14 IMPROVING BUSINESS AGILITY THROUGH TECHNICAL SOLUTIONS: A Case Study on Test-Driven Development in Mobile Software Development

Pekka Abrahamsson
VTT Technical Research Centre of Finland
Oulu, Finland

Antti Hanhineva
Elbit Oy
Oulu, Finland

Juho Jäälinoja
Nokia Technology Platforms
Oulu, Finland

Abstract *This paper maintains that efficient business agility requires actions from all levels of the organization in order to strive for success in a turbulent business environment. Agility and agile software development solutions are suggested as yielding benefit in a volatile environment, which is characterized by continuously changing requirements and unstable development technologies. Test-driven development (TDD) is an agile practice where the tests are written before the actual program code. TDD is a technical enabler for increasing agility at the developer and product project levels. Existing empirical literature on TDD has demonstrated increased productivity and more robust code, among other important benefits. This paper reports results of a case study where a mobile application was developed for global markets, using the TDD approach. Our first results show that the adoption of TDD is difficult and the potential agility benefits may not be readily available. The lessons learned from the case study are presented.*

1 INTRODUCTION

This paper has its roots in the software engineering discipline where agile methods and principles have gained a significant amount of attention recently. Agile software

development ideas can be traced back as early as the 1960s and even beyond (Larman and Basili 2003). Since the mid-1990s, several methods have been proposed to meet the needs of the turbulent business environment (for an overview of the existing methods, see Abrahamsson et al. 2002; Boehm and Turner 2003). Empirical evidence is scarce but quickly emerging. Abrahamsson et al. (2003) present the evolutionary path of agile software development methods and propose that the software engineering and information systems fields have, independently of each other, approached similar conclusions on the state of IS/SE development. The existing methods, to a certain extent, are idealized views, holding a strong prescriptive orientation, on how software and systems should be constructed. The agile movement seeks to provide an alternative view on software development through a set of values and principles (for details, see www.agilemanifesto.org).

The mobile telecommunications industry has shown itself to be comprised of a highly competitive, uncertain, and dynamic environment (Lal et al. 2001). Agile software development solutions can be seen as providing a good fit for the mobile environment, with its the high volatility and tough time-to-market needs. Mobile applications are generally quite small and the majority of them are developed by small software teams. Organizations operating in this type of business environment need to react rapidly to changing market needs. The efforts of organizations attempting to increase their responsiveness will fall short if agility is not pursued at all levels of the organization, including partnered or collaborative development at the interorganizational level. If organizational structures do not support rapid information sharing and short feedback cycles, agility benefits are not achieved. Indeed, a number of organizations are keenly interested in adopting some set of agile practices and principles for use. Test-driven development (TDD) is one of several agile practices. It has become popular with the introduction of the eXtreme Programming (Beck 1999) method. The aim of TDD is to offer agility benefits through an automated unit test suite and more robust code. Extensive automation is required, since agile principles promote common code ownership and expect the system to be always running. Other important benefits have also been suggested. Empirical evidence regarding the application of TDD in different environments is still thin.

This paper reports results from a case study where a mobile application was developed for global markets in a close-to-industry setting, using the controlled case study approach (Salo and Abrahamsson 2004) as the research method. The development team was very successful in achieving the business target. Yet, they applied the TDD approach with poor results. Only 7.8 percent of the code had associated unit tests. While the results remain inconclusive with regard to concrete benefits of TDD, the lessons learned from this case study bear important implications for developers and business managers. These implications are addressed.

The remainder of the paper is organized as follows. The next section introduces briefly the test-driven development approach including a review of the existing empirical body of evidence. This is followed by the description of the empirical research design. The fourth section presents the results of the empirical case, which is followed by discussion on the implications of the results and lessons learned.

2 TEST-DRIVEN DEVELOPMENT

Test-driven development is a programming technique where tests are written before the actual program code (Astels 2003). TDD is an incremental process (Figure 1). First a test is added and then the code to pass this test is written. When the test is passed the code is refactored. Refactoring is a process of making changes to existing, working code without changing its external behavior (Fowler 1999), i.e., the code is altered for the purposes of commenting, simplicity, or some other quality aspect. This cycle is repeated until all of the functionality is implemented.

