Refactoring

Improving code

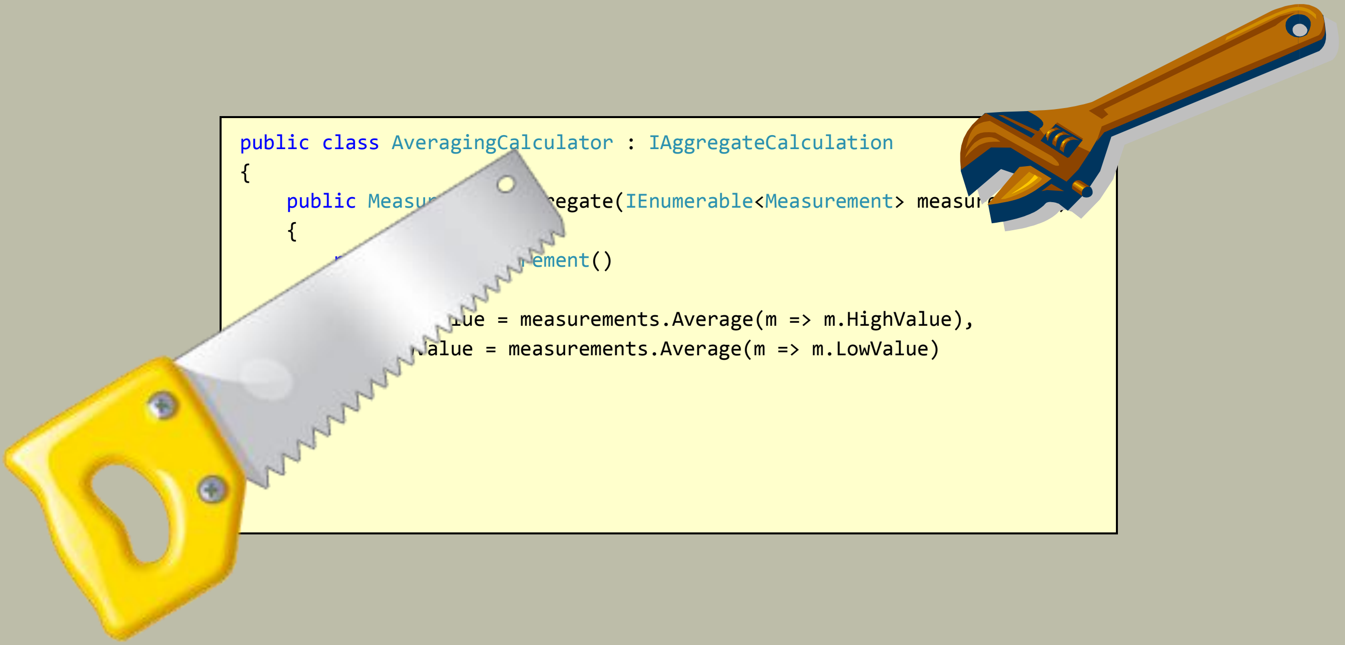
“**Software** must be soft: it has to be easy to change because it **will change** despite our misguided efforts otherwise.”

The Pragmatic Programmers

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
-Martin Fowler et al, Refactoring: Improving the Design of Existing Code, 1999

