On the Effectiveness of the Test-First Approach to Programming

Hakan Erdogmus, Maurizio Morisio, *Member*, *IEEE Computer Society*, and Marco Torchiano, *Member*, *IEEE Computer Society*

Abstract—Test-Driven Development (TDD) is based on formalizing a piece of functionality as a test, implementing the functionality such that the test passes, and iterating the process. This paper describes a controlled experiment for evaluating an important aspect of TDD: In TDD, programmers write functional tests before the corresponding implementation code. The experiment was conducted with undergraduate students. While the experiment group applied a test-first strategy, the control group applied a more conventional development technique, writing tests after the implementation. Both groups followed an incremental process, adding new features one at a time and regression testing them. We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive. We also observed that the minimum quality increased linearly with the number of programmer tests, independent of the development strategy employed.

Index Terms—General programming techniques, coding tools and techniques, testing and debugging, testing strategies, productivity, Software Quality/SQA, software engineering process, programming paradigms.

1 INTRODUCTION

Test-driven Development [1], [2], or TDD, is an approach to code development popularized by Extreme Programming [3], [4]. The key aspect of TDD is that programmers write low-level functional tests before production code. This dynamic, referred to as *Test-First*, is the focus of this paper. The technique is usually supported by a unit testing tool, such as JUnit [5], [6].

Programmer tests in TDD, much like unit tests, tend to be low-level. Sometimes they can be higher level and cross-cutting, but do not in general address testing at integration and system levels. TDD relies on no specific or formal criterion to select the test cases. The tests are added gradually during implementation. The testing strategy employed differs from the situation where tests for a module are written by a separate quality assurance team before the module is available. In TDD, tests are written one at a time and by the same person developing the module. (Cleanroom [7], for example, also leverages tests that are specified before implementation, but in a very different way. Tests are specified to mirror a statistical user input distribution as part of a formal reliability model, but this activity is completely separated from the implementation phase.)

TDD can be considered from several points of view:

- Feedback: Tests provide the programmer with instant feedback as to whether new functionality has been implemented as intended and whether it interferes with old functionality.
- Task-orientation: Tests drive the coding activity, encouraging the programmer to decompose the problem into manageable, formalized programming tasks, helping to maintain focus and providing steady, measurable progress.
- Quality assurance: Having up-to-date tests in place ensures a certain level of quality, maintained by frequently running the tests.
- Low-level design: Tests provide the context in which low-level design decisions are made, such as which classes and methods to create, how they will be named, what interfaces they will possess, and how they will be used.

Although advocates claim that TDD enhances both product quality and programmer productivity, skeptics maintain that such an approach to programming would be both counterproductive and hard to learn. It is easier to imagine how interleaving coding with testing could increase quality, but harder to imagine a similar positive effect on productivity. Many practitioners consider tests as overhead and testing as exclusively the job of a separate quality assurance team. But, what if the benefits of programmer tests outweigh their cost? If the technique indeed has merit, a reexamination of the traditional coding techniques could be warranted.

This paper presents a formal investigation of the strengths and weaknesses of the Test-First approach to programming, the central component of TDD. We study this component in the context in which it is believed to be most effective, namely, incremental development. To undertake the investigation, we designed a controlled

- H. Erdogmus is with the Institute for Information Technology, National Research Council Canada, Bldg. M-50, Montreal Rd., Ottawa, Ontario K1A 0R6, Canada. E-mail: Hakan.Erdogmus@nrc-cnrc.gc.ca.
- M. Morisio and M. Torchiano are with the Dipartimento Automatica e Informatica, Politecnico di Torino, C.so Duca degli Abruzzi, 24, I-10129, Torino, Italy. E-mail: {Maurizio.Morisio, Marco.Torchiano}@polito.it.

Manuscript received 13 Feb. 2004; revised 6 Dec. 2004; accepted 31 Dec. 2004; published online 21 Jan. 2005.
Recommended for acceptance by D. Rombach.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0026-0204.

0098-5589/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

TABLE 1
Summary of Empirical Studies of TDD

Authors	Study Type	Subjects	Study Scope	Test-First focus?	Tests required?	Interfaces provided?	Incr. dev.?	Quality	Prod.
Müller and Tichy [8]	Case Study	Students	XP	No	?	?	N/A	?	?
Müller and Hagner [9]	Cont. Exp.	Students	TDD	Yes	Yes	Yes	No	No diff.	No diff.
Maximilien, Williams and Vouk [10]	Case Study	Prof.	TDD	No	No	No	No	TDD is better	No diff.
Edwards [11]	Case Study	Students	TDD	No	?	?	N/A	TDD is better	Unknown
George and Williams {George, 2003 #91}[12]	Cont. Exp.	Prof.	TDD	No	No	No	No	TDD is better	TDD is worse
This Study	Cont. Exp.	Students	TDD	Yes	Yes	No	Yes	See Sect. 4	See Sect. 4

Abbreviations -- Cont. Exp.: Controlled Experiment; Incr. dev.: Incremental Development; Prof.: Professionals; Prod.: Productivity; ?: Unknown/Unspecified; N/A: not applicable.

experiment in the winter of 2002-2003 and conducted the experiment in an academic setting in the spring of 2003.

The paper is organized as follows: Section 2 reviews the background work, sets the context, and motivates the study. Section 3 explains the goals and hypotheses and describes the experimental design. Section 4 analyzes the data and presents the results. Section 5 discusses threats to validity. Finally, Section 6 revises the theory to better explain the results and Section 7 presents conclusions.

2 Background and Related Work

2.1 Previous Studies

Table 1 provides a summary of previous empirical studies about TDD. The table reports, from left to right, the authors for each study, the type of the study, the type of the subjects used, the scope of the study (whether or not conducted within the larger scope of Extreme Programming (XP)), whether the focus was on the Test-First aspect, whether the control subjects were required to write tests, whether the experiment tasks defined interfaces to comply with, and whether the study was conducted strictly within the context of incremental development. The last two columns report the differences observed between the TDD group and the control group in terms of the quality and productivity measurements taken. The last row characterizes the current study for comparison purposes.

Based on subjective evaluations, Müller and Tichy [8] reported that students considered TDD as the best technique at the end of a university course on Extreme Programming. Later, Müller and Hagner [9] conducted a controlled experiment on TDD in an academic setting. They concluded that TDD neither accelerated development nor produced better quality programs, although it enhanced program understanding.

