Test-driven development

From Wikipedia, the free encyclopedia

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: Requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

American software engineer Kent Beck, who is credited with having developed or "rediscovered"[1] the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.[2]

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999,[3] but more recently has created more general interest in its own right.[4]

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.[5]

Contents

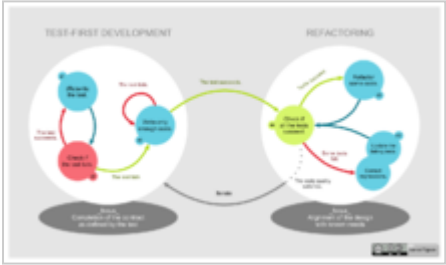
Test-driven development cycle

The following sequence is based on the book *Test-Driven Development by Example*.[2]

1. Add a test

In test-driven development, each new feature begins with writing a test. Write a test that defines a function or improvements of a function, which should be very succinct. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing

framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.



A graphical representation of the test-driven development lifecycle

2. Run all tests and see if the new test fails

This validates that the test harness is working correctly, shows that the new test does not pass without requiring new code because the required behavior already exists, and it rules out the possibility that the new test is flawed and will always pass. The new test should fail for the expected reason. This step increases the developer's confidence in the new test.

3. Write the code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test. The programmer must not write code that is beyond the functionality that the test checks.

4. Run tests

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and for creating clean code.[6][7] By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself,[4] unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

Development style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You aren't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods.[2] In *Test-Driven Development by Example*, Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept such as a design pattern, tests are written that generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Writing the tests first: The tests should be written before the functionality that is to be tested. This has been claimed to have many benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset rather than adding it later. It also ensures that tests for every feature get written. Additionally, writing the tests first leads to a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on software quality.[8] When writing feature-first code, there is a tendency by developers and organisations to push the developer on to the next feature, even neglecting testing entirely. The first TDD test might not even compile at first, because the classes and methods it requires may not yet exist. Nevertheless, that first test functions as the beginning of an executable specification.[9]

Each test case fails initially: This ensures that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has led to the "test-driven development mantra", which is "red/green/refactor", where red means *fail* and green means *pass*. Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the developer's mental model of the code, boosts confidence and increases productivity.

Keep the unit small

For TDD, a unit is most commonly defined as a class, or a group of related functions often called a module. Keeping units relatively small is claimed to provide critical benefits, including:

- Reduced debugging effort – When test failures are detected, having smaller units aids in tracking down errors.
- Self-documenting tests – Small test cases are easier to read and to understand.[8]

Advanced practices of test-driven development can lead to acceptance test–driven development (ATDD) and Specification by example where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process.[10] This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy – the acceptance tests – which keeps them continuously focused on what the customer really wants from each user story.

Best practices

Test structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.

- Setup: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- Execution: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT & UAT.

- Cleanup: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.[8]

Individual best practices

Individual best practices states that one should:

- Separate common set-up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test oracle focused on only the results necessary to validate its test.
- Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits. It may be helpful to review this section during your discussion.[11]

Practices to avoid, or "anti-patterns"

- Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.
- Testing precise execution behavior timing or performance.
- Building “all-knowing oracles.” An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.[11]
- Testing implementation details.
- Slow running tests.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.[12] Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.[13]

Programmers using pure TDD on new ("greenfield") projects reported they only rarely felt the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.[14]

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program.[15] By focusing on the test cases first, one must imagine how the functionality is used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to design by contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, the total code implementation time could be shorter based on a model by Müller and Padberg.[16] Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Madeyski[17] provided an empirical evidence (via a series of laboratory experiments with over 200 developers) regarding the superiority of the TDD practice over the classic Test-Last approach, with respect to the lower coupling between objects (CBO). The mean effect size represents a medium (but close to large) effect on the basis of meta-analysis of the performed experiments which is a substantial finding. It suggests a better modularization (i.e., a more modular design), easier reuse and testing of the developed software products due to the TDD programming practice.[17] Madeyski also measured the effect of the TDD practice on unit tests using branch coverage (BC) and mutation score indicator (MSI),[18][19][20] which are indicators of the thoroughness and the fault detection effectiveness of unit tests, respectively. The effect size of TDD on branch coverage was medium in size and therefore is considered substantive effect.[17]

