Software Qual J (2011) 19:643–661
DOI 10.1007/s11219-011-9130-2

The effectiveness of test-driven development: an industrial case study

Tomaž Dogša · David Batič

Published online: 26 February 2011

Abstract Test-driven development (TDD) is a software development practice, where test cases are incrementally written before implementing the production code. This paper presents the results of a multi-case study investigating the effectiveness of TDD within an industrial environment. Three comparable medium-sized projects were observed during their development cycle. Two projects were driven without TDD practice, while the third one introduced TDD into the development process. The effectiveness of TDD was expressed in terms of external code quality, productivity, and maintainability. Our results indicate that the TDD developers produced higher quality code that is easier to maintain, although we did observe a reduction in productivity.

Keywords Test-driven development · Testing and debugging · Testing strategies · Productivity · Maintainability · Software quality/SQA · Software engineering process

1 Introduction

In test-driven development (TDD) practice (Beck 2003), test cases (preferably automated) are incrementally written before implementing the production code. TDD is also known as Test-First Programming (TFP), Test-First Design (TFD), and Test-Driven Design (TDD) and has been used sporadically for more than a decade. There are several views on the primary goal of TDD in the literature. One view is that the goal of TDD is specification and not validation (Martin et. al. 2002), in other words, it is one way of thinking through the design before writing the functional code. Another view is that TDD is simply a programming technique, the goal of which is to write clean code that works (Beck 2003). Yet

T. Dogša
Faculty of Electrical Engineering and Computer Science, Centre for Verification and Validation of Systems, University of Maribor, Smetanova ul. 17, SI-2000, Maribor, Slovenia
e-mail: tdogsa@uni-mb.si

D. Batič (✉)
Agileon d.o.o., Cesta XIV. Divizije 51, SI-2000 Maribor, Slovenia
e-mail: batic@agileon.eu

another view is that TDD's main goal is not testing software but aiding the programmer and customer during the development process, in order to establish unambiguous requirements (Newkirk and Vorontsov 2004).

The promoters claim that TDD is emerging as one of the most successful development productivity-enhancing techniques recently discovered. Moreover, its advocates claim that it leads to fewer defects, less debugging, more confidence, and better design (Jeffries 1999; Beck 2001; Beck 2003; Newkirk and Vorontsov 2004; Jeffries and Melnik 2007 etc.). Unit tests provide a safety net of regression tests and validation tests for TFD and constant "refactoring" (Fowler et al. 1999), enabling efficient refactoring and integration.[1] However, despite its intriguing properties, skeptics argue that TDD is hard to learn and is counterproductive (Crispin 2006). Besides, software practitioners are often concerned with the lack of upfront design in TDD. Its opponents claim that unit tests leave one critical area uncovered: design correctness; they catch certain types of code-level bugs, but they do not catch the "wrongness" of a design (Stephens and Rosenberg 2004).

The results of case studies, surveys, and controlled experiments are the key input for industrial software process improvement. Most relevant results can be gained only by performing the experiment in an actual industrial environment. Unfortunately, this only rarely takes place, since experiments in industrial environments are either expensive or risky: owing to the high levels of pressure and risk, there is usually neither time nor money for experimentation within the industry.

This paper presents a case study on introducing TDD practice into the incremental development process.[2] The goal of introducing the popular TDD was to improve the development process in a small telecommunication software company.

The remainder of this paper is organized as follows: Section 2 reviews the background and the related work. Section 3 explains the methodology and procedures used in this study. In Section 4, the results are presented and discussed. Section 5 addresses threats to validity. Finally, Section 6 summarizes our findings.

2 Background and related work

2.1 Previous studies and the rationale of this study

Over the past decade, researchers (Maurer and Martel 2002; George and Williams 2003; Mueller and Hagner 2002; Martin et. al. 2002; Maximilien and Williams 2003; Edwards 2003; Erdogmus et. al. 2005; Madeyski 2005; Madeyski and Szała 2007; Bhat and Nagappan 2006; Sanchez et. al. 2007; Siniaalto and Abrahamsson 2007; Jeffries and Melnik 2007) have started conducting studies within academia and industry, on the effectiveness of TDD practice. Only the case studies by Maximilien and Williams (2003), Bhat and Nagappan (2006) and one controlled experiment, see George and Williams (2003), were conducted by professionals within an industrial environment. In general, all these studies reported different results. The majority of previous studies and experiments reported that TDD did not accelerate development, yet produced better code quality. However, a controlled experiment with academic participants (Erdogmus et. al. 2005), providing the subjects with the same experimental task as that investigated by George and Williams (2003), reported that

[1] http://www.extremeprogramming.org/rules/unittests.html

[2] In incremental development, the releases are defined by beginning with one small, functional subsystem and then adding functionality with each new release (Pflegeer and Atlee 2006).

"Test-First" appears to improve productivity but does not achieve better quality, on average. One of the more comprehensive recent reports (Jeffries and Melnik 2007) on the state of TDD research provides an overview of selected empirical studies with industrial and academic participants. The results are sometimes controversial, mostly in the domain of academic studies.

Table 1 provides a summary of selected previous studies on TDD conducted within an *industrial environment* only. The last row characterizes this study for comparison purposes.

We could find only three published industrial case studies on TDD (George and Williams 2003; Maximilien and Williams 2003; Bhat and Nagappan 2006); however, they reported different findings regarding productivity.

We decided to conduct our own industrial multi-case study, in order to further investigate the effectiveness of TDD (see Sect. 3). Although we gathered various data from the process activities and product characteristics (see Appendix 1), we limited this study to the productivity, maintainability, and code quality metrics only. This simplified the comparison of the results with previous work in this area.

