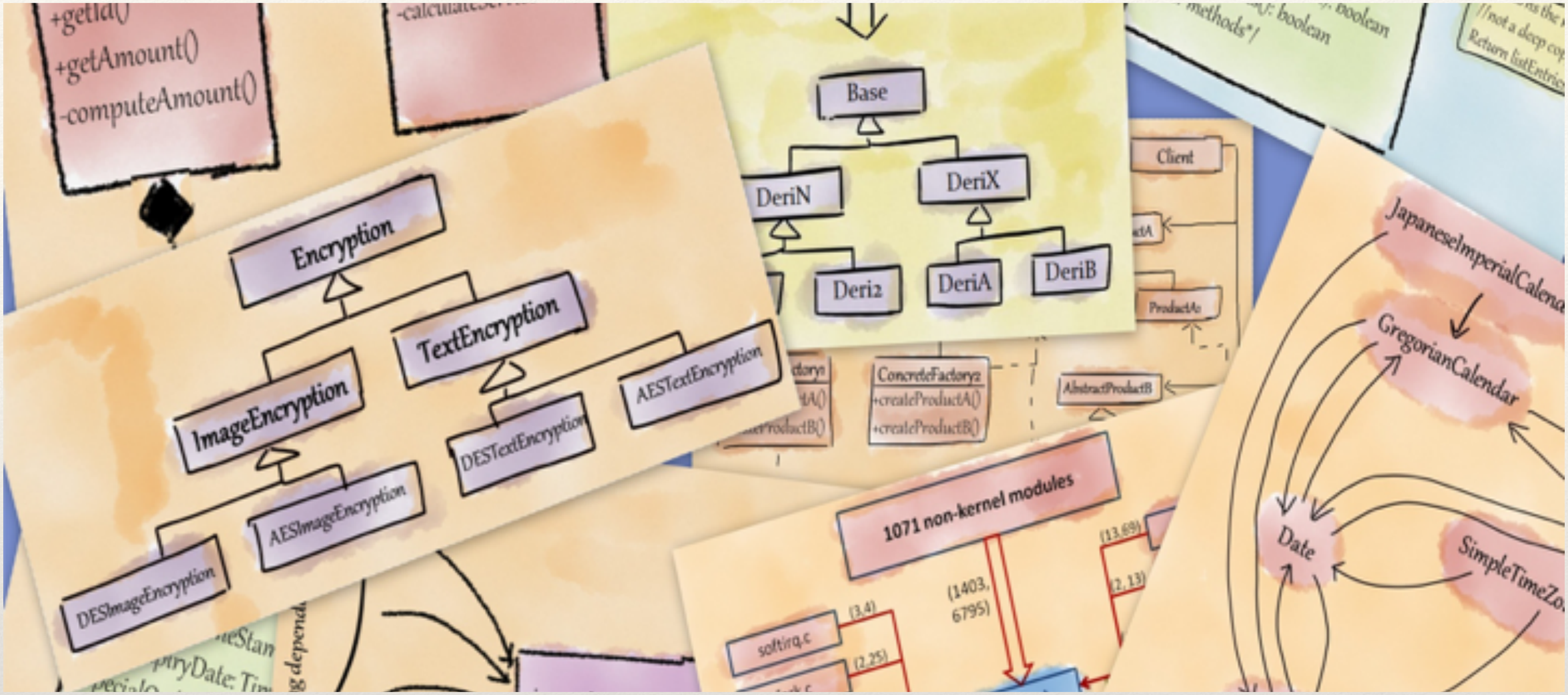



# SOLID Principles in C#

Ganesh Samarthyam
Manoj Ganapathi

"Applying design principles is the key to creating high-quality software!"

Architectural principles:
Axis, symmetry, rhythm, datum, hierarchy, transformation

# Why care about design quality and design principles?

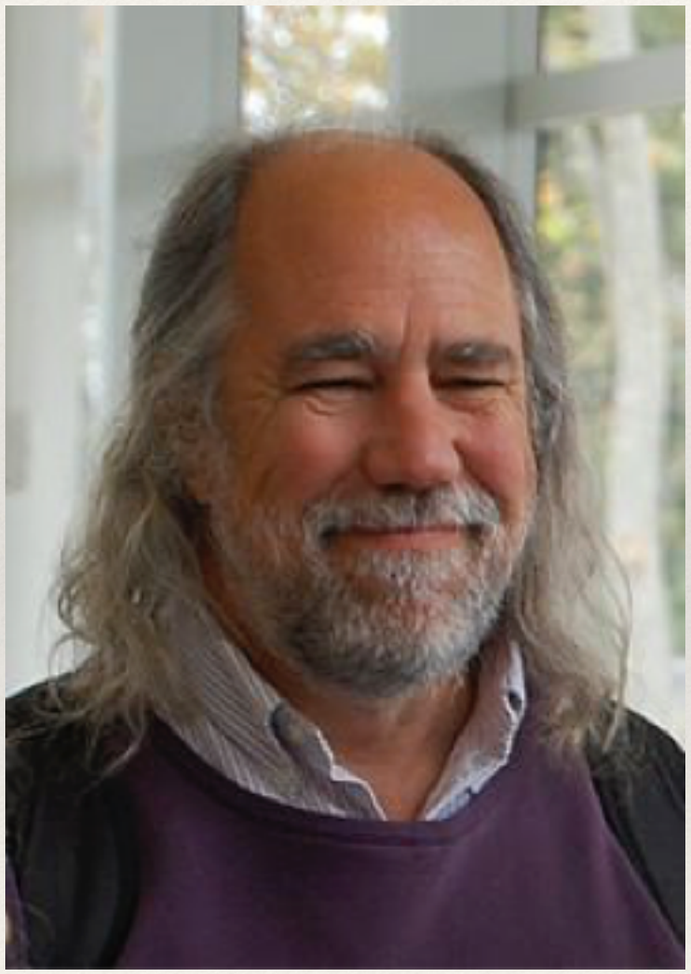
Poor software quality costs more than $150 billion per year in U.S. and greater than $500 billion per year worldwide

- Capers Jones

# The city metaphor

Source: http://indiatransportportal.com/wp-content/uploads/2012/04/Traffic-congestion1.jpg

