Empir Software Eng (2008) 13:289–302
DOI 10.1007/s10664-008-9062-z

Realizing quality improvement through test driven development: results and experiences of four industrial teams

Nachiappan Nagappan · E. Michael Maximilien · Thirumalesh Bhat · Laurie Williams

Published online: 27 February 2008

Editor: Pankaj Jalote

Abstract Test-driven development (TDD) is a software development practice that has been used sporadically for decades. With this practice, a software engineer cycles minute-by-minute between writing failing unit tests and writing implementation code to pass those tests. Test-driven development has recently re-emerged as a critical enabling practice of agile software development methodologies. However, little empirical evidence supports or refutes the utility of this practice in an industrial context. Case studies were conducted with three development teams at Microsoft and one at IBM that have adopted TDD. The results of the case studies indicate that the pre-release defect density of the four products decreased between 40% and 90% relative to similar projects that did not use the TDD practice. Subjectively, the teams experienced a 15–35% increase in initial development time after adopting TDD.

Keywords Test driven development · Empirical study · Defects/faults · Development time

1 Introduction

Test-driven development (TDD) (Beck 2003) is an "opportunistic" (Curtis 1989) software development practice that has been used sporadically for decades (Larman and Basili 2003;

N. Nagappan (✉) · T. Bhat
Microsoft Research, Redmond, WA, USA
e-mail: nachin@microsoft.com

T. Bhat
e-mail: thirub@microsoft.com

E. M. Maximilien
IBM Almaden Research Center, San Jose, CA, USA
e-mail: maxim@us.ibm.com

L. Williams
North Carolina State University, Raleigh, NC, USA
e-mail: williams@csc.ncsu.edu

Gelperin and Hetzel 1987). With this practice, a software engineer cycles minute-by-minute between writing failing unit tests and writing implementation code to pass those tests. Test-driven development has recently re-emerged as a critical enabling practice of agile software development methodologies (Cockburn 2001), in particular Extreme Programming (XP; Beck 2005). However, little empirical evidence supports or refutes the utility of this practice in an industrial context.

In this paper we report on "in vivo" case studies of four software development teams that have adopted the TDD or TDD-inspired practice, one team from IBM (Williams et al. 2003) and three from Microsoft (Bhat and Nagappan 2006). We study TDD as a stand alone software development practice; in that, we do investigate TDD within the prevailing development processes at IBM and Microsoft and not within the context of XP. In this paper, we share details about the projects, the teams, and how they used TDD in their projects. We compare the quality and development time of the product developed by a team using TDD relative to a similar team that did not use TDD. Finally, we provide lessons learned by these four teams.

Case studies can be viewed as "research in the typical," (Kitchenham et al. 1995; Fenton and Pfleeger 1998) increasing the degree to which the results can be compared with those of other industrial teams. However, case studies cannot be performed with the rigor of experiments because they involve real people, real projects, and real customers over a relatively long period of time. In empirical research, replication of studies is a crucial factor to building an empirical body of knowledge about a practice or a process. The four studies presented in this paper contribute towards strengthening of the empirical knowledge of the efficacy of TDD because each case study was conducted in a different context, as shown Fig. 1. The details of the contexts of each case study are discussed throughout the paper.

The rest of this paper is organized as follows. Section 2 gives an overview of TDD and Section 3 the related works. Section 4 explains the context variables of our projects, and Section 5 their individual TDD implementation practices. Section 6 presents the results of the study and Section 7 the threats to validity. Section 8 presents the lessons learned for the benefits of organizations new to TDD or thinking about adopting the practice.