2.2 Test-driven development

TDD practice evolves the design of a system starting from the unit tests of a component. The need for other components is initiated by the test cases written and the implementation of that component. There is no explicit design, nor explicit testing phase. TDD can simply be defined as a simple design and design improvement called "refactoring."

When starting a new task or feature, a list of tests needs to be brainstormed. TDD's "Red, Green, Refactor" mantra is a mnemonic for TDD's development process for implementing each test in the test list (Beck 2003), as follows:

1. Write the test code.
2. Compile the test code (It should fail because you have not yet implemented anything.).
3. Implement just enough to compile without errors.
4. Run the test and see if it fails.
5. Implement just enough to make the test pass.
6. Run the test and see if it passes.
7. Refactor for clarity and to eliminate duplication.
8. Repeat from the top.

A new functionality is not considered properly implemented unless these new unit test cases and every other unit test case ever written for the code base, passes.

3 Methodology and the research approach

3.1 Context of the case study

Over the past years, several Computer and Web Telephony projects have been conducted using the "Synchronize & Stabilize" (S&S) development process model (Jovanović and Dogša 2003). The idea behind the S&S model is simple: to continually synchronize what people are doing as individuals and as members of parallel teams, and to periodically stabilize the product in increments as the project proceeds, rather than once at the end of the project (Cusumano and Selby 1997). Microsoft has been using this approach since the late 1980s (Malik and Palencia 1999).

Table 1 Summary of empirical studies of TDD in industry

Authors	Study type	Incr. dev.	Quality metrics[a]	Quality	Productivity metrics	Productivity	Maintain.
Maximilien and Williams 2003	Case study	No	Defect rate	TDD is better	LOC/person-month	No differences found	N/A
George and Williams 2003	Controlled experiment	No	No. of black-box tests passed	TDD is better	Development time	TDD is worse	N/A
Sanchez, Williams, Maximilien 2007	Case study	No	Defect density	TDD is better	Perception of time spent on development	TDD is worse	N/A
Bhat and Nagappan 2006	Case studies	?	Defect density	TDD is better	Development time	TDD is worse	N/A
Damm and Lundberg[b]2006	Multi-case study	Yes[b]	Fault rate	TDD is better	N/A	N/A	N/A
This study	Case study	Yes	Failure density;Fault density; No. of black-box tests passed	TDD is better	New LOC/total effort	TDD is worse	TDD is better

Incr. dev. Incremental development, *Maintain.* Maintainability, *?* Unknown/unspecified, *N/A* Not applicable

[a] The terminology from each particular case study is used

[b] Some modifications of traditional TDD were applied

S&S development comprises the following phases:

- The product vision, being the foundation for rough functional specification, evolves till the end of the development's lifecycle.
- The development cycle passes milestones; each implicitly contains phases of analysis, design, coding, usability testing, component testing, daily builds, integration and, finally, stabilization. Reaching the "Alpha" milestone means that the product is stable but implements only 1/3 of the planned features. At the "Beta" milestone, 2/3 of the planned features are implemented, while other features are implemented by the last milestone.
- On reaching the last milestone, developers freeze the user interface, perform the last tests, and stabilize the version to be delivered to the customer. The functional specification is then completed.

The S&S development process was slightly modified in order to conduct this case study. The S&S model was simplified by means of a team structure because there was an internal customer, and so the technical coordinator also took over the roles of product and program manager. Ten team members were assigned to each feature team comprising programmer-tester pairs, and finally, there was one additional member dealing with the development, environment, and tooling. The modified development process model did not itself provide a set of rules about how a programmer should write code, but provided a method for permanently synchronizing a particular code segment with those segments implemented by other team members. Daily builds actually forced the developers to write low-level functional tests at the early development stages. Such an operative framework enabled easy TDD adoption. The modification enabled the seamless introduction of either the "test-first" strategy or the conventional development strategy. Hereafter, the conventional process variant will be referred to as the *Test-Last* process. The other process with TDD will be called the *Test-First* process. The S&S development model is depicted in Fig. 1. The Test-Last process differs from the Test-First process only in using a more conventional coding practice instead of TDD; so unit tests are written after the production code.

The regression testing framework JUnit[3] enabled unit test automation during all projects. In order to guarantee that all unit tests were run by all the team members, a toolkit was set up for an automated build. Every 15 minutes, code was automatically extracted from the repository and built; then all unit tests were executed. After each automated build/test run, team members were sent an e-mail with a list of failed tests and of any faults discovered. This automated build and test served as continuous integration and verification (Jeffries 1999).

The development cycle comprised a conventional system test (ST) phase when functional black-box tests based on an evolving functional specification were run by the project teams. No external ST group was assigned; so the ST was designed and performed by the team members. Regression functional testing was performed as soon as the development team had removed all known faults. The acceptance test phase (AT) was performed externally by the customer according to their own test specification, which had not been available to the teams during the development process. The customer accepted the product when 95% of acceptance tests had been passed and no failure had been found with a severity higher than minor—such a state was a precondition for the product entering the maintenance phase.

[3] JUnit—A regression testing framework written by Erich Gamma and Kent Beck, http://www.junit.org.

