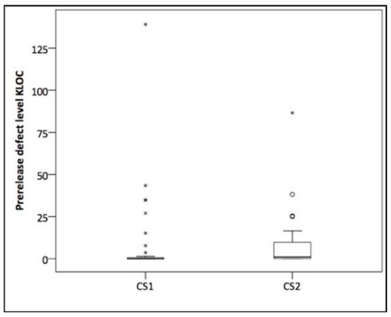
The box plot of the distribution of pre- and post-defect resolution data is shown in Figures 1a and 1b; the numbering of the figures that follow is based on this categorization. In Figure 1a we notice that the defect level of CS2 is consistently higher (before the release) than the defect level of CS1. This could be the result of implementing the test-first approach in CS2, and not necessarily because the CS1 software had fewer bugs.

This seems to be verified by Figure 1b, where we notice that CS1 has more bugs than CS2 after the release. Figure 1b also demonstrates that the medians of the post-release number of bugs per KLOC are nearly the same for CS1 and CS2. The smaller inter-quartile range of CS2 shows that CS2 had a more consistent number of bugs per release, and fewer releases with a large number of bugs (as was the case with CS1).

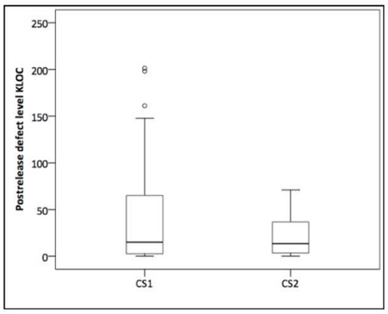
In Figures 1c and 1d we can see the results of the defect lead and throughput category metrics. Figure 1c indicates that the defect resolution duration (the time in days between discovery and resolution) is mostly higher for CS1 than CS2 (although the medians are nearly the same). Furthermore, CS1 has a much longer tail, indicating that a few bugs required more fixing time compared to CS2.

In Figure 1d we see the defect pre-/post-solvability (the proportion of defects uncovered before release). CS2 again outperforms CS1 because a consistently higher percentage of bugs was found before a major release. The developers in CS2 also accurately flagged a bug as solved as soon as they had fixed a defect. This was done to prevent colleagues from fixing the same defect again, and to show the progress to the customer. Hence, this data can be considered reliable.

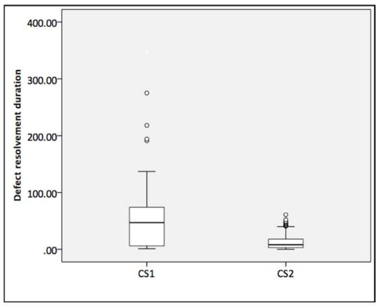
In Figure 1e we see the comparison of development productivity between the two projects. The inter-quartile range of CS1 is almost double that of CS2, while the medians are nearly the same. This implies that while the development productivity was nearly the same, there were instances when the number of added KLOCs was very large for the same number of man days. When we approached the team members for an explanation, they thought it could be due to the introduction of large pre-coded components (containing several hundred code lines apiece).



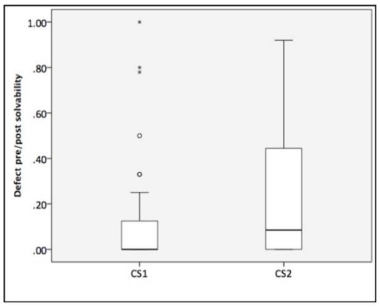
(a)



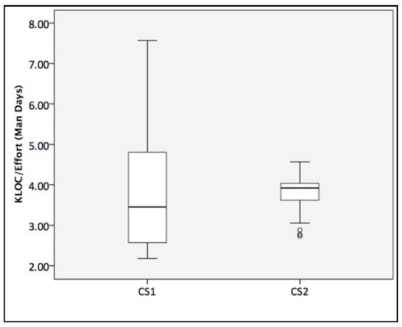
(b)



(c)



(d)



(e)

Figure 1. The descriptive statistics for the metrics described in this article. (a) Pre-release defect level. (b) Post-release defect level. (c) Defect resolution duration in days. (d) Defect pre-/post-solvability. (e) Development productivity.

Inferential Statistics

We first tried to ascertain the difference in means between CS1 and CS2, with respect to the metrics listed in this article. The common statistical method used to test the difference in the distributions of two samples is the Student's t-test. However, the prerequisites for using the t-test are that the two distributions must have the same variance and both populations should follow the normal distribution. We could not use the Student's t-test as Levene's test showed that some of the metrics from CS1 and CS2 did not have equal variances, while the Shapiro–Wilk test for normality demonstrated that normality was violated by almost all the metrics. The reason behind this was the variation in the denominator in all the metrics, namely KLOC. When comparing the KLOC between CS1 and CS2, Levene's test showed a significance of 0.38 (indicating that the population variances are equal) and Shapiro–Wilk showed a significance of 0.00 for both projects (indicating that they are not normally distributed). As the distribution was not normal for both samples, we considered the Mann–Whitney–Wilcoxon (MWW) test, which is non-parametric.