Limitations

Test-driven development does not perform sufficient testing in situations where full functional tests are required to determine success or failure, due to extensive use of unit tests.[21] Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.[22]

Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.[23]

Unit tests created in a test-driven development environment are typically created by the developer who is writing the code being tested. Therefore, the tests may share blind spots with the code: if, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify those parameters. Another example: if the developer misinterprets the requirements for the module he is developing, the code and the unit tests he writes will both be wrong in the same way. Therefore, the tests will pass, giving a false sense of correctness.

A high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.

Tests become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or are themselves prone to failure, are expensive to maintain. This is especially the case with fragile tests.[24] There is a risk that tests that regularly generate false failures will be ignored, so that

when a real failure occurs, it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.

Writing and maintaining an excessive number of tests costs time. Also, more-flexible modules (with limited tests) might accept new requirements without the need for changing the tests. For those reasons, testing for only extreme conditions, or a small sample of data, can be easier to adjust than a set of highly detailed tests.

The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original, or early, tests become increasingly precious as time goes by. The tactic is to fix it early. Also, if a poor architecture, a poor design, or a poor testing strategy leads to a late change that makes dozens of existing tests fail, then it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Test-driven work

Test-driven development has been adopted outside of software development, in both product and service teams, as test-driven work.[25] Similar to TDD, non-software teams develop quality control (QC) checks (usually manual tests rather than automated tests) for each aspect of the work prior to commencing. These QC checks are then used to inform the design and validate the associated outcomes. The six steps of the TDD sequence are applied with minor semantic changes:

1. "Add a check" replaces "Add a test"
2. "Run all checks" replaces "Run all tests"
3. "Do the work" replaces "Write some code"
4. "Run all checks" replaces "Run tests"
5. "Clean up the work" replaces "Refactor code"
6. "Repeat"

TDD and ATDD

Test-driven development is related to, but different from acceptance test–driven development (ATDD).[26] TDD is primarily a developer's tool to help create well-written unit of code (function, class, or module) that correctly performs a set of operations. ATDD is a communication tool between the customer, developer, and tester to ensure that the requirements are well-defined. TDD requires test automation. ATDD does not, although automation helps with regression testing. Tests used in TDD can often be derived from ATDD tests, since the code units implement some portion of a requirement. ATDD tests should be readable by the customer. TDD tests do not need to be.

TDD and BDD

BDD (behavior-driven development) combines practices from TDD and from ATDD.[27] It includes the practice of writing tests first, but focuses on tests which describe behavior, rather than tests which test a unit of implementation. Tools such as Mspec and Specflow provide a syntax which allow non-programmers to define the behaviors which developers can then translate into automated tests.

Code visibility

Test suite code clearly has to be able to access the code it is testing. On the other hand, normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

In object oriented design this still does not provide access to private data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access private fields and methods.[28] Alternatively, an inner class can be used to hold the unit tests so they have visibility of the enclosing class's members and attributes. In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code. This means the released code is not exactly the same as what was unit tested. The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can ensure (among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private methods and data anyway. Some argue that private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Thus it should be sufficient to test any class through its public interface or through its subclass interface, which some languages call the "protected" interface.[29] Others say that crucial aspects of functionality may be implemented in private methods and testing them directly offers advantage of smaller and more direct unit tests.[30][31]

Software for TDD

There are many testing frameworks and tools that are useful in TDD.

xUnit frameworks

Developers may use computer-assisted testing frameworks, such as xUnit created in 1998, to create and automatically run the test cases. Xunit frameworks provide assertion-style test validation capabilities and result reporting. These capabilities are critical for automation as they move the burden of execution validation from an independent post-processing activity to one that is included in the test execution. The execution framework provided by these test frameworks allows for the automatic execution of all system test cases or various subsets along with other features.[32]

TAP results

Testing frameworks may accept unit test output in the language-agnostic Test Anything Protocol created in 1987.