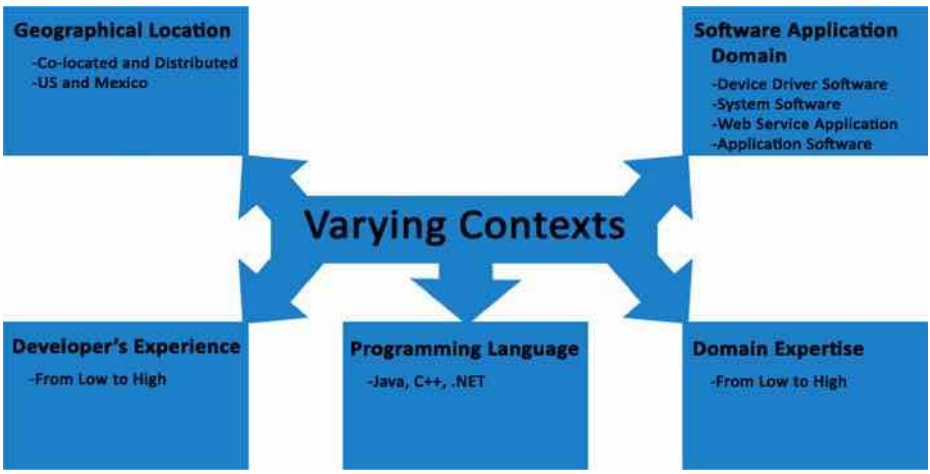

Fig. 1 Varying context factors of the four case studies

2 Test Driven Development

When discussing TDD we consider a task to be a subset of a requirement that can be implemented in a few days or less (Beck and Fowler 2001). As illustrated in Fig. 2, TDD software engineers develop production code through rapid iterations of the following: Minute-by-minute cycles:

- Writing a small number of new and failing automated unit test case(s) for the task at hand
- Implementing code which should allow the new unit tests cases to pass
- Re-running the new unit test cases to ensure they now pass with the new code

Per-task cycles:

- Integrate code and tests for new code into existing code base
- Re-running all the test cases in the code base to ensure the new code does not break any previously running test cases
- Refactoring the implementation or test code (as necessary)
- Re-running all tests in the code base to ensure that the refactored code does not break any previously passing test cases

In TDD, code development is kept within the developer's intellectual control because he or she is continuously making small design and implementation decisions and increasing functionality at a relatively consistent rate. New functionality is not considered properly implemented unless the new unit test cases and every other unit test cases written for the code base run properly. Some possible benefits of TDD are:

- Better Design. TDD is considered as much of (and in some cases, more of) a design process than a test process. Kent Beck, the person responsible for the inclusion of TDD in XP and Ward Cunningham, an early proponent of TDD in an XP context, immediately stated, "Test-first coding is not a testing technique" (Beck 2001; Note: the test-first coding practice eventually came to be known as TDD). They assert that TDD

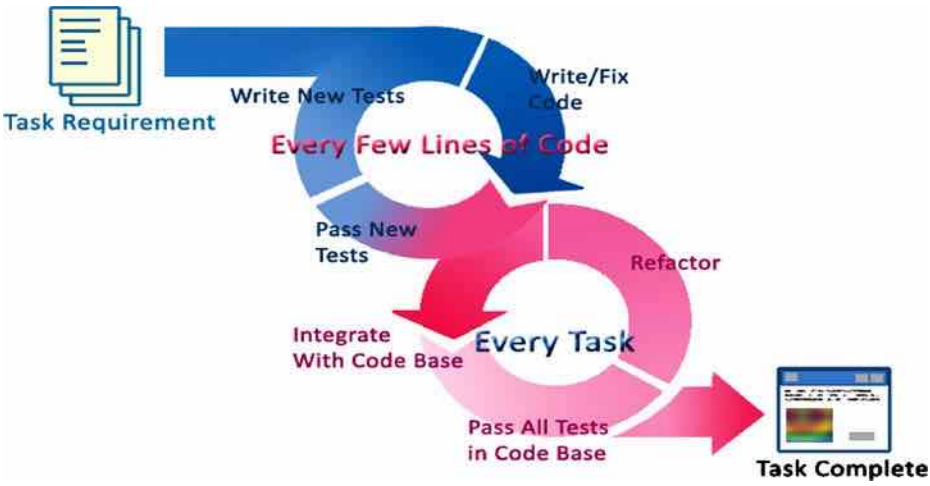

Fig. 2 TDD methodology overview

is an analysis and design technique and that: "Test-first code tends to be more cohesive and less coupled than code in which testing isn't part of the intimate coding cycle" (Beck 2001).