Table 2. Mann–Whitney–Wilcoxon test results for the metrics in this article.

Metric	**Mann–Whitney U**	**Wilcoxon W**	**Z**	**Asymptotic significance (2-tailed)**
Pre-release defect level	258	1038	-0.62	0.009
Post-release defect level	388	619	-0.33	0.739
Defect resolution duration	55555.5	154790.5	-2.57	0.010
Defect pre-/post-solvability	312	942	-1.84	0.067
Development productivity	413	713	-0.78	0.436

Table 2 groups the results based on the relevant metric category. The p-value of the asymptotic significance of the pre-release defect level and defect resolution duration is less than 0.05. Hence, these two metrics demonstrate that CS2 outperforms CS1. Regarding the defect pre-/post-solvability metric, the p-value is just over 0.05, whereas for the post-release defect level and development productivity metrics, we find no significant difference between CS1 and CS2. Note that the MWW test can only determine whether the two distributions are indeed significantly different, and not which distribution has a higher median (or other measures of dispersion). Therefore, we combined the data from Table 2 with Figures 1a and 1c. We see that the number of pre-release defects found in CS2 is significantly greater than in CS1 (Figure 1a). We also see that the defect resolution duration of CS1 is significantly larger than that of CS2. Though the U value of the defect resolution duration appears to be rather large, it is more or less what we can expect given the large number of samples: 414 for CS1 and 474 for CS2 (an expected estimator of U is multiplying half of the sample sizes of the distributions). In the case of defect pre-/post-solvability, we see that the MWW test p-value just exceeds 0.05, though Figure 1d shows that the number of defects uncovered before release is much larger in CS2 than CS1. Hence, CS2 outperforms CS1 in this metric. With respect to the post-release defect level, the MWW test shows that the distribution of the number of bugs per KLOC of the two projects is not significantly different. We see from Figure 1b that the medians of the two distributions are indeed quite close, though the CS1 upper quartile is much larger than that of CS2, so there were more defects occurring in CS1 post-release. This is also the case with the development productivity metric; in Figure 1e, we see that the medians are similar but the lower and upper quartiles are largely different, implying that the development effort put into a few releases in CS1 was much larger than in CS2.

CONCLUSION

The team members at the Dutch SME did perceive an increase in the focus on quality and in the application of tests, while considering customer acceptance. The company now has an infrastructure in place to further evaluate other software process improvement (SPI) initiatives.

Though the inferential statistics are not conclusively in favor of CS2, the descriptive statistics point to an overall improvement in not only finding more defects (defect reduction), but also in shortening the time required to fix the defects (defect lead and throughput). One of the limitations of this research could be the use of KLOC in most of the metrics (in the denominator), as well as the use of KLOC for measuring development productivity (see Figure 1e).[11] Emad Shihab and his colleagues mention that metrics like cyclomatic complexity—and a combination of software metrics—outperform LOC in estimating effort, whereas using solely LOC underestimates effort.[11] However, if the same metric is used to compare different projects, the underestimation of effort could balance out—as it is underestimated for both cases, and we are only interested in the relative comparison of development effort and not the exact development effort of each project. Yet, applying KLOC limited our use of different statistical techniques, as the variance it introduced made the distributions non-normal.

This research adds to the existing set of metrics, and we think our new metrics give a richer and more detailed description of the effects of agile implementations in general.

The contributions of this research are multiple. We have added to the small but hopefully growing body of empirical literature on agile implementation in an industrial setting.[12–14] We have endeavored to provide a rich description of the project setting (contextual factors and product measures in Table 1) that we think is useful to recognize potential validation errors. Finally, we have added to the existing set of metrics, and we think our new metrics give a richer and more detailed description of the effects of agile implementations in general.

SIDEBAR: LITERATURE OVERVIEW

We performed a meta-analysis of the literature reviews on TDD published in the past five years. In the table below, internal quality relates to the quality of the software design (measured by OO metrics, code density, cyclomatic complexity, etc.); external quality refers to the number of pre/post-release defects per given code size; and productivity refers to developer productivity (measured using development time, total LOC divided by total effort, hours per feature/development effort per LOC, etc.). In Table 3, N/A refers to the fact that the particular construct was not analyzed.

Table 3. Comparison of the findings of literature reviews published in the past five years.