Maximilien and Williams [10] observed in an industrial case study that, with TDD, professional programmers were able to achieve a 40-50 percent improvement in code quality without any significant impact on their productivity. TDD programmers were compared with the company baseline where programming involved ad hoc unit testing. These observations are similar to those reported by Edwards [11] based on a classroom case study: Students applying TDD scored higher on their programming assignments while producing code with 45 percent fewer defects. George and Williams [12] later conducted a formal TDD experiment with professional pair programmers. They reported that the TDD pairs' product quality was on average 18 percent higher than that of the non-TDD pairs, while their productivity was 14 percent lower.

2.2 Rationale and Comparison with Previous Studies

We could locate only two published controlled experiments on TDD [9], [12] with different results. Our experiment focuses essentially on the same dependent variables (product quality and programmer productivity) as these experiments, but has a different design. The rationale of our design is explained next.

Müller and Hagner [9] supplied the subjects with method stubs with empty bodies from an existing graphic library and asked the subjects to reimplement the method bodies. While an experiment task with predefined interfaces facilitates assessing the quality of the delivered programs, it could significantly constrain the solution space, making it difficult to reveal the differences between the techniques under study. Our design avoids this pitfall by supplying

TABLE 2
Comparison of Test-First and Test-Last

	Test-First	Test-Last
Who writes and runs tests?	Programmer	Programmer
When are tests written?	Before production code	After production code
When are tests run?	While writing production code, frequently	After writing production code, less frequently
Incremental development?	Yes	Yes
Regression testing?	Yes	Yes

only high-level requirements (in terms of user *stories*) with the experiment task. We overcame the difficulty regarding quality assessment by using a robust black-box acceptance testing approach.

In George and Williams' experiment [12], the control group followed a waterfall-like process, with all testing pushed to the end. As a consequence, most control subjects dropped testing altogether. The result was an apparent, short-term productivity disadvantage for the group that incurred the testing overhead. Writing tests requires time, and when testing is not interleaved with programming, productivity benefits will not be realized in the short-term. Similar effects were also observed in other studies [10], [12], where testing did not play a significant role, was ad hoc, or interpreted as optional by the control subjects. To avoid this bias, we provided the subjects with the same experiment task as that of George and Williams, but decomposed it into a series of features to be implemented in a specific order. The features built on one another and, sometimes, late features modified earlier ones, thus emulating an environment where requirements are incremental and changing. This approach has the advantage that testing and writing production code are easily interleaved, mitigating the risk of deferring testing indefinitely. The objective of our design is *not* to evaluate the effect of the presence of testing, but compare alternative techniques involving opposite testing dynamics.

Additional differences with previous studies concern the measures. While Müller and Hagner [9] addressed internal quality in terms of test coverage and program understanding in terms of usage of existing libraries, we focused on external aspects. The only internal aspect we studied was the amount of tests produced by the programmers. We adopted a productivity measure based on functionality delivered per unit effort according to well-defined, objective criteria. This measure is similar to the one used by Maurer and Martel in their Extreme Programming case study [13], but had an explicit minimum quality requirement. When the productivity measure is not linked to the delivered external functionality or does not satisfy a minimum acceptable quality criterion, any observed productivity gains risk being superfluous. For example, when productivity is measured for a fixed-scope task in terms of problem-solving time alone [9], [10], [12] and for a variable-scope task in terms of a size metric alone, the observed nominal productivity may be high, but real productivity that provides value may be low. In our study, the experiment task had a variable scope (the subjects implemented as many features as possible from a list), and we measured both delivered scope and problem solving time to determine real rather than nominal productivity.

2.3 Test-First versus Test-Last Development

The main characteristics of TDD are incremental development, definition and coding of unit tests by the programmer, frequent regression testing, and writing test code, one test case at a time, before the corresponding production code. Since the last characteristic—writing tests *before* production code—is most central to TDD, we isolate it in our investigation. Therefore, the reference technique that represents conventional practice for the purpose of the investigation also involves incremental development, definition and coding of tests by the programmer, and frequent regression testing, but the programmer writes all the tests for a new system feature *after* the corresponding production code for that feature. From now on, we will refer to the technique under investigation simply as *Test-First* and the reference technique to which Test-First is compared as *Test-Last*. Table 2 summarizes the similarities and differences between these alternative techniques.

The top part of Fig. 1 illustrates the differences in terms of the underlying processes. Test-First advances one test at a time, resulting in an additional inner loop for each user story (a *story* in TDD represents an end-user feature, specified in an informal way). In Test-Last, tests are written in a single burst for a story. Although both techniques apply a similar incremental process above the story level, Test-First is finer grained due to the additional inner loop.

Test-First works as follows:

1. Pick a story.
2. Write a test that expresses a small task within the story and have the test fail.
3. Write production code that implements the task to pass the test.
4. Run all tests.

