# Refactoring for C# Developers

## INTRODUCING REFACTORING AND CODE SMELLS

**Steve Smith**
FORCE MULTIPLIER FOR DEV TEAMS
@ardalis | ardalis.com | weeklydevtips.com

# Objectives

What is *refactoring*?

Why do it?

When and how to do it?

What are *code smells*?

How should you deal with them?

# Refactoring (noun)

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



Source: *Refactoring*, Martin Fowler

# Refactoring (verb)

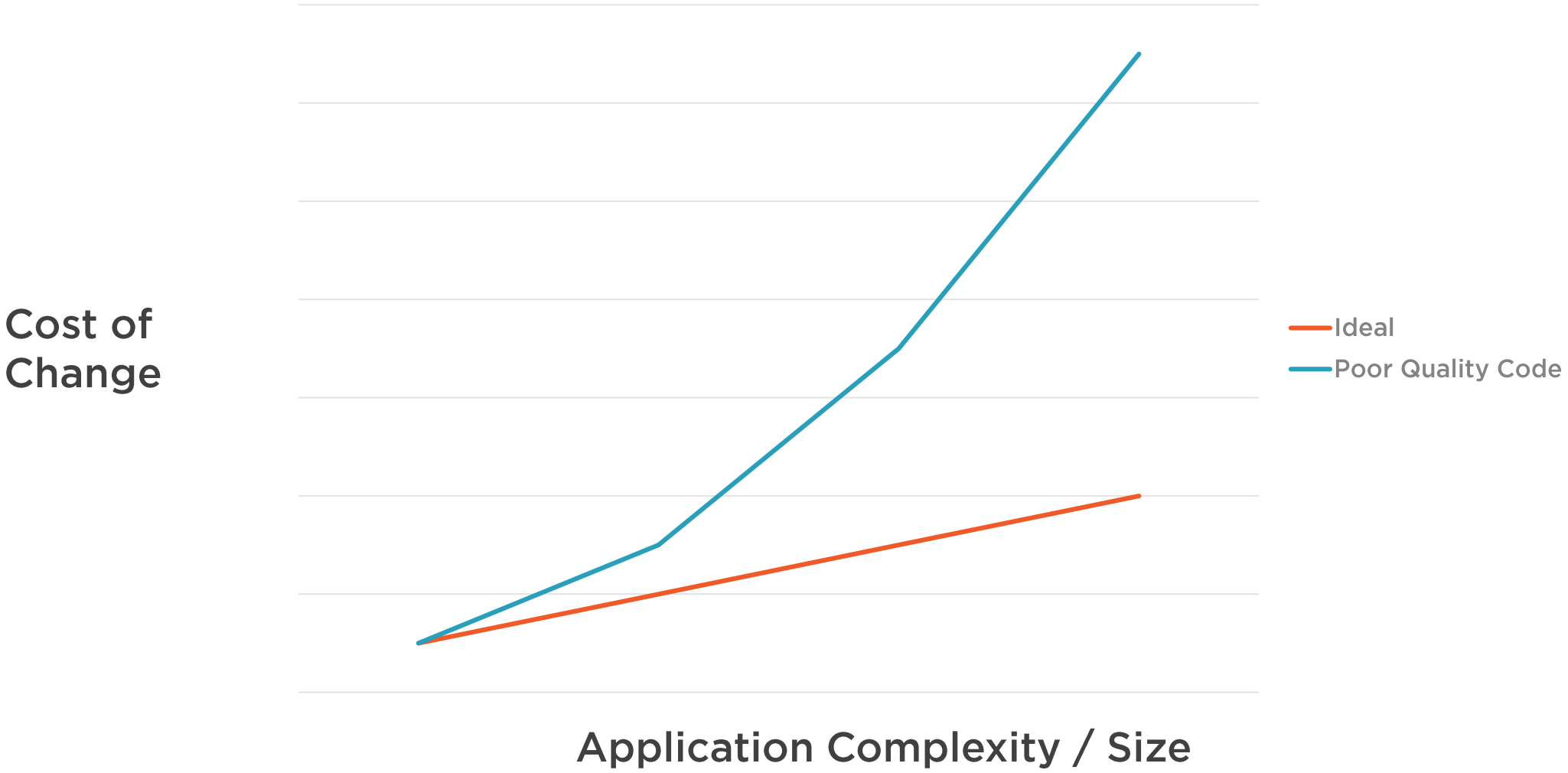
*To restructure software by applying one or more refactorings.*