The practitioner literature on TDD (e.g., Astels 2003; Beck 2003) identifies several potential benefits that can be gained by the application of the programming technique. These benefits are

- Give the developer confidence that the created code works
- Allow efficient refactoring through an extensive safety net
- Enable fast debugging through a test suite that helps to pinpoint defects

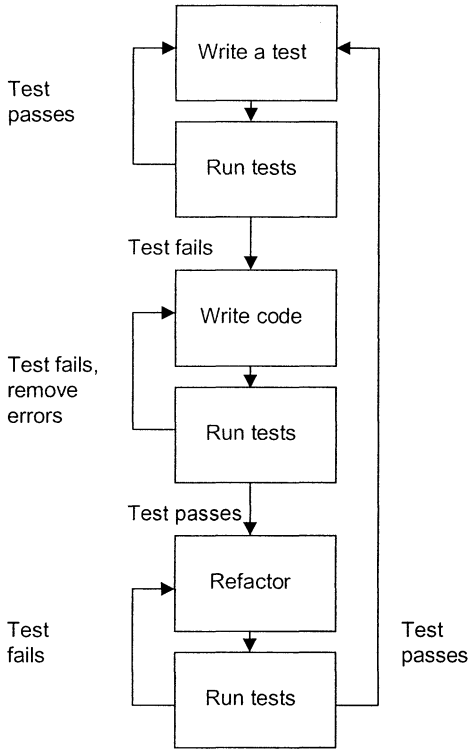

Figure 1. Steps in Test-Driven Development (adapted from Astels 2003; Beck 2003)

- Improve software design by producing less coupled and more cohesive code
- Enable safer changes
- Create up-to-date documentation on the code
- Help developers avoid over-engineering by setting a limit on what needs to be implemented

Every time the tests pass, the developer gets a small dose of positive feedback, making the programming more fun. The unit tests in TDD have three distinct parts: setup, exercise the functionality, and check for postconditions (Astels 2003). The tests are collected into test classes to make running and maintaining the tests easier. TDD relates to refactoring in two ways: after the code is written, the refactoring is used to clean up the code, and when refactoring, the extensive test set built with TDD helps the developer gain certainty that the refactoring did not break the system.

According to quantitative data from recent studies (Edwards 2004; George and Williams 2003; Langr 2001; Maximilien and Williams 2003; Müller and Hagner 2002; Pancur et al. 2003; Williams et al. 2003; Ynchausti 2001), TDD appears to produce higher quality systems but also to increase the development time. A high test coverage is easier to achieve with TDD than with the traditional techniques. TDD forces developers to write unit tests, because the tests are such an essential part of the development that they cannot be left out. The empirical evidence found in the literature shows that the amount of tests in TDD varies from 50 percent less test code than production code to 50 % more test code than production code.

Table 1 summarizes the quantitative empirical body of evidence on test-driven development. Table 1 is divided into five columns, based on the type of finding: TDD versus traditional testing, productivity, quality, test coverage, and ratio of production code versus test code.

Qualitative data from TDD studies (Barriocanal and Urban 2002; Beck 2001; Edwards 2004; George and Williams 2004; Jeffries 1999; Langr 2001; Kaufmann and Janzen 2003; Maximilien and Williams 2003; Müller and Hagner 2002; Pancur et al. 2003; Rasmusson 2003; Williams et al. 2003; Ynchausti 2001) indicate that the test suite produced brings value a system throughout its lifetime. This is due to the fact that the changes are safer to implement in any phase of the system's life cycle. TDD also changes manual debugging to the more-structured task of writing tests. However, TDD is not easy; many developers have prejudices against the practice.

Empirical literature shows that TDD is difficult to use and that it increases the workload of developers, causing them to write less-functional code. These prejudices can be fought with training and support, especially in the beginning of the adoption of TDD. If support is not provided, it is likely that the TDD practice will not work. It also seems that TDD is not suitable for all kinds of development environments; it is highly dependent on the testing framework and requires that the developers using it be motivated and skilled.

Table 2 summarizes the qualitative empirical body of evidence on test-driven development. The first column in Table 2 indicates if a particular finding provides qualitative support (i.e., symbol "↑") for the application of TDD. Symbol "↓," on the other hand, indicates that the finding offers qualitative evidence against TDD. An empty space refers to "neither." This means that a particular finding provides a deeper understanding on a particular aspect with respect the use of TDD in certain environments.

Table 1. Quantitative Empirical Body of Evidence on Test-Driven Development

Type of Study, Reference	TDD versus Traditional Testing	Productivity	Quality	Test coverage	Ratio of Production Code versus Test Code
TDD versus traditional, (Langr 2001)	33% more in tests TDD	–	–	–	50% more tests than code
TDD versus ad hoc testing, (Maximilien and Williams 2003)	No unit tests in ad hoc testing	Minimal impact on productivity in TDD	50 % lower defect rate in TDD	–	50% less tests than code
TDD versus traditional (Edwards 2004)	–	–	45% fewer defects in TDD	–	–
TDD versus traditional (Williams et al. 2003)	52% fewer tests in TDD	Same productivity in both	40% lower defect rate in TDD	–	50% less tests than code
TDD versus iterative test last (Pancur et al. 2003)	–	–	–	92.6 % combined	–
TDD versus waterfall, (George and Williams 2003)	No unit tests in waterfall	TDD took 16 % more time	TDD passed 18% more black box tests	98% method, 92% statement 97% branch	–
TDD versus traditional (Ynchausti 2001)	No unit tests in traditional	TDD took 60 – 100% more time	38 – 267% fewer defects in TDD	–	Equal amount of test and code LOC
TDD versus traditional (Müller and Hagner 2002)	–	"Slight increase on used time in TDD"	"Slight increase on reliability in TDD"	–	–