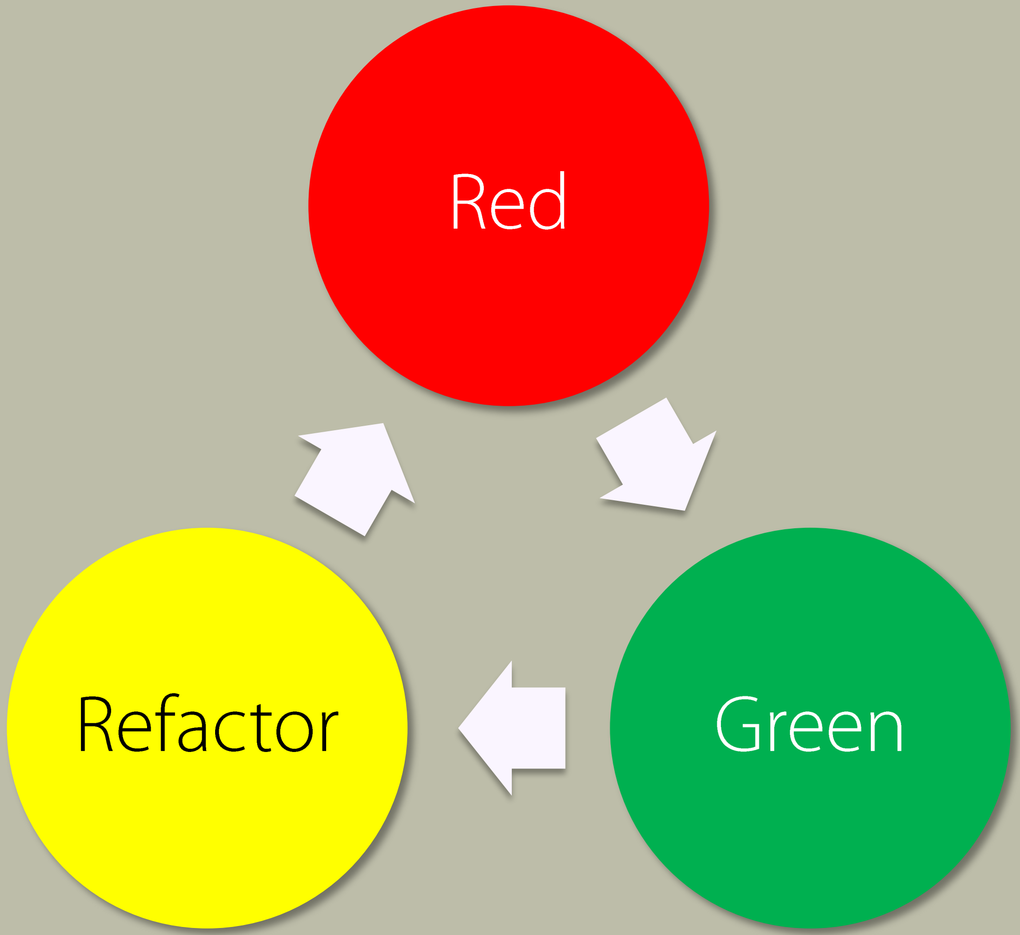
What is it?

- Change the implementation
- Preserve the external functionality



```
public class AveragingCalculator : IAggregateCalculation
{
    public Measu          egate(IEnumerable<Measurement> measur
    {
                          ement()

                  lue = measurements.Average(m => m.HighValue),
                 alue = measurements.Average(m => m.LowValue)
```

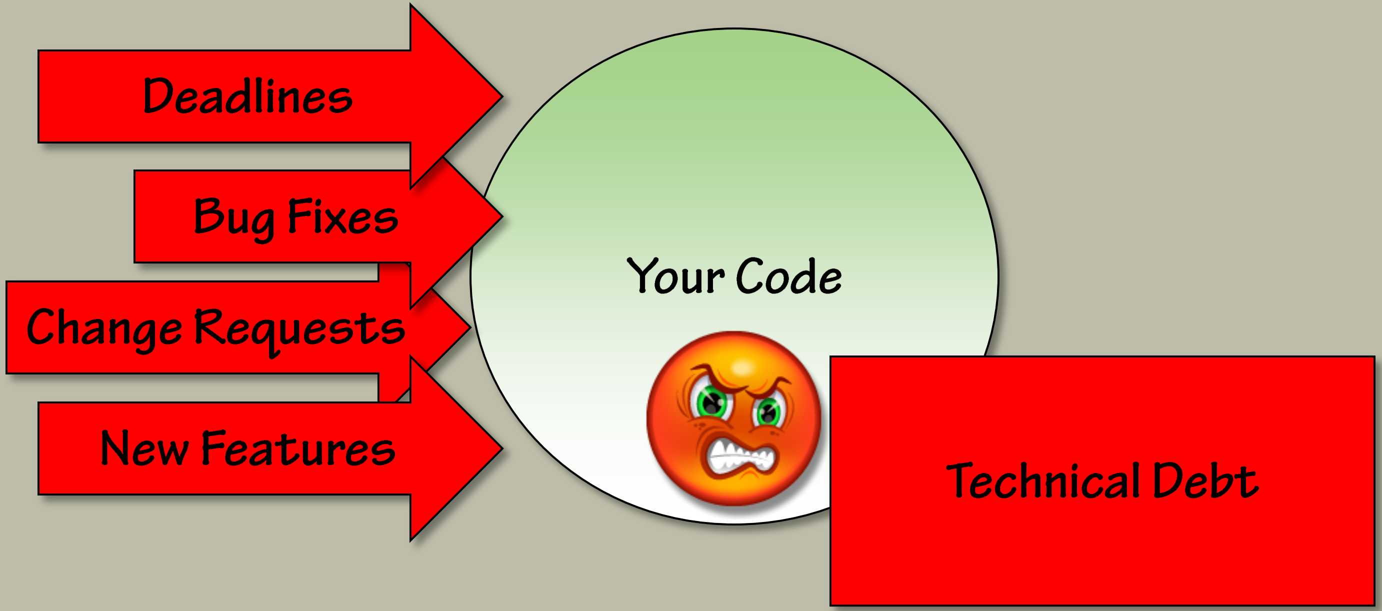
The Mantra



When Are You Done?

