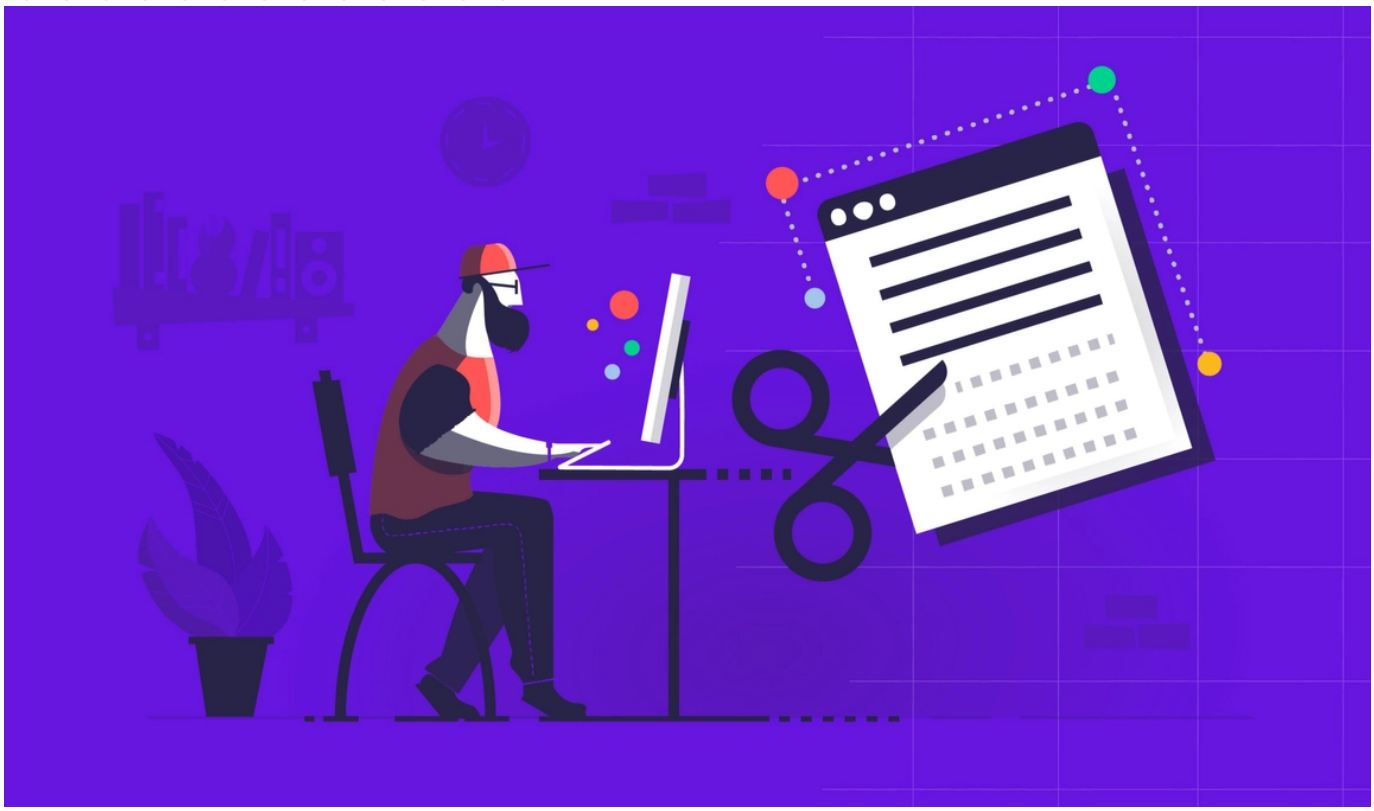
Software Engineering

# How to know when it's time to refactor your software's code

February 26, 2019

Johnny Dunn
Full Stack Developer



Code refactoring is a methodical process of restructuring existing code without modifying any of its external behaviors or outputs. In other words, refactoring changes the nonfunctional parts of code for improved readability and reduced complexity.

