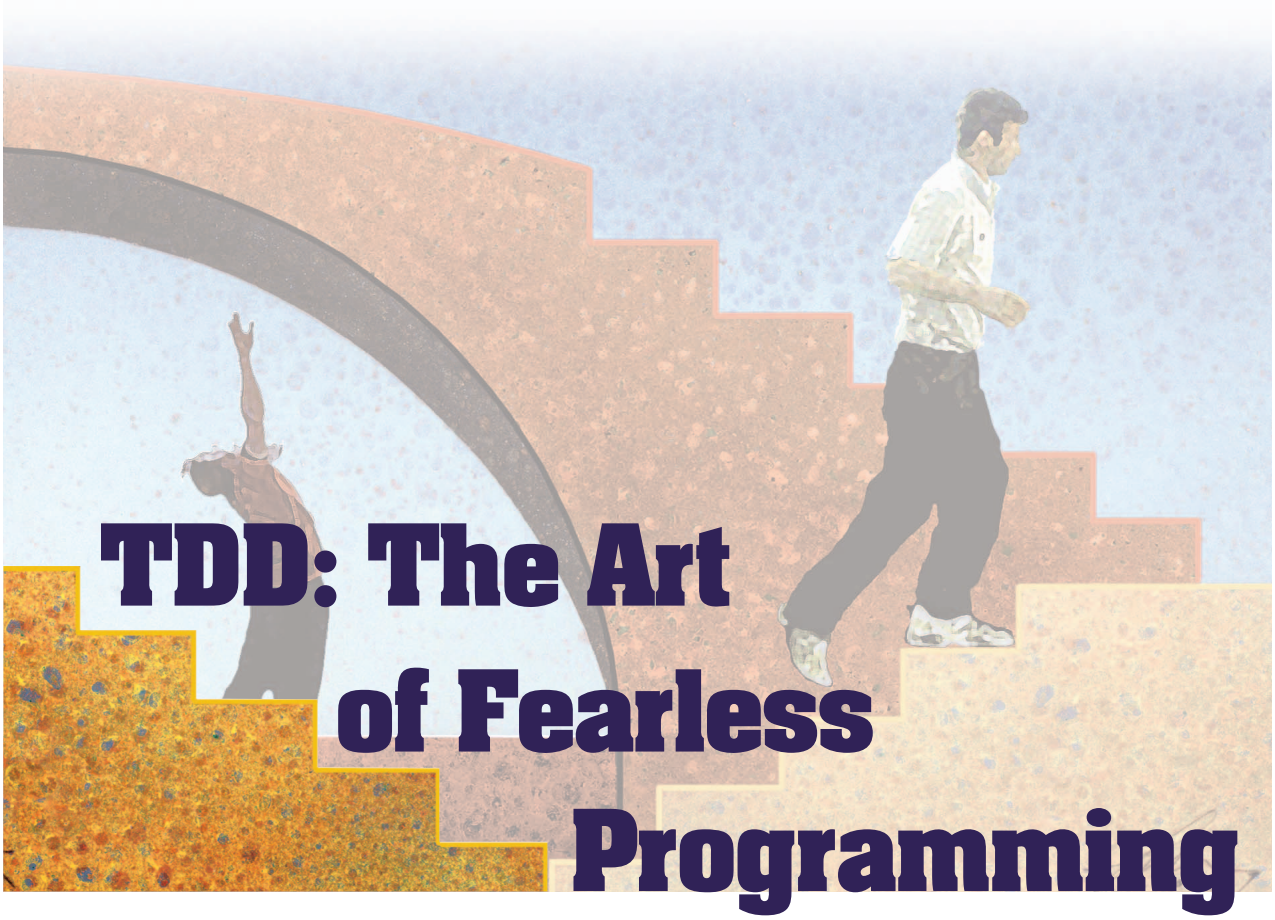


TDD: The Art of Fearless Programming

Ron Jeffries, *independent consultant*

Grigori Melnik, *University of Calgary*

Programmers! Cast out your guilt! Spend half of your time in joyous testing and debugging! Thrill to the excitement of the chase! Stalk bugs with care, and with method, and with reason. Build traps for them. Be more artful than those devious bugs and taste the joys of guiltless programming! —Boris Beizer, 1983

Test-driven development is a discipline of design and programming where every line of new code is written in response to a test the programmer writes just before coding. As TDD practitioners, we think of what small step in capability would be a good next addition to the program. We then write a test specifying just how the program should invoke that capability and what its result should be. The test fails, showing that the capability isn't already present. We implement the code that makes the test pass and then verify that all prior tests are still passing. Finally, we

0740-7459/07/$25.00 © 2007 IEEE

review the code as it now stands, improving the design as we go in an activity called *refactoring*. Then we repeat the process, devising another test for another small addition to the program.

