Test-Driven Development: Concepts, Taxonomy, and Future Direction

Test-driven development creates software in very short iterations with minimal upfront design. Poised for widespread adoption, TDD has become the focus of an increasing number of researchers and developers.

David Janzen
Simex LLC

Hossein Saiedian
University of Kansas

The *test-driven development* strategy requires writing automated tests prior to developing functional code in small, rapid iterations. Although developers have been applying TDD in various forms for several decades,[1] this software development strategy has continued to gain increased attention as one of the core extreme programming practices.

XP is an *agile method* that develops object-oriented software in very short iterations with little upfront design. Although not originally given this name, TDD was described as an integral XP practice necessary for analysis, design, and testing that also enables design through refactoring, collective ownership, continuous integration, and programmer courage.

Along with pair programming and refactoring, TDD has received considerable individual attention since XP's introduction. Developers have created tools specifically to support TDD across a range of languages, and they have written numerous books explaining how to apply TDD concepts. Researchers have begun to examine TDD's effects on defect reduction and quality improvements in academic and professional practitioner environments, and educators have started to examine how to integrate TDD into computer science and software engineering pedagogy. Some of these efforts have been implemented in the context of XP projects, while others are independent of them.

TEST-DRIVEN DEVELOPMENT DEFINED

Although its name implies that TDD is a testing method, a close examination of the term reveals a more complex picture.

The test aspect

In addition to testing, TDD involves writing automated tests of a program's individual units. A unit is the smallest possible testable software component. There is some debate about what exactly constitutes a unit in software. Even within the realm of object-oriented programming, both the class and method have been suggested as the appropriate unit. Generally, however, the method or procedure is the smallest possible testable software component.

Developers frequently implement test drivers and function stubs to support the execution of unit tests. Test execution can be either a manual or automated process and can be performed by developers or designated testers. Automated testing involves writing unit tests as code and placing this code in a test harness or framework such as JUnit. Automated unit testing frameworks minimize the effort of testing, reducing a large number of tests to a click of a button. In contrast, during manual test execution developers and testers must expend effort proportional to the number of tests executed.

Traditionally, unit testing occurred after developers coded the unit. This can take anywhere from a few minutes to a few months. The unit tests might be written by the same programmer or by a designated tester. With TDD, the programmer writes the unit tests prior to the code under test. As a result, the programmer can immediately execute the tests after they are written.

0018-9162/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

TDD assumes that the software design is either incomplete or pliable and open to changes.

The driven aspect

Some definitions of TDD imply that it is primarily a testing strategy. For example, in *JUnit in Action* (Manning Publications, 2003), Vincent Massol and Ted Husted stated that

> Test-driven development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed, and to eliminate duplication. The goal of TDD is 'clean code that works.'

However, according to XP and TDD pioneer Ward Cunningham, "Test-first coding is not a testing technique." TDD is known by various names including test-first programming, test-driven design, and test-first design. The *driven* in test-driven development focuses on how TDD leads analysis, design, and programming decisions. TDD assumes that the software design is either incomplete or pliable and open to changes. In the context of XP, TDD subsumes many analysis decisions. The customer should be "on-site" in XP. Test writing is one of the first steps in deciding what the program should do, which is essentially an analysis step. The Agile Alliance offers another definition that captures this idea (www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm):

> Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

To promote testing to an analysis and design step the practice of refactoring must be introduced. Refactoring changes the structure of an existing body of code without changing its external behavior. A test may pass, but the code may be inflexible or overly complex. By refactoring the code, the test should still pass but the code will be improved.

Understanding that TDD is more about analysis and design than it is about testing is one of the most challenging conceptual shifts for new adopters of the practice. Program testing has traditionally assumed the existence of a program. The TDD idea that a test can be written before the program or that test can aid in deciding what program code to write and what that program's interface should look like is a radical concept for most software developers.

The development aspect

Intended to aid in constructing software, TDD is not in itself a software development methodology or process model. It is a practice, a way of developing software to be used with other practices, in a particular order and frequency and in the context of a process model. TDD has emerged within a particular set of process models. It can be applied as a microprocess within the context of many different process models.

TDD produces a set of automated unit tests that provide some side effects in the development process. The practice assumes the automated tests will not be thrown away once a design decision is made. Instead, the tests become a vital component of the development process, providing quick feedback to any changes to the system. If a change causes a test to fail, the developer knows immediately after making the change, while the test is still fresh in the developer's mind. Among the drawbacks, that developer must now maintain both the production code and the automated tests.

