# Refactoring: Improving the Design of Existing Code

**Martin Fowler**

**fowler@acm.org**

**www.martinfowler.com**
**www.thoughtworks.com**



ThoughtWorks®
The art of heavy lifting.™

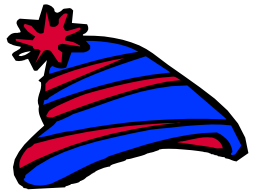
# Definitions of Refactoring

- **Loose Usage**
  - Reorganize a program
- **As a noun**
  - a change made to the internal structure of some software to make it easier to understand and cheaper to modify, without changing the observable behavior of that software
- **As a verb**
  - the activity of restructuring software by applying a series of refactorings without changing the observable behavior of that software.

# The Two Hats

## Adding Function

- Add new capabilities to the system
- Add new tests
- Get the test working



## Refactoring

- Does not add any new features
- Does not add tests (but may change some)
- Restructure the code to remove redundancy

***Swap frequently between the hats, but only wear one at a time***

# Why Refactor

- **To improve the software design**
  - combats “bit rot”
  - makes the program easier to change
- **To make the software easier to understand**
  - write for people, not the compiler
  - understand unfamiliar code
- **To help find bugs**
  - refactor while debugging to clarify the code
- **To program faster**
  - refactoring leads to good design
  - good design lets you program faster

# When should you refactor?

- **To add new functionality**
  - refactor existing code until you understand it
  - refactor the design to make it easy to add
- **To find bugs**
  - refactor to understand the code
- **For code reviews**
  - immediate effect of code review
  - allows for higher level suggestions

***Don't set aside time for refactoring, include it in your normal activities***

# Three strikes and you refactor

- **The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplication thing anyway. The third time you do something similar, you refactor.**

# When should you NOT refactor?

- When the system should be redesigned from scratch
- When you're close to a release deadline

# Refactoring and Performance

- Refactoring leads to lots of small methods, and therefore more indirection
- Doesn't all this indirection slow the program down?
- Yes, but only temporarily
- A well-factored program is easier to optimize

# Refactoring and Performance

**The best way to optimize performance is to first write a well factored program, then optimize it.**

**Most of a program's time is taken in a small part of the code**

**Profile a running program to find these "hot spots"**

You won't be able to find them by eye

**Optimize the hot spots, and measure the improvement**

McConnell Steve, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993

ThoughtWorks®
The art of heavy lifting.™

# Problems with Refactoring

- **We don't know what they all are yet**
- **Database Migration**
  - Insulate persistent database structure from your objects
- **Published Interfaces**
  - Publish only when you need to
  - Don't publish within a development team
- **Without working tests**
  - Don't bother

# Design Decisions

- **Planned design**
  - Consider current needs and possible future needs
  - Design to minimize change with future needs
  - Patch code if unforeseen need appears
- **Evolutionary design**
  - Consider current needs and possible future needs
  - Trade off cost of current flexibility versus cost of later refactoring
  - Refactor as changes appear

martinfowler.com/articles/designDead.html

# Design Decisions

- One benefit of objects is that they make software easier to change.
- Refactoring allows you to improve the design after the code is written
- Up front design is still important, but not so critical