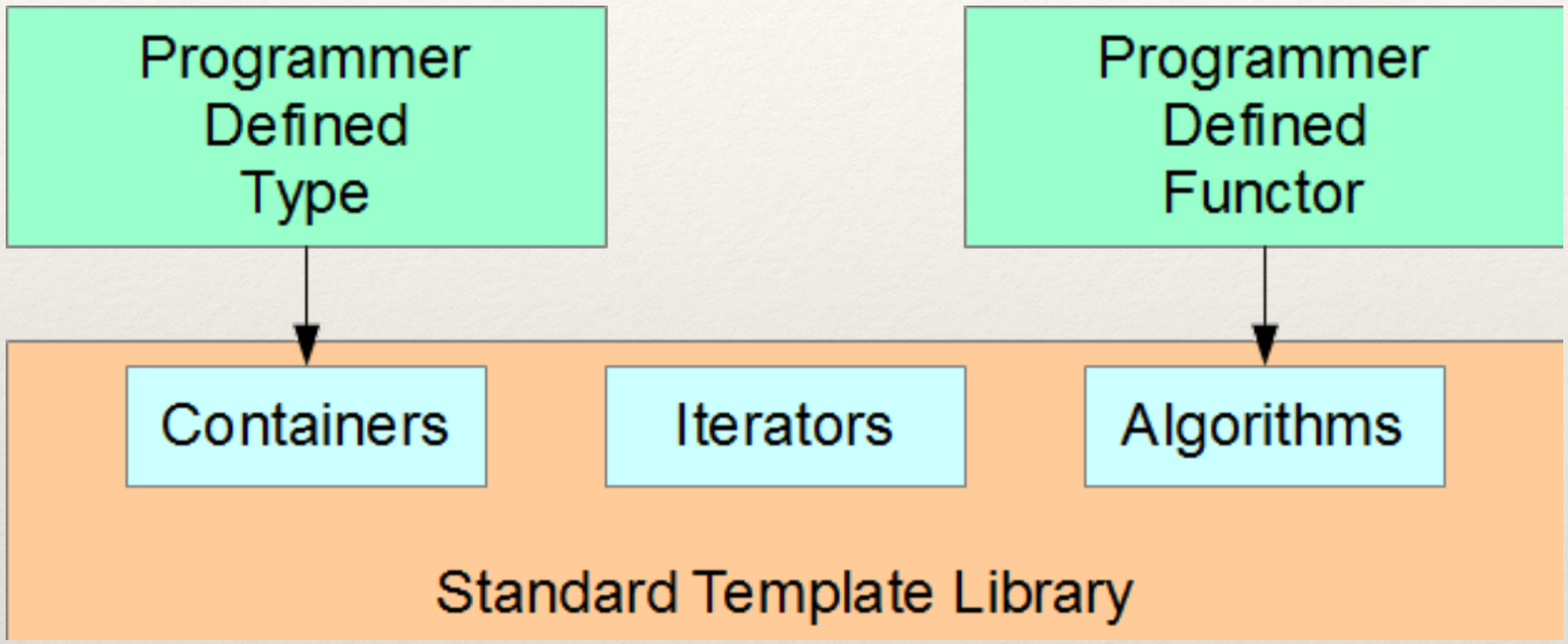
# The city metaphor



*“Cities grow, cities evolve, cities have parts that simply die while other parts flourish; each city has to be renewed in order to meet the needs of its populace… Software-intensive systems are like that. They grow, they evolve, sometimes they wither away, and sometimes they flourish…”*

Grady Booch in the foreword for “Refactoring for Software Design Smells: Managing Technical Debt”, Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Morgan Kaufmann/Elsevier, 2014.

# Example of beautiful design




```
int arr[] = {1, 4, 9, 16, 25}; // some values
// find the first occurrence of 9 in the array
int * arr_pos = find(arr, arr + 4, 9);
std::cout<< “array pos = “<< arr_pos - arr << endl;

vector<int> int_vec;
for(int i = 1; i <= 5; i++)
        int_vec.push_back(i*i);
vector<int>::iterator vec_pos = find (int_vec.begin(), int_vec.end(), 9);
std::cout<< “vector pos = “<< (vec_pos - int_vec.begin());
```

# For architects: design is the key!

# What do we mean by “principles”?

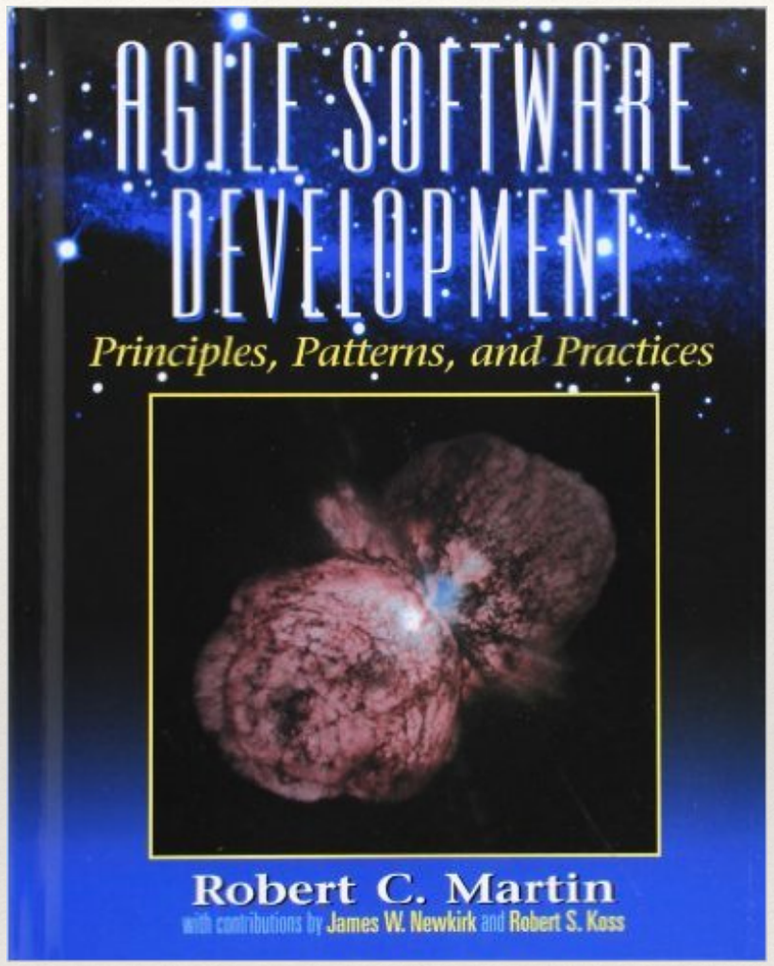
“Design principles are key notions considered fundamental to many different software design approaches and concepts.”
- SWEBOK v3 (2014)

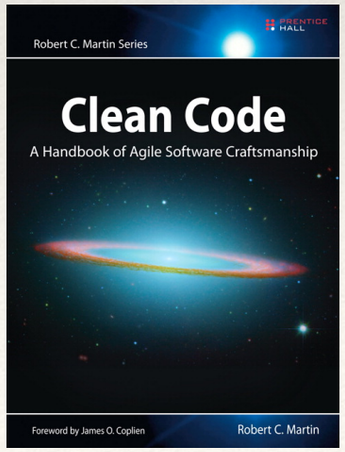
"The critical design tool for software development is a mind well educated in design principles"
- Craig Larman

# Fundamental Design Principles

# Robert C. Martin

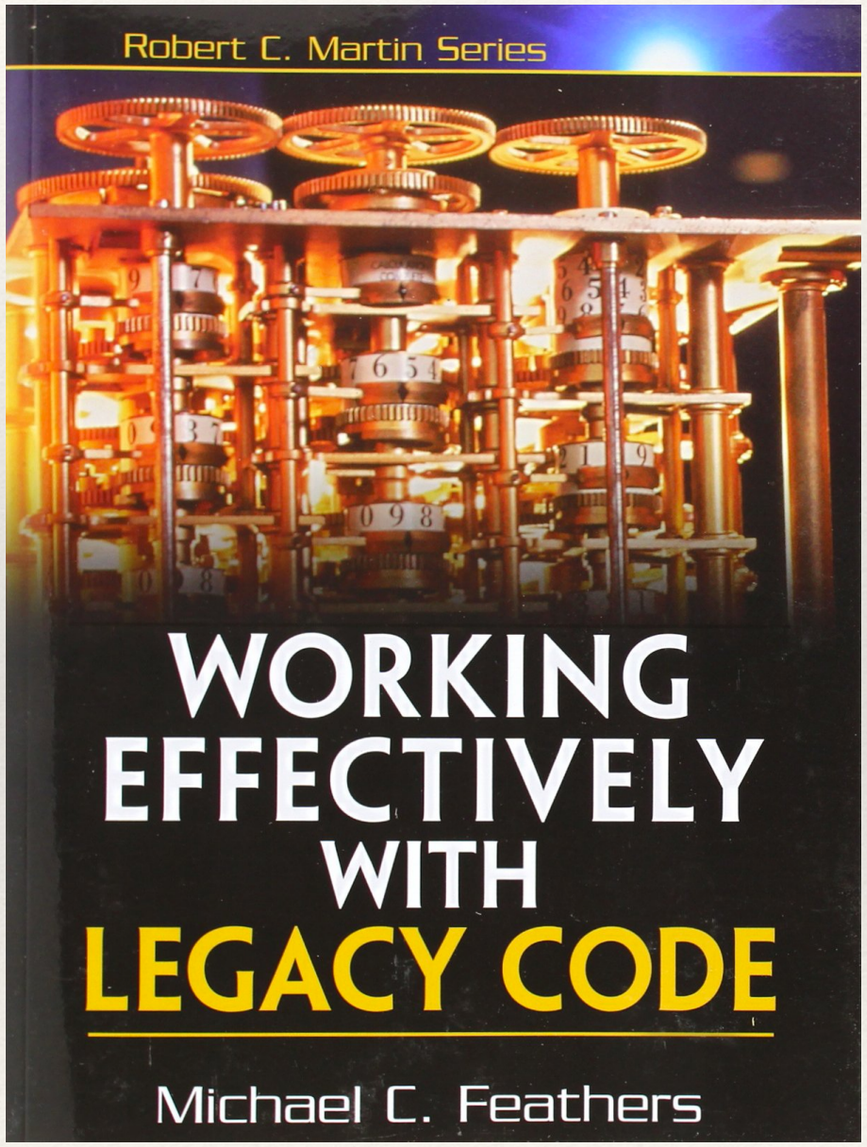
Formulated many principles and described many other important principles



