REFACTORINGS

What is Refactoring?

Refactoring is

- a change made to the internal structure of software
- to make it ***easier to understand*** and ***cheaper to modify***
- without changing its observable behavior.

-- W. Opdyke
-- M. Fowler

Refactoring 1999
Refactoring
Refactoring
Refactoring
Test-Driven
Refactoring in
Refactoring
Refactoring with Microso... Peter Ritchie...
Clean Code in Python: R... Mariano Ana...
Pro PHP Refactoring 2010
Professional refactoring i... Danijel Arse...
Professional Refactoring i... Danijel Arse...
Software Design X-Ra... Adam Tornhi...
CSS Refactoring: ... Steve Lindst...
Refactoring HTML: Impr... Elliotte Rust...

What Motivates to Refactor?

- Refactoring helps you find bugs.
- Gain a better understanding of a code.
- Improve the design of existing code.
- Make it easier to add new features.
- Make coding less annoying.

Believe or not, refactoring helps you program faster!

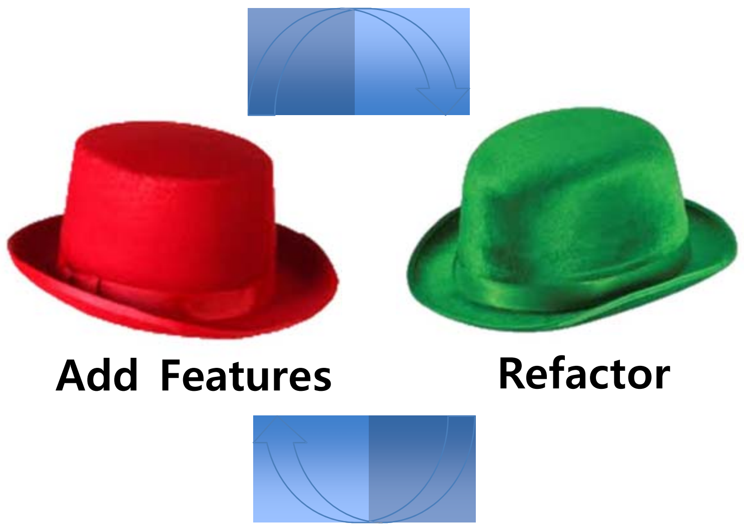
Refactoring Process

```
public class Refactor {
    private Software target;
    public Software refactor() {
        doCleaningWithUnitTest();
        if (noSmell())
            return target;
        return refactor();
    }
    ...
}
```

Refactoring Approaches

- Iterative cycles
- Reveal Intentions
- Small steps – one smell at a time
- Pairs & peers
- Exploratory Refactoring
- Sometimes it gets worse before it gets better.
- Use tools

Iterative Cycles



Reveal Intentions

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."

Clear intent does not mean … Familiar!

Find the square of the second even number which is greater than 7.

```
Integer find(List<Integer> ints) {
  int count = 0; int ans = 0;
  for (Integer num : ints) {
    if (num % 2 == 0) {
      if (num > 7) {
        count++;
        if (count == 2) {
          ans = num * num;
          break;
        }
      }
    }
  }
  return ans;
}
```

It means Clean and Simple!

```
Optional<Integer> find(List<Integer> ints) {
  return(
    ints.stream()
        .filter(n -> n % 2 == 0)
        .filter(n -> n > 7)
        .skip(1)
        .map(n -> n * n)
        .findFirst()
);
```

Small Steps – One Smell At A Time

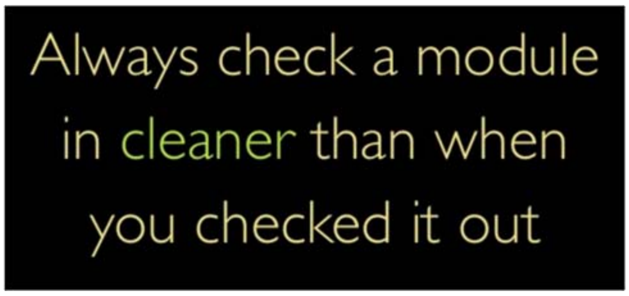
Code smells

"warning sign about potential problems"

God class
Long method
Duplicated code
Intention hiding names
...