As we follow this simple cycle, shown in figure 1, the program grows into being and the design evolves with it. At the beginning of every cycle, the intention is for all tests to pass except the new one, which is "driving" the new code development. At the end of the cycle, the programmer runs all the tests, ensuring that each one passes and hence that every planned feature of the code still works.

TDD is a design and programming activity, not a testing activity per se. Its testing aspect is largely confirmatory, through the regression suite it produces. Professional testers must still perform investigative testing. (The potential for confusion is spawning new terms for the discipline, such as behavior-driven development[1] and example-driven development.[2])

This special issue of *IEEE Software* includes seven feature articles on various aspects of TDD and a Point/Counterpoint debate on the use of mock objects in applying it. Notably, these articles demonstrate the ways TDD is being used in nontrivial situations (database development, embedded software development, GUI development, performance tuning). This signifies an adoption level for the practice beyond the visionary phase and into the early mainstream.

Alleviating fear

In practice, of course, even the best programmers make mistakes. TDD's growing collection of comprehensive tests (the regression suite) tends to detect these problems. No scheme is perfect, but TDD practitioners seem to experience a reduction in defects shipped, plus much faster problem detection.

Anyone who's worked on legacy software recognizes the situation where a system continues to function but becomes more and more outdated until, at some point, it turns into a house of cards. No one wants to touch it because even a minor code change will likely lead to an undesired side effect. With TDD, developers organically develop a test suite while building their applications. This provides a safety net for the whole system, offering reasonable confidence that no part of the code is broken. As a result, TDD helps alleviate the fear of changing the code.

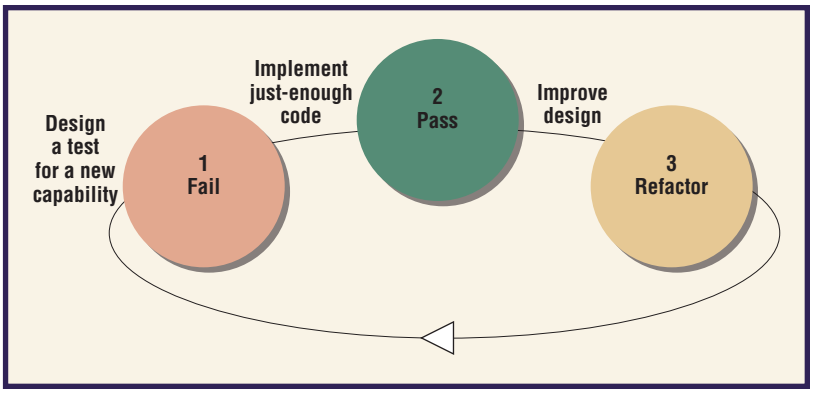


Figure 1. The test-driven-development step cycle: design a failing test, implement code to pass the test, and improve the design via refactoring.

In the past, most developers programmed by first writing code and then testing it. We often performed the tests manually and often gave only a cursory look at whether we had broken any past tests. In spite of what now seems like careless work, we were always surprised when someone found a bug in our code. We might have chalked it up to "just a mistake" or vowed to try harder and be more careful next time. Those tricks rarely worked.

With TDD, things are different. Automated tests specify and constrain each functional bit of the program. While these tests tend to prevent errors and detect them when they do occur, when an error does come up, our best response is to write the test that was missing—the test that would have prevented the defect.

Programmers using TDD become justly confident in the code. As we become more confident, we can relax more as we work. Less stressed, we can focus more on quality because we're keeping fewer balls in the air. We become practiced in thinking about what might not work, at testing whether it does, and at making it work.

Joyous programming

The TDD approach extends the assertion Boris Beizer made in 1983: "The act of designing tests is one of the most effective bug preventers known."[3] As a practice, TDD first appeared as part of the Extreme Programming discipline, described in Kent Beck's *Extreme Programming Explained*, which came out in 1999. In 2002, Beck released *Test-Driven Development: By Example*, and Dave Astels followed soon after with *Test-Driven Development: A Practical Guide*. More books appeared, covering various aspects of the technique, specific tools, and project experiences (see the "Recommended Books" sidebar). TDD

Recommended Books

Kent Beck, *TDD by Example*, Addison-Wesley, 2002 (introductory)
- TDD primer with a basic example, idealized situation, baby-step demonstration. Appropriate for both academia and practitioners new to TDD.