# Michael Feathers

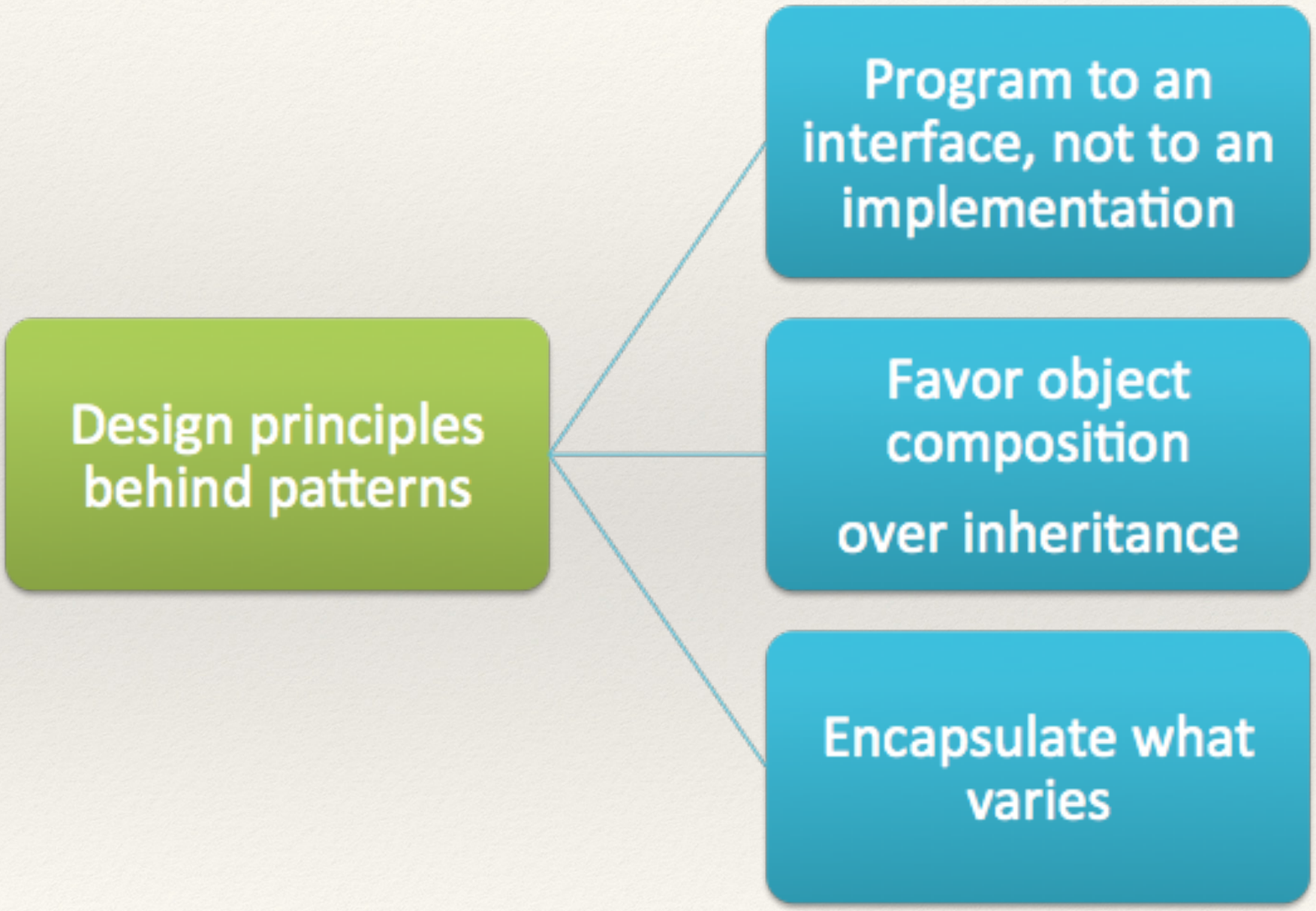
Michael Feathers coined the acronym SOLID in 2000s to remember first five of the numerous principles by Robert C. Martin



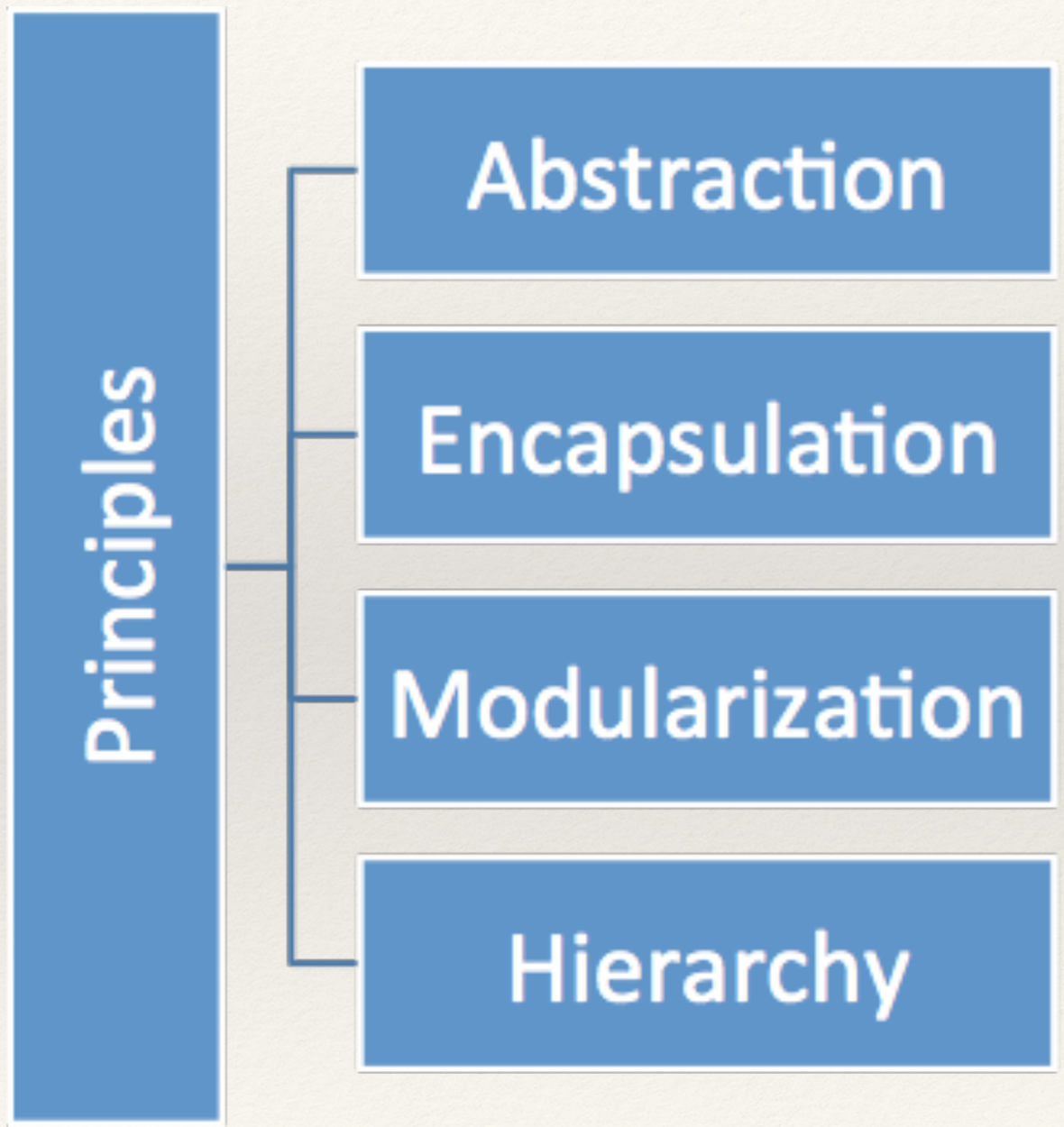
# SOLID principles

| | | |
|---|---|---|
| S | Single Responsibility Principle | Every object should have a single responsibility and that should be encapsulated by the class |
| O | Open Closed Principle | Software should be open for extension, but closed for modification |
| L | Liskov's Substitution Principle | Any subclass should always be usable instead of its parent class |
| I | Interface Segregation Principle | Many client specific interfaces are better than one general purpose interface |
| D | Dependency Inversion Principle | Abstractions should not depend upon details. Details should depend upon abstractions |