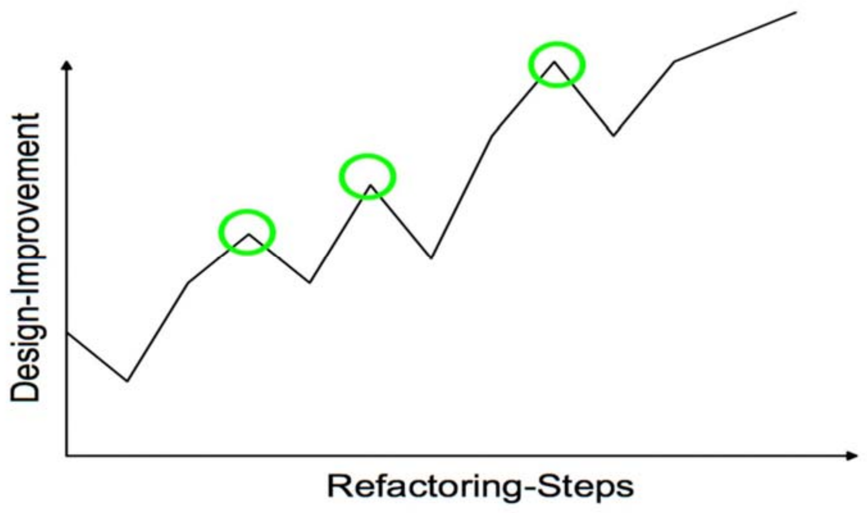
Exploratory Refactoring

- Refactor as you go.
- Clean what you're working on
  - Always leave it in a better state than when you started
  - Add Test
- Persevere!



Always check a module in cleaner than when you checked it out

Things will get worse before it gets better.



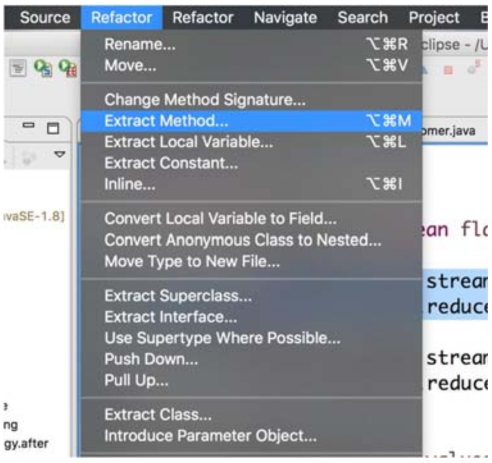
Use Tools

“A fool with a tool is still a fool. However, a genius without a tool is like a mule.”

Tool support is essential to

- Avoid errors
- Check for consistency
- Provide what-if simulations
- Provide and manage key refactorings to engineers
- Increase efficiency

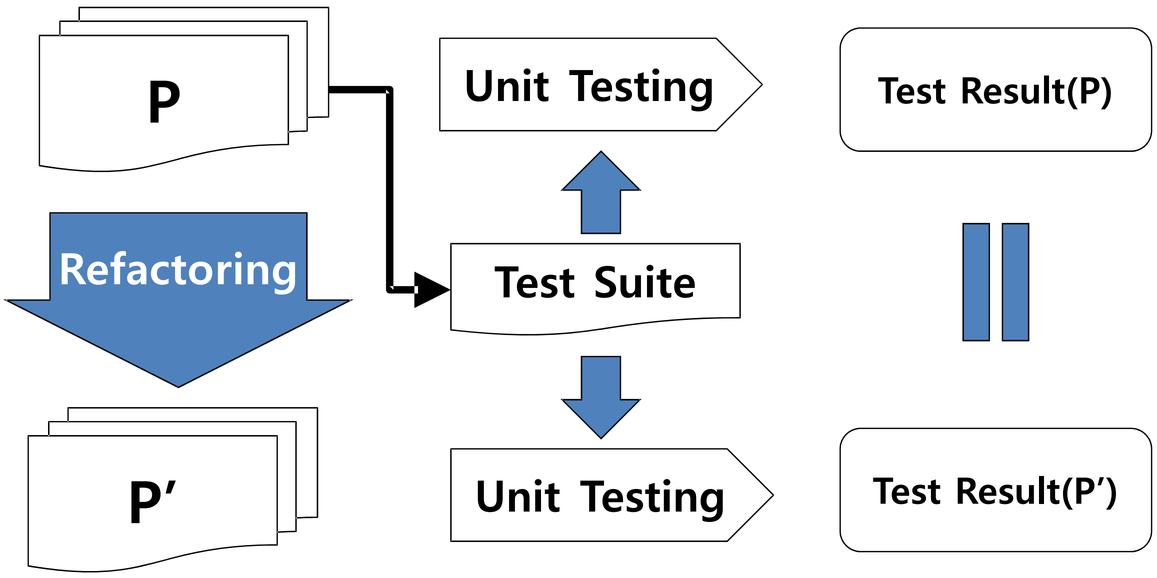
Use Tools



- Java
  - Eclipse JDT / IntelliJ IDEA / NetBeans
- .NET
  - Visual Studio (for .NET)
  - ReSharper (addon for Visual Studio)
  - Refactor Pro (addon for Visual Studio)
  - Visual Assist (addon for Visual Studio with refactoring support for VB, VB.NET. C# and C++)
- DMS Software Reengineering Toolkit
  - Support large-scale refactoring for C, C++, C#, COBOL, Java, PHP and other languages (https://en.wikipedia.org/wiki/DMS_Software_Reengineering_Toolkit)

How to Guarantee Behavior Preservation

- Refactoring is tightly coupled with Unit Testing.