	C. Desai et al.[15]	**B. Turhan et al.[16]**	**A. Causevic et al.[17]**	**S. Kollanus[18]**	**Y. Rafique and V.B. Misic[4]**
Internal quality	Little evidence	Little evidence	Moderate evidence	Inconclusive	N/A

External quality	Moderate evidence	Moderate evidence	N/A	Weak evidence	Little evidence
Productivity	Little evidence	Inconclusive	Evidence for decreased performance	Evidence for decreased performance	Inconclusive

Table 3 broadly suggests there is little to moderate evidence that implementation of TDD in an academic or industrial setting is accompanied by an increase in code quality. However, the evidence for an improvement in productivity is largely inconclusive and the literature indicates, to some extent, that TDD implementation is accompanied by decreased productivity.

REFERENCES

1. P.M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley Professional, 2007.
2. P.B. Crosby, *Quality Without Tears: The Art of Hassle-Free Management*, McGraw-Hill Education, 1995.
3. R. Dion, "Elements of a Process-Improvement Program (Software Quality)," *IEEE Software*, vol. 9, no. 4, 1992, pp. 83–85.
4. Y. Rafique and V.B. Misic, "The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis," *IEEE Trans. Software Eng.*, vol. 39, no. 6, 2013, pp. 835–856.
5. R. Latorre, "A Successful Application of a Test-Driven Development Strategy in the Industrial Environment," *Empirical Software Eng.*, vol. 19, no. 3, 2014, pp. 753–773.
6. T. Bhat and N. Nagappan, "Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies," *Proc. 2006 ACM/IEEE Int'l Symp. Empirical Software Eng.* (ISESE), 2006, pp. 356–363.
7. A.D. Carleton et al., *Software Measurement for DoD Systems: Recommendations for Initial Core Measures*, report, Software Engineering Institute, 1992.
8. N. Nagappan et al., "Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams," *Empirical Software Eng.*, vol. 13, no. 3, 2008, pp. 289–302.
9. *Quality Standards Defect Measurement Manual*, technical manual, United Kingdom Software Metrics Association, 2000; pdfs.semanticscholar.org/2b5b/59fda8fdb51b000008ddd70942d01cd25776.pdf.
10. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Int'l Thomson Computer Press, 1997.
11. E. Shihab et al., "Is Lines of Code a Good Measure of Effort in Effort-Aware Models?," *Information and Software Technology*, vol. 55, no. 11, 2013, pp. 1981–1993.
12. C. Amrit and J. van Hillegersberg, "Detecting Coordination Problems in Collaborative Software Development Environments," *Information System Management*, vol. 25, no. 1, 2008, pp. 57–70.
13. M. Daneva et al., "Agile Requirements Prioritization in Large-Scale Outsourced System Projects: An Empirical Study," *J. Systems and Software*, vol. 86, no. 5, 2013, pp. 1333–1353.
14. C. Amrit, J. van Hillegersberg, and K. Kumar, "Identifying Coordination Problems in Software Development: Finding Mismatches between Software and Project Team Structures," pending publication, 2012; arxiv.org/abs/1201.4142.
15. C. Desai, D. Janzen, and K. Savage, "A Survey of Evidence for Test-Driven Development in Academia," *ACM SIGCSE Bull.*, newsletter, vol. 40, no. 2, 2008, pp. 97–101.
16. B. Turhan et al., "How Effective Is Test-Driven Development?," *Making Software: What Really Works, and Why We Believe It*, O'Reilly Media, 2010, pp. 207–217.

17. A. Causevic, D. Sundmark, and S. Punnekkat, “Factors Limiting Industrial Adoption of Test-Driven Development: A Systematic Review,” *Proc. IEEE 4th Int’l Conf. Software Testing, Verification and Validation* (ICST), 2011, pp. 337–346.
18. S. Kollanus, “Critical Issues on Test-Driven Development,” *Proc. 12th Int’l Conf. Product-Focused Software Process Improvement* (PROFES), 2011, pp. 322–336.

ABOUT THE AUTHORS

Chintan Amrit is an assistant professor in the Department of Industrial Engineering and Business Information Systems (IEBIS) at the University of Twente. His research interests include software development, business intelligence using machine learning, data analysis methods, supply-chain logistics, and mining software repositories. Amrit received a PhD in information systems from the University of Twente. He is an associate editor of *IEEE Access*, a coordinating editor of *Information Systems Frontiers,* and a regular track chair at the European Conference on Information Systems (ECIS). Contact him at c.amrit@utwente.nl.

Yoni Meijberg is a managerial change agent at Topicus. His research interests include software project management and business research methods. Meijberg received a master’s degree in organizational change from RSM Erasmus University and another in industrial engineering from the University of Twente. Contact him at yoni.meijberg@topicus.nl.