Table 2. Qualitative Empirical and Anecdotal Body of Evidence on Test-Driven Development

	Result	**Reference**
↑	The vast test set that comes with TDD helps to refactor with confidence that the code works.	George and Williams 2004; Langr 2001
↑	TDD developers are more confident in their code.	Edwards 2004; Kaufmann and Janzen 2003; Pancur et al. 2003
↑	Test set created via TDD will continue to improve the quality of the system throughout its lifetime.	Maximilien and Williams 2003; Williams et al. 2003
↑	Adding new functionality to the system built with TDD was easier than to a traditionally built system.	Langr 2001; Maximilien and Williams 2003; Williams et al. 2003
↑	TDD produces more testable code, because there is a test already written for it.	George and Williams 2003; Langr 2001
↑	In TDD, the unit testing actually happens, it cannot be left out because it is an essential part of the development.	George and Williams 2003; Maximilien and Williams 2003
↑	Most developers thought that TDD improves productivity and is effective.	George and Williams 2003
↑	The developers' time is more efficiently used writing unit tests than manual debugging.	George and Williams 2004; Williams et al. 2003; Ynchausti 2001;
↓	Resistance at first to use TDD due to inexperience and growth in the amount of work.	Maximilien and Williams 2003; Müller and Hagner 2002
↓	Developers thought that because of writing tests they had time to write less functionality.	Pancur et al. 2003
↓	Nearly half of the developers thought that TDD faces difficulty in adoption.	George and Williams 2003
	TDD training can be used to overcome negative impressions of the TDD practice.	Ynchausti 2001
	When no support for TDD was available, inexperienced developers slipped back to no unit testing development, support at least in the early stages is needed.	Jeffries 1999; Rasmusson 2003
	Given a chance, only 10 percent of students wrote unit tests.	Barriocanal and Urban 2002
↓	The TDD group produced insufficient unit tests.	Kaufmann and Janzen 2003
	If the tests are not automated, they are less likely to be run.	Maximilien and Williams 2003
↓	Graphical user interfaces are hard to build with TDD.	Beck 2001
	Writing test cases for hard-to-test code requires skill and determination from the developers.	George and Williams 2004

3 EMPIRICAL RESEARCH DESIGN

3.1 Research Method

The research approach used in this study contains elements of case study research (Yin 1994), action research (Avison et al. 1999) and experimentation (Wohlin et al. 2000). This type of specific approach has been labeled as the *controlled case study approach* (Salo and Abrahamsson 2004). The term *controlled* is used intentionally. Empirical studies include various forms of research strategies (Basili and Lanubile 1999). *Controlled* is most often associated with the experimentation approach. One central difference between the research strategies is the *level of control.* Following Wohlin et al. (2000, p. 12), "experiments sample over the variables that are being manipulated, while the case studies sample from the variables representing the typical situation." If this is accepted, the experimentation approach can be seen as "a form of empirical study where the researcher has a control over some of the conditions in which the study takes place and control over the independent variables being studied" (Basili and Lanubile 1999, p. 456). Therefore, the use of term controlled in this type of study approach implies that the researchers were in a position to design the implementation environment, i.e., the typical situation (see the next subsection on research setting), beforehand. The developers in this case study developed the product in VTT's laboratory setting close to the researchers.

3.2 Research Setting

A team of four developers was gathered to implement a mobile application for global markets. Three of the four developers were fifth or sixth year university students with industrial experience in software development. One of the developers was an experienced industrial developer. The team worked in a colocated development environment and used a tailored version (i.e., tailored to meet the needs of mobile software development) of the eXtreme Programming method. This paper focuses on one aspect of the used approach, the test-driven development technique.

The project was supported and monitored by a support team in which two of the authors participated. The supporting tasks for TDD consisted of constructing the TDD approach for mobile Java-enabled devices, providing training for the use of the approach, following the TDD process during the project, and assisting in possible problem situations. One of the authors followed the TDD on a biweekly basis and informally discussed the results with the team. On one occasion, one of the authors facilitated the team by participating in test code development, with the goal of providing the team a hands-on example on how the team's TDD practice could be improved.

3.3 Data Collection

Both quantitative and qualitative data were collected. Table 3 indicates the type of data collected, the rationale for its collection, and the interval when it was collected.