Fakes, mocks and integration tests

Unit tests are so named because they each test *one unit* of code. A complex module may have a thousand unit tests and a simple module may have only ten. The unit tests used for TDD should never cross process boundaries in a program, let alone network connections. Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite. Introducing dependencies on external modules or data also turns *unit tests* into *integration tests*. If one module misbehaves in a chain of interrelated modules, it is not so immediately clear where to look for the cause of the failure.

When code under development relies on a database, a web service, or any other external process or service, enforcing a unit-testable separation is also an opportunity and a driving force to design more modular, more testable and more reusable code.[33] Two steps are necessary:

1. Whenever external access is needed in the final design, an interface should be defined that describes the access available. See the dependency inversion principle for a discussion of the benefits of doing this regardless of TDD.

2. The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock. Fake objects need do little more than add a message such as “Person object saved” to a trace log, against which a test assertion can be run to verify correct behaviour. Mock objects differ in that they themselves contain test assertions that can make the test fail, for example, if the person's name and other data are not as expected.

Fake and mock object methods that return data, ostensibly from a data store or user, can help the test process by always returning the same, realistic data that tests can rely upon. They can also be set into predefined fault modes so that error-handling routines can be developed and reliably tested. In a fault mode, a method may return an invalid, incomplete or null response, or may throw an exception. Fake services other than data stores may also be useful in TDD: A fake encryption service may not, in fact, encrypt the data passed; a fake random number service may always return 1. Fake or mock implementations are examples of dependency injection.

A Test Double is a test-specific capability that substitutes for a system capability, typically a class or function, that the UUT depends on. There are two times at which test doubles can be introduced into a system: link and execution. Link time substitution is when the test double is compiled into the load module, which is executed to validate testing. This approach is typically used when running in an environment other than the target environment that requires doubles for the hardware level code for compilation. The alternative to linker substitution is run-time substitution in which the real functionality is replaced during the execution of a test case. This substitution is typically done through the reassignment of known function pointers or object replacement.

Test doubles are of a number of different types and varying complexities:

- Dummy – A dummy is the simplest form of a test double. It facilitates linker time substitution by providing a default return value where required.
- Stub – A stub adds simplistic logic to a dummy, providing different outputs.
- Spy – A spy captures and makes available parameter and state information, publishing accessors to test code for private information allowing for more advanced state validation.
- Mock – A mock is specified by an individual test case to validate test-specific behavior, checking parameter values and call sequencing.
- Simulator – A simulator is a comprehensive component providing a higher-fidelity approximation of the target capability (the thing being doubled). A simulator typically requires significant additional development effort.[8]

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the "real" implementations of the interfaces discussed above. These are integration tests and are quite separate from the TDD unit tests. There are fewer of them, and they must be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

Integration tests that alter any persistent store or database should always be designed carefully with consideration of the initial and final state of the files or database, even if any test fails. This is often achieved using some combination of the following techniques:

- The `TearDown` method, which is integral to many test frameworks.
- `try...catch...finally` exception handling structures where available.
- Database transactions where a transaction atomically includes perhaps a write, a read and a matching delete operation.
- Taking a "snapshot" of the database before running any tests and rolling back to the snapshot after each test run. This may be automated using a framework such as Ant or NAnt or a continuous integration system such as CruiseControl.
- Initialising the database to a clean state *before* tests, rather than cleaning up *after* them. This may be relevant where cleaning up may make it difficult to diagnose test failures by deleting the final state of the database before detailed diagnosis can be performed.

TDD for complex systems

Exercising TDD on large, challenging systems requires a modular architecture, well-defined components with published interfaces, and disciplined system layering with maximization of platform independence. These proven practices yield increased testability and facilitate the application of build and test automation.[8]

Designing for testability

Complex systems require an architecture that meets a range of requirements. A key subset of these requirements includes support for the complete and effective testing of the system. Effective modular design yields components that share traits essential for effective TDD.

- High Cohesion ensures each unit provides a set of related capabilities and makes the tests of those capabilities easier to maintain.
- Low Coupling allows each unit to be effectively tested in isolation.
- Published Interfaces restrict Component access and serve as contact points for tests, facilitating test creation and ensuring the highest fidelity between test and production unit configuration.