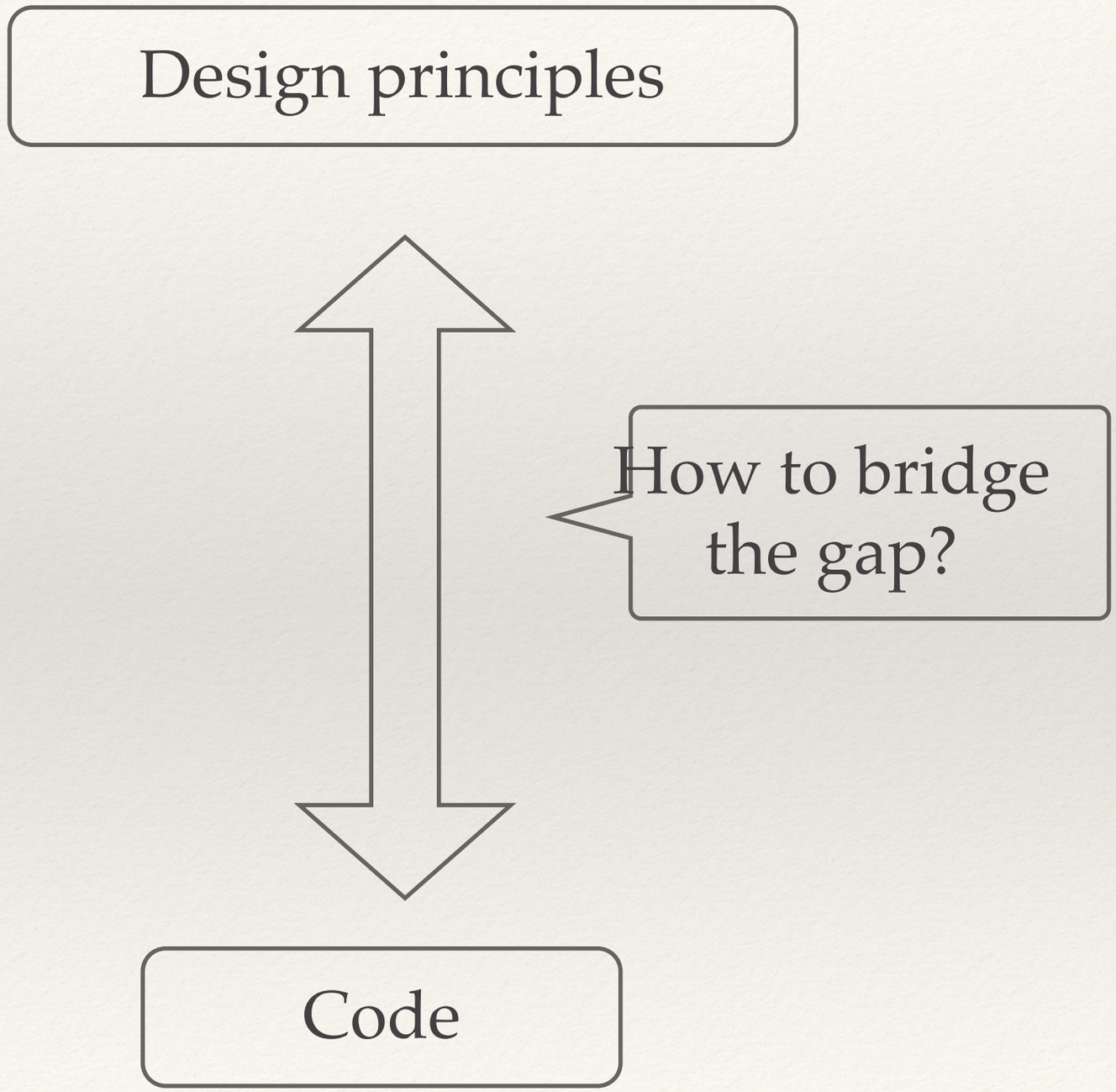
# 3 principles behind patterns



# Booch's *fundamental* principles



# How to apply principles in practice?



# Why care about refactoring?

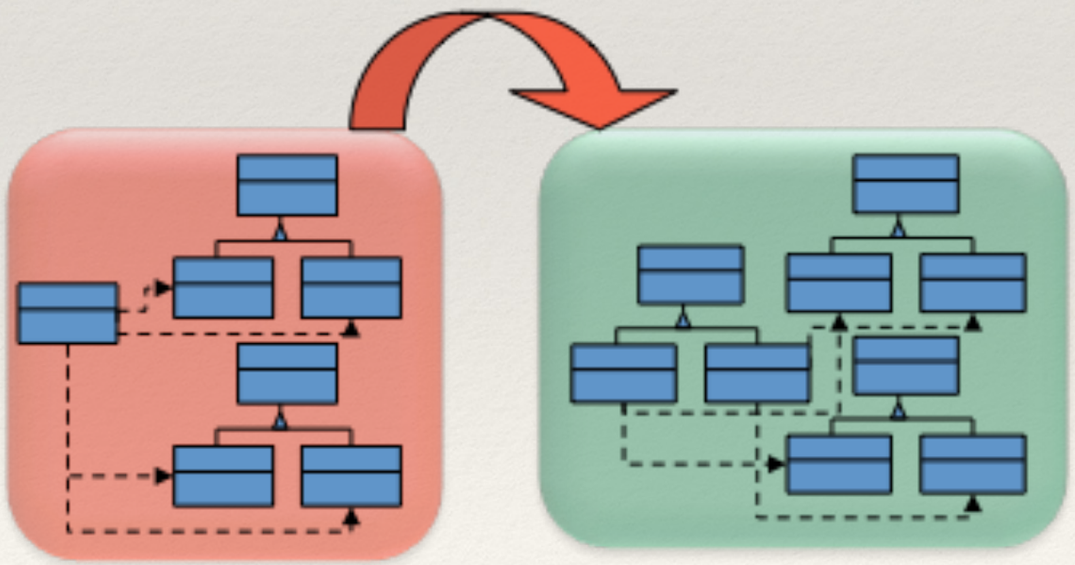
> As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it

- Lehman's law of Increasing Complexity

# What is refactoring?

**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*

**Refactor (verb)**: to restructure software by applying a series of refactorings without changing its observable behavior



# What are smells?

"Smells are certain structures
in the code that suggest
(sometimes they scream for)
the possibility of refactoring."