Table 3. Collected Quantitative and Qualitative Data

Collected Data	Rationale	Type	Collection Interval
Lines of code	Ratio of Test LOC/Application LOC (%)	Quantitative	After each iteration
Effort use	Test development effort used/Application development effort used (%)	Quantitative	Daily
Productivity	LOC/hour	Quantitative	Daily
Structured team interview	Team perception of the use of TDD	Qualitative	After the project
Post-iteration workshop (Salo 2004)	Team perception. Note, a process improvement mechanism (not only TDD issues)	Qualitative	After each iteration
Research notes	Research ideas, observational findings	Qualitative	Daily, during the project

The support team validated the data on a daily or weekly basis, depending on the type of data collected. The purpose of evaluating the effort used for developing tests compared with the effort used in developing the application code is to provide a metric on how much the approach is used in the project.

The qualitative data is collected from three sources: the team interview, post-iteration workshops, and the research notes. The structured interview (recorded and transcribed) was conducted after the project. One of the authors kept systematic research notes with his observations throughout the project. The purpose of collecting qualitative data is to find out if there is a correlation between qualitative and quantitative data collected in the project.

4 CASE STUDY RESULTS

4.1 Case Project Overview

The aim of the project was to produce a production monitoring application for mobile Java devices. The product is an added-value service for the existing production planning system that enables a salesperson to visually view the state of the production anywhere, anytime. The mobile Java application is based on a similar application running on the desktop environment. The project, therefore, aimed at transforming the existing product to a mobile environment with reduced functionality. The limited resources of the mobile devices, however, forced the mobile Java application to act as a browser for the existing data. The application was written in Java 2 Micro Edition, using the MIDP 2.0 profile.

Table 4. The Lines of Code for the Client, Server and the Whole Application

	Test code	Application Code	Total Code	% of Tests from Application
Client	208	2665	2873	7.8
Server	78	972	1050	8.0
Total	286	3637	3923	7.9

The project was conducted in the spring of 2004. The total duration of the project was nine weeks, which includes a system test and fixing phases. The project was divided into five iterations, starting with a 1-week iteration, which was followed by three 2-week iterations, with the project concluding in a final 1-week iteration.

4.2 Quantitative Results

This subsection presents the quantitative results of the case study. Table 4 presents overall data on the application, in terms of the total test and application lines of code for the client, the server and the whole application, including the percentage of test lines of code from the application lines of code. Table 4 highlights the fact that the level of test code is very low.

Figure 2 presents the correlation between test code and application code measured in lines of code on the client side of the application, where TDD was intended to be used. The tools used for the development of the server did not enable the use of TDD, and therefore, the server is excluded from the subsequent analysis. The data is presented by iteration and measured by lines of code.