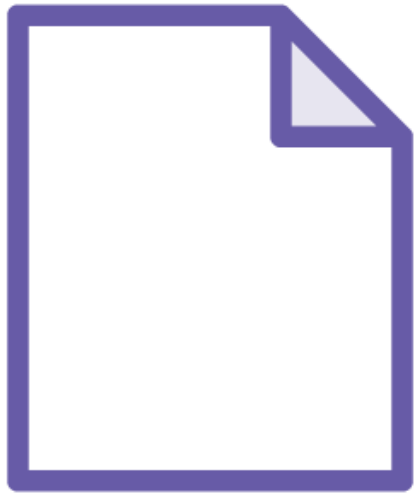
Source: *Refactoring*, Martin Fowler

# Why refactor?

# Why Refactor?



# Refactoring is Editing



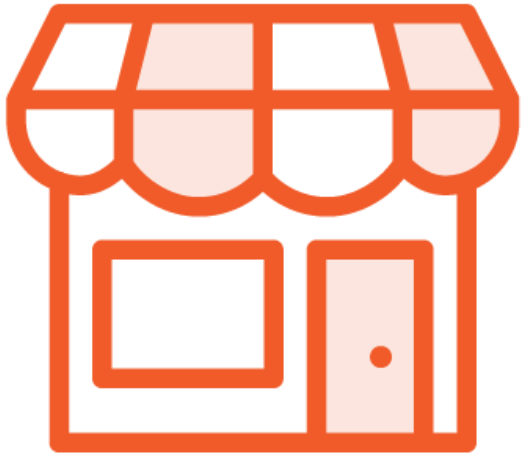
First Draft

First Revision

Final Version

# When Should You Refactor?

You



Your Restaurant

Your Customers

# When to Refactor

After writing a unit test (TDD)

If the current design is causing you pain (PDD)

As part of fixing a bug

As part of a code review

# When Not to Refactor

Current Code Doesn't Work

Massive Technical Debt

Imminent Deadline

“Other than when you are very close to a deadline... you should not put off refactoring because you haven’t got time.”

**Martin Fowler, *Refactoring***

# Boy Scout Rule

Leave your code better than you found it.

# The Refactoring Process

**Commit** (or back up) current working code

**Verify** existing behavior (ideally with automated tests)

**Apply** a Refactoring

**Confirm** original behavior has been preserved

# Writing Characterization Tests

Write a test you know will fail

Use the failing test output to capture current behavior

Update the test to assert the current behavior

Run the test again; it should pass

# Demo

## Writing Characterization Tests

# Refactoring Toward Cleaner Code

Remove Duplication

Improve Naming

Break Up Large Code Elements

Reduce Coupling

Reduce Complexity

Split Responsibility

# Code smell

A code smell is a surface indication that usually corresponds to a deeper problem in the system



Source: https://martinfowler.com/bliki/CodeSmell.html

# Principle of Least Astonishment

"Do what users expect"

Design APIs from the perspective of programmers who will consume them

Be simple

Be clear

*Be* consistent

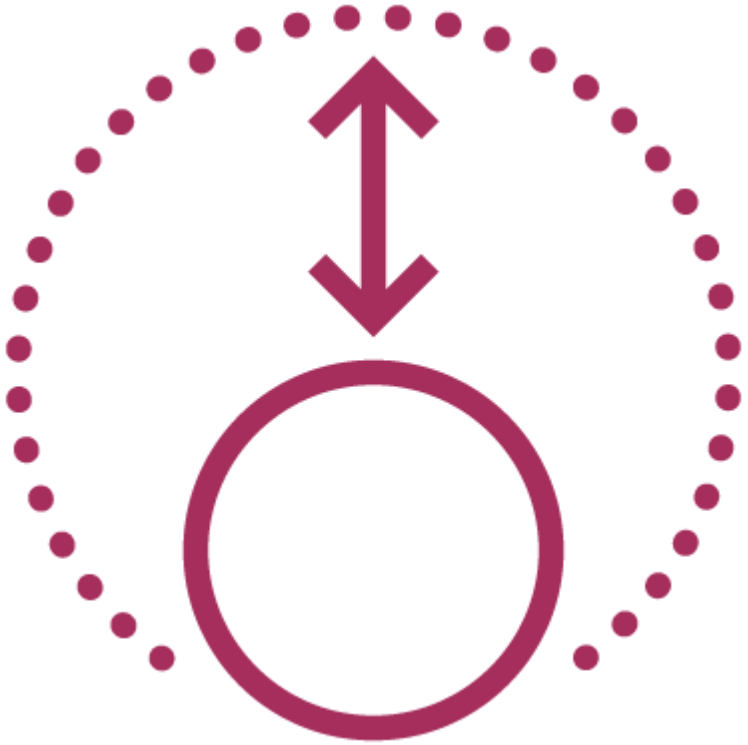
Source: https://en.wikipedia.org/wiki/Principle_of_least_astonishment