A key technique for building effective modular architecture is Scenario Modeling where a set of sequence charts is constructed, each one focusing on a single system-level execution scenario. The Scenario Model provides an excellent vehicle for creating the strategy of interactions between components in response to a specific stimulus. Each of these Scenario Models serves as a rich set of requirements for the services or functions that a component must provide, and it also dictates the order that these components and services interact together. Scenario modeling can greatly facilitate the construction of TDD tests for a complex system.[8]

Managing tests for large teams

In a larger system the impact of poor component quality is magnified by the complexity of interactions. This magnification makes the benefits of TDD accrue even faster in the context of larger projects. However, the complexity of the total population of tests can become a problem in itself, eroding potential gains. It sounds simple, but a key initial step is to recognize that test code is also important software and should be produced and maintained with the same rigor as the production code.

Creating and managing the architecture of test software within a complex system is just as important as the core product architecture. Test drivers interact with the UUT, test doubles and the unit test framework.[8]

See also

- Acceptance testing
- Behavior-driven development
- Design by contract
- Inductive programming
- Integration testing
- List of software development philosophies
- List of unit testing frameworks
- Mock object
- Programming by example
- Sanity check
- Self-testing code
- Software testing
- Test case
- Transformation Priority Premise
- Unit testing
- Continuous test-driven development

References

1. Kent Beck (May 11, 2012). "Why does Kent Beck refer to the "rediscovery" of test-driven development?" (http://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development). Retrieved December 1, 2014.
2. Beck, K. Test-Driven Development by Example, Addison Wesley - Vaseem, 2003
3. Lee Copeland (December 2001). "Extreme Programming" (http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html). Computerworld. Retrieved January 11, 2011.