David Astels, *Test Driven Development: A Practical Guide*, Prentice Hall, 2003 (intermediate)
- Relentlessly practical TDD "how-to" guide with real problems, real solutions, and real code (including building a GUI test-first). Also includes an excellent overview of tools and introduction to mock objects.

James Newkirk and Alexey Vorontzov, *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004 (intermediate)
- Extending TDD to more realistic scenarios (code with Web interfaces, databases, and so on)

Ron Jeffries, *Extreme Programming Adventures in C#*, Microsoft Press, 2004 (introductory)
- Trip through a software engineering project with an expert sitting beside you, sharing the triumphs and failures.

Rick Mugridge and Ward Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall, 2005 (introductory-intermediate)
- Primary resource on customer acceptance testing; half the book written for business experts and the other half for developers.

Martin Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999 (introductory)
- Fundamental introduction to refactoring.

Joshua Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004 (intermediate)
- Guide to improving the design of existing code with pattern-directed refactorings.

Michael Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2004 (advanced)
- Start-to-finish strategies for working effectively with large, untested legacy code bases.

J.B. Rainsberger, *JUnit Recipes*, Manning Publications, 2004 (advanced)
- Concrete, practical expert advice for writing good programmer tests.

Gerard Meszaros, *XUnit Test Patterns*, Addison-Wesley, 2007 (intermediate-advanced)
- Guide on refactoring of both programmer and customer tests; includes a catalog of "smells" with root cause analysis and possible solutions.

Brian Marick, *Everyday Scripting with Ruby: For Teams, Testers, and Users*, Pragmatic Bookshelf, 2007 (introductory)
- Introduction to test-driving Ruby scripts and beyond.

tools now exist for almost every computer language you can imagine, from C++ to Visual Basic, all the major scripting languages, and even some of the more exotic languages—current and past.

TDD has caught the attention of a large software development community that finds it to be a good, fast way to develop reliable code—and many of us find it to be an enjoyable way to work. It embodies elements of design, testing, and coding in a cyclical style based on one fundamental rule: never write a line of code except what's necessary to make the current test pass. The process might sound tedious in the telling, but the practice is rhythmic, quite pleasant, and productive. The swing from test to code to test occurs as frequently as every five or 10 minutes. It's been compared to a waltz, to the smooth grace of skating, and to the seemingly effortless movements of a yin-style martial art. As in all these analogous situations, the practitioner is fully engaged and concentrating, while the work just seems to flow. And like these other arts, the only way to really understand TDD is to practice it.

System-level TDD

Applied at a higher level, TDD is known as executable acceptance TDD[4] or storytest-driven development,[5] and it helps with requirements discovery, clarification, and communication. Customers, domain experts, or analysts specify tests before the features are implemented. Once the code is written, the tests serve as executable acceptance criteria. In this issue, Jennitta Andrea makes a case for better acceptance testing tools in her article, "Envisioning the Next Generation of Functional Testing Tools" (pp. 58–66).

Each test amounts to the first use of a planned new capability. This helps the developer focus on the code in actual use, not just

as implemented. We can often improve a new capability's design when we have a chance to see what we're creating from the first user's viewpoint. Each test makes us think concretely about how the proposed new feature will behave. What are suitable inputs? What behavior will be executed? How will we know what happened? When we turn to writing the code a few minutes later, the concrete example helps us focus on what the code needs to do.

The process is self-correcting. If the tests are too simple, which is rare, the workflow will feel choppy and without challenge. This will encourage the developer to take larger bites. On the other hand, if the tests are too difficult, the longer time between successfully passed tests will alert us that we might be off track.

Once developers gain some skill in TDD, they commonly report less stress during development, better requirements understanding, lower defect insertion rates, less rework, and, as a result, faster production of higher-quality code. Once "test infected," as TDD aficionados are called, a developer rarely wants to go back to the old ways.

TDD practice has a special value as part of agile methods, which are all characterized by iterative delivery of increasingly capable system versions in short cycles—usually fixed lengths of a couple of weeks to a month. At the beginning, a simple architecture and simple design are sufficient to support the system's capability. As it grows, however, the architecture and design need continuous improvement. Agile practitioners might or might not have the complete design in mind from the beginning. Either way, to deliver working system versions every few weeks, they must grow the complete design incrementally, not all at once.