Why Refactor?

- To improve the an “ility” of code
  - Readability
  - Maintainability
  - Even scalability, extensibility



When To Refactor?

- After fixing a failing test (red-green-refactor)
- Before adding a new feature
- After identifying a quality problem

Complex If/Else

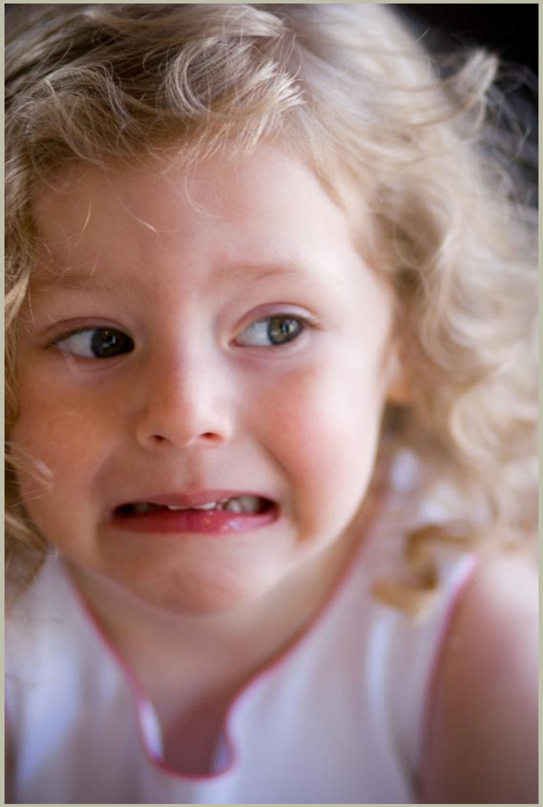
Large class

Duplicated Code

```
public class AveragingCalculator : IAggregateCalculation
{
    public Measurement Aggregate(IEnumerable<Measurement> measurements)
    {
        return new Measurement()
        {
            HighValue = measurements.Average(m => m.HighValue),
            LowValue = measurements.Average(m => m.LowValue)
        };

}

         s AveragingCalculator : IAggregateCalculation

          asurement Aggregate(IEnumerable<Measurement> measurements)

            ew Measurement()

            HighValue = measurements.Average(m => m.HighValue),
            LowValue = measurements.Average(m => m.LowValue)
        };
    }
}

public class AveragingCalculator : IAggregateCalculation
{
    public Measurement Aggregate(IEnumerable<Measurement> measurements)
    {
        return new Measurement()
        {
            HighValue = measurements.Average(m => m.HighValue),
            LowValue = measurements.Average(m => m.LowValue)
        };
    }
}

public class AveragingCalculator :
IAggregateCalculation
{
    public Measurement
Aggregate(IEnumerable<Measurement> measurements)
    {
        return new Measurement()
        {
            HighValue = measurements.Average(m =>
m.HighValue),
            LowValue = measurements.Average(m =>
m.Low     )

               ragingCalculator :
              ulation

    publ   Measurement
Aggregate(IEnumerable<Measurement> measurements)
    {
        return new Measurement()
        {
            HighValue = measurements.Average(m =>
m.HighValue),
            LowValue = measurements.Average(m =>
m.LowValue)
        };
```

When NOT to Refactor?

