WHAT IS CODE REFACTORING? HOW REFACTORING RESOLVES TECHNICAL DEBT

We've all been there before: it's time to add one last function into your program for the next release, but you don't have the time to make it just right – organized, well-structured, and aligned with the rest of the code.

Instead, you have to add the functionality in a bit of a haphazard way, just to get it done, so you do. Luckily, the release goes well, the function is added smoothly, and it's onto the next sprint of work.

But what happens to that code that isn't the cleanest, clearest, or best it can be? We've talked in previous articles about technical debt – the idea that certain work gets delayed during software development in order to deliver on time. Such a short-term solution works for now but isn't the best for the software in the long run. This work then turns into "debt" because it will eventually need to be dealt with.

In this article, we are looking at code refactoring as a way to reduce technical debt.

Defining code refactoring

Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality.

There are many ways to go about refactoring, but it most often comprises applying a series of

standardized, basic actions, sometimes known as micro-refactorings. The changes in existing source code preserve the software's behavior and functionality because the changes are so tiny that they are unlikely to create or introduce any new errors.

The importance of code refactoring

At first, its purpose may seem a little superfluous – sure, code refactoring is improving the nonfunctional attributes of the software, which is nice, but what's the point if it isn't helping the overall functionality?

Experts say that the goal of code refactoring is to turn dirty code into clean code, which reduces a project's overall technical debt.

Dirty code is an informal term that refers to any code that is hard to maintain and update, and even more difficult to understand and translate. Dirty code is typically the result of deadlines that occur during development – the need to add or update functionality as required, even if its backend appearance isn't all that it could or should be. You can often find dirty code by its code smell, as it were.

This is the idea behind technical debt: if code is as clean as possible, it is much easier to change and improve in later iterations – so that your future self and other future programmers who work with the code can appreciate its organization. When dirty code isn't cleaned up, it can snowball, slowing down future improvements because developers will have to spend extra time understanding and tracking the code before they can change it.

Some types of dirty code include:

- Codes, methods, or classes that are so enlarged that they are too unwieldy to manipulate easily
- The incomplete or incorrect application of object-oriented programming principles
- Superfluous coupling
- Areas in code that require repeated code changes in multiple areas in order for the desired changes to work appropriately
- Any code that is unnecessary and removing it won't be detrimental to the overall functionality

Clean code, on the other hand, is much easier to read, understand, and maintain, thereby easing future software development and increasing the likelihood of a quality product in shorter time.

When to refactor

Knowing the right time to refactor isn't too tricky. If you're the developer, you already know where you may have cut corners in your code in order to create the functionality you needed.

If you're part of a team that's sharing a project, it may be harder to prioritize refactoring, so here are some tips:

- **Refactor in accordance with the Rule of 3:**
    - The first time you are doing something, just get it done, even if it's with dirty code, so the software functions as needed.
    - The second time you're doing a similar change, you can do it again the same way – you'll know it a little better, so you may be speedier but the code still won't be perfectly clean.

    - When you encounter this change for the third time, start refactoring.
- **Refactor during code review** – the last chance to clean up code before it is live. Try doing a two-person review so you can fix quick, low-hanging fruit and then better gauge which difficult code change areas are worth the time.
- **Refactor during regularly-scheduled intervals.** This doesn't have to mean dedicating a whole day to it, but rather add it in to your routine – spending the last hour of a workday on refactoring. (Bonus: proactively refactoring means your manager and your team don't have to carve out additional time for it.)

Some refactoring purists maintain that refactoring should not occur while you're addressing bugs, as the purpose of refactoring isn't to improve functionality. But, cleaner code inherently equates to fewer bugs, as bugs are often the result of dirty code. By cleaning code – whether in dedicating refactoring sessions or while addressing bugs – you'll mitigate bugs before they become problems.

Ways of refactoring

There are several ways to refactor code, but the best approach is taking one step at a time and testing after each change. Testing ensures that the key functionality stays, but the code is improved predictably and safely – so that no errors are introduced while you're restructuring the code.

Refactoring used to be a manual process, but new refactoring tools for common languages mean you can speed up the process a little bit. Still, it can be helpful to understand what the tool is actually doing, and if you're in a less common language, great refactoring tools may not be available.

Which techniques to employ often depends on the problems in your code. Here are some common techniques:

- Correcting your composing methods in order to streamline, removing duplication, and make future changes a lot easier in your code.
- Simplifying conditional expressions, which become unnecessary complex over time, and method calls so they are easier to understand, improving interfaces for class interaction.
- Moving features between objects in order to better distribute functionality among classes. This can include safely moving functionality, creating new classes, and hiding implementation details.
- Organizing data to improve handling and class associations so that the classes are recyclable and portable.
- Improving generalization.

Refactoring checklist

What is a good stopping point for clean code? This checklist can help you determine when your code is clean:

- **It is obvious to other programmers.** This can be as simple as creating clearer structures for naming, classes, and methods, or improving more sophisticated algorithms.
- **It contains no duplication.** The chance for human error increases every time you have to double-up on changes
- **It contains minimal moving parts, like number of classes.** Less to remember means less to

maintain and less to clean up.
- **It passes all tests.** Code is dirty even if most of it passes tests.
- **It is easier to maintain.** You'll spend less time on future improvements.

The benefits of refactoring

The main benefit of refactoring is to clean up dirty code to reduce technical debt. Cleaner code is easier to read, for the original developer as well as other devs who may work on it in the future so that it is easier to maintain and add future features. Less complicated code can also lead to improved source-code maintenance, so that the internal architecture is more expressive.

Clean code also means that design elements and code modules can be reused – if code works well and is clean, it can become the basis for code elsewhere.

Refactoring can also help developers better understand code and design decisions. Both beginner-level and more advanced programmers can benefit from seeing how others have worked inside and built up the code as software functionality increased and shifted. This can encourage the sense of collective ownership – that one developer doesn't own the code, but the whole team is responsible for it.

The act of refactoring – changing tiny pieces of code with no front-end purpose – may seem unimportant when compared to higher priority tasks. But the cumulative effect from such changes is significant and can lead to a better-functioning team and approach to programming.