TDD'S HISTORICAL AND MODERN CONTEXTS

Despite the lack of attention in undergraduate curriculum and inconclusive reports of usage in industry, a wide range of software tools exist to support testing, making TDD's emergence possible.

Software development methodologies

A software development process or methodology defines the order, control, and evaluation of the basic tasks involved in creating software. Software process methodologies range in complexity and control from largely informal to highly structured. Developers classify these methodologies as prescriptive or agile and label the specific types as waterfall, spiral, incremental, or evolutionary.

When an organization states that it uses a particular methodology, it often applies a combination of smaller, finer-grained methodologies on a project scale instead. For example, an organization might apply an incremental development model, building small, cumulative slices of a project's features. In each increment, the developers could apply a waterfall or linear method of determining requirements, designing a solution, coding, testing, and then integrating. Depending on the size of the increments and the waterfall's time frame, the process could be labeled differently, with potentially very different results in terms of quality and developer

satisfaction. If we break a software project into N increments where I_i represents each increment, then the equation $\Sigma_{i=1}^{N} I_i$ can represent the entire project. If N is reasonably large, we can label this as an incremental project. However, if $N \leq 2$, we would label this as a waterfall project.

If the increments require modifying a significant amount of overlapping software, we can say that our methodology is more iterative in nature. Specifically, for project P consisting of code C and iterations $I=\Sigma_{i=1}^{N} I_i$, if C_i is the code affected by iteration I_i, and if project P is iterative, then $C_i \cap C_{i+1} \neq \Theta$ for most i such that $1 < i < N$.

Similarly, with incremental and waterfall approaches, we expect a formal artifact, such as a specification document, for documenting the increment's requirements. If the artifact is informal—say, some whiteboard drawings or an incomplete set of UML diagrams and was generated quickly—we would be working in the context of an agile process. The approach and perspective of the architecture or the design would cause us to label the process aspect-oriented, component-based, or feature-driven.

Some individual software developers and smaller teams apply even finer-grained models such as the personal software process or the collaborative software process. The time, formality, and intersection of the steps in software construction can determine the way developers categorize a process methodology.

The order in which construction tasks occur influences a project's label and its quality. Natural and logical, the traditional ordering is requirements elicitation, analysis, design, code, test, integration, deployment, and maintenance. We could, however, consider some possible reorderings even though most do not make sense. For example, we would never maintain a system that hasn't been coded. Similarly, we would never code something for which we have no requirements.

Requirements do not necessarily imply formal requirements. A requirement can be as simple as an idea in a programmer's head. Programmers have applied the prototyping approach when requirements are fuzzy or incomplete. With this approach, we may do very little analysis and design before coding, but ultimately might have to discard the prototype even though it was a useful tool in determining requirements and evaluating design options.

When we closely examine the design, code, and test phases, we see many finer-grained activities. For example, various types of testing take place, including unit, integration, and regression testing. The timing, frequency, and granularity of these tests can vary widely. Some testing can be conducted early—concurrent with other coding activities. Test-driven development reorders these steps to an advantage. By placing fine-grained unit tests before just enough code to satisfy that test, TDD can affect many aspects of a software development methodology.

The order in which construction tasks occur influences a project's label and its quality.

TDD's historical context

Test-driven development has emerged in conjunction with the rise of agile process models. Both have roots in the iterative, incremental, and evolutionary process models used as early as the 1950s. In addition, tools have evolved to play a significant role in supporting TDD.

Early test, early examples. Research on testing has generally assumed the existence of a program to be tested, implying a test-last approach. Moving tests from the end of coding to the beginning, however, is nothing new. Software and test teams commonly develop tests early in the software development process, often with the program logic. The evaluation and prevention life-cycle models integrated testing early in the software development process nearly two decades ago. Introduced in the 1980s, the Cleanroom approach to software engineering included formal verification of design elements early in the development process. Some claim that NASA's Project Mercury applied a form of TDD as early as the 1950s.[1]

Prior to the introduction of XP in 1998, little had been written about the concept of letting small incremental automated unit tests *drive* software development and design processes. Despite the lack of published documentation, many developers have probably used a test-first approach informally. Kent Beck claims he "learned test-first programming as a kid while reading a book on programming. It said that you program by taking the input tape ... and typing in the output tape you expect. Then you program until you get the output tape you expect."[2]

Some argue that TDD merely gives a name and definition to a practice that has been sporadically and informally applied for some time. TDD is more than this. As Beck states, XP takes the known best practices and "turns the knobs all the way up to ten."[2] Many developers might have been thinking and coding in a test-first manner, but TDD does this in an extreme way: always writing tests before code, making tests as small as possible, and never letting code degrade. TDD fits within a process model, and the development of incremental, iterative, and evolutionary process models has been vital to its emergence.