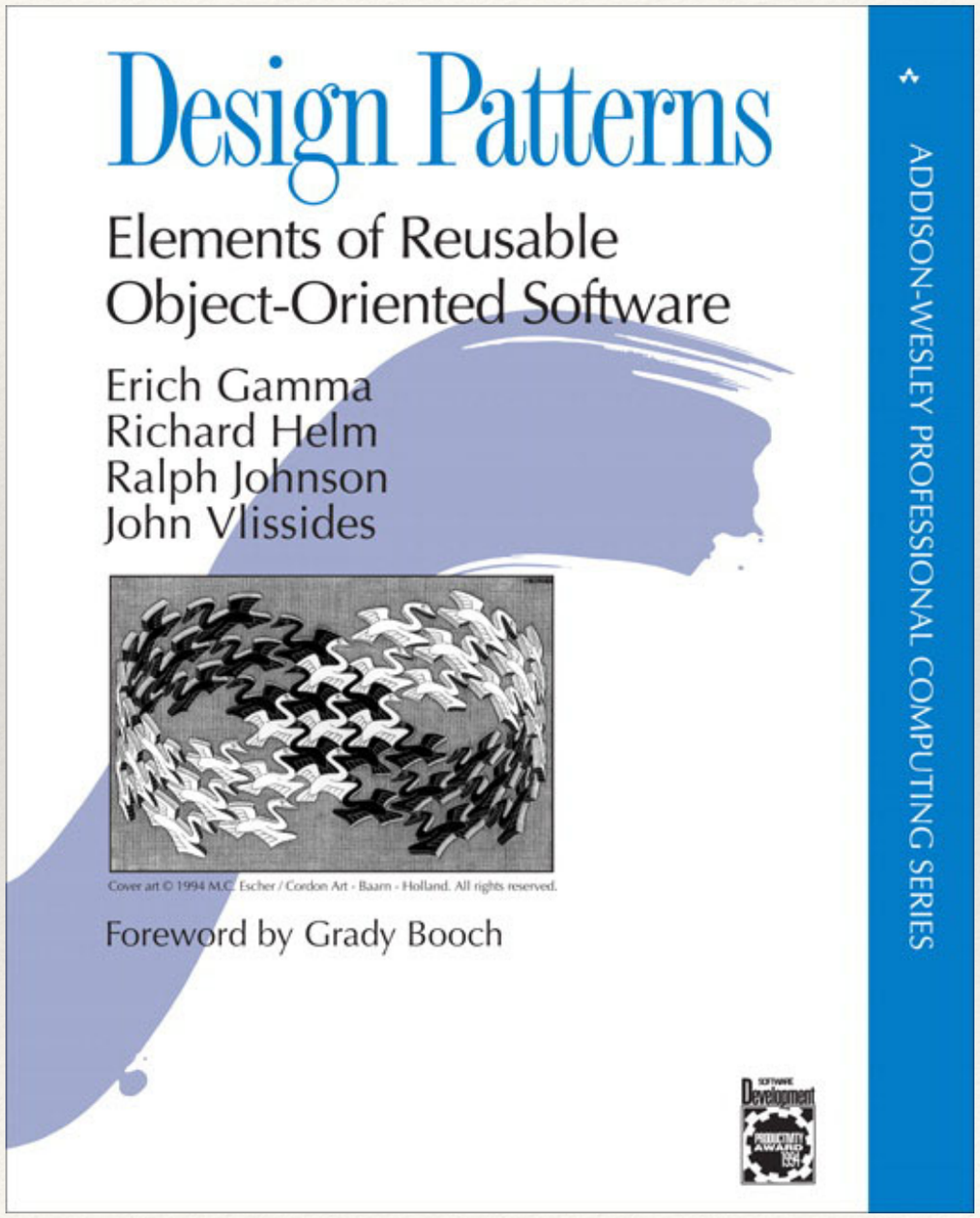
# Granularity of smells

| **Architectural** |
| --- |
| Cyclic dependencies between modules<br>Monolithic modules<br>Layering violations (back layer call, skip layer call, vertical layering, etc) |
| **Design** |
| God class<br>Refused bequest<br>Cyclic dependencies between classes |
| **Code (implementation)** |
| Internal duplication (clones within a class)<br>Large method<br>Temporary field |

# What are design patterns?

*recurrent solutions*
to common
*design problems*

Pattern Name
Problem
Solution
Consequences



# Why care about patterns?

- Patterns capture expert knowledge in the form of proven reusable solutions
  - Better to reuse proven solutions than to "re-invent" the wheel
- When used correctly, patterns positively influence software quality
  - Creates maintainable, extensible, and reusable code

# Design pattern catalog

| | | *Purpose* | | |
|---|---|---|---|---|
| | | **Creational** | **Structural** | **Behavioral** |
| ***Scope*** | **Class** | Factory Method | Adapter (class) | Interpreter<br>Template Method |
| | **Object** | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Flyweight<br>Facade<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

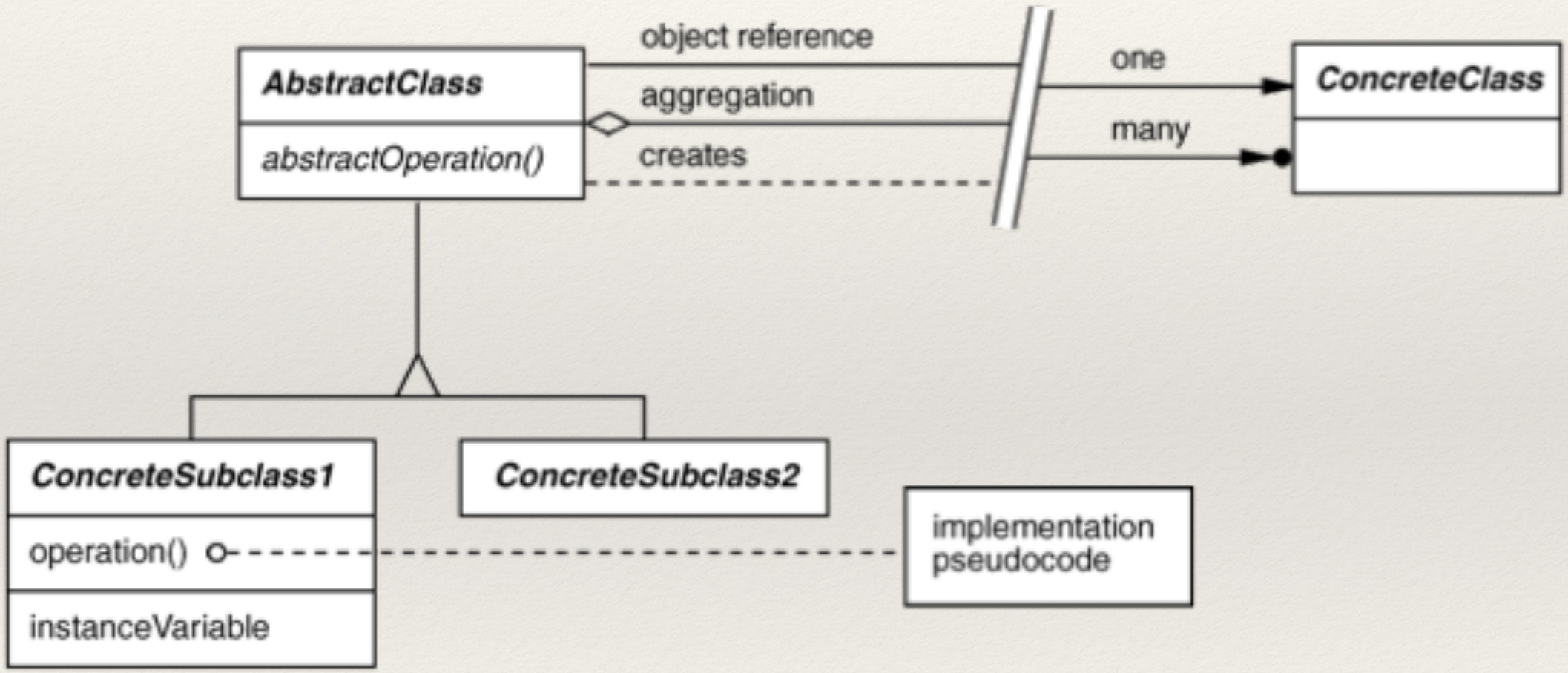
Source: "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