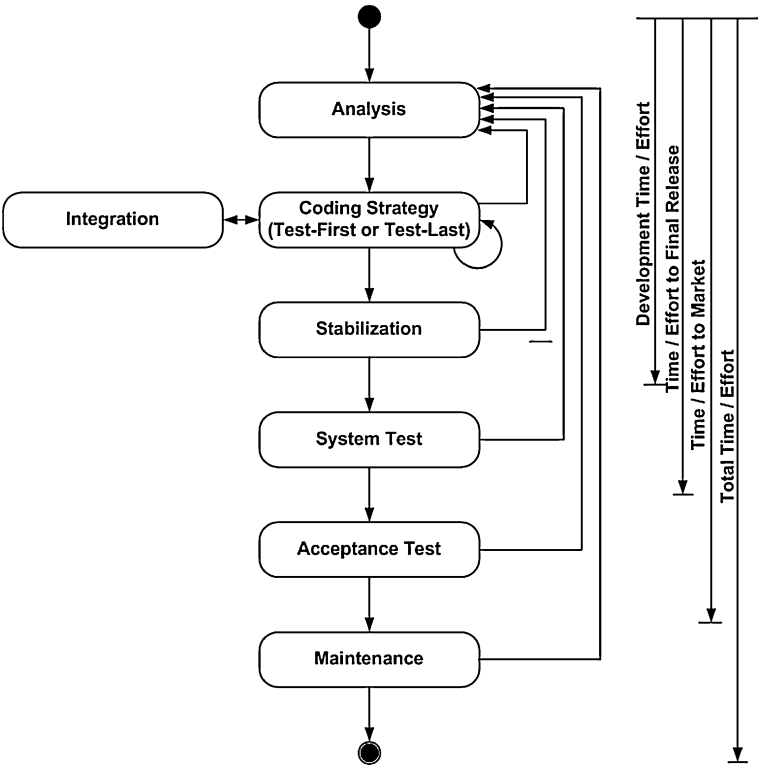

Fig. 1 Development model (simplified)

In all projects, object-oriented design principles were followed and the systems were implemented in Java, using Netbeans IDE on the Windows platform. A version control system was in use to maintain current and historical versions of legacy and production code, as well as all other documentation. A web-based project tracking system recorded progress data and enabled team communication. It enabled measuring of the effort needed for any particular phase of the development cycle. The failure reports from ST and AT were sent to the developers, who had to estimate the severity of the failure and to classify the types of faults. The software product metrics were calculated using a CCCC tool that can also handle Java files.[4]

3.2 Case study details and metrics definition

For the case study, three commercial projects were selected that had well-defined and comparable specifications (high-level requirements were defined according to telecommunication standards) and that were being conducted for the same external customer. The objective for all projects was to develop a simulator for a particular node in IMS (IP Multimedia Subsystem) architecture designed by the wireless standards body 3rd Generation Partnership Project (3GPP), one that simulated call session control function, such that

[4] CCCC—C and C ++ Code Counter http://cccc.sourceforge.net/, a free software tool for the measurement of source code related metrics by Tim Littlefair.

all three simulators were playing the role of a SIP server.[5] The simulators were actually test tools used by the customer in his/her own development process, within the functional and stress test phase of the customer's "high-end" IMS products intended for deployment in the telecommunications market. Any failure of the simulator within the customer development environment would have a direct impact on the customer's development process if not identified on time and could potentially impact the "quality in use" of the customer's IMS product line. The customer had, therefore, money and reputation at risk if the simulators exhibited failures when deployed. Thus, the simulator development teams must use procedures to ensure the highest level of quality. A fully independent testing group at the customer's site performed rigorous acceptance testing in order to verify the software against its specifications.

Equal quality targets and requirements on time to market that were set by the customer for all three projects demanded equal structures and experiences for all teams. In fact, creating a project team was the most important part of the project creation process: in compliance with internal company rules, all project managers monitored the overall performance of the assigned teams as a whole, as well as that of individual team members over time, to ensure that the team remained optimally poised to finish the project successfully. The records on individual experience, skills and training constituted key information used by the project management, who carefully assigned team members to roles and teams (see Sect. 3.1): in order to achieve the required balance, experience and skills were distributed evenly among the teams. A development team of twelve role players was assigned to every project.

All teams were highly motivated with regard to customer satisfaction and to the highest product-quality criterion that was equal for all projects.

The subjects had, on average, over 5 years of professional experience in Java programming and had comparable experience with Test-Last coding practice (see Appendix 1); ad-hoc unit testing using the JUnit Framework has been a widely accepted practice (Test-Last) within the company's R&D departments and was already a part of the internal coding practices before this case study. All involved Java developers attended a mandatory component testing training program (unit testing using the Test-Last approach, refactoring, and continuous integration) and ultimately passed the validation test at the beginning of their Java programming careers. Since then, they have all been continuously involved in Java projects, developing software according to the Test-Last process. Three groups of developers with similar characteristics were formed. All assigned developers had been active participants in different development teams, successfully finishing their Java Test-Last projects directly prior to the start of the projects observed for the purpose of this case study.

Projects A and B were developed according to the Test-Last process, and project C according to the Test-First process. Both processes rely on common agile know-how and skills, the ability to write effective unit tests that is the foundation for both coding practices. TDD is a discipline that differs from Test-Last because tests drive the coding activity. Erdogmus et. al. (2005) refer to this as "test-orientation," and it is one of the most significant characteristics of TDD, one which fully distinguishes Test-First from Test-Last practice. Test-First unit tests are used as the specification for functionality in addition to verification and validation. Adoption of a Test-First mindset is therefore a key factor. It changes the developers' behavior and attitudes during the software development process.

[5] Session Initiation Protocol, Internet Engineering Task Force (IETF), RFC 3261, http://datatracker.ietf.org/doc/rfc3261/).

Some research claims that TDD is also difficult to learn (Crispin 2006). We therefore decided that training of team C on Test-First was needed because they lacked previous experience with Test-First (see Appendix 2, question #4).

Team C was intensively trained in the Test-First approach by solving carefully selected "real-world" problems for 3 weeks before project C started. Theoretical lessons were organized for this purpose, followed by "hands-on" workshops. Several interview sessions were performed during training, and a survey was conducted (see Appendix 2) after the training in order to obtain feedback from developers about the efficacy of the TDD training. An additional survey was conducted after the completion of project C (see Appendix 3). Both surveys were conducted among all 12 team members participating in project C. They were administrated before the start of the case study and were not designed for in-depth qualitative analysis. The questionnaires were anonymous, and the confidentiality of the data was guaranteed. However, it is useful to substantiate our findings with qualitative feedback from the software developers involved in the case study.