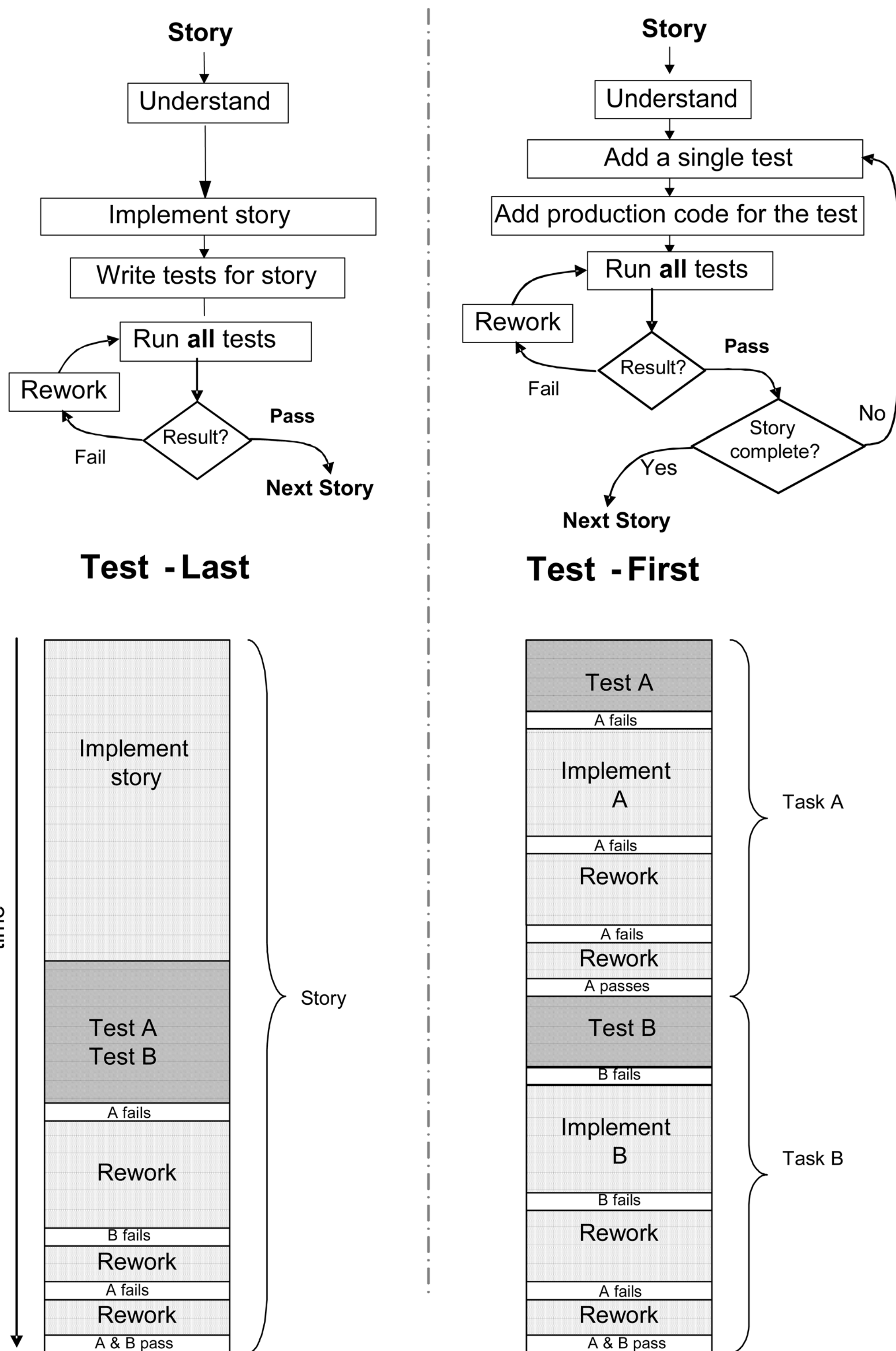


Fig. 1. Dynamics of test-first versus test-last.

5. Rework production code and test code until all tests pass.
6. Repeat 2 to 5 until the story is fully implemented.

Test-Last works as follows:

1. Pick a story.
2. Write production code that implements the story.
3. Write tests to validate the whole story.
4. Run all tests.
5. Rework production code and test code until all tests pass.

The bottom part of Fig. 1 provides an example story that requires two tasks, A and B. The programmer first writes a test for task A, say Test A, runs this test to watch it fail, then codes the corresponding functionality and runs Test A again. If Test A still fails, then the programmer reworks the

code until Test A passes. When Test A passes, task A is complete. Then, the programmer writes a test for task B, say Test B, and codes the corresponding functionality such that both Test A and Test B pass. Thus, at the end of the second cycle, the story has effectively been completed. The Test-Last programmer first writes the production code to implement the whole story, then writes the functional tests. The tests are run, and the programmer reworks the code until all tests pass.

3 Experimental Design

This section first presents the goals and hypotheses, then gives a description of the experiment following the guidelines of the American Psychological Association [14].

3.1 Goals

The goal of the experiment, expressed according to Wohlin et al.'s template [15], is *to compare* Test-First programming *with* Test-Last programming *for the purpose* of evaluating

- external quality and
- programmer productivity

in the context of

- incremental development and
- undergraduate object-oriented programming course.

3.2 Hypotheses

Testing is critical to quality assurance and, in TDD functional, unit-level tests play a special role since they steer the coding activity. Several sources [1], [2], [6], [12] claim that programmers applying a test-first strategy faithfully will tend to write as much test code as they write production code. Such a high ratio of *test-code*-to-*production-code* is considered uncommon with traditional techniques. Thus, our first expectation is that Test-First programmers write more tests per unit of programming effort. This expectation constitutes hypothesis **1T**.

An increased emphasis on testing should increase quality. Most TDD studies support this effect on quality [11], [12]. Since testing plays a more crucial role than in Test-Last and is enforced in Test-First, we expect Test-First programmers to produce higher quality programs. This expectation constitutes hypothesis **1Q**.

Long-term productivity benefits of testing with traditional quality assurance are well recognized: maintaining high quality through testing reduces future effort [16]. Although testing creates additional overhead in the short term, it pays back in the long term. TDD advocates take this argument one step further: They claim that incremental testing increases both long-term and short-term productivity. When testing is interleaved with coding in a fine-grained process, the programmer can maintain focus and receives frequent feedback. The tight feedback loop makes it easier to revise existing functionality and implement new features. Therefore, we expect Test-First programmers to be more productive overall. This expectation constitutes hypothesis **1P**.

These hypotheses assume a direct effect between the independent variable and the dependent variables.

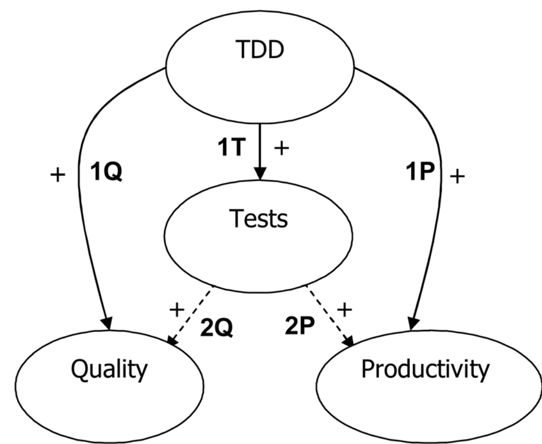


Fig. 2. The two-stage model.

However, in some cases, a direct effect can be hard to detect because of the presence of an intermediate variable. A multistage model allows for a deeper analysis by decomposing the predicted effects into a chain of more elementary components, each of which can be studied in relative isolation. In a chain effect, a variable A depends on an intermediate variable B, which, in turn, depends on a third variable C. The noise introduced in the middle of the chain results in loss of information, making it more difficult to infer a causal relationship between C and A and requiring experimental designs with higher power.

Since tests play such a central role, it is possible that the effectiveness of Test-First hinges on the extent to which the strategy encourages programmers to back up their work with test assets. The more tests are written, the more effective the technique becomes. Therefore, we identify the number of tests written by the programmer as an intermediate variable and extend the theory using a two-stage model. The extended model allows us to address the following questions:

- Is the number of tests important in determining productivity and quality regardless of the technique applied?
- Is Test-First more effective because it encourages programmers to write lots of tests?