TDD developed within the context of iterative, incremental, and evolutionary models.

Iterative, evolutionary, and incremental development. *Iterative* development involves repeating a set of development tasks, generally on an expanding set of requirements.[1] *Evolutionary* approaches involve adaptive and lightweight iterative development.

Being *adaptive* refers to using feedback from previous iterations to improve the software. Being *lightweight* refers to the lack of complete specifications at the beginning of development, thus allowing feedback from previous iterations and from customers to guide future iterations. Lightweight can also refer to other aspects such as a process's level of formality and degree of documentation. The *spiral model* is an evolutionary approach that incorporates prototyping and the cyclic nature of iterative development with *risk-driven-iterations* and *anchor point milestones*. The incremental model produces a series of releases, called *increments*, that provide more functionality with each increment.

TDD developed within the context of such iterative, incremental, and evolutionary models. TDD works because these approaches provide the prerequisite process models. Beck claims that to implement XP, developers must apply all of the incumbent practices—leaving some out weakens the model and can cause it to fail.[3] TDD requires that design decisions be delayed and flexible to influence software design. Each new test might require refactoring and a design change. Automated tests give programmers the courage to change any code and the information they need to know quickly if something has broken, enabling collective ownership.

Automated testing

Software tools have become important factors in the development of modern software systems. Tools ranging from compilers, debuggers, and integrated development environments to modeling and computer-aided software engineering tools have improved and increased developer productivity.

Tools have played an important role in the emergence of TDD, which assumes the existence of an automated unit testing framework. Such a framework simplifies both the creation and execution of software unit tests. *Test harnesses*, basically automated testing frameworks, provide a combination of test drivers, stubs, and interfaces to other subsystems. Often such harnesses are custom-built, although commercial tools do exist to assist with test harness preparation.

Erich Gamma and Kent Beck developed JUnit, an automated unit testing framework for Java. Essential for implementing TDD with Java, JUnit is arguably responsible for much of TDD and XP's wide popularity. JUnit-like frameworks have been implemented for several different languages, creating a family of frameworks referred to as xUnit.

Generally, xUnit lets a programmer write sets of automated unit tests that initialize, execute, and make assertions about the code under test. Individual tests are independent of one another so test order does not matter. The programmer reports the total number of successes and failures. xUnit tests are written in the same language as the code under test and thus serve as first-class clients of the code, while tests can actually serve as documentation for it.

On the other hand, because developers implement xUnit in the target language, that language determines the tool's relative simplicity and flexibility. For example, JUnit is simple and portable partly because it takes advantage of Java's portability through the bytecode/virtual machine architecture. It uses Java's ability to load classes dynamically and exploits Java's reflection mechanism to automatically discover tests. In addition, JUnit provides a portable graphical user interface that has been integrated into popular integrated development environments such as Eclipse.

A wide range of additional tools have emerged to support automated testing, particularly in Java. Some tools simplify the creation of mock objects, or stubs. The stubs replace the needed collaborating objects so that developers can test a particular object. They can use other tools such as Cactus and Derby with JUnit to automate tests that involve J2EE components or databases.

The proliferation of software tools that support TDD shows that it has widespread support and will likely become an established approach. JUnit's simplicity and elegance have been a significant factor in TDD's use, particularly in the Java community. Programmers can develop unit tests easily and execute large test suites with a single button click, yielding quick results about the system's state.

Early testing in academia

The undergraduate computer science and software engineering curriculum provides one indicator of a software practice's widespread acceptance. Sometimes academia has led practice in the field, sometimes it has followed. Software engineering, iterative development, and TDD all seem to follow this latter model.

Much software engineering research originated in academia and then found its way into common

practice. The undergraduate computer science and software engineering curriculum, however, tends to reflect or even lag behind common practice in industry. The choice of programming language has commonly followed business needs. Process models developed in practice later become reflected in curricula.

The 1991 ACM Curriculum Guidelines recommended giving fewer than eight hours each of lecture and lab time to iterative development processes and verification and validation. The 2001 guidelines recommended giving an even smaller amount of time to development processes and software validation—two and three hours each, respectively.

