Engineer

Refactoring

changing code's structure without changing what it does

What it is

Refactoring is changing the internal structure of code without changing what the code does for anyone who uses it. Names, function boundaries and the layout of modules change; inputs, outputs and side effects stay exactly as they were. The verb is to refactor, and one such change is also called a refactoring.

Martin Fowler defines the term in his book Refactoring, and restated the definition on his website in 2004. As a noun, a refactoring is one change to the inner structure of software that makes the code easier to understand and less costly to change, while what it does as seen from outside stays the same. As a verb, to refactor is to carry out a sequence of such changes, with that outside behaviour unchanged throughout. Both halves matter. The purpose is a cheaper next change, and the constraint is that nobody outside the code can tell that anything happened.

Fowler draws the line sharply: “Refactoring isn’t another word for cleaning up code,” he writes on refactoring.com. It names one specific technique. He uses restructuring as the wider word for reorganising code by any means, including ones that do not preserve behaviour at every step. What makes refactoring a technique is its size. Each step is a small transformation that does little by itself and is unlikely to go wrong, and the whole system still works after every one. A long sequence of them can still add up to a large change in structure.

The steps have names. The catalogue on refactoring.com, which accompanies the second edition of the book, lists them with the older names they replace: Extract Function, also known as Extract Method, and its inverse Inline Function; Inline Variable; Rename Variable; Move Function; Split Phase, and dozens more. A shared name lets a developer say what a change was in two words, and lets a reviewer check it against a known recipe.

Use the concept in a decision

Put “What is refactoring?” 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

Software that succeeds keeps changing. New features and fixes are layered on top of each other, and each one tends to make the next a little harder, until, as Fowler describes it, adding anything new takes painfully long. Refactoring is the counterweight: it reshapes the code so that each addition does not bring complexity it does not need. Its return is measured in the cost of the next change, not in the look of the code.

That is why Fowler treats it as part of ordinary programming rather than a task that appears in a project plan. He names three moments. Before adding a feature, he checks whether the code is shaped to make the feature easy, and refactors first if it is not, which he usually finds quicker than forcing the feature in. After the feature works, he refactors the result into a shape that someone returning in a few weeks can follow without puzzling. And when he has to read unfamiliar code to understand it, he refactors so that the next reader does not struggle in the same place. The first of these has its own name, preparatory refactoring, and Fowler credits Kent Beck with the rule behind it: for every change, first reshape the code until the change is easy, which can be the hard part, and only then make it.

Refactoring also keeps risk low in a way that feature work cannot. Fowler passes on another of Beck’s metaphors, the two hats. While wearing the refactoring hat, every change preserves behaviour and the tests stay green throughout. While adding function, tests are added and existing ones break, which is more stressful and riskier. A developer who knows which hat is on knows what a failing test means: during a refactoring, it means the last step was wrong.

The same idea is how most technical debt gets repaid. The Google paper on hidden debt in machine learning systems lists refactoring code first among the ways to pay debt down. Tools help: many languages now have editors that

automate common refactorings. Fowler counts such tools as valuable but not essential, and where they are missing he relies on small steps and frequent tests.

What it looks like in practice

Take a function, written for this article, that totals an order and prints a receipt in one place:

```
function printReceipt(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }

  // print the receipt lines
  console.log(`Order ${order.id}`);
  console.log(`Total: ${total}`);
}
```

A new requirement arrives: the same total must appear on an invoice. Rather than copy the loop, the developer first refactors. Step one is Extract Function: the loop moves into its own function, named for what it computes, and the tests run. Step two extracts the printing the same way, so that a function name replaces the comment, and the tests run again. Step three is Inline Variable: the `total` variable is now used only once, so the call takes its place. After each step the program does exactly what it did before and could be committed on its own:

```
function printReceipt(order) {
  printLines(order.id, orderTotal(order));
}

function orderTotal(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }
  return total;
}

function printLines(id, total) {
  console.log(`Order ${id}`);
  console.log(`Total: ${total}`);
}
```

Only now does the feature go in, and it is one line in the invoice code that calls `orderTotal`. Nothing in the three refactoring steps needed a new test, because nothing observable changed; the tests that already covered the receipt are what prove it.