The extended model is illustrated in Fig. 2. Stage 1 investigates the direct effects between the independent variable (the coding technique used) and dependent variables (productivity and quality). Stage 2 involves the intermediate variable, number of tests per unit effort, shown in the middle (the bubble labeled "Tests"), and two new hypotheses, **2Q** and **2P**. The new hypotheses suggest a causal relationship between the number of tests and quality and productivity, respectively (represented by dashed lines).

- **2Q**: Writing more tests improves quality.
- **2P**: Writing more tests increases productivity.

If the hypotheses 1T and 2Q(P) are accepted and the size of the effect [17] in 1Q(P) is compelling, we can infer that Test-First leads to better quality (productivity) because it encourages programmers to write more tests.

3.3 Experiment Design

We used a standard design with one factor and two treatments [15]. The treatments correspond to the two techniques, Test-First (experiment group) and Test-Last (control group), described in Section 2.4. The assignment of subjects to groups was performed first by stratifying the subjects with respect to their skill level and then assigning them randomly to one of the two treatment groups.

3.4 Subjects

Subjects were third-year students following an eight-week intensive Java course at Politecnico di Torino. The course was part of the Computer Science degree program.

During the course, all students learned about object orientation, Java programming, basic design concepts (including UML), and unit testing with JUnit [5]. The final exam involved a hands-on programming task in the lab.

3.5 Apparatus and Experiment Task

The computer laboratory was equipped with over 40 personal computers. All computers had an internet connection and a standard disk image that was refreshed every week. The image contained a Web browser and Eclipse IDE [18] with the JUnit [5] plugin and CVS [19] client built in. Each student had a confidential CVS account with a private repository.

The experiment task consisted of Robert Martin's Bowling Score Keeper problem that was also used by George and Williams in their TDD study [12]. However, in our experiment, we organized the problem in several small stories, each describing a unique feature of the end product to be delivered by the student.

Each story was externally testable and had a specific set of black-box acceptance tests written by the authors. These acceptance tests were not revealed to the subjects. We wrote a total of 105 black-box acceptance test methods consisting of over 350 JUnit assert statements. The acceptance tests were based on regular expression searches on the screen output of a program. They were robust in their ability to verify requirements concerning form and content separately. At the end of the experiment, these acceptance tests were automatically run on each subject's program.

As part of the experiment task, subjects wrote their own unit tests using JUnit, as prescribed by the technique assigned to them. We refer to the tests written by us to derive the quality and productivity measures as *acceptance tests* and the tests written by the subjects as part of their experiment task as *programmer tests*.

The programming increments (stories) in the experiment task were fine grained to be able to accept and accurately evaluate partially completed programs. A possible caveat of a small granularity is reduced realism due to increased testing overhead in both treatments.

3.6 Procedure

Subjects in both groups received general instruction on low-level functional testing, as well as the principles and use of JUnit in particular. In addition to traditional unit testing, they were taught a specific JUnit pattern for directly testing external functionality through the command line user interface of a program. The subjects used this pattern exclusively during the experiment task. The use of this pattern allowed to eliminate the role that low-level design plays in TDD, thereby isolating the Test-First aspect.

After having been assigned to the two groups, subjects were trained on the two techniques to follow (Test-First and Test-Last as described in Section 2.4). Training was administrated in separate concurrent sessions.

The subjects were asked to implement the stories over several lab sessions following the programming technique assigned to them. They followed a similar protocol for all programming assignments, including the experiment task. The descriptions of the assignments were published online and an empty project was created for each assignment in each student account. In the beginning of a lab session or whenever they wanted to work on an assignment, the subjects first checked out the corresponding project from their private CVS repository. Before they left the lab or when they wanted to stop, they committed the local version of their project back into the CVS repository. They worked on the same assignment over several lab sessions.

Students were encouraged to perform all programming assignments and the experiment task in the computer laboratory during dedicated hours, although they were not forbidden to work outside the laboratory hours. Students received approximately four hours of hands-on experience per week, split into two sessions.

The subjects in both groups tackled the experiment task incrementally, implementing the stories one at a time and in the given order. They were not given any stubs or interfaces to comply with, but were free to use utility classes and patterns from previous assignments. The stories had various difficulty levels and built on top of each other. The end product required only a command-line interface and simple design (it was possible to implement it with a few classes).

The subjects filled in two questionnaires: one before the experiment to assess their experience and skills (the *prequestionnaire*) and one after the experiment to assess the level of acceptance of the techniques, process conformance, and total programming effort (the *postquestionnaire*).

Students were informed of the goals of the study and the procedures before the experiment, but the hypotheses were not revealed. Data was collected only from students who signed an informed consent form. The instructors could not differentiate between participating and nonparticipating students. All students received instruction of equivalent educational value regardless of whether they participated in the experiment and regardless of their assigned group. After the experiment, the class was debriefed and the students were taught the other group's technique.

To address ethical considerations, the following precautions were taken: Participation to the experiment was voluntary and subjects were free to drop out at any point without any consequences. Confidentiality of the data was guaranteed.

3.7 Experiment Variables and Formalized Hypotheses

The main independent variable of the experiment is group affiliation. It indicates whether a subject belongs to the experiment (Test-First) or control (Test-Last) group. The

TABLE 3
Context Variables

Variable name	Description
Subject type	Computer Science, undergraduate students
Programming language	Java
Programming environment	Eclipse, CVS, JUnit
Academic level	Junior object-oriented programming course

other independent variable measures the skill level of a subject based on the subject's prequestionnaire responses, but this variable does not appear in the hypotheses. It was used for balancing the groups.

The main dependent variables are programmer tests per unit effort (TESTS), productivity (PROD), and quality (QLTY).

Productivity is defined as output per unit effort. The number of stories is well-suited for measuring output: For our purpose, it is a superior measure of real output than program size (e.g., lines of code) in that it constitutes a more direct proxy for the amount of functionality delivered. It is still an objective measure since we can compute it automatically based on black-box acceptance tests. If a story passed at least 50 percent of the assert statements from the associated acceptance test suite, then the story was considered to be delivered. The number of stories delivered was normalized by total programming effort to obtain the productivity measure PROD.

The variable TESTS measures the number of programmer tests written by a subject, again, per unit of programming effort. A programmer test refers to a single JUnit test method. Through visual inspection of the subjects' test code, we filtered out ineffective tests, such as empty test methods, duplicated test methods, and useless test methods that passed trivially. Because subjects were free to work as many hours as they wanted, the variation in total programming effort was large (ranging from a few hours to as many as 25 hours). Hence, it was necessary to normalize the number of tests by the total effort expended.