In software-oriented environments, calculating the tradeoffs and benefits of refactoring code can be a tricky balance, as the resulting improvements are generally not obvious to the product users. But there are various types of tell-tale signs, many of which comprise some form of "code smells", that strongly signify the need for some design improvements.

Heeding these signs will save you countless headaches in the future by lessening and preventing technical debt. Refactoring also oftentimes leads to discovering hidden or dormant bugs in your code, preemptively solving issues that could otherwise occur in production or cause problems further down the line during development.

Below is a list of strong arguments and guidelines for determining when to refactor. For the sake of keeping things concise, performance optimization will not be addressed as a reason, although refactoring oftentimes can lead to unintentional benefits including optimization, bug discovery, and future-proofing.

## 1) You seem to be repeating logic or declaring the same structures of code over and over again.


If this is happening, it means you can abstractify and encapsulate the system to make it more modular—can you wrap your code into functions, and keep those functions contained within classes? If you're programming in an object-oriented language/paradigm, generally, modularizing code is one of the most useful things you can do.

Let's take a real-world example of a web application that sends a request to a stock market website to parse and display prices in a dashboard tabularly. What are the conditions or parts of the program that must occur as a user interacts with the app?

One of the first steps involves an HTTP request sent by your app's backend to fetch the data to display. In this case, let's say that the stock site has an API that returns last week's prices in JSON structure. You might start off writing the logic

for this interaction by hardcoding the requisite URL in the same line of code containing the request, something that's straightforward and successfully returns what you want.

Alternatively, if you were thinking about object-oriented principles like modularity and encapsulation, and instead chose to write a function to make the web request, that opens up all sorts of possibilities for quickly extending the functionality of the app we have so far. Still using the stocks API, we can show prices for a specific company to the user by slightly modifying that function or adding another one to call when the user submits an input form corresponding to the company's ticker ID.

With the new function, we then retrieve the submitted ticker value (here's one straightforward way to do this), and make a request to the same API endpoint with the ticker ID passed as a parameter to get a specific listing from the stock site. This is assuming our stocks API is relatively complex and supports this feature in that endpoint.

You could've written another function that accepts a parameter for the ID with another request call to the API, or you could modify the original requests function and make the ID parameter optional and defaulted to an empty string, as shown in the coding example below.

```
function fetchStocks(tickerId="") {
    let url = 'https://www.realstockmarketsite.com/api/pric
    if (tickerId !== "") {
        url = url + '/' + tickerId;
    }
    fetch(url)
    .then((response) => {
        return response.text();
    })
    .then((result) => {
        return result;
```

```
    });
}
```