The requirements were stable for all three projects. Teams A and B performed up-front design using UML, while project C did not involve any "big design up front," often referred to as BDUF. Moreover, there is no evidence of any "little design up front," referred to as LDUF, nor evidence of any other usage of UML by team C.

When planning the roadmaps, a sustainable pace of 40 work hours per developer per week was considered suitable, but allowing teams to work overtime when necessary. Each team member was obliged to report weekly about the time and effort spent on pre-defined tasks, to update progress, to flag issues or conflicts, to submit revised estimates, and to communicate new requirements using the Primavera Progress Reporter.[6] Presence and time worked were tracked by the Access Control System. No remarkable deviations from the 40 h pace were noted during the projects.

All three projects used the protocol stack library developed in Java—an existing code base that does not have a thorough unit test suite. The functionality of the library was treated as a black-box and isolated from the rest of the application code by redirecting all calls to it via wrapper classes. These wrapper classes were equipped with unit tests enabling indirect, low-level testing of that particular legacy code, thus providing a testing safety net to support the agile development practices that the teams sought to adopt.

The final products were installed on-site on a Linux platform with a MySQL database. The customer used all three products intensively, simulating part of the new generation telecommunication network for the purpose of load and stress testing.

Since the objective of our study was to determine the effectiveness of TDD, we chose code quality, maintainability, and productivity as indicators of effectiveness. All these factors play important roles in the economic aspect of software development. If all three increased, this would be a sign that effectiveness had increased as well. From previously published investigations (George and Williams 2003), it was expected that increased emphasis on testing should increase quality and lower productivity. This was the rationale for investigating the following research questions:

- *Code quality:* Does the TDD practice lead to better external code quality?
- *Productivity*: Is development using TDD more productive than development without TDD?
- *Maintainability*: Is it easier to maintain software developed using TDD?

[6] Primavera Progress Reporter is a web-based program that connects project team members across projects, throughout an enterprise.

Case studies, formal experiments, and surveys are three key components of empirical investigation in software engineering. Formal experiments are too expensive in the industrial environment and are replaced by case studies. An advantage of case studies is that they are also easier to plan than experiments, but results are difficult to generalize and harder to interpret (Kitchenham et al. 1995). In order to gather evidence on the effectiveness of TDD, a case study was chosen that was based on carefully collected metrics from three comparable projects conducted within an industrial environment. Two projects that applied the Test-Last process were compared with a project that applied the Test-First process.

The majority of information presented in this case study was gathered over the course of the projects up to the end of the maintenance milestones. The selection of metrics was based on the availability of recorded data. Whenever possible, it was decided to use the same metrics that had been used in similar case studies. The ISO 9126 standard[7] defines three types of quality measures: internal, external, and quality in use. The internal quality metrics reflects the properties of the code as seen by the programmer. Two metrics were recorded: the number of lines of code (LOC) and McCabe's cyclomatic complexity.[8] The software has to be executed in order to estimate the external quality, which is typically done during the testing phase. External quality was measured by failure[9] density, as in Bhat and Nagappan (2006):

$$Q_{e1} = \frac{\#failures}{\#\,new\,source\;\;LOC} \tag{1}$$

The number of failures is the sum of those failures found in the system and during the acceptance test phase. In fact, this is the number of failures before the product has been accepted by the user. The percentage of failed system tests (Q_{e2}) and acceptance tests (Q_{e3}) could also be treated as external code quality indicators. The same measure was used by George and Williams (2003) and Erdogmus et. al. (2005). Failures found after the release (in the maintenance phase) measure quality in use, as they are identified and reported by the user. Since fault or defect density was used for the quality metrics in the majority of the studies in Table 1, we added this to our set of metrics:

$$Q_i = \frac{\#faults}{\#\,new\,source\;\;LOC} \tag{2}$$

Productivity is measured as the ratio of units of output divided by units of effort. Two categories of output are defined in the IEEE 1045–1992 standard: source statements (lines of code) and function points. Since no support for the function points analysis[10] was available, the number of new lines of code[11] was selected for the output. The total effort was measured by the number of staff-hours needed for the final release. The classical definition was chosen for productivity (Fenton and Pfleeger 1998):

[7] ISO/IEC 9,126. (2001). Software engineering—product quality—part 1: Quality model. ISO 2001.

[8] Cyclomatic complexity is a measure of the complexity of code related to the number of ways there are to traverse a piece of code (McCabe 1976).

[9] An unacceptable behavior of the software is defined as a failure. A fault is a defect in the program that, when executed with certain input data, causes a failure.

[10] See Albrecht and Gaffney (1983) for details.

[11] Unit test code was excluded.

$$Productivity = \frac{\#new\ source\ LOC}{total\ effort} \tag{3}$$

Maintainability metrics was defined according to the ISO 9,126 standard as the effort expressed in the staff-hours needed to correct faults, improve performance, or adapt to a changing environment during the maintenance phase. A change request is a generic term for a modification, a correction or an enhancement of code or of a particular system's function. Since the actual effort depends on the type of "change request," we opted for average effort per change request (AE):

$$AE = \frac{maintenanceeffort}{\#changerequests} \tag{4}$$

In our case, the number of change requests was the sum of failure reports and the number of other change requests that were reported after the product was accepted by the user.