The measure of quality is based on defects, counted on a story-by-story basis. We considered only the stories delivered by a subject. The quality of a story is given by the percentage of assert statements passing from the associated acceptance test suite. The quality of each story is then weighted by a proxy for the story's difficulty based on the total number of assert statements in the associated acceptance test suite. Finally, a weighted average is computed for each subject over all delivered stories, giving rise to the measure QLTY. By construction, the range of this variable is 0.5 (50 percent) to 1 (100 percent).

Total programming effort (or program solving time) was obtained from the postquestionnaire on which the subjects reported the total number of hours worked on the experiment task. Their responses were cross-validated by their CVS logs. No significant discrepancies were found. In four cases, the missing values could be reliably estimated from the CVS logs.

Other variables define the context. The context variables are of importance for replication purposes. These are summarized in Table 3.

Table 4 presents the experiment hypotheses formally in terms of the dependent variables. In the table, TF refers to Test-First (experiment group) and TL refers to Test-Last (control group).

4 Data Analysis

4.1 Characterization of Groups

Thirty-five subjects participated in the study. Eleven subjects dropped out or did not check in the final version of their program. Therefore, we retained data from 24 subjects. Eleven of those subjects were in the Test-First (**TF**) group and 13 in the Test-Last (**TL**) group. We found no statistically significant difference between the two groups in terms of both average skill rating and average total effort. However, skill rating could not be used as a blocking variable because some cells did not contain enough data points after mortality (see Table 5).

4.2 Stage 1 Hypotheses (1T, 1Q, 1P)

Fig. 3 shows the boxplots of the variables TESTS, QLTY, and PROD that underlie the hypotheses 1T, 1Q, and 1P, respectively. In the figures, the dots anchored by the horizontal line mark the medians, the floating dots mark the means, the limits of the boxes mark the top and the bottom 25 percent limits, and the whiskers mark the outlier limits. Outliers are indicated by the stars. Table 6 gives the mean and the median values for each group.

For TESTS, both the median and the mean of the TF group are considerably higher (100 percent and 52 percent, respectively) than those of the TL group. (Two subjects in the TL group did not write any tests. When these subjects are excluded, the TF average is still 28 percent higher). For QLTY, the means are very close (the TL mean is 2 percent higher than the TF mean), but the TL median is slightly (10 percent) higher than that of the TF group. Surprisingly, the TL group appears to enjoy a mild quality advantage. For PROD, both the median and the mean of the TF group are markedly higher (21 percent and 28 percent, respectively).

Upon inspecting the 25 percent and 75 percent limits, we observe an apparent advantage for the TF group for the variable TESTS. For PROD, the 75 percent limit of the TF group is much higher, but the 25 percent is slightly lower. For QLTY, both TF limits are lower, suggesting a slight advantage for the TL group. The TF group has a

TABLE 4
Formalized Hypotheses

Model Stage	Name	Null Hypothesis - H_0	Alternative Hypothesis - H_a
1	1T	$TESTS_{TF} = TESTS_{TL}$	$TESTS_{TF} > TESTS_{TL}$ *Test-First Programmers write more tests.*
	1Q	$QLTY_{TF} = QLTY_{TL}$	$QLTY_{TF} > QLTY_{TL}$ *Test-First programmers produce a higher quality product.*
	1P	$PROD_{TF} = PROD_{TL}$	$PROD_{TF} > PROD_{TL}$ *Test-First programmers are more productive.*
2	2Q	$QLTY = \beta_o + \beta_1 TESTS$ $\beta_1 = 0$	$\beta_1 \neq 0$ TESTS can predict QLTY. *Higher number of tests implies higher quality product.*
	2P	$PROD = \beta_o + \beta_1 TESTS$ $\beta_1 = 0$	$\beta_1 \neq 0$ TESTS can predict PROD. *Higher number of tests implies higher productivity.*

higher variation in both TESTS and PROD, but a lower variation in QLTY.

None of the variables were normally distributed according to standard normality tests (although they appeared lognormal). To verify the three Stage-1 hypotheses, we applied the Mann-Whitney U-test [20], which is a robust, nonparametric test. Since all of our hypotheses are directional, we applied the one-tailed version of this test. Because of the small sample size, we set the alpha level to 10 percent. The results of the Mann-Whitney U-test are shown in Table 6 along with the descriptive statistics. The size of the effect is gauged by Cohen's *d* statistic [17].

Hypothesis 1T predicts an improvement in the variable TESTS, the number of tests per unit effort, for the TF group relative to the TL group. The effect is in the predicted direction, the effect size is small [17], but remarkable, and the Mann-Whitney U-test is statistically significant. We therefore reject the null hypothesis.

Hypothesis 1Q predicts a higher quality—as measured by QLTY—for the TF group. The effect is in the opposite direction and the effect size is not meaningful [17]. Thus, we must use a two-tailed test in this case. The test is not statistically significant. We cannot reject the null hypothesis and, therefore, we conclude that there are no significant quality differences between the two groups.

TABLE 5
Distribution of Subjects

Skill Rating	Group		
	Test-First	**Test –Last**	All
Low	3	0	3
Medium	2	5	7
High	6	8	14
All	11	13	24

Hypothesis 1P predicts a higher productivity—as measured by the variable PROD, the number of delivered stories per unit effort—for the TF group. The effect size is small [17] and in the predicted direction, but, according to Mann-Whitney U-test, it is not statistically significant. At this point, we cannot reject the null hypothesis. However, the apparently large differences between the means and particularly the 75 percent limits warrant a deeper investigation, which is undertaken in Stage 2.