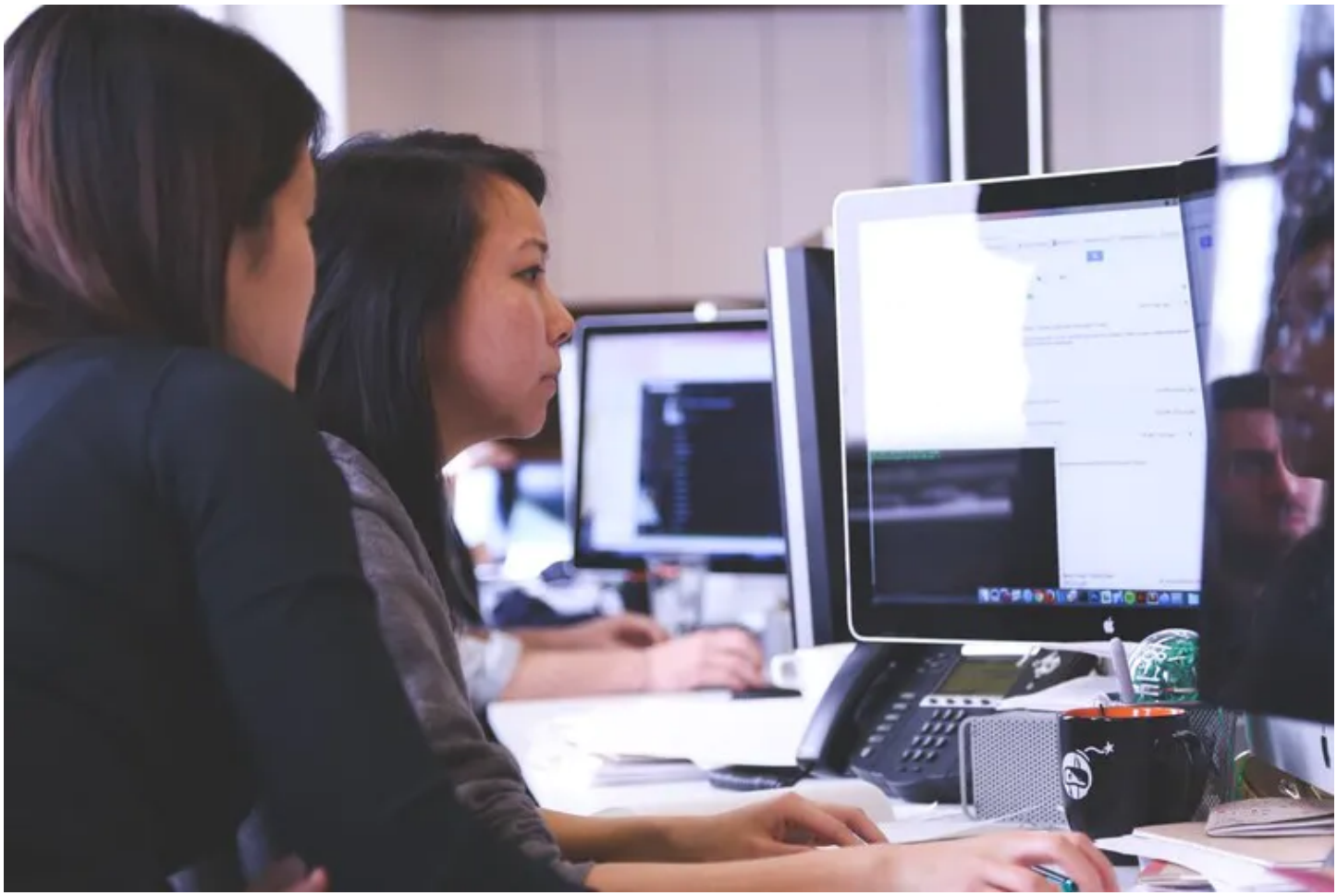
Now you've avoided any duplication of logic in your code, and have maximized readability and brevity, both of which are generally desirable results when refactoring.

If you're building a fairly complex application where you may need to reuse functions in multiple scripts, the best thing to do would be to modularize as much as you can now while you're already in the process, and refactor out the original requests function into smaller components. One function sends a request and returns a response from any URL given to it, and another function acts as the requester to our specific stocks API.

This is an extremely simplified and limited example and doesn't have enough depth to provide significant insight on refactoring real-world software. In a future tutorial that we'll publish and link later, you'll be able to follow along with an example of refactoring a complex app with coding examples, and also review best practices to follow in object-oriented design.



areas in the codebase.



Another telling sign of when you should consider refactoring is if you or your team's getting tripped up over some specific domain or feature in your code. This is often a sign that refactoring is not only necessary but that, if left unchecked, that problematic area could incur massive amounts of technical debt and time costs over time.

Technical debt can have the same effect as an avalanche, where it builds upon itself, multiplying and becoming greater exponentially. Without enough care to creating proper abstractions while writing new modules or features, technical debt can accrue rapidly. In agile-oriented teams that move and iterate quickly, the fallout from accumulative technical debt can occur with negative consequences sooner rather than later.

On a previous team, I was working on extending an automated account registration process for guests visiting a company and for the staffed employees alike. It was designed months prior and had not been made to scale easily. After just a few months, it became a mess of interdependent, poorly abstracted code. Adding new features was difficult, and modifying existing ones was even more so, and the difficulties only grew as the code did and as more developers left their marks on it.