4 Research results

The results of the case study are presented in Table 2. Code quality, productivity, and maintainability are dependent variables, while process type is independent. We selected only those metrics from the project database that might have influenced code quality, productivity, and maintainability. In the analysis, these are treated as extraneous variables. If we want to analyze the effectiveness of TDD, we must control the extraneous variables. The data show that the effort spent on the system test phase is comparable for all three projects. There is a little variance in the lines of code. Significant variability is found in McCabe's cyclomatic number, which is the measure of logical code complexity (McCabe 1976). In our case study, a small variation in code size was expected, since the teams and the requirements were not ideally equal. Fenton and Ohlsson (2000) found no evidence that popular size (LOC) and complexity metrics McCabe) are good predictors of either fault-

Table 2 Quality and productivity indicators

Metric	Projects		
	A Test-last	B Test-last	C Test-first
Q_{e1}: failure density before release [#failures/kLOC]	0.7	0.7	0.5
Q_{e2}: failed system tests [%]	7.8	10.1	5.8
Q_{e3}: failed acceptance tests [%]	6.3	4.0	1.4
Q_{iu}: failure density after release [#failures/kLOC]	0.15	0.15	0.10
Q_4: fault density before release [#faults/kLOC]	2.71	2.30	1.94
Q_5: fault density after release [#faults/kLOC]	0.23	0.24	0.13
Productivity [LOC/staff-hour]	2.3	2.5	1.8
AE: Average maintenance effort [staff-hours/change request]	84	80	59
Sum of all McCabe's cyclomatic complexities	5,879	7,548	4,480
New source LOC	39,554	46,077	38,646
Unit tests LOC/new source LOC	0.38	0.28	0.49

prone or failure-prone modules. This is contrary to what is widely believed: i.e., that failure or fault density is correlated with size. Until there is conclusive evidence of this phenomenon, we can assume that the correlation is weak. Thus, the impact of the observed size variation (less that 20%) on external code quality can be disregarded. The variation in McCabe's cyclomatic complexity is much larger and more difficult to explain. Cyclomatic complexity is actually the number of independent paths through code. More branches and paths make code more complex and therefore more difficult to test, maintain, and understand. This tendency is also observed in Table 2: projects A and B have higher cyclomatic complexity than project C, and consequently also higher maintenance effort. This was also discovered by Janzen and Saiedian (2008), who reported that Test-First programmers consistently produced classes with lower complexity in terms of the number of branches and the number of methods. Therefore, the explanation for the lower cyclomatic complexity of project C could be the result of the TDD approach.

External code quality was measured by failed tests (Q_{e2} and Q_{e3}) and by failure density (Q_{e1} and Q_{iu}). In the system test phase, the most successful was project C, with only 5.8% failed system tests. The same results were also reflected in the acceptance testing (failed acceptance tests) and in maintenance (quality in use Q_{iu}), where project C was again the best. In summary, all quality metrics support the assumption that external quality is better in project C. The same findings were reported in those previous studies listed in Table 1.

The productivity of project C was considerably lower. This decrease in productivity has also been observed by the majority of researchers (see Table 1). George and Williams (2003) reported that TDD developers spent 16% more time on development. The lower productivity was a result of the increased time taken by the TDD team, since the code size was approximately the same for all projects. The ratio 'unit test LOC'/'new source LOC' used as an estimate of testing effort[12] is the highest for project C. In addition, the most substantial development effort (DE), including the effort for writing unit tests, was also observed in project C (see data on DE in Appendix 1). The Test-First approach requires additional and careful preparation of the test unit, which normally results in greater effort in comparison with the Test-Last approach. This is probably one reason for the longer time needed for project C to reach the last milestone ("Final release").

In terms of maintenance effort, project C was again the best, with the lowest average maintenance effort (AE).

On the *"pre-study"* survey that was conducted after the TDD training of team C, we asked developers about their past experience with and opinion about TDD:

- *Existing experience/know-how/skills/practices* (questions 1–5): all developers confirmed their experience in writing unit tests using Test-Last coding practice. The majority of the team had experience in unit test automation and had not favored BDUF in the past. From the answers, we can conclude that the existing experiences of the team allowed for efficient training in Test-Last coding practice.
- *Quality of the training* (questions 6–8): only one developer evaluated the training as average; the others stated that it was good.

In the open-ended responses, several developers noted that, even though they believed that the Test-First approach to be better and would apply it in future, they perceived it as being more difficult than or very different from what they were comfortable with.

On the *"post-study"* survey, we asked the developers for their perception regarding process conformance (question 1), productivity (questions 2–3) and effectiveness

[12] Sanchez et. al. 2007 reported the average ratio across all ten releases as 0.61.

(questions 4 and 6) of the Test-First coding practice, and the influence of test automation on maintainability (question 5). An overwhelming majority of developers believed that they had followed the Test-First strategy faithfully. Also, supporting our findings, a substantial majority believed that Test-First yielded higher quality code and reduced code complexity. It can also be concluded that the majority of developers felt that moderate reduction in productivity was directly related to the specifics of Test-First practice (question 3). All the developers' answers to the question about the positive impacts on maintainability were in favor of test automation, even though some developers had not previously automated their unit tests.

5 Threats to validity

When conducting an experiment or a case study, there is always a set of threats to the validity of the results. Shadish, Cook and Campbell (2002) defined four types of threats: external, internal and statistical conclusion validity, and construct validity.

Threats to *external validity* are conditions that limit our ability to generalize the results of a case study to industrial practice. The strength of our results is that the study was carried out within the industry by professionals in their own environment. However, the generalization potential of the findings is limited, since the sample data cover three projects only. One characteristic of telecommunication software is that requirements are well defined, thus supporting functional test-case design. It would also be too speculative to extend our findings to other software companies developing software that differs from telecommunication software.