Improving the design incrementally is the refactoring step in figure 1. It brings the whole design back into alignment—now just a little bit bigger and better. Changing the design in continual small steps is a good thing, in that we can deliver tangible features as we go along. But frequent design changes also carry the risk that we'll break something that used to work.

The tests we write with TDD have been built, one by one, to cause some new software property to exist and to show that it works. So, as we refactor, we can run all the tests to verify that everything that should work, in fact, still does work. This makes TDD a powerful asset to incremental software development. It becomes a rule of software development hygiene. Robert C. Martin argues for that in his article, "Professionalism and Test-Driven Development," pp. 32–36.

The state of TDD research

TDD also intrigues the research community, and a growing number of studies have investigated its effects. Tables 1 and 2 reflect the current state of TDD research, summarizing the productivity and quality impacts of industry and academic work, respectively.[6–23] The results are sometimes controversial (more so in the academic studies). This is no surprise, given incomparable measurements and the difficulty in isolating TDD's effects from many other context variables. In addition, many studies don't have the statistical power to allow for generalizations. So, we advise readers to consider empirical findings within each study's context and environment. We also invite more researchers to methodically investigate TDD practice and report on its effects, both positive and negative.

All researchers seem to agree that TDD encourages better task focus and test coverage. The mere fact of more tests doesn't necessarily mean that software quality will be better, but the increased programmer attention to test design is nevertheless encouraging. If we view testing as sampling a very large population of potential behaviors, more tests mean a more thorough sample. To the extent that each test can find an important problem that none of the others can find, the tests are useful, especially if you can run them cheaply.

TDD is also making its way to university and college curricula. The IEEE/ACM 2004 guidelines for software engineering undergraduate programs includes test-first as a desirable skill.[24] Educators report success stories when using TDD in computer science programming assignments. In this issue, Bas Vodde and Lasse Koskela describe an effective exercise for introducing TDD to novices—practitioners or students—in their article, "Learning Test-Driven Development by Counting Lines" (pp. 74–79).

Other articles in this special issue give you a taste of TDD's use in diverse and nontrivial contexts: control system design (see Thomas Dohmke and Henrik Gollee, "Test-Driven De-

Once "test infected," as TDD aficionados are called, a developer rarely wants to go back to the old ways.

Table 1

A summary of selected empirical studies of test-driven development: industry participants*

Family of studies	Type	Development time analyzed	Legacy project?	Organization studied	Software built	Software size	No. of participants	Language	Productivity effect	Quality effect
Sanchez et al.[6]	Case study	5 years	Yes	IBM	Point-of-sale device driver	Medium	9–17	Java	Increased effort 19%	40%†
Bhat and Nagappan[7]	Case study	4 months	No	Microsoft	Windows networking common library	Small	6	C/C++	Increased effort 25–35%	62%†
	Case study	≈7 months	No	Microsoft	MSN Web services	Medium	5–8	C++/C#	Increased effort 15%	76%†
Canfora et al.[8]	Controlled experiment	5 hours	No	Soluziona Software Factory	Text analyzer	Very small	28	Java	Increased effort by 65%	Inconclusive based on quality of test
Damm and Lundberg[9]	Multi-case study	1–1.5 years	Yes	Ericsson	Components for a mobile network operator application	Medium	100	C++/Java	Total project cost increased by 5–6%	5–30% decrease in fault-slip-through rate; 55% decrease in avoidable fault costs
Melis et al.[10]	Simulation	49 days (simulated)	No	Calibrated using Klondike-Team and Quinary data	Market information project	Medium	4‡	Smalltalk	Increased effort 17%	36% reduction in residual defect density
Mann[11]	Case study	8 months	Yes	PetroSleuth	Windows-based oil and gas project management with statistical modeling elements	Medium	4–7	C#	n/a	81%§; customer and developers' perception of improved quality
Geras et al.[12]	Quasi-controlled experiment	3 hours	No	Various companies	Simple database-backed business information system	Small	14	Java	No effect	Inconclusive based on failure rates; improved based on no. of tests and frequency of execution
George and Williams[13]	Quasi-controlled experiment	4.75 hours	No	John Deere, Role Model Software, Ericsson	Bowling game	Very small	24	Java	Increased effort 16%	18%#
Ynchausti[14]	Case study	8.5 hours	No	Monster Consulting	Coding exercises	Small	5	n/a	Increased effort 60–100%	38–267%†

Green box = improvement; orange box = deterioration
†Reduction in the internal defect density
‡Simulated in 200 runs
§Reduction in external defect ratio (can't be solely attributed to TDD, but to a set of practices)
#Increase in percentage of functional black-box tests passed (external quality)

