Refactoring

how to change code without breaking it

Refactoring

is a technique of modifying
internal structure of code,
without changing its behaviour.

Refactoring
is based on a series of transformations,
which retain the code semantics.

We refactor
to keep / improve quality
of the application.

PRAGMATISTS

What is quality?

- Readability
  (literate programming)

- Low accidental complexity
  (high cohesion/low coupling)

- Ability to easily modify / extend functionalities

- Maintainability

By continuously improving the design of code, we make it easier and easier to work with.

This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features.

If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.

Joshua Kerievsky,
Refactoring to Patterns

Code smells

- Duplicate code

  identical or similar pieces of code
- Large method

  more than 5-7 lines?
- Large class

  more than 5-7 methods?
- Feature envy

  code uses many methods of a different class
- Inappropriate intimacy

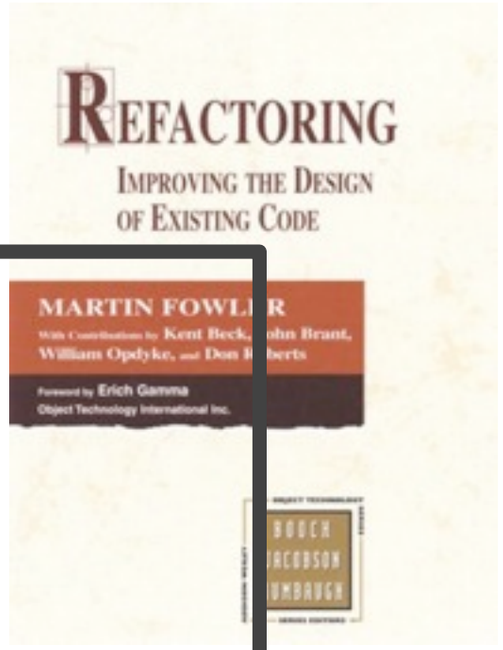
  a class is dependent on the implementation details of another class (Law of Demeter)
- Refused bequest

  a method in a subclass overloads the original method in a way that breaks its contract (Liskov)
- Lazy class

  a class does too little
- Contrieved complexity

  overdesign, use of comples solutions where a simpler one would do
- Excessively long identifiers

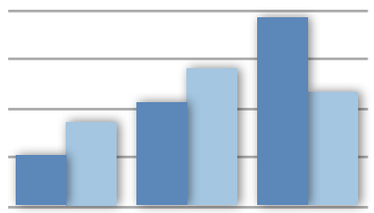
  iReallyLikeTheseLongNamesSinceTheyTellExactlyWhatAMethodDoes

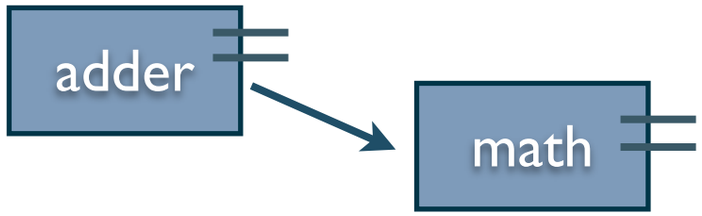


PRAGMATISTS

Increase level of abstraction

- **Encapsulate field**
  access a field through accessor methods
- **Generalize declared type**
  reuse code with more general types
- **Replace conditional with polymorphism**
- **Replace type-checking with State/Strategy**
- **Introduce indirection**






```
public class Adder {
   public int add (int a, int b) {
      return a + b;
   }
}
```

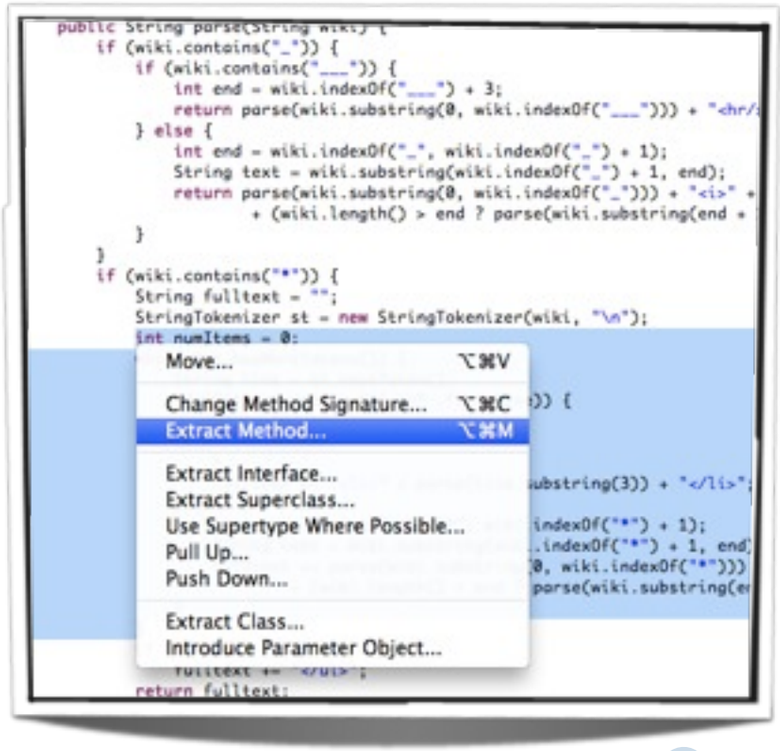
```
function add (var a: integer, b: integer): integer;
begin
   add := a + b;
end;
```

```
mov BL, 1h
fadd ST1
mov EBX, [BL*7]
push EBP
```

```
01100101011001010011001010011010
```