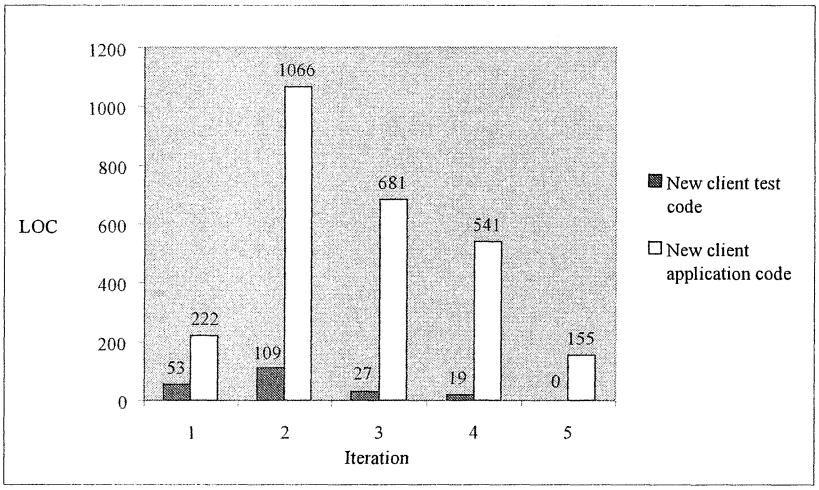

Figure 2. Correlation Between Test Code and Application Code

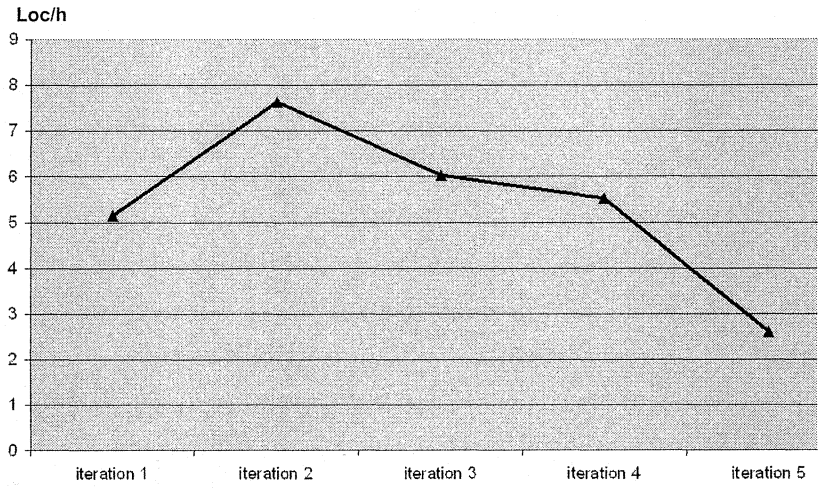

Figure 3. Total Productivity in the Case Project

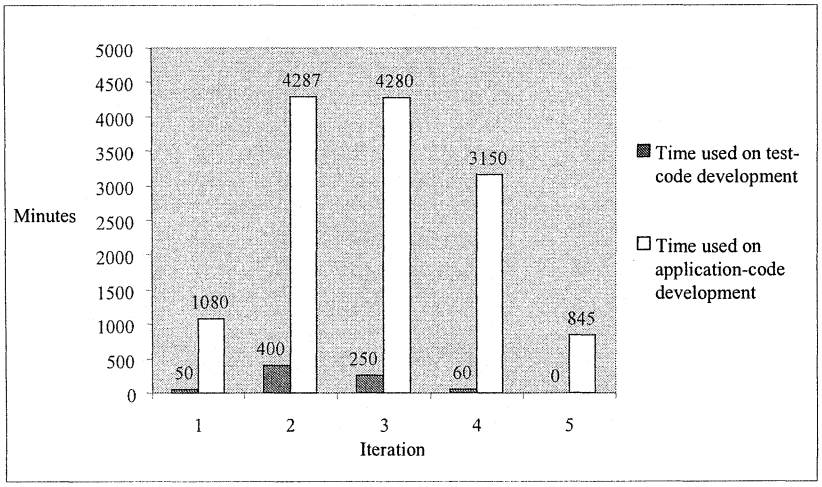

Figure 4. Correlation Between Test-Code Development Time and Application-Code Development Time

As it can be seen in Figure 2, the amount of test code compared to the application code is significantly smaller. The total productivity (Figure 3) is lower in the short iterations (1 and 5) and higher in the longer iterations, however, it drops toward the end of the project. The test code productivity follows roughly the same pattern: in the first

iteration, some test code is written, the maximum productivity is reached on iteration 2, and the amount of test code drops from there, leading to the last iteration which did not produce any test code at all.

Figure 4 presents the correlation between the test code development time and the application code development time. Contrary to the lines of code metric, the use of time metric is presented on the level of the whole application. The data is presented by iteration and measured by minutes used in the respective development modes.

Similar to Figure 2, the amount of time used on test-code development is significantly smaller than the time used in application-code development. The time used on application-code development is shorter in the first and fifth of the 1-week iterations than in the 2-week iterations. The largest amount of time was used for application-code development in the second iteration, and the time drops from there toward the end of the project. The time used on test-code development follows the same pattern as the time used in application-code development: In the beginning of the project, the most time was used for test-code development, and the time used drops from there to the last iteration, where no time was used on the test-code development. The total percentage of time used in test-code development is 5.6 percent.

4.3 Qualitative Results

During the project, the researchers perceived that the developers did not seem to see the benefit of the tests; they regarded them more as a burden. The fact that the team did not see the benefit of the tests was realized, for example, on one occasion when the team used two hours to debug the application without running the tests. Afterward, a member of the team commented that if they had run the tests, they would have caught the defect. The team's attitude toward TDD was also seen in the fact that the developers easily slipped into working in the traditional mode of developing software first and forgetting the tests until the end of the development. Comments included

> I don't think that we could have found faults [with TDD tests] that we could not have found otherwise.

> We just thought that the tests do not offer us any advantage.

The TDD practice clearly had some difficulties in adoption. The team had some negative impressions about it, but they also admitted that some of the reasons for not using TDD were their own.

> The whole TDD practice, where you write tests before the program code, is stupid....The amount of tests will grow so large there is no sense in that.

> Maybe we should have been forced to do the tests.

The limited physical resources on the client end of the application developed in the project forced the mobile application to act as a browser for the existing data. The

application followed the client-server model, and because of the lack of the physical resources on the client end of the application, most of the data processing was done on the server side. So the client's main function was to act as a user interface for the server side. This creates difficulty, which also came up in the team interview.

> TDD is not easily applicable to...user interface development and that is just what we are doing. We had very few things to which it would have suited naturally.

During the project, the researchers became concerned about the low number of test code lines and tried to promote the use of TDD by making the TDD process easier for the developers. One of the authors asked the team if the approach used in the project should be made easier to use. The improvement consisted of making running the tests easier by automating the code changes needed in the main class of the application. The team did not see this as being necessary and claimed that the current way, where the changes had to be done by hand every time the team needed to change between TDD and running the application, was effective enough. The team was under delivery pressure as well.