Internal validity refers to the extent to which the treatment or independent variable(s) were actually responsible for the effects seen in the dependent variable. The complexity of the requirements may have an impact on all dependent variables. In order to eliminate this threat, equal requirements should have been given to all three teams. Since the company could not afford this, three projects were chosen with comparable requirements, according to expert opinion. The results from these projects were three simulators that provided Call Session Control Function using the same legacy code (SIP stack). Additionally, they all had to be built as a plug-in for the test automation framework applied at the customer's site. Deviations during the processing of signaling (depending on the role of a particular simulator) have no noticeable impact on complexity. For this reason, it was assumed that the requirements did not differ significantly.

The structure and experiences of the teams could also have an impact on all dependent variables. Development resource management within an industrial environment is primarily driven by technical requirements and return on investment. When developing comparable "state-of-the-art" products intended for the same external customer, there is no room for deviations from the accepted requirements. Equal requirements for product quality, comparable functional requirements, and high penalties were key factors in producing a balanced structure, together with the experiences of all the development teams involved. In order to avoid the possible Hawthorne effect, the teams were unaware of the study. Another threat to *statistical conclusion* validity is the lack of inferential statistics in the analysis. This was not feasible because of the small sample size.

Process conformance is a threat to *statistical conclusion validity*. TDD requires strict discipline among developers in always writing the test cases first and code later. In order to check conformity within the process, we used a questionnaire conducted at the end of the case study. A questionnaire is not the most objective instrument, but within the

environment of this study, it was the only feasible one (see Appendix 2 and Appendix 3). The developer interviews provided additional insight into their perceptions.

Unequal levels of motivation or informal competition between the teams could also represent a threat to validity. These could potentially arise from the developer's perception in terms of their own opinion on the effectiveness of two different coding practices. The equal reward scheme that was applied on all projects was a safety net for assuring comparable motivation among the teams. During our observations, we noticed a few isolated discussions among the team members of projects A and B on how to assure appropriate test coverage that would compensate for the possible advantages of the Test-First approach used in project C. However, we noted no deviations in the coding behavior of developers in project teams A and B in comparison with previous projects; the number of unit tests normalized per module is also lower in project A and B in comparison with project C (see Appendix 1).

At the beginning of the training, some developers claimed that TDD was one of the most difficult practices to implement. Interviews showed that acquiring the TDD mindset was generally quite difficult. The most frequent concern was about the increase in the development time needed to strictly design and write the low-level test cases prior to the source coding. Our observations as training proceeded and at the end of training (answers to questions #6, #7 and #8) confirmed that by the end of the training, all subjects involved felt comfortable and content with the TDD approach. Only two developers reported that they had slipped into Test-Last coding strategy from time to time (up to 10%), mostly when they were under time pressure (results of the survey at the end of project C; answers to question #1). This small deviation was regarded as a minor "noise" in the measurements that could not significantly change our conclusions.

The major issue in the study could be the unbalanced testing experience. In an ideal case, all teams should have the same experience. This condition was not explicitly verified, based on a rationale comprising the following assumptions:

1. It was assumed that any potential influence of lack of training of team A and team B on their skills was weak. Based on the available project management documentation, human resource records systematically registered in the past, and on the team selection criteria (please refer to Sect. 3.2), it might be concluded that all developers (assigned to teams A, B, and C) had obtained comparable experience with the Test-Last process; they were continuously involved in developing software according to the Test-Last process after they were trained in component/unit testing (according to the Test-Last process) at the beginning of their careers. Also, they had all been developing software within various Java Test-Last projects directly prior to the start of this case study. The overall recognition of their existing and rich experience in Test-Last was one reason that additional training of teams A and B in Test-Last was not approved by the cost-benefit-driven project management; they assumed that additional training in Test-Last would have had no or at least no remarkable influence on their skills and had approved training of team C only. Thus, the influence of "training bias" (the difference in focus on the details of the process) cannot be completely excluded as a threat; to avoid "training bias" for the purpose of the case study, teams A and B should also have been intensively trained for 3 weeks on Test-Last right before the case study.
2. Test-First training performed right before the new project might be also regarded as an advantage of team C because they would have been very aware and conscious of the new technique. However, we assumed that team C was probably disadvantaged, since the experience gained from the Test-First training (over the duration of 3 weeks), even

though it was intensive and conducted right before the start of the study, probably cannot equalize or outperform the experience gathered through exercising the Test-Last practice on real projects continuously for several years. Coding practices, Test-Last and Test-First fully rely on the agile principles and related skills, particularly the ability to write effective unit tests, which is the foundation skill for both practices. As described above, we dealt with comparable experience in foundation skills. However, the know-how and especially the amount of experience in the process of coding as well as the depth of mind-set adoption were different: since all had considerable experience with Test-Last and its mind-set, while team C had no experience with Test-First, we could not avoid intense training of team C prior to starting the study.

According to these assumptions, we expected that the Test-Last teams would perform better in comparison with the Test-First team. We also expected that this could be one of the main causes if the results of this study were the reverse. Since team C was ultimately better and at the same time, in our opinion, disadvantaged regarding past experience, we believe that any potential threat to validity arising from "training bias" could be disregarded.

Threats to *construct validity* refer to the extent to which the case study setting actually reflects the construct under study. A potential threat in this study is the construct validity of the dependent variable code quality. Measuring quality by counting failures or passed acceptance tests constitutes a widely accepted metrics based on well-founded concepts in software engineering. External quality depends on the number of unrevealed faults and the quality of testing. Acceptance testing was performed externally by the customer according to his/her own test specifications, which were unavailable to the team. The number of test cases (see Appendix 1) was almost the same for the Test-Last projects. Even though the Test-First project had been tested with more test cases, it had the lowest number of failed system and acceptance tests.