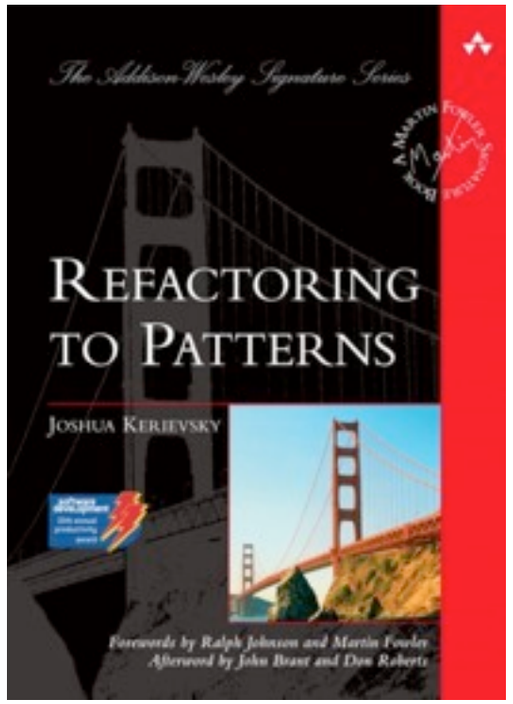
Divide code into logical parts

- **Extract method**
  create a method from some part of code
- **Inline**
  place a method's code in place of its invocation
- **Extract class**
  move some fields to a separate class
- **Extract superclass / interface**
  move some methods to superclass / interface



Naming and code location

- **Rename Method / Field**
- **Move Method / Field**
- **Pull Up / Push Down**
- **Convert local variable to field**
- **Introduce Parameter**
- **Introduce Parameter Object**
- **Convert Member Type to Top Level**



PRAGMATISTS

Catalogue of refactorings

Object creation

- *Replace constructors with creation methods*
- *Inline Singleton*

Simplifying code

- *Replace conditional logic with strategy*
- *Move embellishment to Decorator*
- *Replace conditional dispatcher with Command*

Code generalisation

- *Form Template Method*
- *Replace hard-coded notifications with Observer*

Security

- *Limit instantiation with Singleton*
- *Introduce Null-Object*

Accumulation

- *Move accumulation to collecting parameter*
- *Move accumulation to Visitor*

Exercise
Replace conditional with polymorphism

```
public class Employee {
    private Type type;
    private double base;
    private double achievements;
    private double companyResult;
    private double achievementsFactor;

    enum Type {
        SALES, HR, WORKER, CEO
    }

    public double getSalary() {
        switch (type) {
        case SALES:
            return getBase() + getAchievementsFactor() * getAchievements()
                    + getCompanyResult() * 0.0000001;
        case HR:
            return getBase() + getCompanyResult() * 0.0000002;
        case WORKER:
            return getBase();
        case CEO:
            return getBase() + getAchievements() * getAchievementsFactor()
                    + getCompanyResult() * 0.01;
        default:
            throw new IllegalStateException("Employee type unspecified");
        }
    }
}
```

```java
class Employee {
    private EmployeeType type;

    public double getSalary() {
        return type.getPaymentStrategy().getSalary(this);
    }
}
```

```java
enum EmployeeType {
    SALES(...),
    HR(...),
    WORKER(...),
    CEO(...);
}
```

```java
abstract class PaymentStrategy {

    public abstract double getSalary(Employee employee);
}
```

```java
class PaymentStrategyHr
 extends PaymentStrategy
```

```java
class PaymentStrategyCeo
 extends PaymentStrategy
```

```java
class PaymentStrategySales
 extends PaymentStrategy
```

```java
class PaymentStrategyWorker
 extends PaymentStrategy {
    @Override
    public double getSalary(Employee employee) {
        return employee.getBase();
    }
}
```