# Classifications of Code Smells

| | | |
|---|---|---|
| Bloaters | Object-Orientation abusers | Change preventers |
| Dispensables | Couplers | Obfuscators |



Source: http://mikamantyla.eu/BadCodeSmellsTaxonomy.html

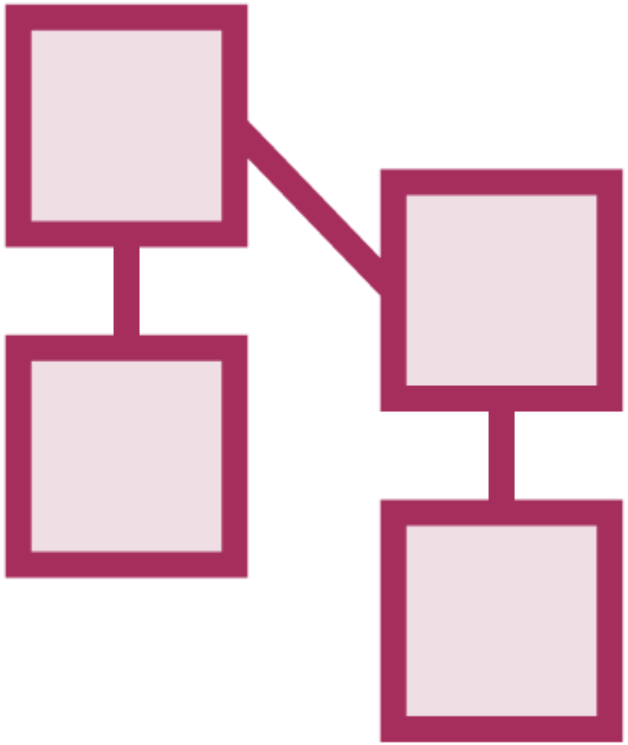


# Bloaters

Make codebase bigger than necessary

Usually impact code slowly over time

Are prevented by lean, focused code

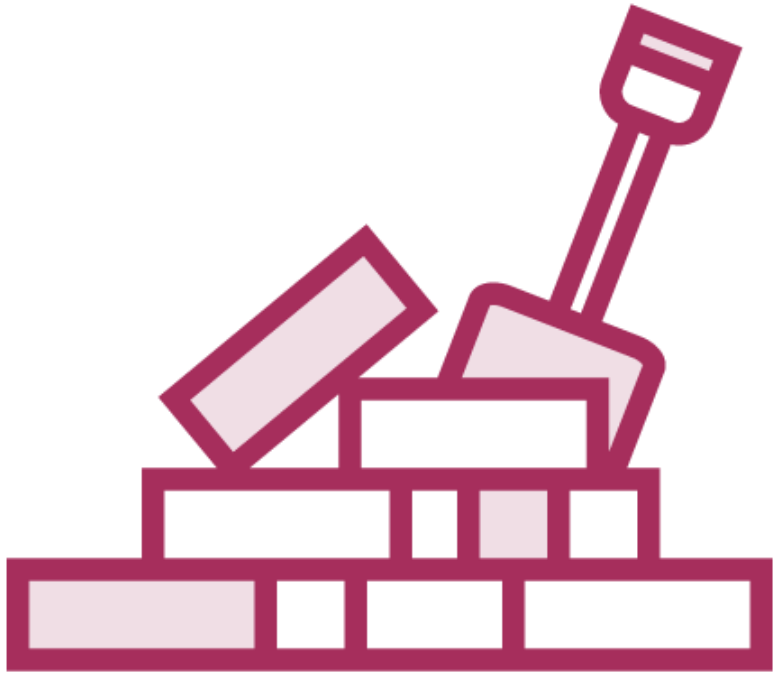


# Object-Orientation Abusers

Break polymorphism

Create inappropriate tight coupling

Require repetition



# Change Preventers

Touch many parts of the system

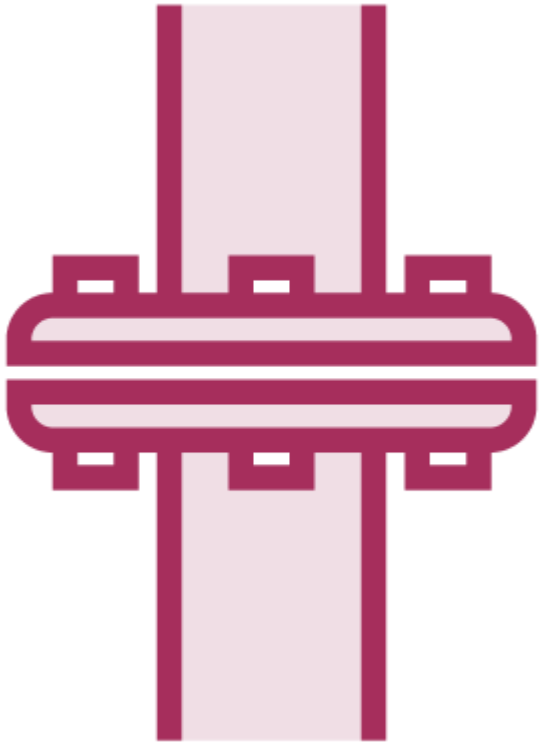
Create tight coupling

