TEST-DRIVEN DEVELOPMENT

Professionalism and Test-Driven Development
by Robert C. Martin, pp. 32–36. A professional software developer ships clean, flexible code that works—in time. Unfortunately, many software developers use high-stress heroics to ship late, buggy, messy, and bloated code. Test-driven development is a discipline that helps developers behave in a more professional manner.

Test-Driven Development of Relational Databases
by Scott W. Ambler, pp. 37–43. Developers can use test-driven development with database schema just as they use it with application code. Implementing test-driven database development (TDDD) involves three relatively simple steps: database refactoring, database regression testing, and continuous database integration. From a technical viewpoint, TDDD is straightforward. However, cultural challenges can make it difficult to adopt.

Test-Driven Development of a PID Controller
by Thomas Dohmke and Henrik Gollee, pp. 44–50. With test-driven development, developers don’t write new code until an automated test has failed, and duplicate functions, tests, or code fragments are always removed. TDD can lead to better-designed, higher-quality systems. sUnit combines the features of the xUnit testing frameworks and the Simulink graphical programming language to apply TDD to control-system design. Development of a controller for a simplified vehicle system illustrates this approach.

Test-Driven GUI Development with TestNG and Abbot
by Alex Ruiz and Yvonne Wang Price, pp 51–57. Regardless of test-driven development’s well-known benefits, it has suffered slow adoption in GUI development. Because GUIs are one of an application’s most important components, testing them is essential for improving the entire system’s safety and robustness. Testing GUIs is difficult, but several recommendations and practices can simplify test-driven GUI development for Java Swing applications.

Envisioning the Next Generation of Functional Testing Tools
by Jennitta Andrea, pp. 58–66. An explanation of test-driven development often begins by describing the red-green-refactor cycle. This slogan is so catchy and the description so simple that practitioners and tool developers tend to focus only on this localized cycle. Experience has shown that a successful functional test-driven development strategy must span the entire application life cycle and must be supported by effective tools. This article is a call to improve the state of the art of functional test-driven development by reflecting on the state of the art, describing processes and phases in the full application life cycle, and painting a vision for the next generation of functional testing tools.

Incorporating Performance Testing in Test-Driven Development
by Michael J. Johnson, Chih-Wei Ho, E. Michael Maximilien, and Laurie Williams, pp. 67–73. Performance design and performance testing are necessarily different from functional test case design. A rigorous test-driven design methodology isn’t practical for all performance measurement. A test-first approach to performance provides some advantages in a TDD environment. Experience with applying early performance testing in a TDD framework for a device-driver development project provides insight into the test-first approach. The results show a trend of performance improvement throughout the development life cycle and better performance compared to an earlier release.

Learning Test-Driven Development by Counting Lines
by Bas Vodde and Lasse Koskela, pp. 74–79. Test-driven development is an agile development practice that changes every minute of developers’ daily lives. That’s a big change! How can you best train developers in such a pervasive practice? One method is to write tests for counting lines of code. While developing the tests, developers run into problems that force them to reevaluate their design. This experience provides valuable insights into TDD and its benefits.

FEATURES

Determining Criteria for Selecting Software Components: Lessons Learned
by Juan Pablo Carvallo, Xavier Franch, and Carme Quer, pp. 84–94. Software component selection relies on the suitability and completeness of the criteria used to evaluate and compare the components. Experiences from determining such criteria for several industrial projects provide important lessons.

Collaboration, Process Control, and Fragility in Evolutionary Product Development
Tor Erlend Fægri and Geir Kjetil Hanssen, pp. 96–104. This descriptive, longitudinal case study reports on the experiences from a medium-sized packaged software product supplier in their improvisation-like transition from a plan-based process to Evo—an agile, evolutionary process. The study focuses on the organizational implications for the supplier, including customer collaboration, process control, and psychological comfort. Evolutionary product development improves stakeholder engagement and developer responsiveness over plan-based processes but requires cross-organizational discipline and involved management.