Undergraduate texts give little attention to comparative process models. Texts provide limited coverage of software design and testing techniques. The topics of software design and testing are often relegated to a software engineering course that may not be mandatory for all students.

Extreme programming's place in undergraduate education has been the topic of much debate. Some argue strongly for using XP to introduce software engineering to undergraduates. Others argue that XP and agile methods offer only limited benefits. Given the different opinions on using XP in the undergraduate curriculum, TDD has received limited exposure at this level. Some educators have called for increased design and testing coverage. Others see TDD as an opportunity to incorporate testing in the curriculum, rather than relegating it to an individual course.

TDD tools have, however, found their way into early programming education. BlueJ, a popular environment for learning Java, incorporates JUnit and adds help for building test cases at an early stage in a programmer's learning cycle. Proponents have advocated using JUnit for early learning of Java because it abstracts the bootstrapping mechanism of `main()`, allowing the student to concentrate on the use of objects early.

TDD has yet to achieve widespread acceptance in academia, at least partly because faculty who do not specialize in software engineering are not likely to be familiar with it. TDD instructional materials that target undergraduate courses remain basically nonexistent.

Recent context

XP and agile methods have received much attention in the past few years. Even though conclusive documentation is lacking, anecdotal evidence indicates that TDD usage is rising.

Agile methods. These methods clearly have roots in the incremental, iterative, and evolutionary methods. Pekka Abrahamsson and colleagues[4] provide an evolutionary map of nine agile methods and describe how they focus on simplicity and speed while emphasizing people over processes.

XP proposes using TDD as an integral component for developing high-quality software.

Probably the most well-known agile method, XP is often used in combination with other agile methods such as Scrum. XP proposes using TDD as an integral component for developing high-quality software. The highly disciplined practice of TDD and the simple, lightweight nature of agile processes give rise to an interesting conflict. Potential TDD adopters often express concern regarding the time and cost of writing and maintaining unit tests. Although he concedes that automated unit tests are not necessary for absolutely everything, Beck insists that XP cannot work without TDD because it provides the glue that holds the process together.[3]

Adoption measures. Measuring the use of a particular software development methodology is hard. Many organizations might be using the methodology without talking about it. Others might claim to be using a methodology when in fact they are misapplying it. Worse yet, they might be advertising its use falsely. Surveys might be conducted to gauge a method's usage, but often only those who are much in favor or much opposed to the methodology will respond.

A 2002 survey reported that out of 32 survey respondents across 10 industry segments, 14 firms used an agile process.[5] Of these, five were in the e-business industry. Most of the projects using agile processes were small, involving 10 or fewer participants and lasting one year or less. A 2003 survey reported that 131 respondents claimed they used an agile method.[6] Of these, 59 percent claimed to be using XP and implied they were using TDD. Both surveys revealed positive results from applying agile methods, with increases in productivity and quality and reduced or minimal changes in costs.

XP has accumulated a substantial body of literature. Most of this involves the promotion of XP or explains how to implement it. Many experience reports present only anecdotal evidence of XP's benefits and drawbacks. Although the existence of these reports indicates that XP is being adopted in many organizations, it remains unclear if these same organizations will continue to use XP over time or, if they have, if they will move on to other methods.

Although XP's popularity implies a growing adoption of TDD, we have no idea how widely it is being used. Organizations may be using XP with-

Table 1. Summary of TDD research in industry.

Study	Type	Number of companies	Number of programmers	Quality effects	Productivity effects
George[8]	Controlled experiment	3	24	TDD passed 18% more tests	TDD took 16% longer
Maximilien[9]	Case study	1	9	50% reduction in defect density	Minimal impact
Williams[10]	Case study	1	9	40% reduction in defect density	No change

out adopting all of its practices or they may be applying the practices inconsistently. On a project at ThoughtWorks, Jonathan Rasmusson, an early XP adopter, estimates one-third of the code was developed using TDD.[7] In the same report, Rasmusson stated, "If I could only recommend one coding practice to software developers, those who use XP or otherwise, it would be to write unit tests."

In this ThoughtWorks project, developers used 16,000 lines of automated unit tests on 21,000 lines of production code. Many tests were written in both test-first and test-last iterations.

Despite the possibility of adopting XP without it, TDD seems to be a core XP practice. Anecdotal evidence indicates that TDD is commonly included when only a subset of XP is adopted.

