Refactoring

What refactoring really is

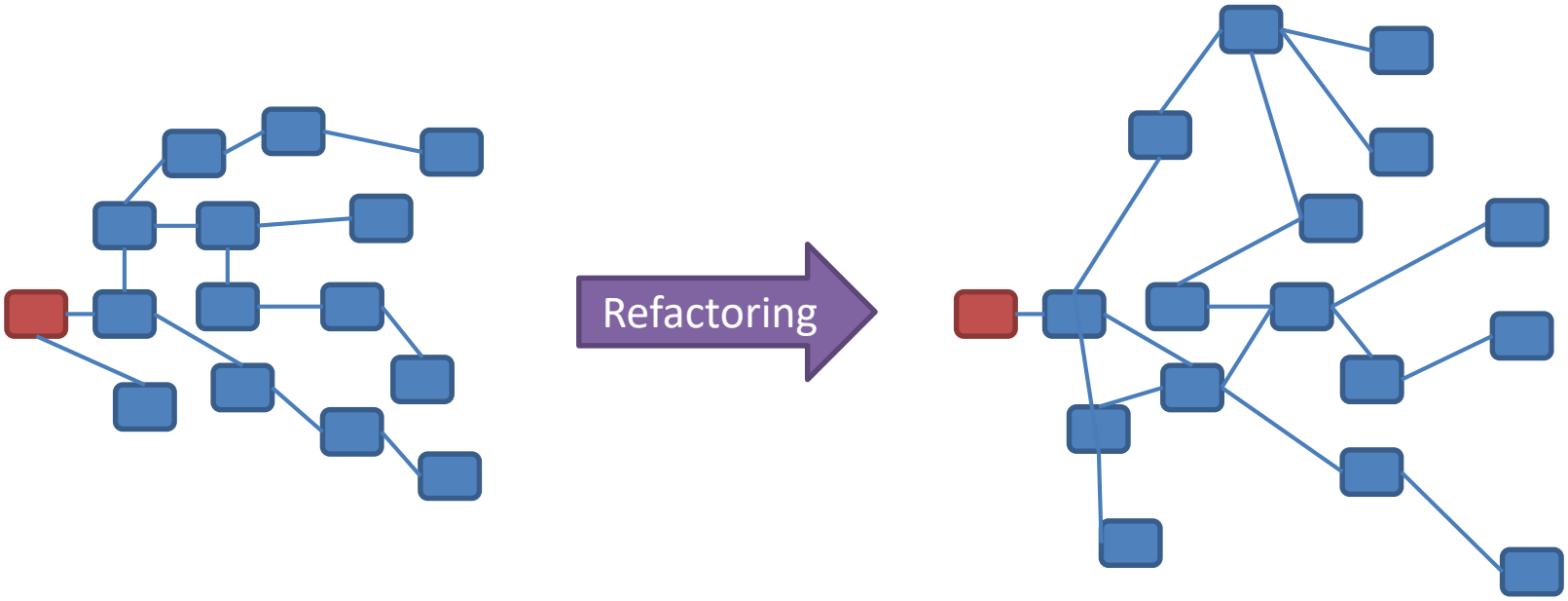
We've seen a number of refactorings already today. Now, let's put the term on a solid foundation.

Two insights help on the way to understanding refactoring:

- There are **several ways to write** a solution to a problem. All of them yield the same result, but they are structured differently. Let's call each such way to write a solution a **factoring**.
- Some factorings **are better than others**, for moving forward in a certain direction. Loosely, you want your factoring to reveal the things you expect to change in the future, putting them within easy reach for modification.

What refactoring really is

Going from one factoring to another is called **refactoring**. It is the act of letting out a bit of steam from the code, making it more suitable to move forward.



Refactor mercilessly

EDUMENT

“Extreme projects do not use Big Design Up Front. Therefore they upgrade their designs continuously."

http://c2.com/cgi/wiki?RefactorMercilessly

Basically, when you see something that would be improved by a refactor, do it. **Refactor mercilessly**.

It may seem a strange investment of our time to refactor code. But doing so often speeds up regular development, finds bugs earlier, gets us closer to the domain model, and reveals other potential improvements.

You don't have time **not** to refactor.

Refactor mercilessly

EDUMENT

The test suite is what gives us the **courage** change anything we wish to.

A successful refactor, by definition, takes the test suite from all-green to all-green.