Lack separation of concerns

# Dispensables

Provide little or no value

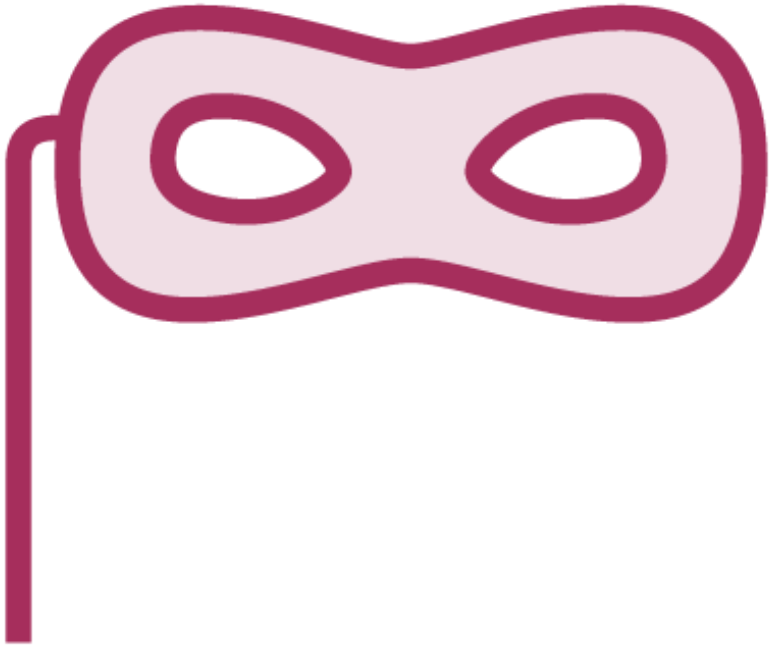
Can be safely removed with little/no effort



# Couplers

Introduce excessive coupling

Tie unrelated part of the system together



# Obfuscators

Impede clear communication

Hide intent

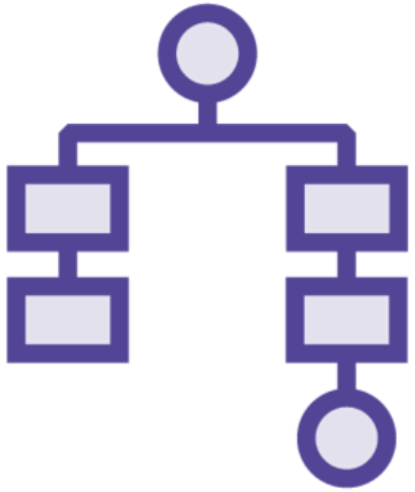
Confuse the reader

# Code Smells by C# Organizational Hierarchy

```
var a = Process();
```

Statement Smells

Method Smells



Class Smells

# Key Takeaways

**Refactoring** improves design without changing behavior

Refactor while adding value

Recognize common code smells

Organize code smells by impact and code hierarchy