Although this was a relatively "straightforward" check-in process, in theory, the actual technical requirements weren't steady in the beginning and had to be updated multiple times, and unfortunately, there wasn't enough development time available to do any proper refactoring. If we had refactored early and abstracted and designed well from the beginning, the feature shouldn't have been too difficult to update and extend as future requirements came in.

Instead, however, making changes to the code involved going through multiple areas within the system, as many of the backend interactions had behaviors that were intertwined, or dependent on each other, since our team had an easier time copying and pasting what was already there, rather than adequately abstracting and encapsulating classes.

Eventually, there came a point when the resources expended on updating the original system would outgrow the cost of a few devs rewriting from scratch. And that's exactly what happened; however, this time, our team had learned to refactor early and continuously, especially when we would notice problematic patterns within a specific domain.

## 3) Debugging issues seems to take you longer than it should, and similarly, you're debugging randomly because you're not sure how to solve it systematically.


Spending an extraordinarily long time to debug issues is a very common symptom of codebases in desperate need of refactoring. It frequently occurs in parallel with the previous item on the list. If you're working independently on a feature or you're the sole developer for a product, you won't have a fallback system of other developers who can help regulate and refactor the commits, or impart their own perspectives on what's working or not working well. All this makes it more difficult for a developer to correctly assess and determine when to refactor.

If a significant portion of your debugging time involves navigating the codebase or its output randomly — i.e. you're spending too much of your time blindly looking around and setting breakpoints in your IDE, in an attempt to understand what's transpiring inside the stack trace — this points to underlying issues with the system's transparency that need to be addressed.

Are you having trouble following how the value of a variable changes throughout your script? Is it unclear what the output of a given function will return when you know the exact values passed to it? Then you're doing too much with a component that can separate into smaller, more distinctive parts.

Don't be afraid of throwing out code if it's giving you problems. It's easy and great to have pride in what you're writing, especially if you're solving problems that are interesting and challenging, but it's also easy to get stuck with a particular way of doing things. Being stubborn with your code restricts your perspective, the tools you're working with, and eventually the overall system you're building.

## 4) It's been a while since the last time your team's refactored anything, or you're planning on integrating another feature / component.

Even the most reliable team of devs won't always produce the most desirable architecture of a system the first time. As teams grow, inconsistencies with individual coding conventions/styles often lead to instability within the codebase, a scenario calling for an imminent need of refactoring, and perhaps a more rigorous enforcing of coding standards and tools.

Practice having consistent code reviews, where all relevant members of your team can have their say on what tradeoffs and benefits different implementations offer for end users as well as the developers themselves. Doing this makes code refactoring a natural and even seamless process at work, and catches coding inconsistencies early that could make trouble for a codebase later on.

Even after undergoing a recent code review or refactor, if your team is about to integrate some different module or new feature with your current code, you should be conducting periodic checks and quick reviews throughout the process as, oftentimes, after integrating something new, unexpected outcomes can result and adjustments are needed. This is especially true if it's a new feature whose system has been abstracted or hidden away from you (e.g., a third-party cloud API).

Integration, in general, is a very complicated process, but you can ask yourself the following to help smooth things along and potentially refactor if needed:

- Did the structure of any of your modules or classes act differently or needed modification after adding a new feature?
- Does the flow of the program seem more complicated or less transparent now than it was before?
- Is it taking you longer to add new code or to modify existing functions since the last moving pieces were integrated?

All the above points to poor object-oriented design in your system: tightly coupled components with low cohesiveness, meaning that things are dependent on each other where it isn't necessary. This quickly leads to unexpected and undesirable behavior.

Knowing when and why you need to refactor can be the hardest part of writing code, especially when you're moving quickly with requirements that could easily change with every new sprint. However, the process of code refactoring also needs to be flexible yet methodical and conscientious, as every application has its own niche set of requirements.

We have gone through a typical process of how you should think when refactoring code in part one of this article; however, in most real-world scenarios, web-facing applications are much more complex, and so refactoring requires more thought and delicateness.