- Efficiency. The fine granularity of test-then-code cycle gives continuous feedback to the developer. With TDD, faults and/or defects are identified very early and quickly as new code is added to the system. Consequently, the source of the problem is also more easily determined since the problems have not been in existence for long. We contend that the efficiency of fault and defect removal and the corresponding reduction in the debugging and maintenance time compensates for the additional time spent writing and executing test cases.
- Test Assets. The automated unit test cases written with TDD are valuable assets to the project. Subsequently, when the code is enhanced, modified, or updated, running the automated unit tests may be used for the identification of newly introduced defects, i.e., for regression testing.
- Reducing Defect Injection. Unfortunately, maintenance fixes and "small" code changes may be nearly 40 times more error prone than new development (Humphrey 1989), and often, new faults are injected during the debugging and maintenance phases. The ease of running the automated test cases after changes are made should also enable smooth integration of new functionality into the code base and therefore reduce the likelihood that fixes and maintenance changes introduce new defects. The TDD test cases are essentially a high-granularity, low-level, regression test. By continuously running these automated test cases, one can find out whether a change breaks the existing system early on, rather than leading to a late discovery.

3 Related Works

We briefly summarize the relevant empirical studies on TDD and their high level results. George and Williams (George and Williams 2003a, b) performed a structured experiment involving 24 professional programmers from John Deere, Rolemodel Software, and Ericsson, to investigate the efficacy of TDD. One group developed a small Java program using TDD while the other control group used a waterfall-like approach. Experimental results indicated that TDD programmers produce higher quality code because they passed 18% more functional black-box test cases. On the other hand, the TDD programmers took 16% more time. However, the programmers in the control group often did not write the required automated test cases after completing their code.

Müller and Hagner (2002) found that a TDD-variant did not help produce a higher quality system. In this study, the programmers (graduate computer science students) had to write their complete set of automated unit test cases before writing any production code rather than in the highly iterative manner of TDD. Erdogmus et al. (2005) performed a controlled investigation regarding test-first and test-last programming using 24 undergraduate computer science students. They observed that TDD improved programmer productivity but did not, on average, help the engineers to achieve a higher quality product. Their conclusions also brought out a valid point that the effectiveness of the test-first technique depends on the ability to encourage programmers to enhance their code with test assets. Müller and Tichy (2001) investigated XP in a university context using 11 students. From a testing perspective they observed that, in the final review of the course, 87% of the students stated that the execution of the test cases strengthened their confidence in their code.

Janzen and Seiedian (2006) conducted an experiment with undergraduate students in a software engineering course. Students in three groups completed semester-long programming projects using either an iterative test-first (TDD), iterative test-last, or linear test-last approach. Results from this study indicate that TDD can be an effective software design approach improving both code-centric aspects such as object decomposition, test coverage, and external quality, as well as developer-centric aspects, which includes productivity and confidence.

Our work builds up on the prior empirical work in this area. Few case studies examine the use of TDD in an industrial context. Our paper contributes to the empirical base of knowledge about the TDD practice by combining results of four empirical studies carried out in different industrial contexts and environments.

4 The Projects and the Teams

In this section, we provide information about the four products and teams and the contexts into which the projects were executed. As a cautionary note, as with all empirical research, the results should not be taken to be valid in the general sense, only in the context in which the case study was conducted.

4.1 IBM

The IBM case study was conducted with a group that has been developing device drivers for over a decade. They have one legacy product that has undergone seven releases since late 1998. This legacy product was used as the baseline in our case study. In 2002, the group developed the device drivers on a new platform and moved to a completely new architecture. In our case study, we compare the seventh release of the legacy platform with the first release of the new platform. Because of its longevity, the legacy system handles more classes of devices, on multiple bus connectivity, with more vendors' devices than the new system. Hence, while not a true control group, the legacy software can still provide a valuable relative insight into the performance of the TDD methodology since the two products exposed the same interfaces for the same set of devices—a JavaPOS[1] interface. The device drivers from the legacy code base were wrapped using JNI to expose the Java™ interface, and in the new system, the device drivers were mostly entirely implemented in the Java technology.