The use of xUnit testing frameworks provides another possible indicator of TDD's use. JUnit, the first such framework, has enjoyed widespread popularity (www.junit.org). No JUnit adoption statistics are directly available. The Eclipse core distribution, a popular integrated development environment primarily used for Java development includes JUnit. A press release issued in February 2004 on the Eclipse Web site states that the Eclipse platform has recorded more than 18 million download requests since its inception. Although duplicate requests from the same developer can occur, the figure is still substantial. Certainly not all Eclipse developers use JUnit, nor do all JUnit adopters use TDD, but it is likely that the popularity of XP, JUnit, and Eclipse combined implies a certain degree of TDD adoption.

EVALUATIVE TDD RESEARCH

Since the introduction of XP, many practitioner articles and books on applying TDD have been written. There has been relatively little evaluative research on the benefits and effects of TDD, however.

Research on TDD can be categorized broadly by context. In particular, TDD research is classified as industry if the study or research was conducted primarily with professional software practitioners. It is classified as academia if the practitioners are primarily students and the work takes place in the context of an academic setting. Academic research also includes studies in which students work on a project for a company but in the context of an academic course.

TDD in industry

A few evaluative research studies have been conducted on TDD with professional practitioners. North Carolina State University seems to be the only source of such a study to date. Researchers at NCSU have performed at least three empirical studies on TDD in industry settings involving fairly small groups in at least four different companies.[8-10] These studies examined defect density as a measure of software quality, although some survey data indicated that programmers thought TDD promoted simpler designs. In one study, programmers' experience with TDD varied from novice to expert, while programmers new to TDD participated in the other studies.

These studies showed that programmers using TDD produced code that passed 18 percent to 50 percent more external test cases than code produced by corresponding control groups. The studies also reported less time spent debugging code developed with TDD. Further, they reported that applying TDD had an impact that ranged from minimal to a 16 percent decrease in programmer productivity—which shows that applying TDD sometimes took longer. In the case that took 16 percent more time, researchers noted that the control group wrote far fewer tests than the TDD group.

Table 1 summarizes these studies and labels each experiment as either a case study or a controlled experiment.

TDD in academia

Several academic studies have examined XP as a whole, but a few focused on TDD. Although many of the TDD studies published in academic settings are anecdotal, the five studies shown in Table 2 specifically report on empirical results. When referring to software quality, all but one[11] study focused on the ability of TDD to detect defects early. Two of the five studies reported significant improvement in software quality and programmer productivity.[11,12] One reported a correlation between the number of tests written and productivity.[13] In this particular study, students using test-first methods wrote more tests and were significantly more productive. The remaining two studies[14,15] reported no significant improvement in either defect density or productivity.

All five of these relatively small studies lasted a semester or less and involved programmers who had little or no previous experience with TDD.

Table 2. Summary of TDD research in academia.

Controlled experiment	Number of programmers	Quality effects	Productivity effects
Kaufmann[11]	8	Improved information flow	50% improvement
Edwards[12]	59	54% fewer defects	n/a
Erdogmus[13]	35	No change	Improved productivity
Müller[14]	19	No change, but better reuse	No change
Pančur[15]	38	No change	No change

FACTORS IN SOFTWARE PRACTICE ADOPTION

A variety of factors play into the widespread adoption of a software practice. These include motivation for change, economics, availability of tools, training and instructional materials, a sound theoretical basis, empirical and anecdotal evidence of success, time, and even endorsements of the practice by highly regarded individuals or groups.

These factors complicate TDD's current state. Current software development practice provides a clear motivation for change and thus TDD seems poised for growth. Software development involves a complex mix of people, processes, technology, and tools that struggle to find consistency and predictability. Projects continue to go over schedule and budget, which makes practitioners eager to find improved methods.

Tool support for TDD is strong and improving for most modern languages. Tools such as JUnit, MockObjects, and Cactus are mature and widely available. Much of this tool development has targeted Java, an increasingly popular language in both commercial applications and academia.

Economic models have noted the potential for positive improvements for XP and TDD, but recognized that additional research is needed. This is especially true regarding speed and defects and when TDD is combined with pair programming. The interplay between academic and industry practitioners for acceptance is an interesting one. Research indicates that it takes five to 15 years for academic development to succeed in commercial practice—and the reverse holds true. Research shows how TDD can improve programming pedagogy, yet few instructional resources exist. The JUnit incorporation into BlueJ and the corresponding programming textbook indicates improvement may be on its way, however.