Look out for a companion piece to be published soon, entitled *How to modularize and refactor code with object-oriented principles*, which will be an in-depth guide on code refactoring with practical examples and comprehensive information on writing clean object-oriented code.

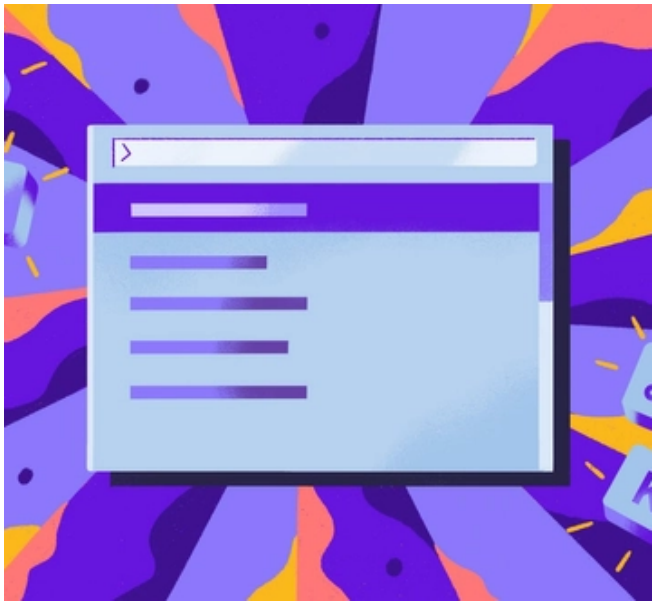
Do you have any other tell tale signs for when it's time to refactor your code? Let us know on Twitter!

Topics:

Tech Debt

Share this Clubhouse story

## More stories from Clubhouse



Clubhouse News and Updates

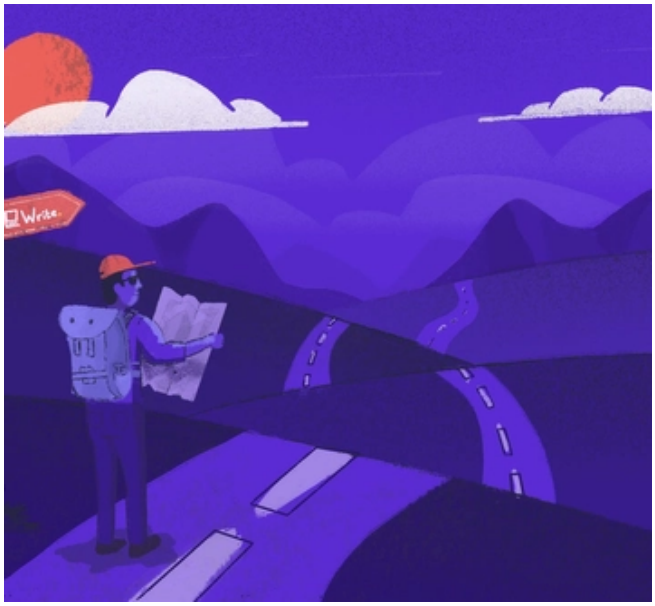
Clubhouse News and Updates

## Execute commands more efficiently with the Action Bar

February 5, 2020

## Collaborate faster at scale with Group @mentions and notifications

February 3, 2020




Clubhouse News and Updates

## Update on the Clubhouse Write private beta and what's next

January 28, 2020

### Product

Features
Pricing
Enterprise
Integrations
Write beta

### Company

About Us
Customers
Careers
Press
Contact
Brand Guidelines
FAQ

### Developers

REST API Docs
Webhook API Docs
Community
Open Source Projects
Developer How-Tos
Write for Clubhouse

### Resources

Help Center
Blog
Webinars
Status
Referral Program
Release Notes

### The Clubhouse Blog

Collaborate faster at scale with Group @mentions and notifications

Execute commands more efficiently with the Action Bar







Privacy Policy Terms of Use Security Your Cookies