# Design pattern catalog

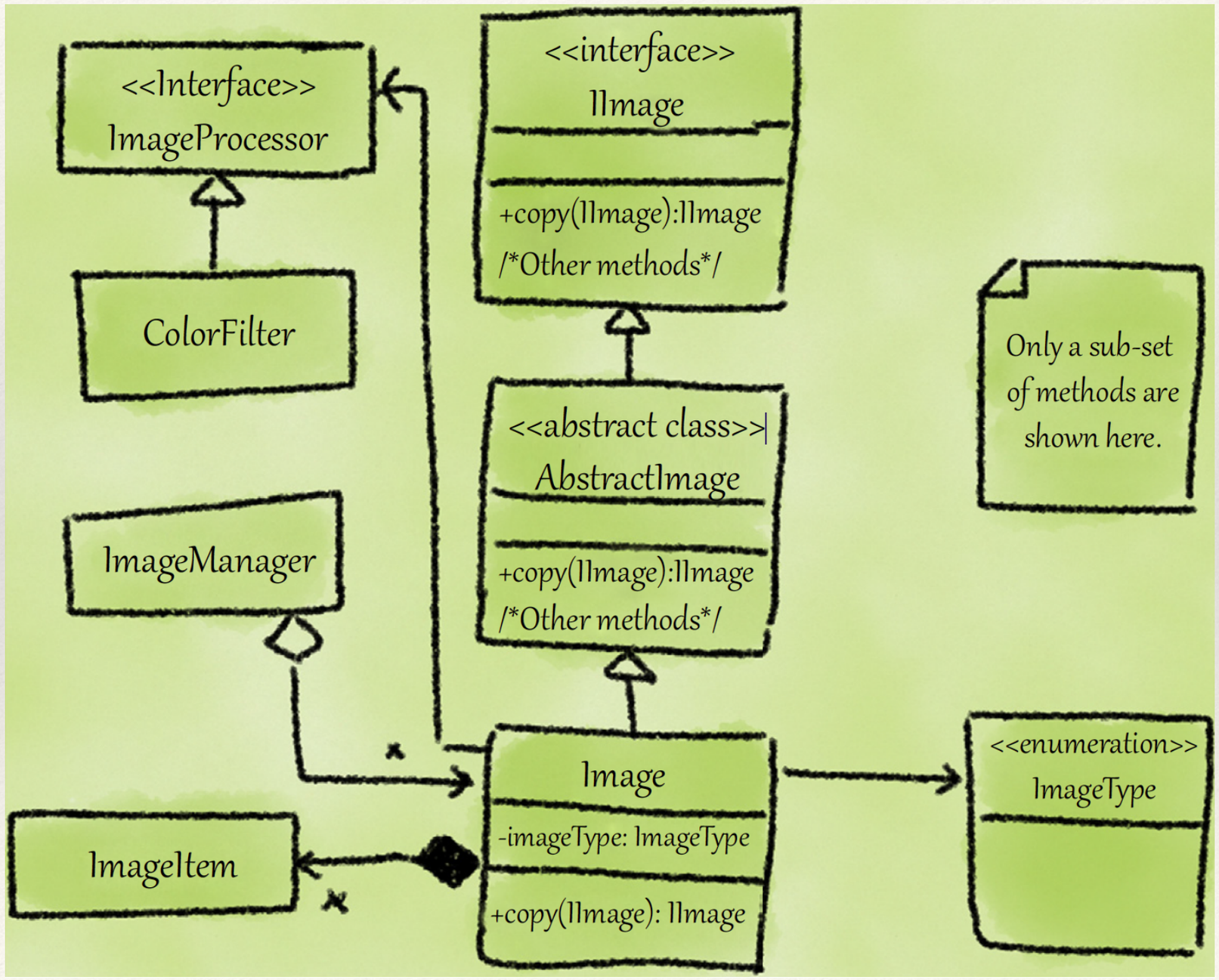
| | | |
|---|---|---|
| Creational | Deals with controlled object creation | *Factory method*, for example |
| Structural | Deals with composition of classes or objects | *Composite*, for example |
| Behavioral | Deals with interaction between objects/classes and distribution of responsibility | *Strategy*, for example |

Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

# 5 minutes intro to notation



# An example




Source: "Refactoring for Software Design Smells: Managing Technical Debt", Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Morgan Kaufmann/Elsevier, 2014

# What's that smell?

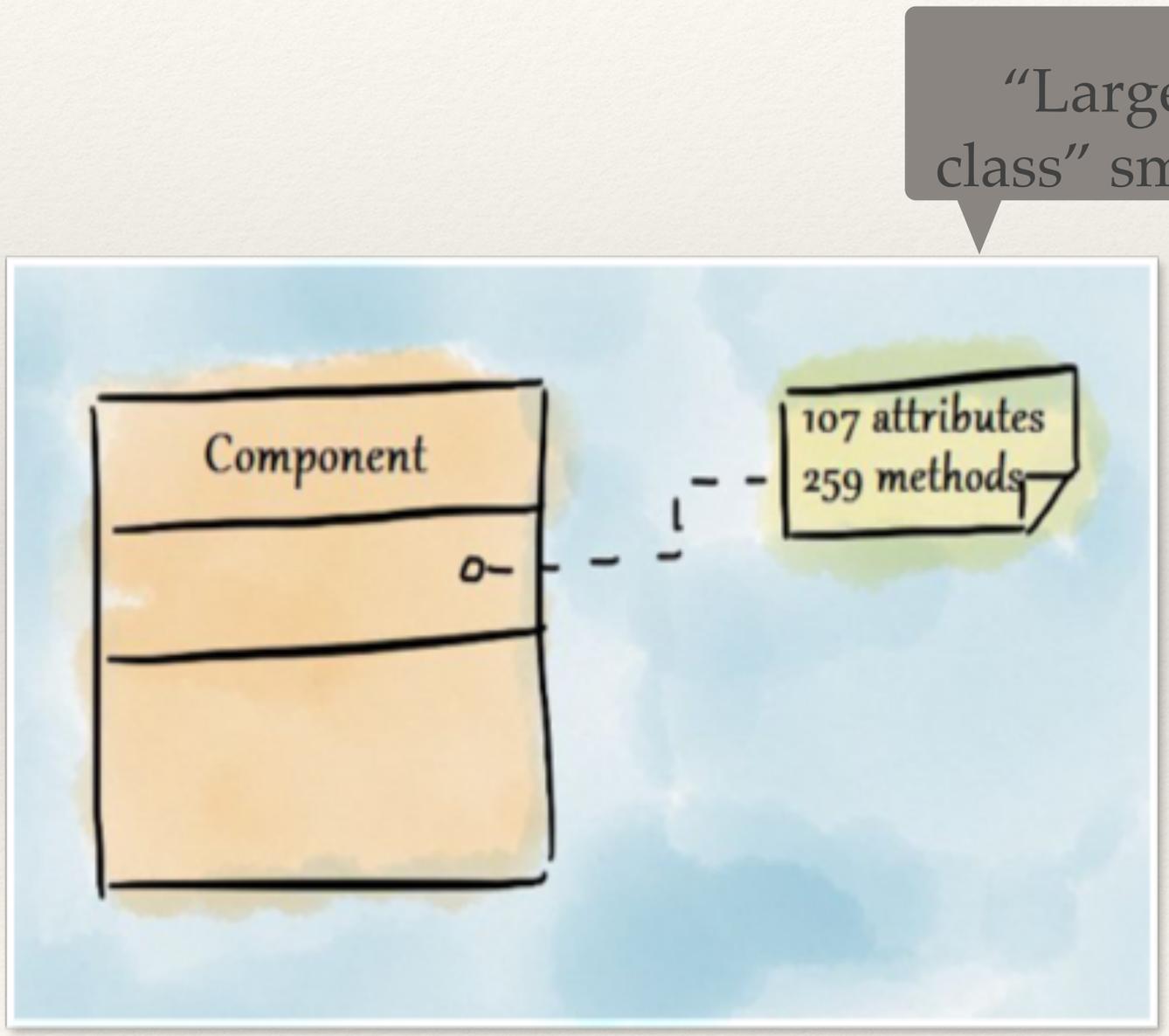
```
 7  namespace System.Web.Security {
 8      using  System.Web;
 9      using  System.Web.Configuration;
10      using  System.Security.Principal;
11      using  System.Security.Permissions;
12      using  System.Globalization;
13      using  System.Runtime.Serialization;
14      using  System.Collections;
15      using  System.Security.Cryptography;
16      using  System.Configuration.Provider;
17      using  System.Text;
18      using  System.Configuration;
19      using  System.Web.Management;
20      using  System.Web.Hosting;
21      using  System.Threading;
22      using  System.Web.Util;
23      using  System.Collections.Specialized;
24      using System.Web.Compilation;
25
26      public static class Membership...
594  }
```