The adoption of TDD faces many challenges. First, TDD requires a good deal of discipline on the programmer's part. Hence, programmers may require compelling reasons before they try it. Second, TDD is still widely misunderstood, perhaps because of its name, but many still think erroneously that TDD addresses testing only, and not design. Third, TDD doesn't fit every situation. Developers and managers must determine when to apply TDD and when to do something else.

Additional research and the availability of training and instructional materials will likely play an important role in determining how widespread TDD will become.

UNDERSTANDING TDD'S EFFECTS

Further research must be done to determine and understand TDD's effects. To date, research has focused on TDD as a testing technique to lower defect density. Empirical studies should be conducted to evaluate TDD's effect on software design quality and to examine characteristics such as extensibility, reusability, and maintainability.

Even with the focus on defect density, there have been only a small number of studies conducted, and those on only small samples. One industry study with more than 10 participants involved a small application that took only one day to complete. The results were suspect because the control group wrote a minimal number of tests.

The few academic studies that have examined defect density produced inconsistent results. The largest study reported a 54 percent reduction in defect density with beginning programmers. Two other reasonably large studies with advanced programmers did not provide any significant reduction in defect density. One study hinted at better designs.

Future studies should consider the effectiveness of TDD at varying levels in the curriculum and the programmer's maturity. The studies can also examine how TDD compares to test-last methods that fix the design ahead of time, as well as iterative test-last methods that build an emergent design.

XP-EF,[16] a framework for consistently conducting and assessing case studies on XP projects, currently under development, seems appropriate for adaptation into a TDD case studies framework. Given such a framework, researchers can conduct multiple case studies and controlled studies. Programmer productivity and software quality should be examined. Adoption issues such as learning curves, suitability and fit, and motivation must be addressed. Additionally, research is needed to examine the effects of combining TDD with other practices such as pair programming and code inspection.

Both industry and academic settings can benefit from more research. In particular, academic stud-

ies need to examine whether TDD improves or at least does not hinder learning. Can TDD be incorporated into the undergraduate curriculum in a way that improves students' ability to design and test? If so, then TDD must be written into appropriate student texts and lab materials.

Even if XP fades in popularity, TDD may persist. Additional research is needed on TDD's ability to improve software quality and on its place in undergraduate computer science and software engineering curricula. If TDD finds its way into academia, students could enter software organizations with increased discipline and improved software design and testing skills, increasing the software engineering community's ability to reliably produce, reuse, and maintain quality software.

References

1. C. Larman and V.R. Basili, "Iterative and Incremental Development: A Brief History," *Computer*, June 2003, pp. 47-56.
2. K. Beck, "Aim, Fire," *IEEE Software*, Sept./Oct. 2001, pp. 87-89.
3. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
4. P. Abrahamsson et al., "New Directions on Agile Methods: A Comparative Analysis," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 244-254.
5. D.J. Reifer, "How Good are Agile Methods?" *IEEE Software*, July/Aug. 2002, pp. 16-18.
6. Shine Technologies, "Agile Methodologies Survey Results," 17 Jan. 2003; www.shinetech.com/download/attachments/98/ShineTechAgileSurvey2003-01-17.pdf?version=1.
7. J. Rasmusson, "Introducing XP into Greenfield Projects: Lessons Learned," *IEEE Software*, May/June, 2003, pp. 21-28.
8. B. George and L. Williams, "A Structured Experiment of Test-Driven Development," *Information and Software Technology*, vol. 46, no. 5, 2004, pp. 337-342.
9. E.M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 564-569.
10. L. Williams, E.M. Maximilien, and M. Vouk, "Test-Driven Development as a Defect-Reduction Practice," *Proc. 14th Int'l Symp. Software Reliability Eng.* (ISSRE 03), IEEE Press, 2003, pp. 34-45.
11. R. Kaufmann and D. Janzen, "Implications of Test-Driven Development: A Pilot Study," *Companion of the 18th Ann. ACM Sigplan Conf. Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, 2003, pp. 298-299.
12. S.H. Edwards, "Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance." *Proc. Int'l Conf. Education and Information Systems: Technologies and Applications* (EISTA 03), Aug. 2003; http://web-cat.cs.vt.edu/grader/Edwards-EISTA03.pdf.
13. H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of Test-First Approach to Programming," *IEEE Trans. Software Eng.*, Mar. 2005, pp. 226-237.
14. M.M. Müller and O. Hagner, "Experiment about Test-First Programming," *IEE Proc. Software*, IEE Publications, 2002, pp. 131-136.
15. M. Pančur et al., "Towards Empirical Evaluation of Test-Driven Development in a University Environment," *Proc. Eurocon 2003: The Computer as a Tool*, The IEEE Region 8, vol. 2, 2003, pp. 83-86.
16. L. Williams, L. Layman, and W. Krebs, *Extreme Programming Evaluation Framework for Object-Oriented Languages*, v. 1.4, tech. report TR-2004-18, North Carolina State Univ., 2004.

