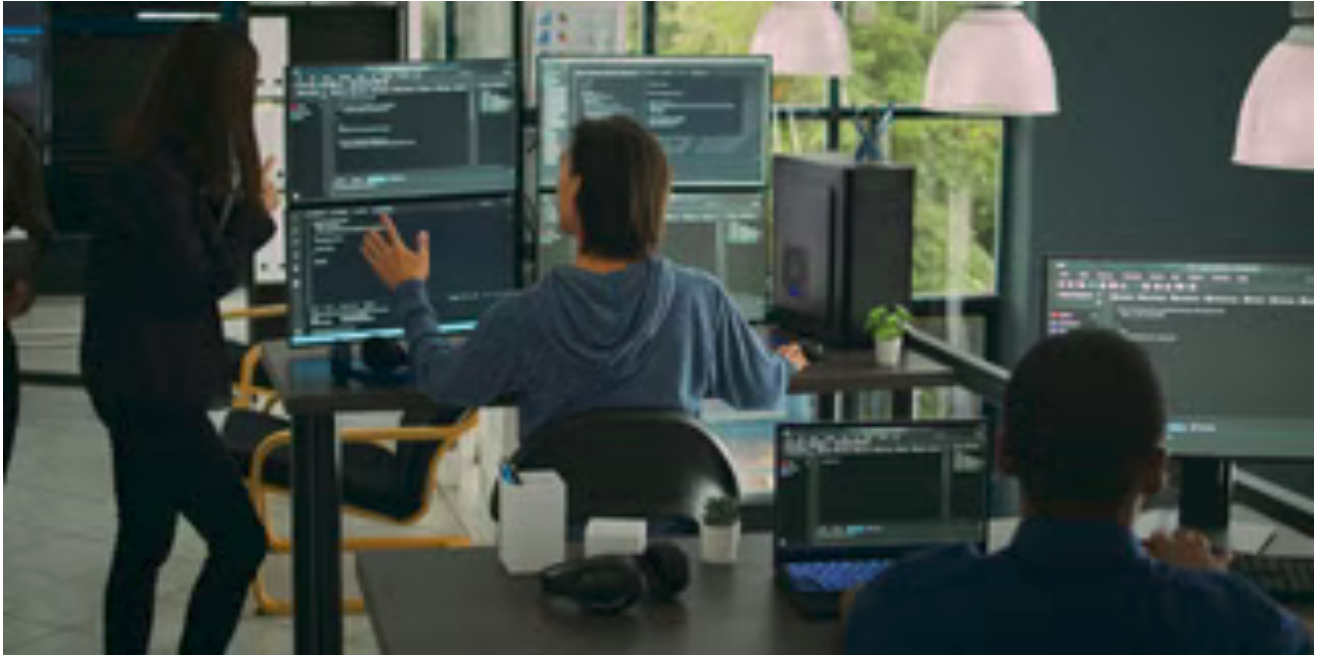


What Is Code Refactoring? | IBM

By Cole Stryker and Rina Diane Caballar

Published 30 April 2025

Updated 06 April 2026



Code refactoring defined

Code refactoring is a [software development](#) practice that alters the internal structure of software code without modifying its external behavior or impacting its functions. These minor changes aim to make code more readable and maintainable.

Martin Fowler popularized the practice through his book *Refactoring*, which was first published in 1999. Code refactoring can help eliminate “[code smells](#),” which Fowler defines as “a surface indication that usually corresponds to a deeper problem in the system.” He adds that code smells are “quick to spot or sniffable” and cites long methods and classes containing only data and no behavior as instances of code smells.

Examples of code refactoring include fixing incorrect formatting, renaming nondescriptive variables, removing duplicate or unused functions and breaking up large, long methods into smaller, more manageable blocks. These trivial, behavior-preserving modifications are less likely to break code or introduce errors, but their incremental effect can help optimize software performance.

The latest tech news, backed by expert insights

Stay up to date on the most important—and intriguing—industry trends on AI, automation, data and beyond with the Think newsletter. See the [IBM Privacy Statement](#).