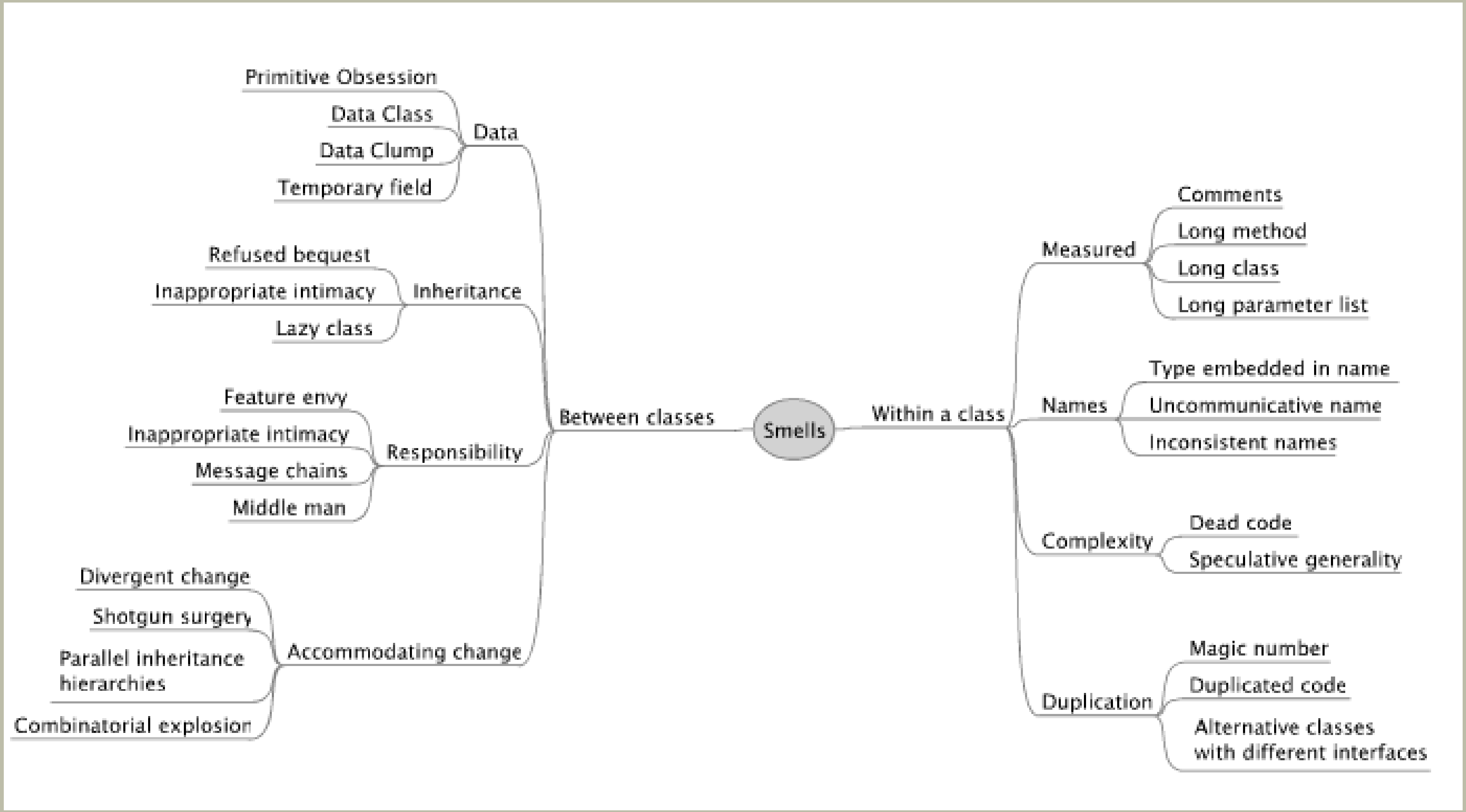
- When you don’t have tests!



```
public class AveragingCalculator : IAggregateCalculation
{
    public Measurement Aggregate(IEnumerable<Measurement> measurements)
    {
        return new Measurement()
        {
            HighValue = measurements.Average(m => m.HighValue),
            LowValue = measurements.Average(m => m.LowValue)
        };
    }
}
```

OdeToCode.com

Code Smells (Fowler Refactoring)



