Refactoring: an opinionated introduction

This is written from the perspective of a developer working with Ruby and Javascript - dynamically typed languages which have limited automated refactoring support in most modern IDEs.
Dr L J Noble, July 2024

Definitions

Noun: a specific, reversible code transformation which solves a particular code smell.

Verb: the process of applying a series of refactorings

"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour… It is a disciplined way to clean up code that minimises the chances of introducing bugs."
-- Martin Fowler, Kent Beck

Improving the design of existing code
… in a sequence of small transformations
… that preserve the important behaviour of the system
… which you can complete relatively quickly
… and which give you inexpensive options to change direction.

Disambiguation

Refactoring is not the same as rewriting (also called restructuring, re-architecting, or replatforming). This can be a valuable exercise, but it is not refactoring.

Other things that are commonly confused with refactoring:

- Fixing any bugs that you find while working on a piece of code.
- Optimization.
- Tightening up error handling and adding other defensive code like validating inputs before processing, or logging to support debugging.
- Making the code more testable – although this may happen as the result of refactoring.

Once you start using Branch by Abstraction, or the Strangler Fig Pattern, and you are working with old code and new code, with detours and scaffolding to support the interim structure, then although it's technically refactoring, it is neither quick nor safe, and should not be treated as a routine coding practice.

When to refactor: Make the change easy, then make the easy change

Writing code so that the team can keep up a sustainable pace is your job. It's not something you should have to ask permission to do.

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
--Martin Fowler, Refactoring

- use good names
- remove duplication
- get rid of code smells
- keep methods small
- keep complexity low
- optimise code for reading (not writing)

The only viable reason for refactoring is an economic one: in order to make it possible to deliver more features more rapidly, to be more responsive to the changes that people need. This is why it's the right thing to do professionally, because otherwise we are deliberately slowing down development and negatively affecting our customers and the company bottom line.

The idea that new code is better than old is absurd - old code has been tested (if not in a test suite, then certainly in production). When you throw away old code and rewrite from scratch you lose (potentially) years of bug fixes and knowledge about how real users interact with the system. You have absolutely no guarantees (beyond hubris) that you will do a better job than the original coders.

Be very careful before starting a Refactoring Project: you are likely trying to fix issues that were caused by creating the wrong abstractions in anticipation of a future that never arrived - by attempting to predict the future again.

The scope of your refactoring work should be driven by the change or fix that you need to make – what do you need to do to make the change safer and cleaner? Don’t refactor for the sake of refactoring. Don’t refactor code that you aren’t changing or preparing to change. And stop refactoring as soon as you are no longer certain that you are improving the code!

Spend ten minutes a day doing cleanups that are too small to matter, and over time you see results which really do matter. If you haven't learned to do the daily tidying, then any big cleanup effort will be unsustainable (and frankly pointless).

"One reason why developers are so keen to rewrite code is that it is much harder to read code than it is to write it. It is easier and more fun to write a new function than work out how an existing one is used."
-- Joel Spolsky

“It’s like I want to go 100 miles east but instead of just traipsing through the woods, I’m going to drive 20 miles north to the highway and then I’m going to go 100 miles east at three times the speed I could have if I just went straight there. When people are pushing you to just go straight there, sometimes you need to say, 'Wait, I need to check the map and find the quickest route.' The preparatory refactoring does that for me.”
-- Preparatory Refactoring (Jessica Kerr)

When to rewrite rather than refactor

If you are using a language or a platform which cannot move forward (no longer supported, obsolete, etc) and thus prevents future change, your only option may be to rewrite. This is, however, a risky approach as it essentially prevents any development of the product for the duration of the rewrite. Before you start a Big Rewrite, you need to understand and document all the business logic in the application you intend to rewrite. Generally, with a legacy application, you have neither understanding nor documentation.

Depending on how testable and maintainable the existing code is, it may be safer to test drive replacement code than attempt to cover existing code with tests. After replacing a few sections of code with test-driven replacements, however, it may become safer and quicker to refactor rather than continue the rewrite.

Caveat: refactoring for understanding

If you are struggling to understand a particular piece of code, scratch refactoring (refactoring to understand) may be justifiable. Try renaming methods and variables once you understand their purpose, or splitting complex conditional statements, or inlining methods which make the code hard for you to reason about.

Don’t bother reviewing and testing all of these changes. The point is to move fast – this is a quick and dirty prototype to give you a view into the code and how it works. Learn from it and throw it away.

Requirements for safe refactoring

- Code in version control.
- An automated test suite which you trust to detect regressions. That test suite *must* be green. Familiarity with code smells and the refactorings required to resolve them.
- The discipline to follow the prescribed refactoring steps, including running the test suite after each small step - especially if you work in a language or environment which does not have good automated refactoring support.
- Practice

If you are not the only developer working on the codebase, a clear agreement with your colleagues about the desired end point.

Be specific about what ‘better’ code looks like in the context of your team and application. Consider agreeing on naming conventions, structured logging standards, whether generic code belongs in /lib or app/lib, which auto formatter and linter you will all use, etc.

Pitfalls to avoid when refactoring

- Combining refactoring with behaviour change - this includes automated linting and reformatting!
- Refactoring more than you need to achieve the current task.
- Committing changed code just because you changed it - if it's not an obvious improvement then all you're doing is replacing familiar code with unfamiliar code.
- Engaging in a refactoring tug of war: you apply a refactoring, then someone else applies the inverse refactoring, and so on.
- Attempting to predict the future.
- Being seen as careless because you refactor, rather than get it “right” the first time.