Infrastructure code is refactored as well, and there the stakes are higher, because the names refer to running resources. Terraform's documentation warns that, by default, giving a resource a new address in the configuration tells Terraform to tear down the one that exists and build a fresh one. Renaming `aws_instance.a` to `aws_instance.web` for readability would therefore replace a server. Since Terraform v1.1, a `moved` block records the old address instead, and the next plan updates the address without destroying anything:

```
resource "aws_instance" "web" {
  # (configuration unchanged)
}

moved {
  from = aws_instance.a
  to   = aws_instance.web
}
```

On earlier versions, the documentation points to the `terraform state mv` command for the same move. Either way the refactoring is only safe once the plan has been read and shows a move, not a destroy and a create.

Keeping such changes separate from feature work is easier when the history says which is which. The commit standard described in this site's portfolio entry on holding four repositories to one history standard starts every commit message with a conventional prefix naming the kind of change, enforced by a commit-message hook. The page, on work carried out in July 2026, counts 26 refactors in the largest repository's history beside 156 feature commits and 133 fixes. The same page is honest that the hook checks the format and not the content. A commit labelled as a refactor still has to be shown by its tests and its review to have left behaviour alone.

Where it goes wrong

Behaviour changes ride along with the refactoring. A developer tidies a function and fixes a bug in it in the same commit. When a test fails, nobody can tell which of the two caused it, and a reviewer has to read every line as a possible change in behaviour. Wear one hat at a time: refactor in one commit, change behaviour in the next, and if a test fails while refactoring, undo the last step instead of fixing forward.

A large restructuring is called refactoring. A branch spends weeks with the build broken while the code is “being refactored”. That is restructuring, in Fowler’s terms, and it carries the risk refactoring is designed to avoid. Break the work into steps that each leave the system working and can be merged on their own; where that is not possible, call it a rewrite and plan it as one.

There are no tests to show nothing changed. Behaviour preserved is a claim, and without tests it is a guess. Before touching code that has no tests, write tests that pin down what it does today, even where that looks wrong, and fix the oddities in separate changes afterwards. Fowler’s rule of thumb from his preparatory refactoring example applies here: whenever the urge comes to run the code and inspect its output by eye, write a test instead.

A rename in infrastructure code replaces the resource. A tidy new name in Terraform reads like a refactoring in review, and the plan quietly proposes to destroy a database and create an empty one. Record every rename and every move into or out of a module with a `moved` block, and treat any destroy in the plan of a change meant as a refactoring as a failed step.

An old `moved` block is deleted. Moved blocks accumulate, and removing them looks like housekeeping. The Terraform documentation calls it a breaking change: any configuration that still uses the old address would then get a plan that destroys the object rather than moving it. Keep the historical blocks, chain them when an object is renamed twice, and remove them only from a private module once everyone using it has applied the version that contains them.

Refactoring waits for a dedicated phase. A team that schedules a “refactoring sprint” for later keeps adding features to code that resists them, and a sprint planned for later is easy to cut when a deadline moves. Fowler’s advice runs the

other way: refactor as part of each change, first to make the change easy and afterwards to leave the result clear, so that the work never needs a slot of its own in the plan.

Further reading

- Definition Of Refactoring — Martin Fowler's noun and verb definitions, as given in his book.
- Refactoring — Fowler's site for the book, with why refactoring is part of daily programming and the full catalogue.
- Extract Function — the catalogue entry for one of the most common refactorings, with its inverse and its first-edition name.
- An example of preparatory refactoring — Fowler reshapes his own code in small steps before adding a feature, and explains the two hats.
- Refactor modules — Terraform's documentation on `moved` blocks, renames, splitting modules and why old blocks should stay.



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 refactoring?” 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/refactoring/