4.2 Microsoft

The three case studies at Microsoft were distributed across three of the main product families, Windows, MSN, and Visual Studio (Bhat and Nagappan 2006). The TDD and non-TDD projects are developed by teams led by project managers with similar levels of responsibilities and reporting to the same higher-level manager. As a result, our comparisons involve projects in a similar usage domain. We therefore avoid making unequal comparisons, as would be the case in comparing, for example, computer game and nuclear reactor software. Also, these projects were analyzed post hoc, and the developers did not know during development that their work was going to be assessed. Therefore, the case study did not interfere with the developers' performance. There was no enforcement

[1] http://www.javapos.org

and monitoring of the TDD practice; and decisions to use TDD were made as a group. More detail is provided on the projects completed by each of the three teams:

- Windows: This case study was performed in the Networking team. The networking common library was written by the tool development team as a re-usable set of modules for different projects within the networking tools team. Some examples are generic packet manipulation routines, command line manipulation, memory allocation and tracking routines, random number generator, and timer classes. This library is used by more than 50 internal applications.
- MSN: This case study involves a Web services application in the MSN division. MSN is a business division of Microsoft that works on Web service applications that range from online shopping to news and Web search technologies.
- Developer Division: This project is part of the Visual Studio (VS) software development system. Visual Studio provides a range of tools for individual developers and software development teams. Visual Studio allows for the development of stand-alone applications, Web service applications, and so on.

4.3 Examining the Range of Teams and Products

Tables 1 and 2 respectively show a summary of a comparison between the team and the product measures. The team factors enable us to understand the composition of these four teams. The product factors present a comparative view of the size and effort associated with the projects.

Across the four projects, we observe that the teams are experienced; the majority of the team members had around 6–10 years experience. In the MSN team, the developers are experienced but only have moderate domain and programming language expertise. The IBM project has four developers who were inexperienced. Further, the IBM project was developed in a distributed context with five developers in Raleigh, NC and four in Guadalajara, Mexico. No one on the new IBM team knew TDD beforehand, and three were somewhat unfamiliar with Java. In the IBM case, all but two of the nine full-time developers were novices to the targeted devices.

Table 1 Summary of team factors

Metric description		IBM: Drivers	Microsoft: Windows	Microsoft: MSN	Microsoft: VS
Team size		9	6	5–8	7
Team location		USA and Mexico	Redmond, WA	Redmond, WA	Redmond, WA
Experience level	>10 years	0	0	1	5
	6–10 years	3	5	7	0
	<5 years	6	1	0	2
Domain expertise (low, medium, high)		Medium to low	High	Medium	High
Language expertise (low, medium, high)		Medium to low	High	Medium	High
Programming language		Java	C/C++	C++/C#	C#
Program Manager's expertise (low, medium, high)		High	High	Medium	Medium
Team location		Distributed	Collocated	Collocated	Collocated

Table 2 Summary of product factors

Metric description	IBM: Drivers	Microsoft: Windows	Microsoft: MSN	Microsoft: VS
Source KLOC	41.0	6.0	26.0	155.2
Test KLOC	28.5	4.0	23.2	60.3
Test KLOC/source KLOC	0.70	0.66	0.89	0.39
Percent block coverage from unit tests (%)	95	79	88	62
Development time (in man-months)	119	24	46	20
Comparable project—metric description				
Source KLOC	56.6	4.5	149.0	222.0
Development time (in man-months)	45	12	144	42
Team size	5	2	12	5

From the data in Table 2, we observe that the projects are of different sizes ranging from six thousand lines of code (KLOC) to 155 KLOC and in development time from 24 to 119 person man-months. The test-to-source line of code (LOC) ratio and code coverage varies, reflecting the environmental and usage requirements that play an important part in defining these metrics. These projects were non-legacy applications, i.e., written completely or mostly from scratch.

5 TDD Implementations

In this section, we provide more detail about the TDD practice used by each team.

5.1 IBM

The unit testing approach of the legacy group can be classified as largely ad-hoc. The developers would code a prototype of the utility classes (classes which collaborate with other classes) and of classes that are expected to be reused; and then create a design via UML class and sequence diagrams (Fowler 2000). This design was then followed by an implementation stage that sometimes caused design changes, and thus some iteration between the design and the coding phases. Unit testing followed as a post-coding activity. In all cases, the unit test process was not formal and was not disciplined. More often than not, there were resource and schedule limitations that constrained the number of test cases developed and run. Most of the unit tests developed were also not reused during the subsequent Functional Verification Test (FVT) phase, when a defect was found, or when a new release of the software was developed.