Refactoring in a Legacy Codebase: a special case

Read the book - Working Effectively with Legacy Code (Michael C. Feathers)

- Identify pain points
- Break dependencies
- Cover the desired piece of code that's about to change with tests
- Make your changes
- Refactor all you want afterwards when it's safe to do so

Refactoring Workflow

Why do you want to refactor the code?

- it's ugly and makes me sad to look at
- it doesn't use the most modern paradigms
- it's inefficient
- it doesn't make sense
- it's difficult to test
- every time we change it, we introduce new bugs
- everybody is too scared to touch it

Are you working on, or about to work on, this area of the codebase?

- no
    - step away from this code - you'll do more good focussing on code which is about to change, and there's always the risk of an undetected regression
- yes

What does your test suite look like in this area of the codebase?

- What test suite?
    - write some tests, and if that's not possible consider the "replace algorithm" refactoring
- I've got unit tests for every method: private and public!
    - write some higher level tests so you can change the implementation safely
- I've got feature- and/or integration-level tests for the most important functionality

Are your tests all green?

- no, I need to refactor to fix things
    - back out of your changes to the last point where tests were green, and commit
- no, the tests apply to the old behaviour before I made changes
    - it is not safe to combine refactoring with behaviour change, so do one or the other
- yes, I changed the tests to make them pass after I made code changes
    - keep behaviour change and refactoring in separate commits
- yes

Do you understand the code you are going to be refactoring?

- No
    - If you can, speak to expert users or experienced colleagues to understand the business use-case and happy path of the code from a user perspective
    - Apply the simplest, least-risky refactorings: renaming variables, classes and methods
    - Consider scratch refactoring purely for understanding (do not commit this!)
    - Consider pairing with another developer
- Yes

Do you know which code smell you are going to start by fixing?

- sorry, what?
    - code smells are your pointer to the correct, safe, refactoring
- yes

Do you know how to carry out the safe refactoring steps?

- steps?
    - safe refactorings have been documented: don't try to reinvent the wheel
- yes, my IDE has a refactoring menu
    - excellent, but be aware that even the best tooling can make mistakes
- yes, I have a reference book/website/cribsheet
    - excellent, take it carefully

OK, you've completed the refactorings to fix this code smell - are your tests green?

- no, I'll need to fix them
    - something has gone wrong: stop and work out if your tests were at the wrong level, or the refactoring was not actually safe
- Yes
    - commit

Any more code smells that you can see, or that this refactoring has exposed?

- yes
    - Will fixing them make the change you intend easier? if not, stop and bask in the glow of 'good enough' code
- no
    - stop and bask in the flow of 'good enough' code

Are you completely satisfied with your code?

- Yes
    - mistake, you've probably overengineered it and likely taken too long to do so, too
- no
    - good, aim to design and code just well enough to avoid nasty surprises when implementing the next feature

Assuming that after refactoring you have quite understandable code that works in an obvious manner, that doesn’t contain kludges and that is quite easy-to-use. Stop! There is an infinite list of potential improvements between good code and ideal code - and attempting to achieve the latter will prevent you from actually shipping things that customers will use. Progress, rather than perfection.

Once you are done, review your own PR before opening it up for formal review - the overview may help you see where you have made local improvements and missed the bigger potential gains. You need to switch your mindset from creator to publisher in order to get this broader perspective, and a formal review can help with the switch.

Bonus Point: if you can see a “thing” which clearly needs to be extracted, but you aren’t yet clear on exactly what to call it, have a set of generic names to hand which have nothing to do with your problem domain and which you can easily replace once you are clear on the correct name to use.
E.g. Badger, Snake, Mushroom; or Applesauce, Hotsauce, Weaksauce

References

https://github.com/docljn/self-study/tree/master/refactoring (includes notes and extended references)

Further Reading

https://refactoring.com/catalog/ (support for the Martin Fowler ‘Refactoring’ book)
https://refactoring.guru/refactoring/catalog
https://databaserefactoring.com/
https://classnames.paulrobertlloyd.com/
https://martinfowler.com/articles/extract-data-rich-service.html
https://www.geepawhill.org/series/many-more-much-smaller-steps/

Refactoring (Martin Fowler)
Five Lines of Code: How and when to refactor(Christian Clausen)
Refactoring to Patterns (Kerievsky)
Working Effectively with Legacy Code (Michael C. Feathers)
Your Code as a Crime Scene (Tornhill)

Further Watching

▶ RailsConf 2016 - Get a Whiff of This by Sandi Metz
▶ REFACTORING: What You Need To Know | Guided Learning Hour and code at GitHub - emilybache/Tennis-Refactoring-Kata: This is a Refactoring Kata based on the rules of Tennis
▶ Can you even refactor in Javascript? Or Python? | Everyday Coding Expertise
▶ What Would Martin Fowler Do? Javascript Code Refactoring Demo
Git-driven Refactoring
▶ Gilding the Rose: Refactoring-Driven Development - Kevlin Henney - ACCU 2023
▶ Martin Fowler @ OOP2014 "Workflows of Refactoring" and slides at Workflows of Refactoring
▶ Tidy First? Kent Beck on Refactoring