Table 2

A summary of selected empirical studies of TDD: academic participants*

Family of studies	Type	Development time analyzed	Legacy project?	Organization studied	Software built	Software size	No. of participants	Language	Productivity effect	Quality effect
Flohr and Schneider[15]	Quasi-controlled experiment	40 hours	Yes	University of Hannover	Graphical workflow library	Small	18	Java	Improved productivity by 27%	Inconclusive
Abrahamsson et al.[16]	Case study	30 days	No	VTT	Mobile application for global markets	Small	4	Java	Increased effort by 0% (iteration 5) to 30% (iteration 1)	No value perceived by developers
Erdogmus et al.[17]	Controlled experiment	13 hours	No	Politecnico di Torino	Bowling game	Very small	24	Java	Improved normalized productivity by 22%	No difference
Madeyski[18]	Quasi-controlled experiment	12 hours	No	Wroclaw University of Technology	Accounting application	Small	188	Java	n/a	–25 to –45%†
Melnik and Maurer[19]	Multi-case study	4-month projects over 3 years	No	University of Calgary/ SAIT Polytechnic	Various Web-based systems (surveying, event scheduling, price consolidation, travel mapping)	Small	240	Java	n/a	73% of respondents perceive TDD improves quality
Edwards[20]	Artifact analysis	2–3 weeks	No	Virginia Tech	CS1 programming assignment	Very small	118	Java	Increased effort 90%	45%†
Pančur et al.[21]	Controlled experiment	4.5 months	No	University of Ljubljana	4 programming assignments	Very small	38	Java	n/a	No difference
George[22]	Quasi-controlled experiment	1-3/4 hours	No	North Carolina State University	Bowling game	Very small	138	Java	Increased effort 16%	16%†
Müller and Hagner[23]	Quasi-controlled experiment	10 hours	No	University of Karlsruhe	Graph library	Very small	19	Java	No effect	No effect, but better reuse and improved program understanding

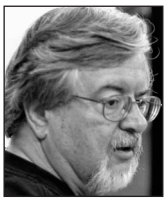
Green box = improvement; orange box = deterioration
†Increase in percentage of functional black-box tests passed (external quality)

velopment of a PID Controller," pp. 44–50), GUI development (Alex Ruiz and Yvonne Wang Price, "Test-Driven GUI Development with TestNG and Abbot," pp. 51–57), and database development (Scott W. Ambler, "Test-Driven Development of Relational Databases," pp. 37–43). In addition, Michael J. Johnson, Chih-Wei Ho, E. Michael Maximilien, and Laurie Williams inspect the aspect of incorporating performance testing in TDD (pp. 67–73). In Point/Counterpoint (pp. 80–83), Steve Freeman and Nat Pryce debate Joshua Kerievsky on the role of mock objects in test-driving code.

TDD is becoming popular across all sizes and kinds of software development projects. A Cutter Consortium survey of companies about various software process improvement practices identified TDD as the practice with the second-highest impact on project success (after code inspections).[25] Of course, like any other programming tool or technique, TDD is no silver bullet. However, it can help you become a more effective and disciplined developer—fearless and joyful, too.

Happy reading!

About the Authors



Ron Jeffries is an independent Extreme Programming author, trainer, coach, and practitioner and proprietor of www.xprogramming.com, one of the longest-running and certainly the largest one-person site on XP, comprising over 200 articles at this time. His research interests center on agile software development. He's one of the 17 original authors and signatories of the Agile Manifesto (www.agilemanifesto.org). He has master's degrees in mathematics and in computer and communication science. Contact him at ronjeffries@acm.org.

Grigori Melnik is a software engineer, researcher, and educator, currently affiliated with the University of Calgary and SAIT Polytechnic. His research interests include empirical evaluation of agile methods, executable acceptance-test-driven development, domain-driven design, e-business software engineering, the Semantic Web, and distributed cognition in software teams. He's the research chair of the Agile 2007 conference and the program academic chair of Agile 2008. Contact him at melnik@cpsc.ucalgary.ca.

Acknowledgments

We thank the 32 groups of authors who responded to our call for papers. Selecting seven articles from this pool would have been impossible without reviewers who contributed their expertise and effort. Our profound gratitude goes to all of them.