4. Newkirk, JW and Vorontsov, AA. *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004.
5. Feathers, M. Working Effectively with Legacy Code, Prentice Hall, 2004
6. Beck, Kent (1999). *XP Explained, 1st Edition*. Addison-Wesley Professional. p. 57. ISBN 0201616416.
7. Ottinger and Langr, Tim and Jeff. "Simple Design" (http://agileinaflash.blogspot.com/2009/02/simple-design.html). Retrieved 5 July 2013.
8. "Effective TDD for Complex Embedded Systems Whitepaper" (http://www.pathfindersolns.com/wp-content/uploads/2012/05/Effective-TDD-Executive-Summary.pdf) (PDF). Pathfinder Solutions.
9. "Agile Test Driven Development" (http://www.agilesherpa.org/agile_coach/engineering_practices/test_driven_development/). Agile Sherpa. 2010-08-03. Retrieved 2012-08-14.
10. Koskela, L. "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007
11. "Test-Driven Development for Complex Systems Overview Video" (http://www.pathfindersolns.com/resources/industry-glossary/tdd/test-driven-development-complex-systems-overview-video/). Pathfinder Solutions.
12. Erdogmus, Hakan; Morisio, Torchiano. "On the Effectiveness of Test-first Approach to Programming" (http://nparc.cisti-icist.nrc-cnrc.gc.ca/npsi/ctrl?action=shwart&index=an&req=5763742&lang=en). Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005. (NRC 47445). Retrieved 2008-01-14. "We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive."
13. Proffitt, Jacob. "TDD Proven Effective! Or is it?" (http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx). Retrieved 2008-02-21. "So TDD's relationship to quality is problematic at best. Its relationship to productivity is more interesting. I hope there's a follow-up study because the productivity numbers simply don't add up very well to me. There is an undeniable correlation between productivity and the number of tests, but that correlation is actually stronger in the non-TDD group (which had a single outlier compared to roughly half of the TDD group being outside the 95% band)."
14. Llopis, Noel (20 February 2005). "Stepping Through the Looking Glass: Test-Driven Game Development (Part 1)" (http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1). Games from Within. Retrieved 2007-11-01. "Comparing [TDD] to the non-test-driven development approach, you're replacing all the mental checking and debugger stepping with code that verifies that your program does exactly what you intended it to do."
15. Mayr, Herwig (2005). *Projekt Engineering Ingenieurmässige Softwareentwicklung in Projektgruppen* (2., neu bearb. Aufl. ed.). München: Fachbuchverl. Leipzig im Carl-Hanser-Verl. p. 239. ISBN 978-3446400702.
16. Müller, Matthias M.; Padberg, Frank. "About the Return on Investment of Test-Driven Development" (http://www.ipd.kit.edu/KarHPFn/papers/edser03.pdf) (PDF). Universität Karlsruhe, Germany. p. 6. Retrieved 2012-06-14.
17. Madeyski, L. "Test-Driven Development - An Empirical Evaluation of Agile Practice", Springer, 2010, ISBN 978-3-642-04287-4, pp. 1-245. DOI: 978-3-642-04288-1
18. The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment. (http://madeyski.e-informatyka.pl/download/Madeyski10c.pdf) by L. Madeyski *Information & Software Technology 52(2): 169-184 (2010)*
19. On the Effects of Pair Programming on Thoroughness and Fault-Finding Effectiveness of Unit Tests (http://madeyski.e-informatyka.pl/download/Madeyski07.pdf) by L. Madeyski *PROFES 2007: 207-221*
20. Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites. (http://madeyski.e-informatyka.pl/download/Madeyski08.pdf) by L. Madeyski *Software Process: Improvement and Practice 13(3): 281-295 (2008)*
21. "Problems with TDD" (http://dalkescientific.com/writings/diary/archive/2009/12/29/problems_with_tdd.html). Dalkescientific.com. 2009-12-29. Retrieved 2014-03-25.
22. Hunter, Andrew (2012-10-19). "Are Unit Tests Overused?" (https://www.simple-talk.com/dotnet/.net-framework/are-unit-tests-overused/). Simple-talk.com. Retrieved 2014-03-25.
23. Loughran, Steve (November 6, 2006). "Testing" (http://people.apache.org/~stevel/slides/testing.pdf) (PDF). HP Laboratories. Retrieved 2009-08-12.
24. "Fragile Tests" (http://xunitpatterns.com/Fragile%20Test.html).
25. Leybourn, E. (2013) *Directing the Agile Organisation: A Lean Approach to Business Management*. London: IT Governance Publishing: 176-179.
26. *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Boston: Addison Wesley Professional. 2011. ISBN 978-0321714084.
27. "BDD" (http://guide.agilealliance.org/guide/bdd.html). Retrieved 2015-04-28.

28. Burton, Ross (2003-11-12). "Subverting Java Access Protection for Unit Testing" (http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html). O'Reilly Media, Inc. Retrieved 2009-08-12.
29. van Rossum, Guido; Warsaw, Barry (5 July 2001). "PEP 8 -- Style Guide for Python Code" (https://www.python.org/dev/peps/pep-0008/). Python Software Foundation. Retrieved 6 May 2012.
30. Newkirk, James (7 June 2004). "Testing Private Methods/Member Variables - Should you or shouldn't you" (http://blogs.msdn.com/jamesnewkirk/archive/2004/06/07/150361.aspx). Microsoft Corporation. Retrieved 2009-08-12.
31. Stall, Tim (1 Mar 2005). "How to Test Private and Protected methods in .NET" (http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx). CodeProject. Retrieved 2009-08-12.
32. "Effective TDD for Complex, Embedded Systems Whitepaper" (http://www.pathfindersolns.com/download-whitepaper/?download=EffectiveTDDforComplexEmbeddedSystems.pdf&title=Effective_TDD_for_Complex_Embedded_Systems) (PDF). Pathfinder Solutions.
33. Fowler, Martin (1999). *Refactoring - Improving the design of existing code*. Boston: Addison Wesley Longman, Inc. ISBN 0-201-48567-2.

External links

- TestDrivenDevelopment on WikiWikiWeb
- Bertrand Meyer (September 2004). "Test or spec? Test and spec? Test from spec!". Archived from the original on 2005-02-09.
- Microsoft Visual Studio Team Test from a TDD approach
- Write Maintainable Unit Tests That Will Save You Time And Tears
- Improving Application Quality Using Test-Driven Development (TDD)