David Janzen *is the president of Simex LLC, a consulting and training firm, and an associate professor of computer science at Bethel College. He is currently a PhD candidate at the University of Kansas, where he received an MS in computer science. Janzen is a member of the IEEE Computer Society and the ACM. Contact him at david@simexusa.com.*

Hossein Saiedian *is a professor and associate chair in the Department of Electrical Engineering and Computer Science, and a member of the Information and Telecommunication Technology Center at the University of Kansas. His research interests include software process improvement, object technology, and software architecture. Saiedian received a PhD in computer science from Kansas State University. He is a senior member of the IEEE, and a member of the IEEE Computer Society and the ACM. Contact him at saiedian@eecs.ku.edu.*

Editor: Diomidis Spinellis
Athens University of Economics and Business, dds@aueb.gr

Continuous Integration and Its Tools

Mathias Meyer

CONTINUOUS INTEGRATION HAS been around for quite some time now. The idea was originally conceived as part of the practices collectively known as extreme programming. At its very core, continuous integration is a set of principles that apply to the daily workflow of development teams. Let's walk through the basics.

All code must be kept in a repository. Git is a common tool, but you'll also find Subversion, Perforce, Mercurial, and an abundance of other choices in active use today. Surprisingly, you'll still find places where using a code repository and versioning the source code aren't yet part of common practice. In a continuous integration life cycle, when someone checks his or her revised code into the repository, an automated system picks up the change, checks out the code, and runs a set of commands to verify that the change is good and didn't break anything. This tool is meant to be the unbiased judge of whether a change works or not, thereby preventing the "it works on my machine" syndrome before the code hits production.

For all this to work, the build must be automated, and thanks to modern Web frameworks and build tools, this is an easy-to-achieve task. A build ideally involves more than just compiling—it should also include a thorough test suite to help verify that the code still works with every change. The objective is to increase test coverage with every new change, but also to keep the test suite fast. If tests run longer than 10 minutes, developer productivity drops, slowing down the process of shipping new features or bug fixes to the customer.

Commit Daily, Commit Often

A core practice of continuous integration is that all developers commit to the mainline branch daily. With modern, distributed version control systems such as Git, the temptation is high to work in a local branch, maybe pushing your work to a remote repository and eventually merging your changes. What if, instead, your developers made all their changes on the mainline branch? Some even go as far as discarding work that's not committed by the end of the day, to make sure the habit really sticks.

When your team makes changes in smaller increments and integrates them into the mainline regularly, the benefits go beyond just the development process. Smaller changes, shipped to production quickly, are a lot easier to debug when something breaks. This is one of the key values of continuous integration, and it's where the name comes from. Rather than living in branches for long chunks of time, changes are continuously integrated.

Build Early, Build Often

How do you verify whether someone's changes broke something in the code or whether the changes work in the larger