(A version control system that doesn't suck helps, too.)

You Ain't Gonna Need It

EDUMENT

Certain actions work ***against*** agile development, consuming time and energy without any corresponding reward.

Over engineering is like that:

trying to come up with a solution that not only serves you **now**, but also **later**.

You Ain't Gonna Need It

Trying to code for the future happens so often and is such an easy trap to fall into, that the saying **You Ain't Gonna Need It** (abbreviated **YAGNI**) was invented. It's there to remind you to focus on needs here-and-now.

Therefore: Always implement things when you **actually** need them, never when you just **foresee** that you need them.

This principle is sometimes expressed as **The Simplest Thing That Could Possibly Work**, or **Better Sorry Than Safe**.

It puts the focus on exploration, not up-front design.

Once And Only Once

Sometimes, we end up with the **same behavior** being **duplicated** in many places in our system.

Often, the duplication starts off seeming too small to be worth factoring out:

just a line or two.

Typically, the duplication is highlighted at the point a new requirement emerges. The big clue is **copy-pasting** an update to several places.

Once And Only Once

The danger is that you **forget to update one of the duplicates** when doing changes later on.

Therefore: Program logic should be represented **once and only once** in code.

Once and Only Once: quote

EDUMENT

“Code wants to be simple. [...] When I began working in this \[Once And Only Once\] style, I had to give up the idea that I had the perfect vision of the system to which the system had to conform.

*Instead, I had to accept that I was only the vehicle for **the system expressing its own desire for simplicity**. My vision could shape initial direction, and my attention to the desires of the code could affect how quickly and how well the system found its desired shape, but the system is riding me much more than I am riding the system."*

-- Kent Beck

XP simplicity rules

EDUMENT

What do we mean when we say "the simplest thing"? The XP people have an answer ready for this. Simple code:

- **Runs all the tests**.
- Contains **no duplication**.
- **Expresses all the ideas** you want to express.
  - Do not put the implementation of **unrelated ideas** in the same method.
  - Classes should **organize ideas** in a readily understandable way.
  - Use **appropriate names** so you don't have to explain method, member or class names with additional documentation.
  - Methods and classes should be implemented so they can be **understood totally** from their public interfaces. This not only allows for up-front testing, but decreases coupling.
- **Minimizes classes and methods**.

Single Responsibility Principle

EDUMENT

"*Write programs that do one thing and do it well.*"

- Doug McIlroy, inventor of Unix pipes

Keep classes focused. Don't make them do too much.

It's less a question of **size** - classes can run long for legitimate reasons - than about **purpose**. Make sure that each class has a single **responsibility**. If it has more, factor them out.

A nice way to spot if a class does too much is to start looking at which methods in the class use which fields.

Cohesion

Another way to say the same thing is that we want to optimize for the **cohesion** of the class.

Cohesive classes have fields and methods that conceptually belong together.

*"**cohesion** refers to the degree to which the elements of a **module** belong together."*

- Wikipedia

Feature Envy

Feature envy happens when a method seems more interested in a class other than the one it is in. Maybe it should actually be in that other class instead?

"**Tell, don't ask**" touches on this problem. Feature envy may come through as a method asking some other object lots of questions and then making a decision.

But it doesn't have to; the method may do lots of **tell** and still be a bit too preoccupied with the other class.

Feature Envy

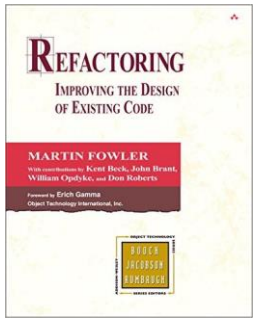
At its worst, feature envy leads to **Inappropriate Intimacy**, where the method simply makes assumptions about the internals of another class.

A method with its hands in the cookie jar of another class will be too tightly coupled to that class, and will very likely break when the class is modified.

Books on refactoring

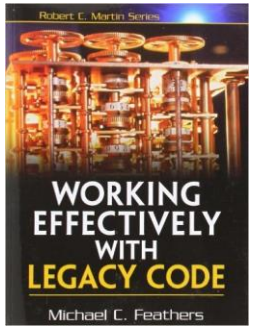
Refactoring: Improving the Design of Existing Code

Martin Fowler
ISBN: 0201485672



Working Effectively with Legacy Code

Michael Feathers
ISBN: 0131177052



Exercise #3