4.3 Stage-2 Hypotheses (2Q, 2P)

Hypothesis 2Q predicts a linear relationship between TESTS and QLTY independent of the group. The scatterplot, shown in Fig. 4, however, suggests that the relationship is not linear. The Spearman correlation is low and statistically insignificant. We cannot reject the null hypothesis in its present form. However, remarkably, all data points lie approximately on or above a diagonal line (dashed line in Fig. 4). Although a few tests can still achieve a high level of quality, writing more tests increases the minimum quality achievable. Analytically, the relationship between TESTS and QLTY can be expressed by the following inequality:

```
QLTY  0.1*TESTS + 0.55
```

Hypothesis 2P predicts a linear relationship between TESTS and PROD. The linear trend is visible in the scatterplot of Fig. 5. The Spearman coefficient is highly statistically significant (0.587 with a p-value of 0.003). Linear regression results in the model:

```
PROD = 0.255 + 0.659*TESTS
```

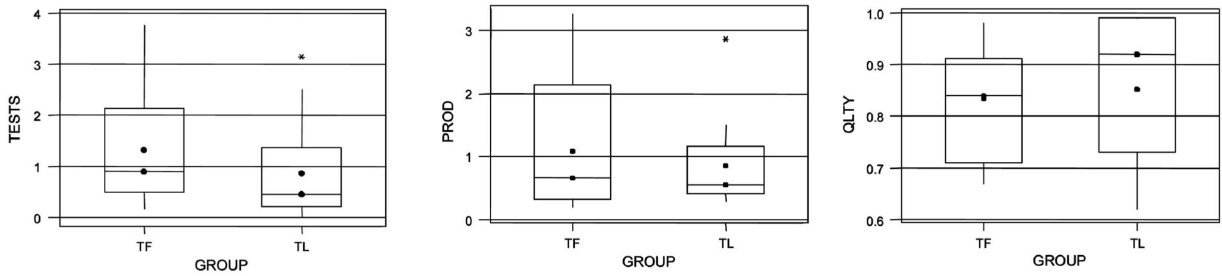
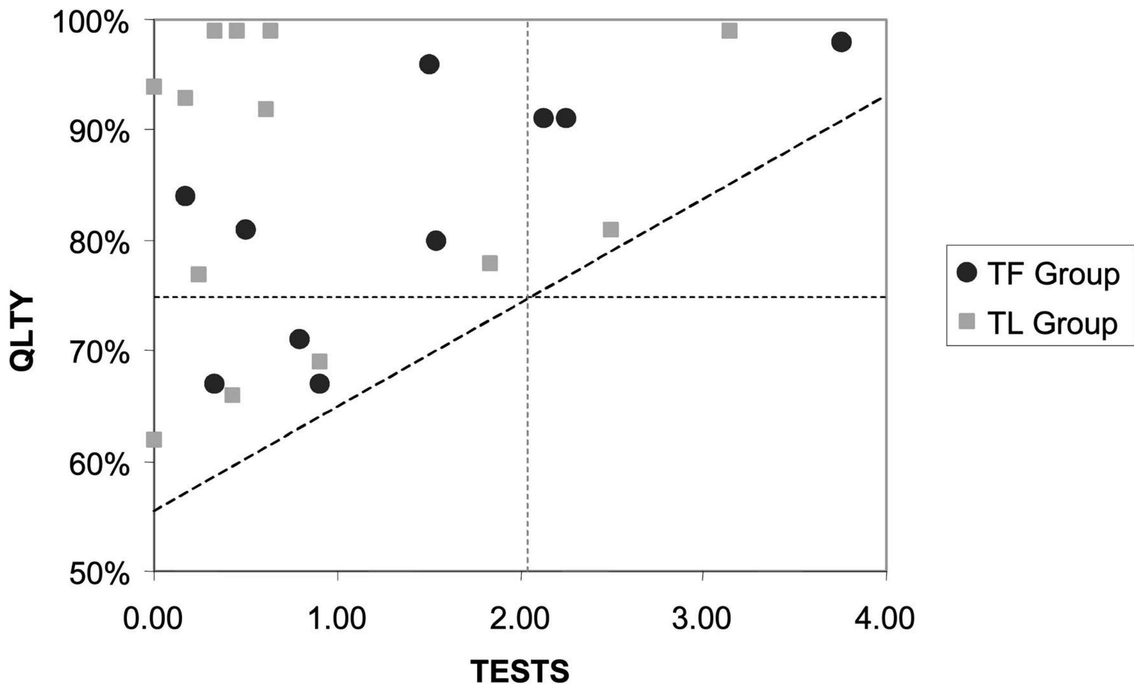


Fig. 3. Boxplots for number of tests, quality, and productivity.

TABLE 6
Results of Stage 1

Hypothesis (Variable)	Median		Mean *(StdDev)*		Mann-Whitney Exact p-value
	Test-First	Test- Last	Test-First	Test –Last	
1T (TESTS)	0.90	0.45	1.31 *(1.07)*	0.86 *(1.00)*	0.09
1Q (QLTY)	0.84	0.92	0.83 *(0.11)*	0.85 *(0.14)*	0.25
1P (PROD)	0.67	0.55	1.09 *(1.05)*	0.85 *(0.71)*	0.48



The variable TESTS explains a large part of the variation in the variable PROD (adjusted $R^2 = .60$), and the regression coefficient (β_1) is highly statistically significant with a p-value of less than 0.001. When the control and experiment groups are analyzed separately, the slope β_1 of the regression line fluctuates only slightly and remains statistically significant. Therefore, we reject the null hypothesis and accept the alternative hypothesis 2P that the number of tests can predict productivity *independent of the group*.

5 Threats to Validity

Process conformance. As with most empirical studies, an important threat is represented by the level of conformance of the subjects to the prescribed techniques. We took the

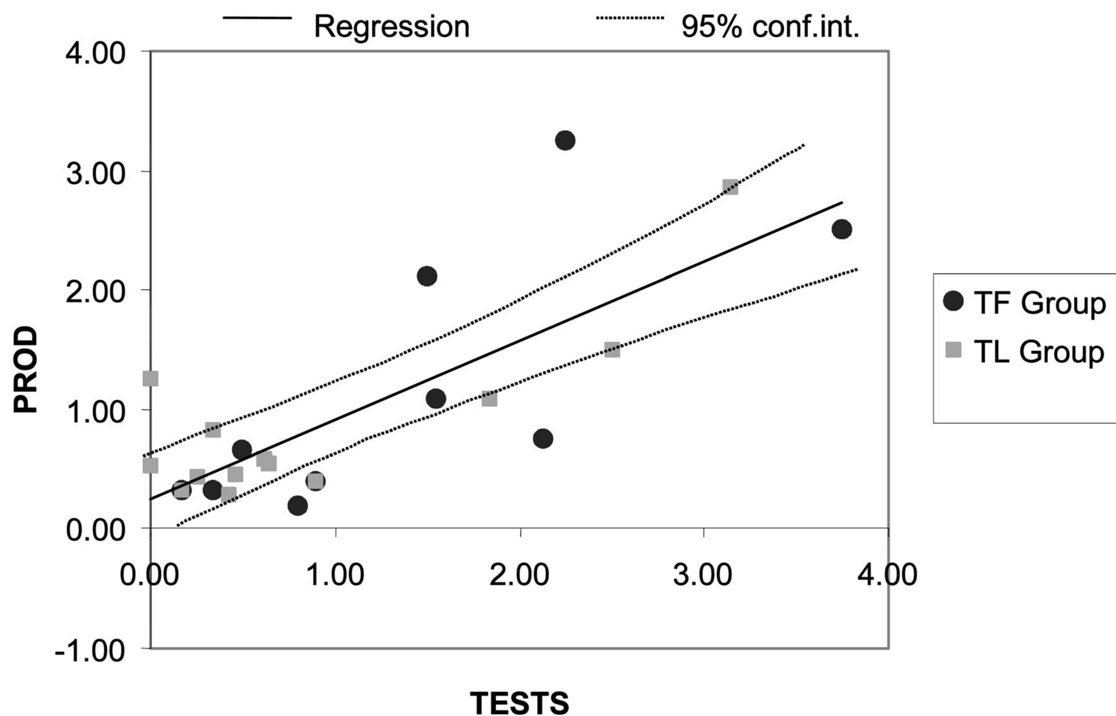


Fig. 5. Regression analysis of TESTS vs. PROD.