With the TDD group, test cases were developed mostly up front as a means of reducing ambiguity and to validate the requirements, which for this team was a full detail standard specification. UML class and sequence diagrams were used to develop an initial design. This design activity was interspersed with the up-front unit test creations for developed classes. Complete unit testing was enforced—primarily via reminders and encouragements. We define *complete testing* as ensuring that the public interfaces and semantics of each method (the behavior of the method as defined by the specification) were tested utilizing the JUnit[2] unit-testing framework. For each public class, there was an associated public test class; for each public method in the class there was an associated public test method in the

[2] http://junit.org

corresponding unit test class. The target goal was to cover at least 80% of the developed classes by automated unit testing.

Since TDD was geographically distributed in two different locations, to guarantee that all unit tests would be run by all members of the team, an automated build and test systems was set up in both geographical locations. Daily, the build systems extracted all the code from the library build the code and ran all the unit tests. The Apache ANT[3] build tool was used. After each automated build and test run cycle, an email was sent to all members of the teams listing all the tests that successfully ran as well as any errors found. This automated build and test served as a daily integration and validation heartbeat for the team.

With both the legacy and new projects, when the majority of the device driver code was implemented and passed all their own unit tests and those in the code base also passed, the device drivers were sent to FVT. The external FVT team (different from both development legacy and "new project" teams) had written black box test cases based on the functional system specification. More than half of the FVT tests were automated in part; the remaining tests were split fairly evenly between fully automated and fully manual. Both the legacy and new teams faced the same degree of schedule pressure.

5.2 Microsoft

The TDD team at Microsoft did most of their development using a hybrid version of TDD. By hybrid we mean that these projects as with almost all projects at Microsoft had detailed requirements documents written. These detailed requirements documents drove the test and development effort. There were also design meetings and review sessions. This explains our reason to call this a hybrid-TDD approach, as agile teams typically do not have design review meetings. The three teams studied did not use agile software development practices except for TDD. The legacy teams did not use any agile methodologies nor the TDD practice.

The development methodology at Microsoft is a hybrid of the waterfall approach (Royce 1970). Our meaning of hybrid denoted not purely waterfall but with one with variations. First, the requirements are gathered from various sources, e.g., customers, partners, stakeholders, and market study results. Program managers come up with an extensive list of features and write the requirements. Based on iterations with the development teams, the final set of requirements is written. This serves as the functional requirements checklist for the features to be implemented. From here on, teams have design discussions and design reviews where preliminary design is completed and implementation (with integrated unit, system, stress, and functional testing) starts. Then the requirements are fully implemented and developed; this phase is otherwise known as "code-complete." During the development and testing process a variety of tools are used in addition to the TDD practices, such as static analysis tools (Larus et al. 2004), dependency analysis tools (Srivastava et al. 2005), as well as fault-injection tools.

From the context of TDD, the unit test framework used was run from the command line. The unit test framework enables the developer to figure out the status of the test run from the command line. The log files obtained from the runs are used for deeper analysis and highlight passing and failing tests (Bhat and Nagappan 2006). Further, some teams also implemented a mechanism that would identify when a unit test failed and sent an automated email to the developers whose code or test failed. This is done to help ensure timely fixing of the code.

The teams checked in source and unit test code into an integrated version control system through which the system is built. The three projects at Microsoft had different build

[3] http://www.apache.org/jakarta/ant

processes as they had different execution and usage requirements. The teams did not have any particular target coverage measures. Functionality was the main driver for the resulting code coverage. The completion was determined by satisfying the requirements stated in the initially developed requirements document. The teams recognized the different coverage requirement needs for system level code, core architectural components, GUIs, and so on.

6 Quality and Productivity Results

We measure the quality of the software products in terms of defect density computed as defects/thousand lines of code (KLOC).

> "When software is being developed, a person makes an error that results in a physical fault (or defect) in a software element. When this element is executed, traversal of the fault or defect may put the element (or system) into an erroneous state. When this erroneous state results in an externally visible anomaly, we say that a failure has occurred" (IEEE 1988).

We use the above-stated definition of defect as an error made by a developer that results in a physical fault in the system and normalize it per KLOC in order to derive our quality measure named defect density.

We now discuss our outcome measures. We compare the TDD and non-TDD project for a change in defect density, as shown in Table 3. We use defect density as an approximate overall indicator of the quality of the code from the point of integration stability. Defect information is mined from IBM's and Microsoft's bug databases. The defects measured for Microsoft include all defects reported post-integration, such as design defects, code review defects, test defects, integration defects, stress-test defects, customer defects, security defects, static analysis tool defects, and performance defects, as well as customer-reported problems. The IBM defects cover the same categories but are flagged to indicate if the defect was identified from an internal user (e.g., tester) or from external users at a customer location. The latter tend to have higher priorities since they directly impact overall customer satisfaction.