# Hands-on exercise

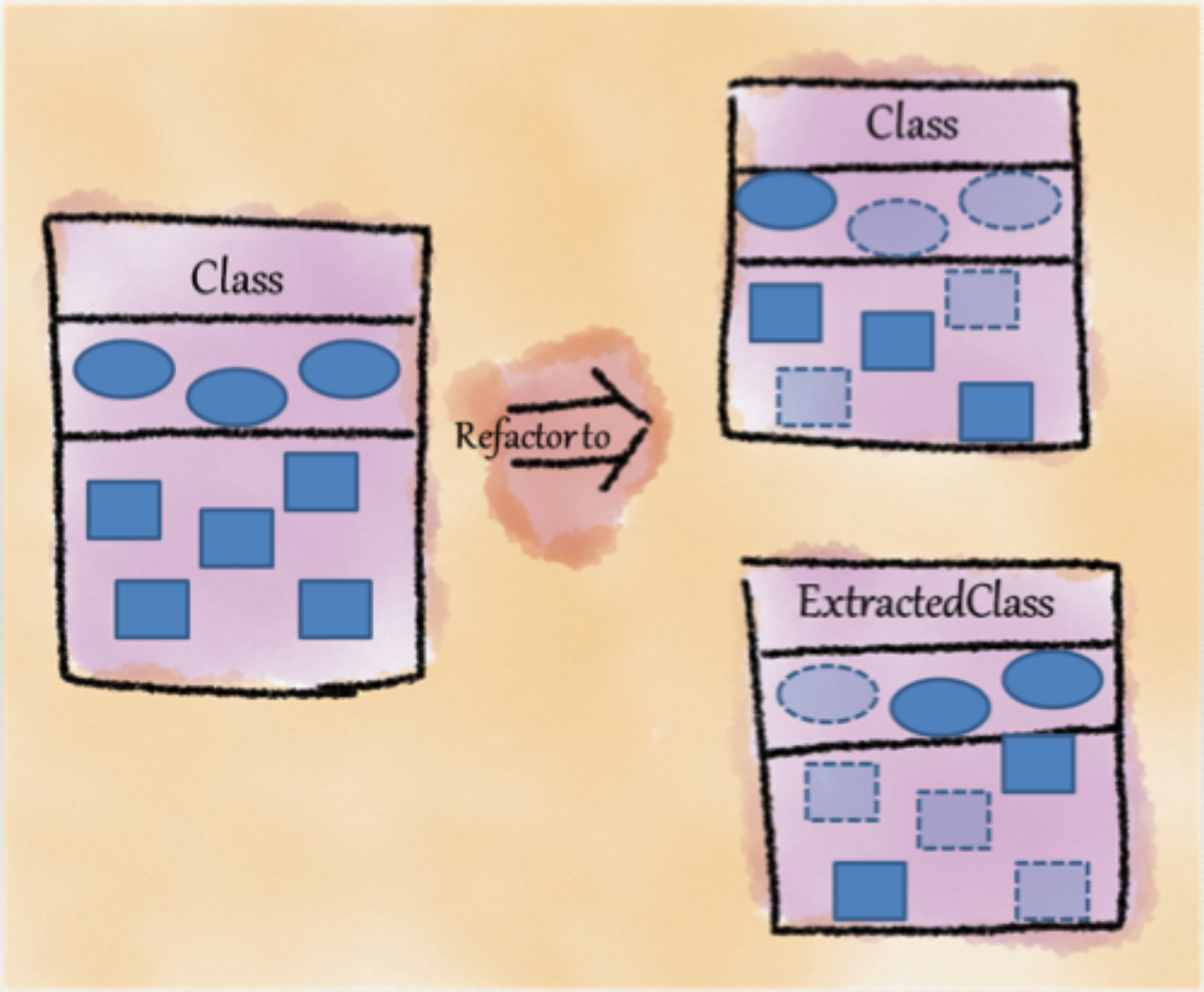
```
1 reference
public class FormatFlags
{
    0 references
    private FormatFlags()
    {
    }

    public static int LEFT_JUSTIFY = 1 << 0;
    public static int UPPERCASE = 1 << 1;
    public static int ALTERNATE = 1 << 2;
}
```

Solution Location: Solid-Master->Solid->FormatFlags.cs

# What's that smell?



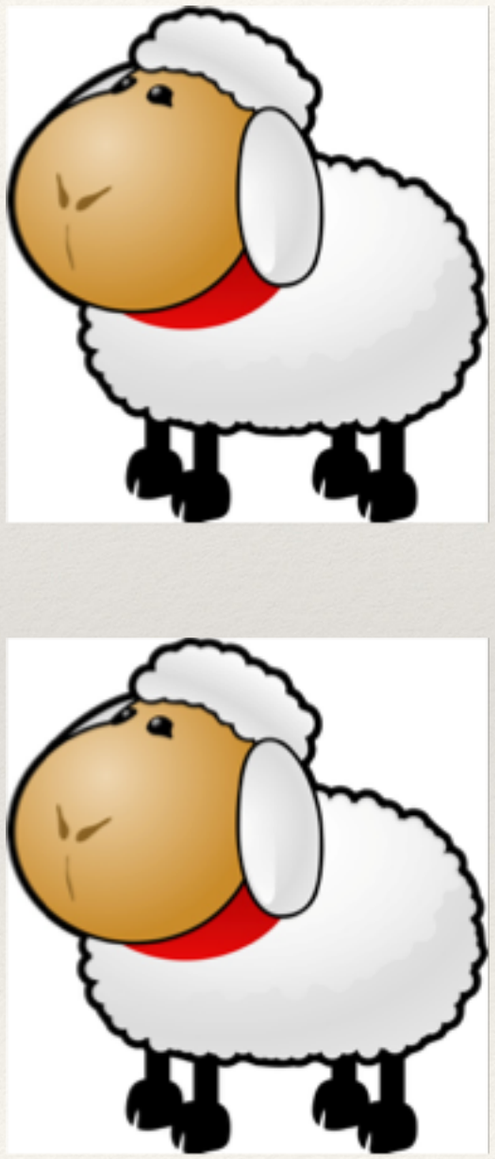
# Refactoring with “extract class”



# Single Responsibility Principle

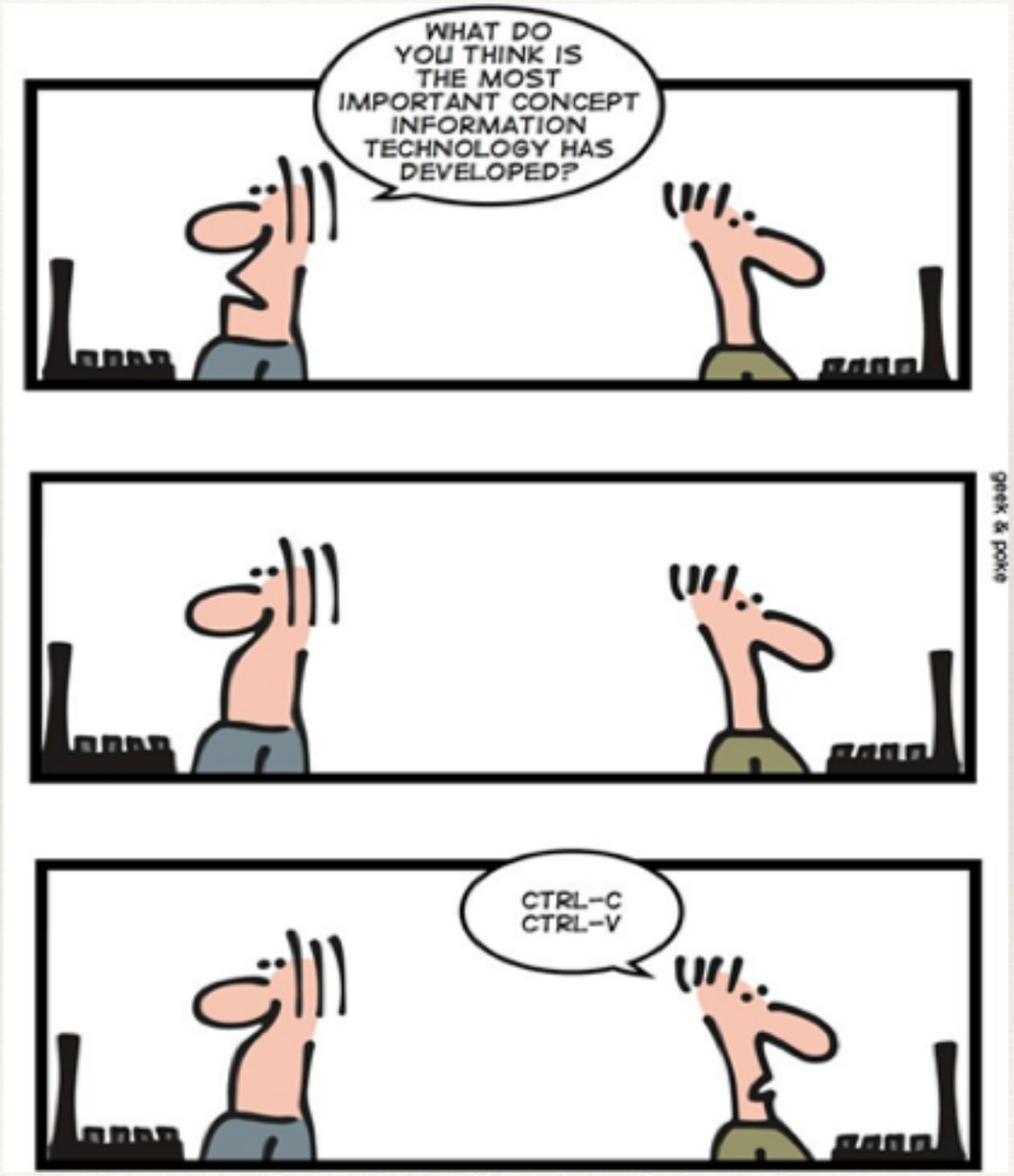
There should be only one reason for a class to change