Code refactoring techniques

Refactoring might seem like a straightforward endeavor, but some tactics can help software developers take a more strategic approach:

Abstraction

Composition

Moving features

Red-green refactor

Simplification

Abstraction

Abstraction is a foundational concept in object-oriented programming. It entails generalizing objects so the complex details are hidden and only the essential information remains.

In code refactoring, abstraction is typically implemented for large codebases. It consists of 2 mechanisms:

The **pull-up method** pulls code from a subclass and moves it up in the hierarchy to an abstract class or superclass. This allows for less code duplication and better reusability of any shared attributes or functions.

The **push-down method** pushes code from an abstract class or superclass down to a subclass for logic that isn't reusable or only applies to a particular subclass.

Composition

This modular approach breaks down or divides huge chunks of code into smaller ones to make them simpler and more manageable. The extraction method and inline method are 2 approaches to composition:

The **extraction** approach takes parts of an existing method and moves them into a new method. This can be done for huge methods containing different functions, for example, so each function can have its own stand-alone method.

The **inline** approach replaces a call to a method with the body or content of the method itself, with the

method then deleted. This usually applies to methods having just a few lines of code and called by only a single class, for instance.

Moving features

In this technique, attributes, methods and other features are moved between classes to lessen dependencies and enhance cohesiveness within the functions of a class and among classes. This redistribution helps achieve a more logical and balanced design of existing code, making it easier to extend and maintain.

Red-green-refactor

Red-green-refactor borrows from test-driven development, wherein tests are written before the source code itself. It's an iterative strategy that allows for continuous refactoring and testing.

This 3-stage process follows these steps:

During the **red phase**, developers write tests to validate a specific software behavior or function. These tests are initially intended to fail because the code hasn't been created yet.

In the **green phase**, programmers write the code for the specified behavior or function. This can be the minimum code required to pass the tests, because the goal here is speed over quality.

Finally, the **refactor phase** is when refinement happens, making any necessary improvements for cleaner, clearer, more efficient code while still preserving its behavior and passing all related tests.

Simplification

The objective here is to simplify code and its associated logic. This can be in the form of reducing the number of parameters in a method, renaming overly long variables or methods, combining conditional expressions that lead to the same result, decoupling complex conditional fragments or even using polymorphism in place of conditionals.

Benefits of code refactoring

Think of code refactoring as tidying a room each day to make cleaning up at the end of the week much easier and faster. The goal of code refactoring is to reduce **technical debt**, which accumulates as a result of programmers taking shortcuts, such as duplicating logic, failing to follow coding standards or using unclear variable names.

Here are a few advantages that software development teams can gain from code refactoring:

Decreased complexity

Enhanced maintainability

Improved code readability

Swifter speed

Decreased complexity

Refactoring can lead to simpler code. This helps developers better comprehend large codebases. Newly hired programmers also benefit as they can quickly grasp the inner workings of unfamiliar code.

Enhanced maintainability

Code refactoring lays the groundwork for better maintenance. Clear code requires less effort when [debugging](#), implementing a new function, updating existing features or upgrading to the latest technologies. Like preventive maintenance in manufacturing, code refactoring allows for minor fixes now to prevent major bugs in the future.

Improved code readability

When code is refactored, it's cleaner, making it easier to understand and work with. Refactored code is also smoother to navigate, helping streamline the software development process.

Swifter speed

Refactoring might not have as significant an impact as actual code optimizations targeted for performance. However, simpler, less bulky code can still contribute to more efficient software and quicker runtimes.

Challenges of code refactoring

Refactoring can result in clear and clean code, but the process is not without its drawbacks. Here are some challenges that development teams might encounter when refactoring code:

Allocating developers

Introducing bugs

Legacy code

Scope creep

Time constraints

Allocating developers

Teams must decide who will be part of the code refactoring process and what their roles and responsibilities will be. This can take programmers away from critical software development work, and smaller teams might not be able to afford that tradeoff.

Introducing bugs

Even the slightest code change carries the possibility of creating a new bug or resurfacing an existing one. A more complex code refactoring also brings the chance of breaking or altering features and functions.

Legacy code

[Legacy code](#) refers to old codebases that still serve their purpose but were developed using now outdated technologies and are no longer actively supported or maintained. It might pose dependency problems and compatibility issues during refactoring. This requires a more in-depth code analysis and a detailed plan to address any changes affecting legacy code.