All the teams demonstrated a significant drop in defect density: 40% for the IBM team; 60–90% for the Microsoft teams. One would expect the legacy IBM product to have exhibit a lower defect because it has a substantial portion of legacy code that has already been tested and exposed to the field in prior releases. For the new IBM project developed using TDD, most of the code written is new and has not been exposed to the field. Part of the intuition behind the difference can be attributed to the fact that with the legacy system the additional code, while being added to a stable production code base, would cause cascading defects that were hard to identify early on. As opposed to, the iterative nature of TDD that helps identify and control the injection of faults.

Table 3 Outcome measures

Metric description	IBM: Drivers	Microsoft: Windows	Microsoft: MSN	Microsoft: VS
Defect density of comparable team in organization but not using TDD	W	X	Y	Z
Defect density of team using TDD	0.61W	0.38X	0.24Y	0.09Z
Increase in time taken to code the feature because of TDD (%) [Management estimates]	15–20%	25–35%	15%	25–20%

Another interesting observation from the outcome measures in Table 3 is the increase in time to develop the features attributed to the usage of the TDD practice, as subjectively estimated by management. The increase in development time ranges from 15% to 35%. From an efficacy perspective this increase in development time is offset by the by the reduced maintenance costs due to the improvement in quality (Erdogmus and Williams 2003), an observation that was backed up the product teams at Microsoft and IBM.

7 Threats to Validity

In this section we list threats that potentially would invalidate our results and findings, in order to limit the scope our claims as well as explain their utility. Developers using TDD might have been more motivated to produce higher quality code as they were trying out a new process and might have been motivated to prove that TDD worked efficiently for them. To an extent, this concern is alleviated because these were professional programmers all of whom were more concerned with completing their tasks than with the quality improvement results of a new development practice. Furthermore, the TDD and non-TDD teams did not know that they would be part of a study anytime during their development cycle.

The projects developed using TDD might have been easier to develop, as there can never be an accurate equal comparison between two projects except in a controlled case study. In our case studies, we alleviated this concern to some degree by the fact that these systems were compared within the same organization (with the same higher-level manager and sub-culture). Therefore, the complexity of the TDD and non-TDD projects are comparable. Another related issue is the comparison of new projects (done with TDD) with an enhancement to a legacy systems (not done with TDD). The legacy systems might have had existing problems that could cause legacy systems to have higher defect density than the new TDD systems and/or the legacy system could have a lower defect density because the code has been tested and exposed to the field in prior releases.

As previously mentioned, case studies are used to collect data through observation of a project in an unmodified, contextual setting (Zelkowitz and Wallace 1998). Case studies (or $N=1$ experiments [Harrison 1997]) can be viewed as "research in the typical" (Kitchenham et al. 1995; Fenton and Pfleeger 1998) and are an evaluative technique carried out with realistic tasks and realistic subjects in a realistic environment (Fenton et al. 1994; Sjøberg et al. 2002). However, comparisons made via case studies can never be perfect due to the complex contexts of both the compared projects. Additionally, a family of case studies is likely not to yield statistically significant results, though Harrison (2004) observes that statistical significance has not been shown to impact industrial practice.

Drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted (Basili et al. 1999). Researchers become more confident in a theory when similar findings emerge in different contexts (Basili et al. 1999). Towards this end, we intend that our case study will help contribute towards strengthening the existing empirical body of knowledge in this field (Müller and Tichy 2001; Müller and Hagner 2002; George and Williams 2003a, b; Williams et al. 2003; Erdogmus et al. 2005) by replicating across different context variables and environments. Several teams at Microsoft and at IBM employ TDD as part of the development process. We intend to collect future data from all these projects and continue build an empirical body of knowledge in this field.