6 Conclusions

This paper presents experiences obtained from the introduction of TDD into the S&S model. Comparison of three projects conducted within an industrial environment indicates that, at least in our environment, TDD practice appears to improve external code quality and quality in use. Productivity, however, is remarkably lower when TDD is applied. The main advantage is the maintainability, which is considerably better for the following reasons:

- TDD automatically creates a thorough unit-test safety net that provides seamless, low-level regression testing whenever the source code is changed.
- TDD may also lead to lower cyclomatic complexity and, in our opinion, better design, since a developer writes only as much source code as is really needed. Implicitly, better design expressed by lower complexity reduces the effort needed for debugging during the maintenance phase.

An important consideration is that the quality improvement was also recognized by the developers on the survey at the end of project C (answers to questions #4 and #6, Appendix 3). However, their opinions do not support our conclusions about design improvement (answers to question #4, Appendix 3).

These results need to be viewed within the limitations of this case study. Further empirical studies in the industry (more data points) could provide more reliable evidence for TDD efficacy, and thus strengthen or disprove these findings.

Appendix 1

See Table 3.

Table 3 Projects details

	Project A	Project B	Project C
Process	Test-last	Test-last	Test-first
Team size (TS)	12	12	12
Average experience level (total/Java programming) [years]	8.7/5.0	8.3/4.9	8.5/5.1
No. of use cases (NUC)	61	62	63
No. of modules (Java classes/interfaces) (NM)	511	565	794
No. of unit tests (NUT)	2,054	2,387	3,734
Code base (reused code) LOC (RLOC)[a]	20,013	20,013	20,013
New source LOC (SLOC)	39,554	46,077	38,646
Unit tests LOC (ULOC)	15,206	13,001	19,023
Total new LOC (TNLOC = SLOC + ULOC)	54,760	59,078	57,669
Total LOC (TLOC)	74,773	79,091	77,682
ULOC/SLOC	0.38	0.28	0.49
Sum of all McCabe's cyclomatic complexities of SLOC)	5,879	7,548	4,480
No. of system test cases (NSTC)	243	248	274
No. of failure reports in the system test phase (FRST)	19	25	16
No. of faults in the system test phase (FST)	89	93	70
No. of acceptance test cases (NATC)	112	124	143
No. of failure reports in the acceptance test phase (FRAT)	7	5	2
No. of faults in the acceptance test phase (FAT)	18	13	5
No. of change requests (failures) in the maintenance phase (FRMP)	6	7	4
No. of faults in the maintenance phase (FM)	9	11	5
No. of change requests (enhancement, modifications) in the maintenance (CRMP)	13	11	10
Total number of change requests (TNCR = CRMP + FRMP)	19	18	14
Total number of faults (TNF = FST + FAT + FM)	116	117	80
Development time (DT) [days]	123	119	134
Time to final release (FRET) [days]	183	189	224
Time to market (MT) [days]	191	193	228
Maintenance period (MP) [days]	260	260	260
Development effort (DE) [staff-hours]	11,765	10,987	14,654
Effort until final release (FRE) [staff-hours]	17,501	18,204	21,567
System Test effort (STE) [staff-hours]	5,718	5,537	6,238
Maintenance effort (ME) [staff-hours]	1,600	1,455	820
Overall effort (OE = FRE + ME) [staff-hours]	19,101	19,659	22,387
Test effort/development effort [%]	48.6	50.4	49.3

[a] The same SIP stack legacy code was used in all three projects

Appendix 2

See Table 4.

Table 4 Survey after TDD training

#	Question (to team C)	Answer	Result
1	How long have you been a Java developer?		5.1 years (average)
2	Are you in favor of BDUF ("Big Design Up Front")?	a. Yes	Yes = 5
		b. No	No = 7
3	Had you ever written unit tests before this training?	a. Yes	Yes = 12
		b. No	No = 0
4	Based on the answer to #3, what coding strategy did you use?	a. Test-first	Test-first = 0
		b. Test-last	Test-last = 11
5	Based on the answer to #3, do you automate your unit tests?	a. Yes	Yes = 7
		b. No	No = 5
6	Please specify the quality of your TDD training:	a. Poor	Poor = 0
		b. Average	Average = 1
		c. Good	Good = 11
7	Based on the examples and "know-how" gathered during training, do you think writing unit tests helps the developer produce better code quality ?	a. Yes	Yes = 8
		b. No	No = 3
8	Would you apply TDD in your future projects?	a. Yes	Yes = 9
		b. No	No = 2
		c. Comment:	

Appendix 3

See Table 5.

Table 5 Final survey

#	Questions (to team C)	Answer	Result
1	How faithfully have you been following the "Test-First" strategy in coding?	a. Please specify [%]:	98% (average)
		b. Comment:	
2	How much effort do you think writing unit tests adds to your traditional effort?	a. None	None = 0
		b. <10%	<10% = 5
		c. 10–20%	10–20% = 6
		d. 20–30%	20–30% = 0
		e. >30%	>30% = 0

Table 5 continued

#	Questions (to team C)	Answer	Result
3	How much time do you feel applying the "Test-First" strategy adds to your time when you apply the "Test-Last" strategy?	a. None	None = 0
		b. <5%	<5% = 2
		c. 5–10%	5–10% = 7
		d. 10–20%	10–20% = 1
		e. >20%	>20% = 0
		f. Comment:	
4	What are the major benefits of the "Test-First" approach (Check all that apply):	"Cleaner" code	"Cleaner" code = 6
		Better design	Better design = 3
		Lower complexity	Lower complexity = 7
		Fewer "Code Smells"[a]	Fewer "Code Smells" = 5
		Specify:	Specify: quality = 9
5	Do you think that unit test automation is helpful in terms of maintainability?	a. Yes	Yes = 11
		b. No	No = 0
6	Do you believe that TDD improves code quality in general?	a. Yes	Yes = 9
		b. No	No = 2

[a] Code smell is any symptom in the source code of a program that could indicate a deeper problem—see Fowler et. al. (1999)

References

Albrecht, A. J., & Gaffney, J. E., Jr. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering, SE-9*(6), 639–648.

