Test Driven Development

**Test-driven development** (**TDD**) is a [software development process](http://en.wikipedia.org/wiki/Software_development_process) that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated [test case](http://en.wikipedia.org/wiki/Test_case) that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally [refactors](http://en.wikipedia.org/wiki/Code_refactoring) the new code to acceptable standards.

TDD encourages simple designs and inspires confidence.[[1]](http://en.wikipedia.org/wiki/Test-driven_development)

Test-driven development is related to the test-first programming concepts of [extreme programming](http://en.wikipedia.org/wiki/Extreme_programming), begun in 1999,[[2]](http://en.wikipedia.org/wiki/Test-driven_development) but more recently has created more general interest in its own right.[[3]](http://en.wikipedia.org/wiki/Test-driven_development)

Programmers also apply the concept to improving and [debugging](http://en.wikipedia.org/wiki/Software_bug) [legacy code](http://en.wikipedia.org/wiki/Legacy_code) developed with older techniques.[[4]](http://en.wikipedia.org/wiki/Test-driven_development)

**Test-driven development cycle**



**1. Add a test**

**2. Run all tests and see if the new one fails**

**3. Write some code**

**The next step is to write some code that causes the test to pass.**

**4. Run tests**

**If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.**

**5. Refactor code**

**Now the code should be cleaned up as necessary.**

**Development style**

There are various aspects to using test-driven development, for example the principles of "keep it simple stupid" ([KISS](http://en.wikipedia.org/wiki/KISS_principle)) and "[You aren't HYPERLINK "http://en.wikipedia.org/wiki/You\_aren%27t\_gonna\_need\_it"gonna HYPERLINK "http://en.wikipedia.org/wiki/You\_aren%27t\_gonna\_need\_it" need it](http://en.wikipedia.org/wiki/You_aren%27t_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.[[1]](http://en.wikipedia.org/wiki/Test-driven_development) In [*Test-Driven Development by Example*](http://en.wikipedia.org/wiki/Test-Driven_Development_by_Example), Kent Beck also suggests the principle "[Fake it till you make it](http://en.wikipedia.org/wiki/Fake_it_till_you_make_it)".

To achieve some advanced design concept, such as a [design pattern](http://en.wikipedia.org/wiki/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.

Write the tests first. The tests should be written before the functionality that is being 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 worrying about it later. It also ensures that tests for every feature get written. Additionally, writing the tests first drives a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on the quality of the product.[[7]](http://en.wikipedia.org/wiki/Test-driven_development) When writing feature-first code, there is a tendency by developers and the development organisations to push the developer on to the next feature, neglecting testing entirely. The first test might not even compile, at first, because all of the classes and methods it requires may not yet exist. Nevertheless, that first test functions as an executable specification.[[8]](http://en.wikipedia.org/wiki/Test-driven_development)

First fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has been coined the "test-driven development mantra", known as 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 programmer'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 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 have improved readability and facilitate rapid understandability.

Advanced practices of test-driven development can lead to [Acceptance test-driven development](http://en.wikipedia.org/wiki/Acceptance_test-driven_development) (ATDD) and [Specification by example](http://en.wikipedia.org/wiki/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. 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 that user story.

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.
* 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.

<http://en.wikipedia.org/wiki/Test-driven_development>