following precautions to improve process conformance: The subjects were informed of the importance of following the proper procedures. In the postquestionnaire, we asked the subjects whether they applied the technique assigned and we discarded those data points for which the subject answered "No" to this question. The questionnaires were anonymous and confidentiality was guaranteed. In addition, we applied a formal criterion to gauge conformance in the experiment group: Subjects who did not write any tests for at least half the time (i.e., for half the stories they implemented) were deemed nonconformant. In the final analysis, four data points (three based on the postquestionnaire responses and one based on the formal criterion) were discarded.

Mono-operation bias. Only one task was used in the experiment, and the task was small and had unique technical characteristics. Although we designed the task to specifically fit in the target context (incremental development), it is not clear to what extent we can generalize the results to programming tasks with a larger scope, different technical characteristics, and requiring a different incremental process. In particular, the fine-grained incremental process enforced through the use of artificially small stories could have presented a bias. Small increments tend to increase the overhead of testing. If the technique employed affects the testing overhead to different extents, the results could have been skewed by this design choice. The impact of this extra cost, particularly on productivity, is as yet unclear.

Moreover, we deliberately isolated an aspect of TDD that we consider both central and unique (the *Test-First* dynamics) at the expense of others (for example, we controlled for the role of unit tests as a substitute for up-front design by encouraging the subjects to adopt a particular functional testing pattern).

External validity. The external validity of the results could be limited since the subjects were students. Runeson [21] compared freshmen, graduate, and professional developers and concluded that similar improvement trends persisted among the three groups. Replicated experiments by Porter and Votta [22] and Höst et al. [23] suggest that students may provide an adequate model of the professional population. However, not in all contexts are the trends similar among populations representing different skill and professional maturity levels. For example, Arisholm and Sjøberg [24] reported opposite trends for students and professionals when evaluating an advanced object-oriented design technique. While junior students performed worse with the advanced technique, professionals performed better with it. Senior students were in the middle. The authors concluded that the effectiveness of the technique depended to a large extent on skill. A possible explanation is that advanced techniques require advanced levels of mastery and, thus, higher-skill people are better at leveraging an advanced technique. This effect is certainly also observed in our study: The productivity improvement in students with higher skill ratings were, on average, more dramatic than the productivity improvement in students with lower skill ratings. This observation works to our advantage. The general perception of TDD and test-first programming is that its effective application requires mastery and discipline. If junior students can effectively apply it with little experience and exposure, then experienced students and professionals could potentially achieve more dramatic improvements.

6 Discussion and Theory Building

The results of the experiment support an alternative theory of the Test-First technique that is mainly centered on productivity rather than on quality. We now revise the theory to better explain the results.

6.1 Quality

Why did more testing by Test-First subjects not result in a proportional increase in quality? When the programming task is small, the scope is limited, and the requirements are predecomposed into small, well-articulated features, then ad hoc strategies may compensate for lack of automated, low-level functional tests. Programmers could just as easily get equivalent feedback by visually inspecting the output of their program.

A second explanation is also plausible. As seen in Fig. 4, the quality level attained for a given range of tests has a high variation when the number of tests is low. Here, skill is a possible factor. Students who wrote few programmer tests achieved varying degrees of quality, from very low to very high, possibly depending on their skill level. Perhaps skill could compensate for lack of testing to a certain extent and higher skilled programmers wrote more effective tests. If these postulates hold, Fig. 4 suggests that skill should become less and less influential in determining quality as the number of tests increase.

Programmer tests nevertheless still play a central role in quality assurance. Although the amount of tests did not predict quality in absolute terms with student subjects, it could predict minimum quality.

6.2 Productivity

Testing and programming are tightly integrated activities that support incremental development. Incremental development allows for demonstrating real, rather than nominal, productivity benefits of Test-First. We believe that the observed productivity advantage of Test-First subjects is due to a number of synergistic effects:

- *Better task understanding*. Writing a test before implementing the underlying functionality requires the programmer to express the functionality unambiguously.
- *Better task focus*. Test-First advances one test case at a time. A single test case has a limited scope. Thus, the programmer is engaged in a decomposition process in which larger pieces of functionality are broken down to smaller, more manageable chunks. While developing the functionality for a single test, the cognitive load of the programmer is lower.
- *Faster learning*. Less productive and coarser decomposition strategies are quickly abandoned in favor of more productive, finer ones.
- *Lower rework effort*. Since the scope of a single test is limited, when the test fails, rework is easier. When rework immediately follows a short burst of testing and implementation activity, the problem context is still fresh in the programmer's mind. With a high number of focused tests, when a test fails, the root cause is more easily pinpointed. In addition, more frequent regression testing shortens the feedback cycle. When new functionality interferes with old functionality, this situation is revealed faster. Small problems are detected before they become serious and costly.

Test-First also tends to increase the variation in productivity. This effect is attributed to the relative difficulty of the technique, which is supported by the subjects' responses to the post-questionnaire and by the observation that higher-skill subjects were able to achieve more significant productivity benefits.

7 Conclusions and Future Work

We evaluated an important component of Test-Driven Development (TDD) through a controlled experiment conducted with undergraduate students. The evaluation focused on the *Test-First* aspect of TDD, where programmers implement a small piece of functionality by writing a unit test *before* writing the corresponding production code. Since Test-First is essentially an incremental strategy, we designed the experiment to fit in an incremental development context.

Our main result is that Test-First programmers write more tests per unit of programming effort. In turn, a higher number of programmer tests lead to proportionally higher levels of productivity. Thus, through a chain effect, Test-First appears to improve productivity. We believe that advancing development one test at a time and writing tests before implementation encourage better decomposition, improves understanding of the underlying requirements, and reduces the scope of the tasks to be performed. In addition, the small scope of the tests and the quick turnaround made possible by frequent regression testing together reduce debugging and rework effort.