References

1. D. Astels, "A New Look at Test-Driven Development," 2006, http://blog.daveastels.com/files/BDD_Intro.pdf.
2. B. Marick, "Driving Software Projects with Examples," 3 July 2004, www.exampler.com.
3. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold Electrical, 1983.
4. F. Maurer and G. Melnik, "Driving Software Development with Executable Acceptance Tests," *Cutter Consortium Report*, vol. 7, no. 11, 2006, pp. 1–30.
5. T. Reppert, "Don't Just Break Software, Make Software: How Storytest Driven Development Is Changing the Way QA, Customers, and Developers Work," *Better Software*, vol. 9, no. 6, 2004, pp. 18–23.
6. J. Sanchez, L. Williams, and E.M. Maximilien, "A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry," to appear in *Proc. Agile 2007 Conf.*, IEEE CS Press, 2007.
7. T. Bhat and N. Nagappan, "Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies," *Proc. Int'l Symp. Empirical Software Eng.* (ISESE 06), ACM Press, 2006, pp. 356–363.
8. A. Canfora et al., "Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals," *Proc. Int'l Symp. Empirical Software Eng.* (ISESE 06), ACM Press, 2006, pp. 364–371.
9. L. Damm and L. Lundberg, "Results from Introducing Component-level Test Automation and Test-Driven Development," *J. Systems and Software*, vol. 79, no. 7, 2006, pp. 1001–1014.
10. M. Melis et al., "Evaluating the Impact of Test-First Programming and Pair Programming through Software Process Simulation," *J. Software Process Improvement and Practice*, vol. 11, 2006, pp. 345–360.
11. C. Mann, "An Exploratory Longitudinal Case Study of Agile Methods in a Small Software Company," master's thesis, Dept. Computer Science, Univ. of Calgary, 2004.
12. A. Geras et al., "A Prototype Empirical Evaluation of Test Driven Development," *Proc. 10th Int'l Symp. Software Metrics*, (METRICS 04), IEEE CS Press, 2004, pp. 405–416.
13. B. George and L. Williams, "An Initial Investigation of Test Driven Development in Industry," *Proc. ACM Symp. Applied Computing*, ACM Press, 2003, pp. 1135–1139.
14. R.A. Ynchausti, "Integrating Unit Testing into a Software Development Team's Process," *Proc. 2nd Int'l Conf. Extreme Programming and Flexible Processes in Software Eng.* (XP 01), 2001, pp. 79–83.
15. T. Flohr and T. Schneider, "Lessons Learned from an XP Experiment with Students: Test-First Needs More Teachings," *Proc. 7th Int'l Conf. Product Focused Software Process Improvement* (Profes 06), LNCS 4034, Springer, 2006, pp. 305–318.
16. P. Abrahamsson, A. Hanhineva, and J. Jäälinoja, "Improving Business Agility through Technical Solutions: A Case Study on Test-Driven Development in Mobile Software Development, Business Agility and Information Technology Diffusion," *IFIP TC8 WG 8.6 Int'l Working Conf.*, Int'l Federation for Information Processing, 2005, pp. 1–17.
17. H. Erdogmus et al., "On the Effectiveness of the Test-First Approach to Programming," *IEEE Trans. Software Eng.*, vol. 31, no. 3, 2005, pp. 226–237.
18. L. Madeyski, "Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality," *Software Engineering: Evolution and Emerging Technologies*, K. Zielinski and T. Szmuc, eds., IOS Press, 2005, pp. 113–123.
19. G. Melnik and F. Maurer, "A Cross-Program Investigation of Students' Perceptions of Agile Methods," *Proc. 2nd Int'l Conf. Software Eng.* (ICSE 05), ACM Press, 2005, pp. 481–489.
20. S. Edwards, "Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action," *ACM SIGSCE Bull.*, 2004, pp. 26–30.
21. M. Pančur et al., "Towards Empirical Evaluation of Test-Driven Development in a University Environment," *IEEE Region 8 Proc. EUROCON 2003*, vol. 2, IEEE Press, 2003, pp. 83–86.
22. B. George, "Analysis and Quantification of Test Driven Development Approach," master's thesis, Dept. Computer Science, N. Carolina State Univ., 2002.
23. M. Müller and O. Hagner, "Experiment about Test-First Programming," *IEE Proc. Software*, vol. 149, no. 5, 2002, pp. 131–136.
24. Joint Task Force on Computing Curricula, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, tech. report, IEEE CS and ACM, 2004; http://sites.computer.org/ccse.
25. K. El Emam, "Evaluating ROI from Software Quality," *Cutter Consortium Report*, vol. 5, no. 1, 2004, p. 20.