Beck, K. (2001). Aim, fire. *IEEE Software, 18*(5), 87–89.

Beck, K. (2003). *Test driven development: by example*. USA: Addison Wesley Professional.

Bhat, T., Nagappan, N. (2006). Evaluating the efficacy of test-driven development: industrial case studies. *Proceedings of International Symposium, Empirical Software Engineering (ISESE 06)*, ACM Press, 356–363.

Crispin, Lisa. (2006). Driving software quality: how test-driven development impacts software quality. *IEEE Software, 23*(6), 70–71. doi:10.1109/MS.2006.157.

Cusumano, M. A., & Selby, R. W. (1997). How microsoft builds software. *Communications of the ACM, 40*(6), 53–61.

Damm, L.-O., & Lundberg, L. (2006). Results from introducing component-level test automation and test-driven development. *Journal of Systems and Software, 79*, 1001–1014. Elsevier Science Inc.

Edwards, S. H. (2003). Using test-driven development in the classroom: providing students with concrete feedback on performance. *Proceedings of International Conference on Education and Information Systems: Technologies and Applications (EISTA '03)*.

Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering, 31*(3), 226–237.

Fenton, N. E., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering, 26*(8), 797–814.

Fenton, N. E., & Pfleeger, S. L. (1998). *Software metrics: A rigorous & practical approach* (2nd ed.). Boston, USA: PWS Publishing Company.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. USA: Addison Wesley.

George, B., Williams, L. (2003). An initial investigation of test driven development in industry. *ACM Symposium on Applied Computing (SAC)*, March.

Janzen, D. S., & Saiedian, H. (2008). Does test-driven development really improve software design quality? *IEEE Software, 25*(2), 77–84.

Jeffries, R. E. (1999). Extreme Testing. *Software Testing and Quality Engineering, 1*(2), 23–26, March/April.

Jeffries, R., Melnik, G. (2007). TDD: The art of fearless programming. *IEEE Software,* May/June, 24–30.
Jovanović, D., Dogša, T. (2003). Comparison of software development models and their usage in computer-telephony systems. In *ConTEL 2003: Proceedings of the 7th international conference on telecommunications*, University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia (vol. 2, pp. 587–592), June 11–13.
Kitchenham, B., Pickard, L., & Pfleeger, S. L. (1995). Case studies for method and tool evaluation. *IEEE Software, 12*(4), 52–62.
Madeyski, L. (2005). Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. *Frontiers in Artificial Intelligence and Applications.Proceeding of the 2005 conference on Software Engineering*, vol. 130, pp. 113–123.
Madeyski, L., Szała, Ł. (2007). The impact of test-driven development on software development productivity—An empirical study. *Software Process Improvement 14th European Conference, EuroSPI 2007*, Potsdam, Germany, September 26–28, 2007 in Proceedings lecture notes in computer science, Springer Berlin/Heidelberg, vol. 4764.
Malik, S., Palencia, J. R. (1999). Synchronize and Stabilize vs. Open-source–An analysis of 2 team-organization models when developing consumer level shrink-wrapped software. Computer Science 95.314A Research Report, Carleton University, Ottawa, Ontario, Canada.
Martin, R. C., Newkirk, J. W., Kess, R. S. (2002). *Agile software development, principles, patterns, and practices* (1st ed.). Prentice Hall, Englewood Cliffs.
Maurer, F., Martel, S. (2002). On the productivity of agile software practices: An industrial case study. Technical report, University of Calgary, Department of Computer Science, Calgary, Alberta, Canada.
Maximilien, M., Williams, L. (2003). Assessing test-driven development at IBM. *Proceedings of 25th International Conference on Software Engineering* (pp. 564–569).
McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320.
Mueller, M. M., & Hagner, O. (2002). Experiment about Test-First Programming. *IEE Proceedings–Software, 149*(5), 131–136.
Newkirk, J., Vorontsov, A. (2004). *Test-driven development in microsoft NET*. Microsoft Press.
Pfleeger, S. L., & Atlee, J. M. (2006). *Software engineering: Theory and practice*. New Jersey, USA: Prentice Hall.
Sanchez, J., Williams, L., Maximilien, E. M. (2007). A Longitudinal Study of the Use of a Test-Driven development Practice in Industry. *AGILE 2007 Conference (AGILE 2007)*, 13–17 August, Washington, DC, USA. IEEE Computer Society 2007, ISBN 0-7695-2872-4.
Shadish, W. R., Cook, T. D., & Campbell, D. T. (2002). *Experimental and Quasi-experimental designs for generalized causal inference*. Boston, MA: Houghton Mifflin Company.
Siniaalto, M., Abrahamsson, P. (2007). A comparative case study on the impact of test-driven development on program design and test coverage. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society, Washington, pp. 275–284, ISBN ~ ISSN: 1938–6451.
Stephens, M., Rosenberg, D. (2004). The Irony of Extreme Programming. *Dr. Dobb's Journal*, May 2004.

Author Biographies

Tomaž Dogša is an associate professor at the University of Maribor in Slovenia. His research interests include software and hardware testing, testing tools development and circuit simulation. He received his Ph.D. in computer science at the University of Maribor. He also leads the Centre for Verification & Validation of Systems. Contact him at tdogsa@uni-mb.si.

David Batič is the practice leader for agile development and managing director of Agileon d.o.o., Slovenia (www.agileon.eu). He is also a postgraduate student at the Faculty of Electrical Engineering and Computer Science at University of Maribor. His research interests include software testing and reliability, test automation, test-tools, software process, and particularly agile software development. Contact him at batic@agileon.eu