8 Conclusions and Discussion

Our experiences and distilled lessons learned, all point to the fact that TDD seems to be applicable in various domains and can significantly reduce the defect density of developed software without significant productivity reduction of the development team. Additionally, since an important aspect of TDD is the creation of test assets—unit, functional, and integration tests. Future releases of these products, as they continue using TDD, will also experience low defect densities due to the use of these test assets. Specifically, recent contact with the IBM team indicated that in one of the subsequent releases (more than five releases since the case study) some members of the team (grown 50% since the first release) have taken some shortcuts by not running the unit tests, and consequently the defect density increased temporally compared to previous releases. As part of our future studies of TDD and non-TDD teams we plan on incorporating effort measurement in-process, collecting design effort (along with design measures), testing effort, and refactoring effort. Another interesting future work plan is to investigate TDD in legacy systems. Most research to date has focused on new systems. We plan on evaluating the incremental effort required to do TDD in legacy systems working with pre-existing dependencies.

Additionally, we have gained experiences and lessons with transitioning to TDD that we wish to share with other teams. Some of the practices can work well without TDD too and can prove to be beneficial to the teams. These were not part of the empirical evaluation study and are qualitative recommendations based on our learning experiences.

- *Start TDD from the beginning of projects*. Do not stop in the middle and claim it doesn't work. Do not start TDD late in the project cycle when the design has already been decided and majority of the code has been written. TDD is best done incrementally and continuously.
- *For a team new to TDD, introduce automated build test integration towards the second third of the development phase—not too early but not too late*. If this is a "Greenfield" project, adding the automated build test towards the second third of the development schedule allows the team to adjust to and become familiar with TDD. Prior to the automated build test integration, each developer should run all the test cases on their own machine.
- *Convince the development team to add new tests every time a problem is found*, no matter when the problem is found. By doing so, the unit test suites improve during the development and test phases.
- *Get the test team involved and knowledgeable about the TDD approach*. The test team should not accept new development release if the unit tests are failing.
- *Hold a thorough review of an initial unit test plan*, setting an ambitious goal of having the highest possible (agreed upon) code coverage targets.
- *Constantly running the unit tests cases in a daily automatic build (or continuous integration)*; tests run should become the heartbeat of the system as well as a means to track progress of the development. This also gives a level of confidence to the team when new features are added.
- *Encourage fast unit test execution and efficient unit test design*. Test execution speed is very important since when all the tests are integrated, the complete execution can become quite long for a reasonably-sized project and when using constant test executions. Tests results are important early and often; they provide feedback on the current state of the system. Further, the faster the execution of the tests the more likely developers themselves will run the tests without waiting for the automated build tests

results. Such constant execution of tests by developers may also result in faster unit tests additions and fixes.
- *Share unit tests*. Developers' sharing their unit tests, as an essential practice of TDD, helps identify integration issues early on.
- *Track the project using measurements*. Count the number of test cases, code coverage, bugs found and fixed, source code count, test code count, and trend across time, to identify problems and to determine if TDD is working for you.
- *Check morale of the team at the beginning and end of the project*. Conduct periodical and informal surveys to gauge developers' opinions on the TDD process and on their willingness to apply it in the future.

Acknowledgements At IBM we would like to thank the Raleigh and Guadalajara development teams; in particular, Julio Sanchez of Guadalajara and Dale Heeks from the FVT team. At Microsoft we would like to thank the Windows, MSN, and DevDiv teams that participated in this study without whom this work would not have been possible and the agile development community at Microsoft for valuable feedback on earlier work. This work done by Dr. Williams was supported by the National Science Foundation under CAREER Grant Nos. 0346903.

References

IEEE (1988) IEEE Std 982.2-1988 IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software. IEEE Computer Society, Washington, DC

Basili VR, Shull F et al (1999) Building knowledge through families of experiments. IEEE Trans Softw Eng 25(4):456–473

Beck K (2001) Aim, fire. IEEE Softw 18:87–89

Beck K (2003) Test driven development—by example. Addison-Wesley, Boston

Beck K (2005) Extreme programming explained: embrace change. Addison-Wesley, Reading, MA

Beck K, Fowler M (2001) Planning extreme programming. Addison-Wesley, Reading, MA

Bhat T, Nagappan N (2006) Evaluating the efficacy of test-driven development: industrial case studies. International Symposium on Empirical Software Engineering, Rio de Janeiro

Cockburn A (2001) Agile software development. Addison-Wesley Longman, Reading, MA

Curtis B (1989) Three problems overcome with behavioral models of the software development process (panel). In: Proceedings of the International Conference on Software Engineering, Pittsburgh, PA. ACM, Pittsburgh, PA