Test-First programmers did not achieve better quality on average, although they achieved more consistent quality results. We attribute the latter observation to the influence of skill on quality, which Test-First tended to dampen. Writing more tests improved the minimum quality achievable and decreased the variation, but this effect does not appear to be specific to Test-First.

In summary, the effectiveness of the Test-First technique might very well hinge on its ability to encourage programmers to back up their code with test assets. Future experiments could focus on this ability.

Extra care must be taken before attempting to generalize these observations to other contexts. Many factors pertaining to the experimental setting could have biased the results (see Section 5).

The study can benefit from several improvements before replication is attempted. The most significant one is securing a larger sample size to guarantee a high-power design in order to tolerate the observed mortality rate of 30 percent. Next important, improvement concerns better support for instrumentation and process conformance. New tools should help with this latter issue [25].

Acknowledgments

The authors are indebted to the following persons: Dr. Howard Johnson hypothesized about the central role of the

number of tests. Dr. Khaled El-Emam provided advice on design, theory, and analysis. Dr. Janice Singer provided comments on the revisions and organization. Professor Laurie Williams shared insights and material from her research. Andrea Capiluppi gave technical and logistic support. Professor Daniele Romano provided comments on the analysis. Serra Erdogmus helped with editing and interpretation of results.

REFERENCES

[1] K. Beck, *Test-Driven Development: by Example.* Addison Wesley, 2003.

[2] D. Astels, *Test Driven Development: A Practical Guide.* Prentice Hall, 2003.

[3] K. Beck, *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[4] D. Astels, G. Miller, and M. Novak, *A Practical Guide to Extreme Programming.* Prentice Hall, 2002.

[5] JUnit.org, www.junit.org, Year?

[6] K. Beck and E. Gamma, "Test Infected: Programmers Love Writing Tests," *Java Report,* vol. 3, pp. 51-56, 1998.

[7] R.C. Linger, H.D. Mills, and M. Dyer, "Cleanroom Software Engineering," *IEEE Software,* Sept. 1987.

[8] M.M. Müller and W.F. Tichy, "Case Study: Extreme Programming in a University Environment," *Proc. Int'l Conf. Software Eng. (ICSE),* 2001.

[9] M.M. Müller and O. Hagner, "Experiment about Test-First Programming," *Proc. Conf. Empirical Assessment in Software Eng. (EASE),* 2002.

[10] E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," *Proc. Int'l Conf. Software Eng. (ICSE),* 2003.

[11] S.H. Edwards, "Using Test-Driven Development in the Classroom: Providing Students with Concrete Feedback on Performance," *Proc. Int'l Conf. Education and Information Systems: Technologies and Applications (EISTA '03),* Aug. 2003.

[12] B. George and L. Williams, "An Initial Investigation of Test Driven Development in Industry," *Proc. ACM Symp. Applied Computing,* 2003.

[13] F. Maurer and S. Martel, "On the Productivity of Agile Software Practices: An Industrial Case Study," technical report, Univ. of Calgary, Alberta, 2001, http://ebe.cpsc.ucalgary.ca/ebe/attach?page=Root.PublicationListpercent2FMaurerMartel2002c.pdf.

[14] APA, *Publication Manual of the Am. Psychological Association,* fourth ed. Washington DC: Am. Psychological Association, 1994.

[15] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction.* Kluwer Academic, 2000.

[16] B. Boehm, "Industrial Metrics Top-10 List," *IEEE Software,* pp. 84-85, Sept. 1987.

[17] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences,* second ed. Hillsdale, N.J.: Lawrence Earlbaum and Associates, 1988.

[18] "Eclipse Platform Technical Overview," vol. 2004, Object Technology Int'l, 2003, http://www.eclipse.org/whitepapers/eclipseoverview.pdf.

[19] P. Cederqvist, "Version Management with CVS," Bristol, U.K.: Network Theory Ltd., 1993.

[20] H.B. Mann and D.R. Whitney, "On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other," *Annals of Math. Statistics,* 1947.

[21] P. Runeson, "Using Students as Experiment Subjects—An Analysis on Graduate and Freshmen Student Data," *Proc. Seventh Int'l Conf. Empirical Assessment and Evaluation in Software Eng. (EASE '03),* 2003.

[22] A. Porter and L. Votta, "Comparing Detection Methods for Software Requirements Inspection: A Replication Using Professional Subjects," *Empirical Software Eng.,* vol. 3, pp. 355-380, 1995.

[23] M. Höst, B. Regnell, and C. Wohlin, "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment," *Empirical Software Eng.,* vol. 5, pp. 201-214, 2000.

[24] E. Arisholm and D.I.K. Sjøberg, "A Controlled Experiment with Professionals to Evaluate the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," Technical Report 2003-6, Simula Research Laboratory, Norway, June 2003, http://www.simula.no/photo/tr20036sep10.pdf.

[25] H. Erdogmus and Y. Wang, "The Role of Process Measurement in Test-Driven Development," *Proc. XP/Agile Universe Conf. Extreme Programming and Agile Methods,* Aug. 2004.

Hakan Erdogmus received the doctoral degree in telecommunications from Université du Québec and the masters degree in computer science from McGill University, Montréal. He is a senior research officer with the software engineering group at NRC's Institute for Information Technology in Ottawa. His current research interests are centered on agile software development and software engineering economics.

Maurizio Morisio received the PhD degree in software engineering and the MSc degree in electronic engineering from the Politecnico di Torino. He is an associate professor at the Politecnico di Torino, Turin, Italy. He spent two years working with the Experimental Software Engineering Group at the University of Maryland, College Park. His current research interests include experimental software engineering, wireless Internet software engineering, software reuse metrics and models, agile methodologies, and commercial off-the-shelf processes and integration. He is a consultant for improving software production through technology and processes. He is a member of the IEEE Computer Society.

Marco Torchiano received the MSc and PhD degrees in computer engineering from the Politecnico di Torino. He is an assistant professor at Politecnico di Torino, Italy; previously, he was a postdoctoral research fellow at Norwegian University of Science and Technology (NTNU), Norway. His research interests include component-based software engineering, COTS-based development, software architecture, maintenance, and design patterns. He is a coeditor of the COTS area on the IEEE SE Portal. He is a coauthor of the forthcoming book *Software Development—Case Studies in Java* (Addison-Wesley). He is a member of the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.