Scope creep

It can be tempting to fix more than what's necessary, especially when no plan is in place or when no clear objectives have been set. When tackling logical restructuring in particular, the scope can expand when programmers haven't taken the time to examine the code and determine which parts can be refined.

Time constraints

Code refactoring can entail much time, which most development teams lack. They need to balance the need for refactoring with meeting project deadlines and consider when and how much to refactor.

Tips for more efficient code refactoring

Before embarking on a code refactoring journey, consider these tips to help address the challenges of the process:

Timing matters

Planning is key

Analyze and standardize

Test and document

Timing matters

When to do code refactoring is as important as the why or how. It can be part of a software development team's periodic maintenance activities, or it can be integrated into code reviews.

Refactoring is also a must before adding new features, implementing substantial updates, switching to newer tech stacks or upgrading [application programming interfaces \(APIs\)](#) or libraries. It creates a more scalable and adaptable framework to build upon in the future.

Planning is key

Code refactoring can be time-consuming, making planning essential. Development teams must consider their goals and scope. They can take small steps, with a few days focused on tiny changes such as deleting dead code, fixing formatting or removing duplicates. If it's a more involved cleanup, then refactoring becomes a project with timelines and a longer time frame.

Analyze and standardize

Simple refactoring tasks might not require much analysis. **For techniques that touch on logic, however, understanding all related code is vital.** Determining the rationale behind the code structure can help programmers make more informed decisions and intentional modifications.

Also, following a team's coding standards and design principles can retain the integrity and maintain the architecture of the codebase.

Test and document

Refactoring is not only about improving code; it's also about making sure those improvements work. That's why testing is important to confirm that the software itself and its behavior remain intact.

While developers can run their own integration and unit tests, including the QA team is crucial. They can run functional tests to verify features and regression tests to check that the refactored code doesn't introduce any bugs or break any functions.

Documenting changes must also be part of the process. This makes it easier to track modifications and facilitates smoother code refactoring in the future.

Tools and resources for code refactoring

Several tools can help speed up and automate code refactoring. Here are some popular ones:

Integrated development environments (IDEs)

Static code analyzers

Other resources

Integrated development environments (IDEs)

Many of today's IDEs already have built-in support for automated refactoring without breaking any code or introducing bugs. Some even provide AI-driven refactoring recommendations.

IDEs that can be used for code refactoring include IntelliJ IDEA for programming languages based on the Java Virtual Machine (JVM), PyCharm for Python and the ReSharper Visual Studio extension for C#, C++ and .NET.

Static code analyzers

Static analyzers assess code without running them. These analyzers detect common programming flaws and code quality issues, helping developers fix them early in the software development process.

Examples of static code analyzers include Codacy and the [open source](#) PMD that both support multiple programming languages, JArchitect for Java, NDepend for .NET and the RuboCop linter and formatter for Ruby.

Other resources

[Refactoring.com](#) is Martin Fowler's website that contains an online catalog of refactoring methods from his book. [Refactoring.Guru](#) is another website that discusses refactoring techniques and design patterns.

AI for code refactoring

[Artificial intelligence \(AI\)](#) can assist with the code refactoring process. [Generative AI](#) applications are powered by [large language models \(LLMs\)](#) that can analyze complex or huge codebases and understand the semantics and context behind them. Based on that, they promptly provide real-time refactoring recommendations.

[IBM® Bob](#), for instance, identifies bugs and areas for improvement. It then suggests targeted fixes that align with a team's established coding conventions, helping simplify and accelerate code refactoring. Other similar AI refactoring tools include Amazon CodeGuru Reviewer, GitHub Copilot and OpenAI's ChatGPT.

For those dealing with legacy code, [IBM watsonx Code Assistant for Z](#) blends generative AI and [automation](#) to help developers [modernize their applications](#). The watsonx Code Assistant for [Z Understand](#) allows programmers to refactor their application into more modular and reusable services. It employs code analysis algorithms to modernize [COBOL](#) applications.

As with any AI system, developers must still review the outputs of AI-driven code refactoring tools

for accuracy. Testing is also required to make sure any suggested changes work as expected.