Yet, as stated before, the team was inexperienced with TDD. They thought that they could have achieved better results if they had had more time to get acquainted with the practice and possibly had more support while developing

> This was the first time for all of us to try TDD. We probably would have been more capable of using the practice if we had used it previously to develop something [for which] it is better suited.

> Maybe if we had a bit longer time to do the training we could have been more capable of using TDD....Also if a member of the support team would have been with us while developing, it would have helped us to do TDD.

Although the team considered that TDD had difficulties when developing this kind of application, they felt that it could provide advantages in a different kind of application area

> TDD could save time at later development phases when adding functionality to application; the developer could use the tests to see if it broke the existing functionality.

> I think TDD is good for testing logic...test set could be run to verify if the application broke or not.

The qualitative findings offer interesting results. While the team observed that TDD could provide them some help, they were not very keen in utilizing the practice if not made mandatory. Even when the research team proposed a significant improvement opportunity for the TDD approach in mobile environment, the team refused to give it a try. Yet, the team conceived the TDD practice was useful with testing the logic and verifying the functionality of the software.

5 DISCUSSION

Technical agile solutions, such as TDD, are designed for the type of volatile development environment presented in this study. However, as the results show, this study is inconclusive with regard to the concrete benefits of TDD. We cannot, therefore, determine whether TDD positively or negatively affected the software development. Yet, the project was a remarkable business success, producing a fully marketable mobile application in a very short time frame. Our findings are of importance for practitioners who aim at using agile solutions in their development settings as well as for researchers who conduct case studies and experiments in the area. In the following, the results are mapped against the existing empirical body of evidence, after which the implications in terms of concrete lessons learned during the study are addressed.

5.1 Mapping the Results to Existing Empirical Body of Evidence

TDD studies have shown that TDD projects generally produce somewhere from 50 percent less to 50 percent more test code than application code (Langr 2001; Maximilien and Williams 2003; Williams et al. 2003; Ynchausti 2001). In this study, the ratio was only 7.8 percent. This could also indicate that TDD is poorly applicable for the mobile Java environment due to technical challenges. Yet, the particular approach designed for this study was pre-tested by the research team and found feasible. More importantly, the low amount of test code can be explained by observing the qualitative data, where the development team clearly indicated reluctance for adoption of TDD for actual use, due to reasons of difficulty, inexperience, and application domain. In particular, the developers expressed that TDD was not suitable the kind of application that the project involved (i.e., a browser type with a strong focus on user interfaces). TDD authors have brought this up earlier (e.g., Beck 2001). Prior experience in unit testing generally, and TDD in particular, has been found to contribute to the adoption rate. Our study is in line with these findings. The development team had not been exposed to test-first design or development prior to the project. George and Williams (2004) also propose that the adoption of the TDD practice requires determined and skilled developers.

In terms of effort used, results show that in the first iteration, the team used up to 30 percent of effort for TDD. This dropped quickly in the subsequent iterations. Qualitative evidence points out that the team found TDD provide them little or no added value, for the to reasons explicated above. It should be noted that the server side of the software was developed in the desktop environment, and the team used JUnit as the testing tool in that project. Although having a different, more sophisticated tool for TDD available, the team still did not manage to produce tests.

5.2 Lessons Learned

Half way through the case project, the research team realized that the TDD technique was not going to be systematically used within the project. Some measures

(i.e., extra training and mentoring support) were used to ease the adoption of the technique but, as the results show, the situation was not improved. Therefore, it is important to understand the reasons for the reluctance to adopt the TDD technique in practice.

5.2.1 Lack of Motivation

A proper use of TDD requires that developers write about the same amount of test code as actual production code. Therefore, it requires a lot of motivation and discipline to author the extra code in a tightly scheduled project. Clearly, our developers acknowledged the extra work needed to be done, but did not see the benefits of TDD. One reason for this may have been the fact that the developers did not have to live through the maintenance phase of the product, where new features would be added without breaking the existing solution. In addition, the developers perceived the quality issues as being of less importance in such a small project.

It would have been possible to put more pressure on the team with regard to the use of TDD *per se*. Yet, the project was under business delivery pressure and the end product was their primary concern, as is the case in industry. Moreover, we find that motivation to use and acceptance of a new technology should emerge from use and actual benefits. In our case, the team did not achieve an early victory with the process innovation, which hindered effectively further application opportunities.