Erdogmus H, Williams L (2003) The economics of software development by pair programmers. Eng Econ 48 (4):283–319

Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Trans Softw Eng 31(3):226–237

Fenton NE, Pfleeger SL (1998) Software metrics: a rigorous and practical approach. Cole Brooks, Pacific Grove, CA

Fenton N, Pfleeger SL et al (1994) Science and substance: a challenge to software engineers. IEEE Softw 11 (4):86–95

Fowler M (2000) UML Distilled. Addison Wesley, Reading, MA

Gelperin D, Hetzel W (1987) Software quality engineering. In: Proceedings of the Fourth International Conference on Software Testing, Washington, DC, June

George B, Williams L (2003a) An initial investigation of test-driven development in industry. In: Proceedings of the ACM Symposium on Applied Computing, Melbourne, FL

George B, Williams L (2003b) A structured experiment of test-driven development. Inf Softw Technol (IST) 46(5):337–342

Harrison W (1997) N=1: an alternative for empirical software engineering research? Empir Software Eng 2 (1):7–10

Harrison W (2004) Propaganda and software development. IEEE Softw 21(5):5–7

Humphrey WS (1989) Managing the software process. Addison-Wesley, Reading, MA

Janzen D, Saiedian H (2006) On the influence of test-driven development on software design. In: Proceedings of the Conference on Software Engineering Education and Training, Turtle Bay, HI

Kitchenham B, Pickard L et al (1995) Case studies for method and tool evaluation. IEEE Softw 12(4):52–62
Larman C, Basili V (2003) A history of iterative and incremental development. IEEE Comput 36(6):47–56
Larus J, Ball T, Das M, Deline R, Fahndrich M, Pincus J, Rajamani S, Venkatapathy T (2004) Righting software. IEEE Softw 21(3):92–100
Müller MM, Hagner O (2002) Experiment about test-first programming. IEEE Proc Softw 149(5):131–136
Müller MM, Tichy WF (2001) Case study: extreme programming in a university environment. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE2001)
Royce WW (1970) Managing the development of large software systems: concepts and techniques. IEEE WESTCON, Los Angeles, CA
Sjøberg D, Anda B et al. (2002) Conducting realistic experiments in software engineering. In: Proceedings of the International Symposium on Empirical Software Engineering, Nara, Japan. IEEE Computer Society, Washington, DC
Srivastava A, Thiagarajan J, Schertz C (2005) Efficient integration testing using dependency analysis. Technical report: MSR-TR-2005-94. Microsoft Research, Redmond, WA
Williams L, Maximilien EM et al. (2003) Test-driven development as a defect-reduction practice. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering, Denver, CO. IEEE Computer Society, Washington, DC
Zelkowitz MV, Wallace DR (1998) Experimental models for validating technology. Computer 31(5):23–31

Nachiappan Nagappan is a researcher in the Software Reliability Research group at Microsoft Research. He received his MS and PhD from North Carolina State University in 2002 and 2005, respectively. His research interests are in software reliability, software measurement and empirical software engineering.

Dr. E. Michael Maximilien (aka "max") is a research staff member at IBM's Almaden Research Center in San Jose, California. Prior to joining ARC, he spent ten years at IBM's Research Triangle Park, N.C., in

software development and architecture. He led various small- to medium-sized teams, designing and developing enterprise and embedded Java™ software; he is a founding member and contributor to three worldwide Java and UML industry standards. His primary research interests lie in distributed systems and software engineering, especially Web services and APIs, mashups, Web 2.0, SOA (service-oriented architecture), and Agile methods and practices. He can be reached through his Web site (maximilien.org) and blog (blog.maximilien.com).

Thirumalesh Bhat is a Development Manager at Microsoft Corporation. He has worked on several versions of Windows and other commercial software systems at Microsoft. He is interested in software reliability, testing, metrics and software processes.

Laurie Williams is an associate professor of computer science at North Carolina State University. She teaches software engineering and software reliability and testing. Prior to joining NCSU, she worked at IBM for nine years, including several years as a manager of a software testing department and as a project manager for a large software project. She was one of the founders of the XP Universe conference in 2001, the first US-based conference on agile software development. She is also the lead author of the *Pair Programming Illuminated* book and a co-editor of *the Extreme Programming Perspectives* book.