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools

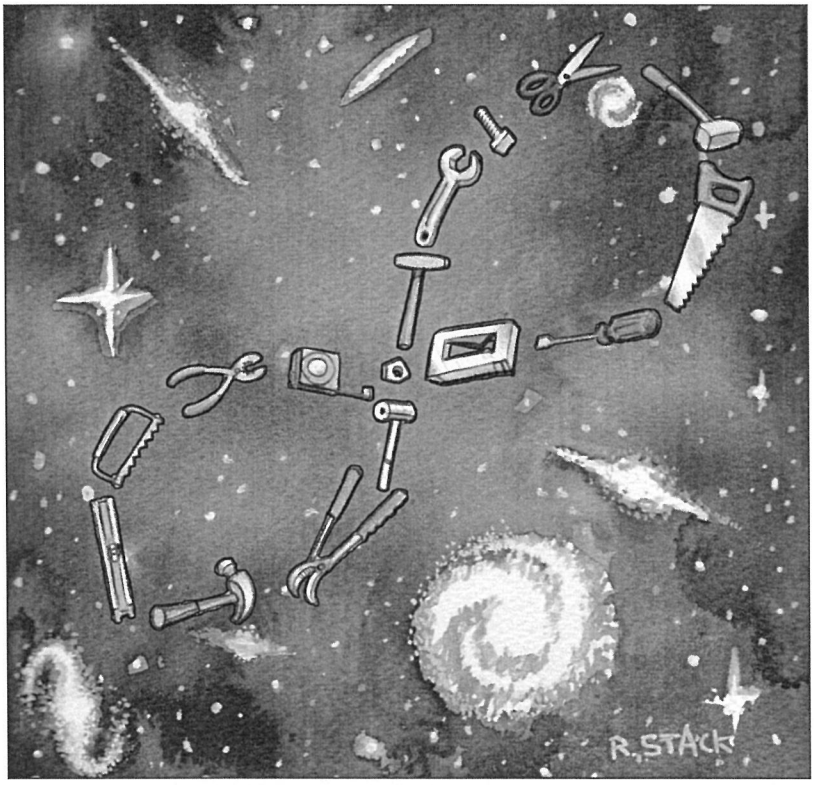
0740-7459/14/$31.00 © 2014 IEEE

context of the entire codebase? Beyond version control, a continuous integration server is one of the more important tools a development team can put to good use. Its sole purpose is to check the code repository for changes, check out the code if it spots any, and run a list of commands to trigger the build.

The definition of a build varies widely depending on the language and framework used. With compiled languages such as Java, C++, Erlang, and the like, a build is a compilation step with an additional run of a test suite. For dynamic languages such as Ruby, Python, and JavaScript, a test suite is commonly the only thing that's run as part of the build.

A continuous integration server is unbiased. Its tasks boil down to telling the team whether the most recent changes still pass the stages it's configured to run. An abundance of tools are available, from self-hosted systems such as Jenkins, TeamCity, and Bamboo to hosted products like CloudBees and Travis CI. They're all very easy to set up, so there's no excuse not to use a continuous integration server for your projects.

The last step of a fully automated build is the ability to deploy to production, which requires an automated deployment process that every developer should be able to run just like the continuous integration server. With an automated build in place, everyone can deploy to staging or production, anytime. What if this could even become part of your build process? Imagine shipping all new code directly to the staging system for instant testing!



Keep It Green

With everyone committing to the mainline, a green build is an important responsibility. A green build is the result of a code change passing all the steps involved in the build successfully. When developers break the build, they're blocking everyone else on the team from committing their changes with confidence. When the build is broken and other developers keep making changes and committing them, it's a lot harder to figure out whether or not they failed the build, making debugging the failed build harder with every new commit.

At Toyota, everyone on the factory floor can halt the production line if something breaks or holds them up. In your development team, a broken build should spur a similar habit. Getting it back to green should be the highest priority. A green build is your insurance policy that changes can be deployed to production at anytime.

A Matter of Culture

Beyond having good tooling and a fast test suite in place, continuous integration is really about fostering responsibility and providing customer value. The server to run the build is but a small part of it. The practices that continuous integration includes have a much bigger picture in mind: a responsible team that ensures new features can be shipped to production and the customer quickly, and that makes sure all members can get their work done and check in new code.

When new changes are continuously merged into the mainline branch, there's a deeper responsibility for everyone on the team to make sure that those changes not only pass the build but that they also have minimal impact on the

production environment. Rather than build new and bigger features on long-lived feature branches, developers start thinking about how they can build them in ways that let them merge their work into the mainline branch regularly. Feature flips are a popular way of reducing the impact of new features and providing ample means to try them out before shipping to all customers. They have the added benefit that when something breaks in production, features can easily be switched off to reduce the load on the system while the issue is investigated.

But how do you notice that something is broken? How can you know that a change doesn't have a negative impact on the production environment or on your customers? Good monitoring is essential: logging, metrics, alerting, and exception tracking will help you ship changes with confidence. Thanks to an abundance of hosted services, it's just as easy to get started with monitoring as it is with continuous integration servers.

Feature flips and monitoring aren't practices originally considered with continuous integration, but they're invaluable additions to the process of shipping early and often. With a responsible team focused on shipping and increasing customer value, continuous integration can be a catalyst for long-lasting change, just by adding a few simple practices to your team's workflow. When you add these habits together along with feature flips, fully automated and fast deploys, and monitoring, you're ready to ship value to your customers anytime.

MATHIAS MEYER works at Travis CI, a hosted continuous integration and deployment platform for open source and private repositories. Contact him at meyer@paperplanes.de or via Twitter (@roidrage).

See **www.computer.org/software-multimedia** for multimedia content related to this article.