5.2.2 Developers' Inexperience

The development team spent a considerable amount of time in solving technical issues related to the mobile development environment and programming solutions (i.e., use of architectural patterns, threads, etc.). Only one of the team members was an expert in mobile Java programming. Moreover, the application domain was filled with domain-specific details with regard to production planning system operations. Experience with these issues would most likely have eased the adoption of the test-driven mind set. TDD is also a personal-level development practice and, therefore, may be more difficult to adopt than other agile development techniques such as rapid release cycles, agile modeling, and constant communication. The learning curve appears to be steeper in the case of TDD than in the other agile practices. While the team used a so-called green field approach (i.e., they adopted many agile techniques at once), it may well be that the project's time-frame was too tight for the most difficult practices. A more effective strategy would have been to introduce fewer new techniques on a first-of-a-kind project and recommend TDD on the following projects, when the developers would be more experienced with the other new development techniques.

5.2.3 Immature Development Environment for TDD

The TDD method relies on using an extensive set of tests that are constantly executed during development. It must be possible to run the test suite automatically without too much effort. The tools for implementing TDD in the case project's

development environment were found to be immature. In addition, the development included significant user interface implementation, an area where the tools for executing TDD are only beginning to come into more general use.

5.2.4 Absence of a Mentor

A brief basic training of the concepts of TDD was provided prior to project launch. This turned out to be an overly optimistic approach for introducing a new technique in practical terms. TDD is not learned in a one-day course. We suspect that mastery of the technique requires several months of intense use. For a short project, such as the case study presented here, where developers were not familiar with the technique, a mentor within the project team is required. Constant advice and motivation from the mentor would have eased use, even in times when resistance occurred.

6 CONCLUSIONS

The mobile telecommunications industry has proved to be highly competitive, uncertain, and dynamic environment. Industries operating in such a turbulent market-place is particularly interested in trying out technical agile solutions. This study reported the results of a case study, where a development team attempted to use test-driven development in a mobile development environment with little success. Nevertheless, the project was highly successful in the business sense.

For business managers and others, this study bears important implications. In particular, this study points out that the adoption of a certain agile technique or approach is not a straightforward, silver-bullet solution. Business managers should stay alert in the midst of the hype before mandating the use of agile solutions in their organizations. Developers should keep their heads up as well. This case has demonstrated that very few if any of the technical agile solutions can be adopted and used without proper, systematic software process improvement tactics. While this study fails to provide empirical evidence either for or against test-driven development, it highlights the obstacles hindering adoption. We believe that the results of our study are applicable in other environments and agile techniques. Agile improvements at the technical level require as careful planning and follow-up as any other software engineering innovation. An interesting avenue for future research would be the use innovation theories, such as León's (1996) innovation adoption profiles, to analyze the adoption of agile solutions in practical settings. Concrete empirical evidence should still be collected, however.

This study maintains that business agility cannot be achieved without considering all organizational levels, including development teams and personnel. Software engineering research and practice has produced technical solutions, which have been the focus of this paper. Information systems research is likely to provide the needed extension to the organizational and interorganizational levels. Yet, even low-level agile changes are not easily implemented. We plan to continue the validation of agile solutions in future case studies.

REFERENCES

Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. *Agile Software Development Methods: Review and Analysis*, Espoo, Finland: Technical Research Centre of Finland, VTT Publications 478, 2002 (available online at http://www.vtt.fi/inf/pdf/publications/2002/P478.pdf).

Abrahamsson, P., Warsta, J., Siponen, M. T., and Ronkainen, J. "New Directions on Agile Methods: A Comparative Analysis," in *Proceedings of the 25th International Conference on Software Engineering*, Los Alamitos, CA: IEEE Computer Society Press, 2003, pp. 244-254.

Astels, D. *Test-Driven Development: A Practical Guide,* Upper Saddle River, NJ: Prentice Hall, 2003.

Avison, D., Lau, F., Myers, M., and Nielsen, P. A. "Action Research," *Communications of the ACM* (42:1), 1999, pp. 94-97.

Barriocanal, E. G., and Urban, M.-A. S. "An Experience in Integrating Automated Unit Testing Practices in an Introductory Programming Course," *ACM SIGCSE Bulletin* (34), 2002, pp. 125-128.

Basili, V. R. , and Lanubile, F. "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering* (25), 1999, pp. 456-473.

Beck, K. "Aim, Fire," *IEEE Software* (18:5), 2001, pp. 87-89.

Beck, K. "Embracing Change with Extreme Programming," *IEEE Computer* (32:10), 1999, pp. 70-77.

Beck, K. *Test-Driven Development: By Example*, New York: Addison-Wesley, 2003.

Boehm, B., and Turner, R. *Balancing Agility and Discipline: A Guide for the Perplexed,* Boston: Addison-Wesley, 2003.

Edwards, S. H. "Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action," in *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, New York: ACM Press, 2004, pp. 26-30.

Fowler, M. *Refactoring: Improving the Design of Existing Code*, Boston: Addison Wesley Longman, 1999.