OdeToCode.com

Common Refeactorings

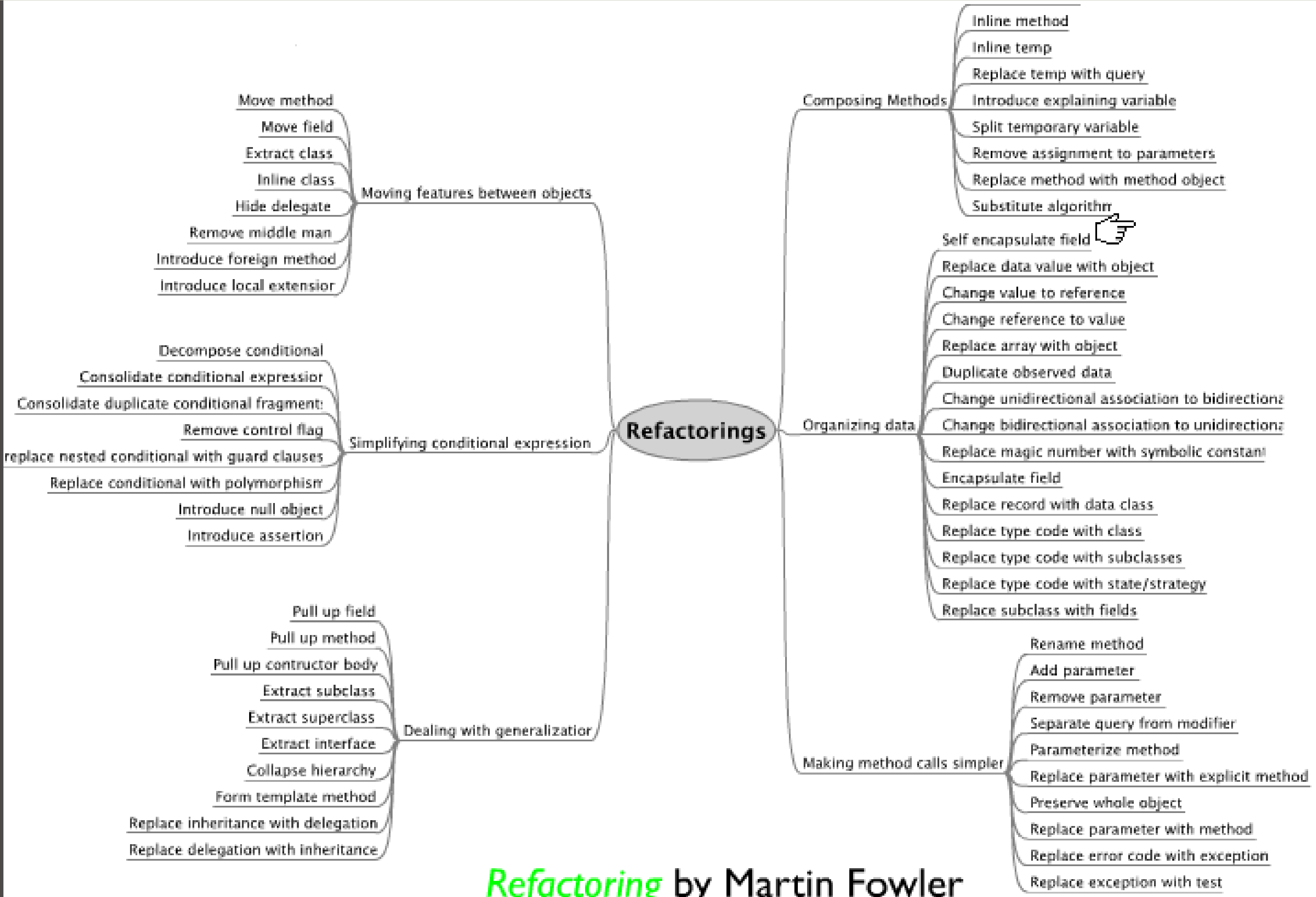
- Extract method
- Rename
- Introduce parameter

Refactoring To Abstractions

- Extract interface
- Extract superclass

Refactoring To Design Patterns

- Test have a knack of finding the flaws in an API
- Decorator
- Command
- Strategy
- Builder
- Façade


Refactorings
Moving features between objects
Move method
Move field
Extract class
Inline class
Hide delegate
Remove middle man
Introduce foreign method
Introduce local extensior
Simplifying conditional expression
Decompose conditional
Consolidate conditional expressior
Consolidate duplicate conditional fragment:
Remove control flag
replace nested conditional with guard clauses
Replace conditional with polymorphisr
Introduce null object
Introduce assertion
Dealing with generalizatior
Pull up field
Pull up method
Pull up contructor body
Extract subclass
Extract superclass
Extract interface
Collapse hierarchy
Form template method
Replace inheritance with delegation
Replace delegation with inheritance
Composing Methods
Inline method
Inline temp
Replace temp with query
Introduce explaining variable
Split temporary variable
Remove assignment to parameters
Replace method with method object
Substitute algorithrr
Organizing data
Self encapsulate field
Replace data value with object
Change value to reference
Change reference to value
Replace array with object
Duplicate observed data
Change unidirectional association to bidirectiona
Change bidirectional association to unidirectiona
Replace magic number with symbolic constan
Encapsulate field
Replace record with data class
Replace type code with class
Replace type code with subclasses
Replace type code with state/strategy
Replace subclass with fields
Making method calls simpler
Rename method
Add parameter
Remove parameter
Separate query from modifier
Parameterize method
Replace parameter with explicit method
Preserve whole object
Replace parameter with method
Replace error code with exception
Replace exception with test

Refactoring by Martin Fowler

The Synergy Between Testability and Design

- **Test can tell you about design problems**
  - Iceberg classes
  - State hidden in methods
  - Difficult setup
  - State leaks across tests
  - Environmental dependencies
  - Framework frustration
  - Difficult mocking
  - Hidden effects
  - Test thrash

Summary