Refactoring and Design

- Upfront Design *versus* Refactoring
- Upfront design might causes "**needless complexity**"

With design I can think very fast,
but my thinking is full of little holes.

-- Alistar Cockburn

Refactoring and Performance

- Remember 80/20 rule.
- Benefits of well-factored program:
  1. First, it gives you time to spend on performance tuning.
  2. Second, with a well-factored program you have finer granularity for your performance analysis.

Make it work. Make it right. Make it faster.

Refactoring and Performance
(Cont'd)

"Premature optimization is the root of all evil"

"… pointed out that it is essentially impossible to predict where the bottlenecks are in a program--you need to use a profiler to actually measure what the code is doing"

-- Donald Knuth

Refactoring and Performance
(Cont'd)

Even if you know exactly what is going on in your system, measure performance, don't speculate.
You'll learn something, and nine times out of ten, it won't be that you were right!

-- Ron Jeffries

Patterns *vs.* Refactorings

Based on characteristics, scope, and the impact:

- Architectural Pattern
- Design Pattern
- Idioms

- Architectural Refactoring
- Design Refactoring
- Code Refactoring

Smells & Refactorings

Based on characteristics, scope, and the impact:

- Architecture Smells
- Design Smells
- Code Smells

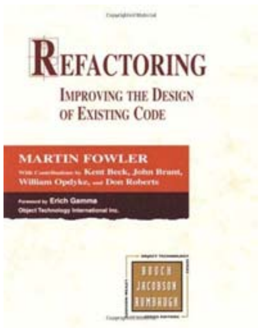
- Architecture Refactoring
- Design Refactoring
- Code Refactoring

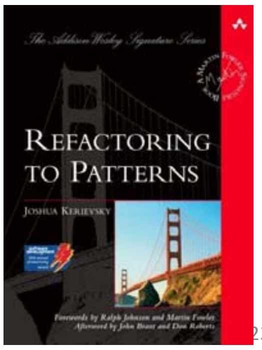
Code smells have limited scope (typically confined to a class (or file)) and have a limited local impact.

On the other hand, architecture smells span to multiple components, have a system level impact.

Low-level *vs*. Composite Refactoring

- Low-level refactorings
  - Most of classic code refactorings
  - Much of the work performed by low-level refactorings involves moving code around.
    - Extract Method/Pull Up Method/Extract Class/Move Method
- Composite refactorings
  - Composite refactorings are high-level refactorings composed of low-level refactorings.


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma


The Addison Wesley Signature Series
REFACTORING TO PATTERNS
JOSHUA KERIEVSKY
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

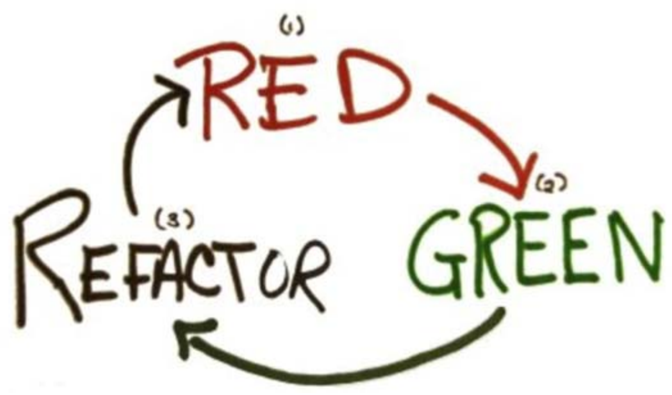
Composite Refactoring

- Between applying low-level refactorings, run (unit) tests to confirm functionality preservation.
- Testing is thus an integral part of composite refactoring → *Safety Net !!*

Test-Driven Refactoring

- Testing can also be used to "**rewrite and replace**" old code with the new code.
- When composite refactoring isn't possible to improve a design, test-driven refactorings can help you produce a better design safely and effectively.
  - **Substitute Algorithm** [F]
  - **Encapsulate Composite with Builder** (96)
  - **Replace Implicit Tree with Composite** (178)
  - **Move Embellishment to Decorator** (144)

TDD Red/Green/Refactor cycle



1. Create a unit tests that fails.
2. Write production code that makes that test pass.
3. Clean up the mess you just made.

Three Laws of TDD

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

When Not to Refactor?

- Existing code is such a mess that rewrite from scratch would be easier than refactoring.
- When the current code just does not work.
- When you are close to a deadline.
  - – Technical Debts