# Refactoring Legacy Code V.S. Staying Focused On The Current Project

**sihui.io**/refactoring-vs-staying-focused

Sihui Huang

January 31, 2019

When I work on projects, I often run into legacy code that can be improved—to be more readable, more testable, or more comply with the current coding style. My urge to refactor the code is especially strong after spending a good amount of time trying to understand a piece of obscure code. That code hurts my brain, and I don't want the same thing happens to other developers. There is the famous Boy Scout Rule: **Leave Code Better than You Found It**.

But at the same time, I also want to stay focused and make progress on my current project. I worry that I might get distracted and spend too much time refactoring.

This is a dilemma I often face. **I want to be the good samaritan leaving code better than I found it, an engineer turning all the messy code I come across into something clean and elegant. But I also want to stay pragmatic and ship projects quickly.**

Balancing between refactoring the legacy code I come across and staying focused on my current project can be hard.

A recent conversation with my manager on this topic provided me guidance to find that balance. Here are a few points I find helpful.

- **You don't have to make the code perfect. Small improvements are still useful.** After I read a piece of legacy code, my brain immediately thinks about how it should be ideally. But refactoring that code from its current state to the ideal state sometimes requires an extensive refactoring. At its best, the required refactoring can be time-consuming. At its worst, it can turn into a rabbit hole that causes a big delay on the project. When I faced cases like this, I used to either take a bet and do the extensive refactoring anyway or hold my nose and let the code stay stinky. Now I realize there is a happy middle point: I don't have the make the code perfect. I can spend as much or as little time as I can afford to make the code a bit better. Small improvements I do can be still useful for the developers that come next. And that developer may have time to improve the code further, building upon the improvements I make. [1]
- **Time-box your refactoring.** Time-boxing is a great technique to avoid rabbit holes. Before starting to refactor, do a quick estimation about how much time you can spend on refactoring legacy code without delaying the project too much. Write that number down. Time-boxing your refactoring gives you the peace of mind knowing the main project will stay on track. And that makes the refactoring experience more enjoyable.
- **Bake the refactoring time into project estimation.** Things can be even easier if you bake the refactoring time into project estimation before starting a project. While estimating how much time a project might take, take a quick look at the code you need to touch: when was it first written? how readable and extensible it is? how much test coverage it has? Unless there's a hard requirement about when the project needs to be shipped, you should bake some refactoring time into the project's timeline. After all, as we keep building upon a legacy codebase, refactoring it as we go is the best way to keep it healthy. And a healthy codebase, in turn, makes building new features faster and safer.
- **Refactoring is educational.** Refactoring a piece of code is the best way to understand it. Compared to staying afar and reading a piece of code, coming in and modifying it gives you a much deeper and intimate relationship with the code. When you try to add your touch to the code, you will start to notice things you missed before. You may see the constraints and may even understand why the author originally wrote the code this way. Although refactoring a piece of code doesn't seem to add any immediate value to the business, it's an opportunity for you to gain a piece of domain knowledge you currently don't have. Gaining that domain knowledge is invaluable to both you and the business.
- **Refactoring is a muscle you can build: the more you do it, the faster you become.** One last reason to refactor legacy code is to use it as an opportunity to

strengthen your “refactoring muscle”. Investing in building your “refactoring muscle” has a self-reinforcing compound effect: the more you refactor, the better you are at refactoring, making your next refactoring faster. The more you do it, the better the codebase is, making building future features faster.

Luckily, my next project relates to code from almost three years ago. I expect to face many cases requiring me to balance between refactoring legacy code and making progress on the project. I will share my new learning on the blog. Subscribe to follow along!

## Enjoyed the article?

[1] (PS: There’s no perfect nor ideal code. The moment you make a change, you see more improvements you can make.)

---

My career plan for the year is to grow into a tech lead. I’m excited about all the learnings ahead and would love to share this journey with you in a brutally honest fashion. I will be sharing my weekly learning on the blog.

In the next few months, I will focus on growing in the following areas, so you can expect to see posts related to them:

- focusing on the big picture of the project instead of near-term implementation details;
- balancing my efforts between leading projects and coding;
- work-life balance for long-term productivity;
- the human side of software development: making sure everyone riding with me enjoys the ride and feels fulfilled and inspired.

## Enjoyed the article?

## Related Posts