Engineer

Test-driven development

writing the test before the code, one small test at a time

What it is

Test-driven development is a way of programming in which the test for a piece of behaviour is written before the code that provides it, one small test at a time. Kent Beck, who describes his own part in it as a rediscovery, calls it a programming workflow. In 2023, after finding that programmers did not agree on what the term meant, he set the workflow down in a short post titled Canon TDD, and he writes the abbreviation, TDD, almost everywhere in it.

The workflow has five steps:

- List the cases the change has to handle.
- Pick exactly one case from the list and write it as a concrete test that runs.
- Make that test pass without breaking any earlier one, and put on the list every new case the work brings to light.
- If the design of the implementation can be better, refactor it now; this step is optional.
- Return to step 2 until nothing is left on the list.

  The list is analysis of behaviour, not a design document: the basic case first, then the variations, then the ways the change must not break what already works. The test written in step 2 is a real automated test, with setup, an invocation and assertions, and it fails when it first runs, since the behaviour it describes is still to be written. Beck calls step 3 going from red to green, from a failing test to a passing one.

It is just as useful to know what the workflow is not. Writing every test before writing any code is a frequent complaint about TDD, and Beck answers it directly: that is not TDD, and a critique of it is a critique of something else. He is equally

relaxed in the other direction. A team that works differently and gets good results is welcome to, and he says there is no gold star for following the steps exactly.

Use the concept in a decision

Put “What is test-driven development?” to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

Beck states what the workflow is for. A programmer who changes the behaviour of a system should end up with a system in which the change preserves existing behaviour while satisfying the new requirements, the code is ready for the next change, and the programmer and their colleagues feel confident about all three. Because each test is written while its change is small and fresh, the safety net grows at the same pace as the code instead of being promised for later.

The second reason is design. Beck separates two kinds of design decision: how a piece of behaviour is invoked, and how the system implements it. Writing the test first forces the interface decisions, because the test has to call something and has to say what comes back. The implementation decisions wait for the refactoring step, when passing tests protect them. A test that is awkward to write is early news that the interface will be awkward to use.

The third reason is the double check. The expected value in a test is worked out by a person from the requirement, and the code arrives at it a second way. When the two agree, two independent routes have reached the same answer, and Beck says that double checking is where much of TDD’s value as a check comes from.

There is also a sense of when to stop. Beck’s advice is to keep testing and coding until fear about how the code behaves has turned into boredom. The habit is much older than the post. Beck places his rediscovery between 1994, when he wrote SUnit for a consulting client, and October 1995, when he demonstrated TDD to Ward Cunningham at the OOPSLA conference in Austin. His paper Simple

Smalltalk Testing describes a simple testing framework for Smalltalk, and Python's unittest documentation still points to it as Beck's original paper on the pattern that unittest shares.

What it looks like in practice

Take a content check that should report banned buzzwords in a website's copy. The work starts with the test list, written as plain notes:

```
- a banned word is reported
- a banned word with a capital letter is reported
```

Step 2 turns exactly one item into a test. The first item comes first for a reason: it gives the check a known banned word and asserts that the word is reported. The test fails until the check can find it. It goes in a file called `test_wording.py`, which pytest collects because the name matches `test_*.py`:

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

Running `pytest -q` fails at once, because there is no `find_banned` to import. That failure is the point: the test has been seen to fail, so a pass later on means something. The simplest code that makes it pass is enough for now, in `wording.py`:

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

The test passes. The next item becomes the next test:

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

It fails, and because pytest reports the intermediate values of an assertion, the output shows an empty list where `["Leverage"]` was expected. Comparing `word.lower()` instead of `word` makes it pass. While writing that change, the programmer notices that `text.split()` leaves punctuation attached, so "leverage," with a comma would slip through. That goes on the list rather than into the code, and becomes the third test:

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

It fails, and one more line makes all three pass:

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

Step 4 is optional, and here there is little to improve: two lines, each with one job. Every item on the list now has a passing test, so the work is done. Each expected value in the tests was written from the list item, never pasted from a run.

The order of that list was deliberate, and the site's own record shows why. A project that wrote tests for its own checkers, work carried out in August 2026, found that its custom checks, among them a spell check, a voice check and a figure-consistency check, had run green for months, and that a checker which has only ever seen clean input cannot be told apart from one that reads nothing. The fix was to plant the defect each checker exists to find: the spell check is fed a deliberately misspelled word and the test fails if the run comes back clean, the voice check is fed prose in the first person and must reject it, and the buzzword check is fed a banned word. Two of the checkers turned out to read a narrower set of files than their documentation claimed. The page does not describe TDD, but its lesson is the one TDD builds in from the first step: a test is worth something only once it has been seen to fail.

Where it goes wrong

All the tests are written first. Turning every item on the list into a test before any of them passes is the misreading Beck singles out. When the first test forces a rethink, every speculative test needs rework, and a programmer on the sixth test without a single pass, he writes, meets depression or boredom. Write one test, make it pass, then pick the next. If a new test shows that the work so far cannot handle a case, choose between carrying on and starting again; Beck's advice is to start again and take the cases in another order.

The test has no assertion. A test that calls the code and checks nothing raises the coverage figure and proves nothing. Every test needs setup, an invocation and an assertion, and Beck suggests trying, now and then, to write a test backwards from its assertions.

The assertion is deleted. When a test will not pass, deleting its assertion makes it pretend to. Make it pass for real, or take the item back to the list and rethink the order.

Expected values are pasted from the output. Running the code, copying what it returned and pasting that into the test turns a double check into a single one: the test now confirms only that the code does what it did. Work the expected value out from the requirement, by hand if necessary.

Refactoring while making it pass. Changing structure and behaviour in the same step is what Beck calls wearing two hats. Make it run, then make it right: reach green with the simplest change, and tidy up only once the tests pass.

Tidying too far, or abstracting too soon. Tidying feels good, and it can feel safer than facing the next test. Stop when the code is good enough for this session, and leave two similar pieces of code alone until the pattern is clear. Beck's line on this is short: "Duplication is a hint, not a command."

The tests know the implementation. Tests that call private methods or check which calls the code made, rather than what it produced, break whenever the internals change. Erik Kuefler's chapter on unit testing in *Software Engineering at Google* calls them brittle tests: they fail on an unrelated change that introduces no real bug. In TDD that is doubly costly, because the refactoring step

depends on tests that stay green while the internals move. Test through the public interface, check the resulting state rather than the interactions, and name each test after the behaviour it checks.

Further reading

- Canon TDD — Kent Beck's own statement of the workflow, from 2023: the five steps, the split between interface and implementation design, and the mistakes at each step.
- Simple Smalltalk Testing: With Patterns — Beck's paper on a simple testing framework for Smalltalk, with its fixtures, test cases and suites, in an archived copy.
- Unit Testing — Erik Kuefler's chapter of *Software Engineering at Google* on tests that survive refactoring, testing through public APIs and testing behaviours.
- Get Started — pytest's introduction: installing it, a first failing test with its report, and how test files are found.
- unittest: Unit testing framework — the Python standard library's framework, which names Beck's paper as the original description of the pattern it shares.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is test-driven development?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/test-driven-development/