George, B., and Williams, L. "An Initial Investigation of Test Driven Development in Industry," in *Proceedings of the ACM Symposium on Applied Computing*, New York: ACM Press, 2003, pp. 1135-1139.

George, B., and Williams, L. "A Structured Experiment of Test-Driven Development," *Information and Software Technology* (46:5), 2004, pp. 337-342.

Jeffries, R. E. "Extreme Testing," *Software Testing & Quality Engineering* (1:2), March/April 1999, pp. 23-26.

Kaufmann, R., and Janzen, D. "Implications of Test-Driven Development A Pilot Study," in *Proceedings of the Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, New York: ACM Press, 2003, pp. 298-299.

Lal, D., Pitt, D. C., and Beloucif, A. "Restructuring in European Telecommunications: Modeling the Evolving Market," *European Business Review* (13:3), 2001, pp. 152-156.

Langr, J. "Evolution of Test and Code via Test-First Design," paper presented at the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Tampa Bay, FL, 2001.

Larman, C., and Basili, V. R. "Iterative and Incremental Development: A Brief History," *IEEE Software* (20), 2003, pp. 47-56.

León, G. "On the Diffusion of Software Technologies: Technological Frameworks and Adoption Profiles," in *Diffusion and Adoption of Information Technology*, K. Kautz and J. Pries-Heje (Eds.), Padstow, Cornwall, England: TJ Press Ltd., 1996, pp. 96-116.

Maximilien, E. M., and Williams, L. "Assessing Test-Driven Development at IBM," in *Proceedings of the International Conference on Software Engineering (ICSE)*, Los Alamitos, CA: IEEE Computer Society Press, 2003, pp. 564-569.

Müller, M. M., and Hagner, O. "Experiment About Test-First Programming," *IEEE Proceedings Software* (149:5), 2002, pp. 131-136..

Pancur, M., Ciglaric, M., Trampus, M., and Vidmar, T. "Towards Empirical Evaluation of Test-Driven Development in a University Environment," in *Proceedings of EUROCON 2004*, Ljubljana, Slovenia, IEEE Computer Society, 2003, pp. 83-86.

Rasmusson, J. "Introducing XP into Greenfield Projects: Lessons Learned." *IEEE Software* (20:3), 2003, pp. 21-28.

Salo, O., and Abrahamsson, P. "Empirical Evaluation of Agile Software Development: A Controlled Case Study Approach," in *Proceedings of the 6th International Conference on Product Focused Software Process Improvement*, F. Bomarius and H. Ilda (Eds.), Kansai Science City, Japan: Springer, 2004, pp. 408-423.

Williams, L., Maximilien, E. M., and Vouk, M. "Test-Driven Development as a Defect-Reduction Practice," in *Proceedings of the 14th International Symposium of Software Reliability Engineering (ISSRE'03)*, Los Alamitos, CA: IEEE Computer Society Press, 2003, pp. 34-48.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. *Experimentation in Software Engineering*, Boston: Kluwer Academic Publishers, 2000.

Yin, R. K. *Case Study Research Design and Methods*, Thousand Oaks, CA: Sage Publications, 1994.

Ynchausti, R. A. "Integrating Unit Testing into a Software Development Team's Process," in *Proceedings of the XP 2001 Conference*, Caglieri, Italy, 2001, pp. 79-83.

ABOUT THE AUTHORS

Pekka Abrahamsson is a senior research scientist at VTT Technical Research Centre of Finland. He received his Ph.D. from University of Oulu, Finland, in 2002. His current responsibilities include managing the AGILE-ITEA project (http://www.agile-itea.org), which involves 22 organizations from 9 European countries. The project aims at utilizing agile innovations in the development of embedded systems. His research interests are currently focused on the development of mobile information systems, applications and services, business agility and agile software production. He has coached several agile software development projects in industry and authored more than 40 scientific publications focusing on software process and quality improvement, commitment issues, and agile software development. He is the principal author of the Mobile-D methodology for mobile application development. Pekka can be reached at Pekka.Abrahamsson@vtt.fi.

Antti Hanhineva is a software designer at Elbit Oy in Finland. He received his M.Sc. from University of Oulu, Finland, in 2004. Prior to joining Elbit, he worked at VTT Technical Research Centre of Finland. While at VTT he coached several projects on test-driven development and testing related issues in mobile development environments. He is a coauthor of the Mobile-D methodology for mobile application development. Antti can be reached at antti.hanhineva@elbit.fi.

Juho Jaalinoja is a software engineer at Nokia Technology Platforms. Prior to joining Nokia, he worked as a research scientist at VTT Technical Research Centre of Finland. His research areas include software process improvement and agile methods. He received his M.Sc. in Information Processing Science from University of Oulu, Finland, in 2004. Juho can be reached at Juho.Jaalinoja@nokia.com.