# Duplicated Code





# CTRL-C and CTRL-V



# Types of Clones

## Type 1

- **exactly identical** except for variations in whitespace, layout, and comments

## Type 2

- **syntactically identical** except for variation in symbol names, whitespace, layout, and comments

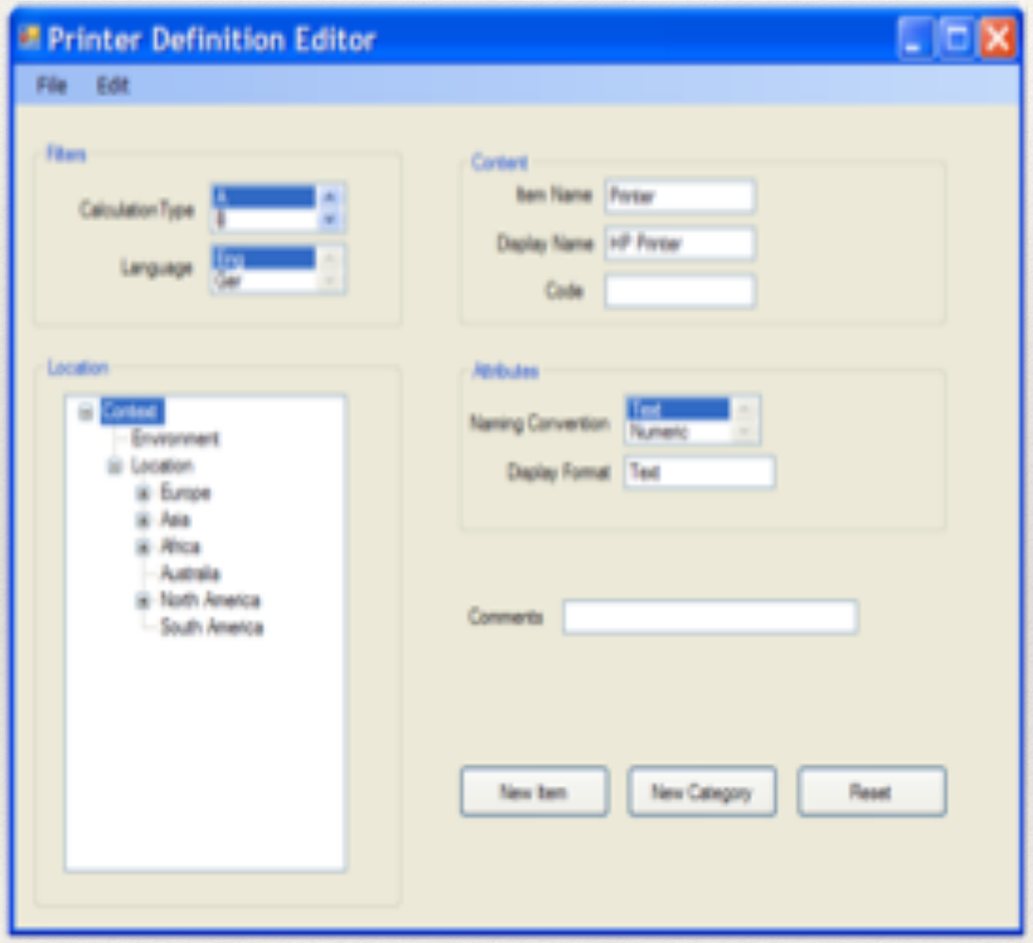
## Type 3

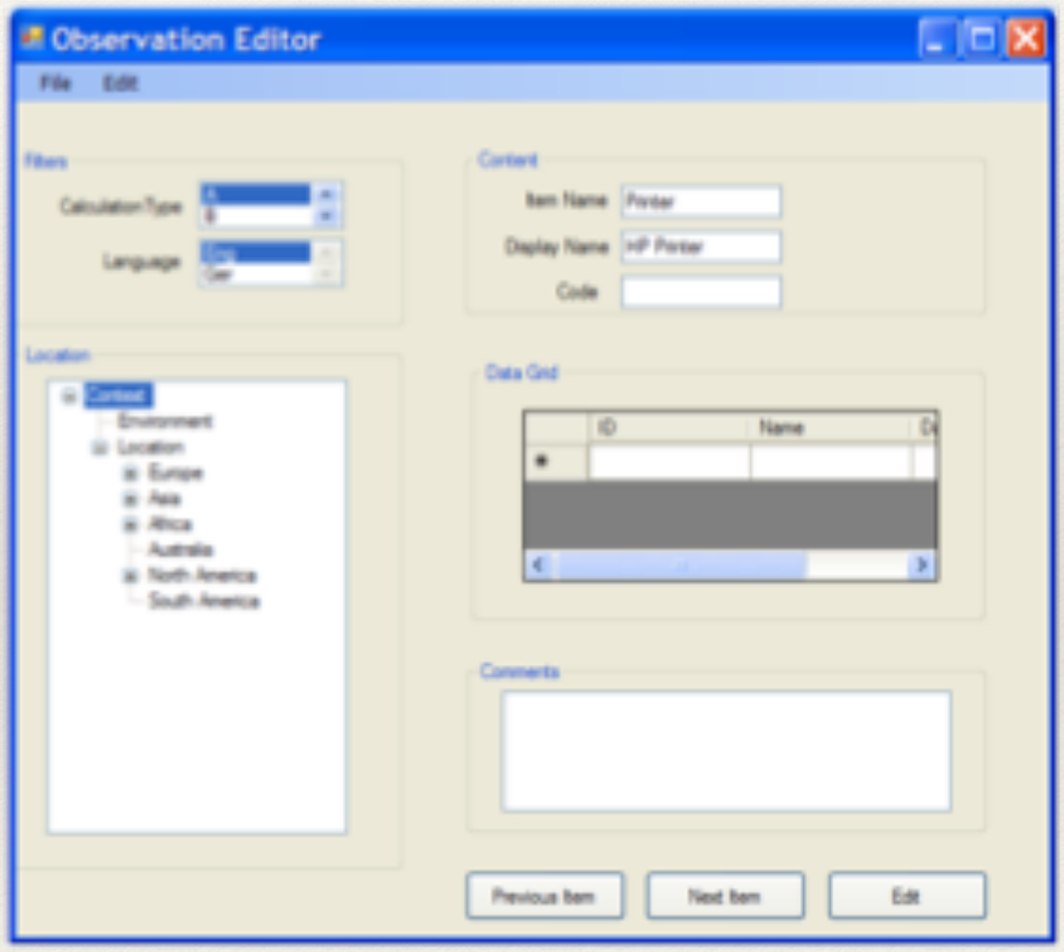
- identical except some **statements changed, added, or removed**

## Type 4

- when the fragments are **semantically identical** but implemented by syntactic variants

# How to deal with duplication?



# Refactoring "incomplete library classes" smell

| min/max | open/close | create/destroy | get/set |
|---|---|---|---|
| read/write | print/scan | first/last | begin/end |
| start/stop | lock/unlock | show/hide | up/down |
| source/target | insert/delete | first/last | push/pull |
| enable/disable | acquire/release | left/right | on/off |

# What's that smell?

```
2 references
private static string TranslateDevice(int type, int userLanguage)
{
    switch (type)
    {
        case MEDICAL:
            return userLanguage == EN ? "Medical" : "Medisch";
        case AGRICULTURAL:
            return userLanguage == EN ? "Agricultural" : "Agrarisch";
        case REFINARY:
            return userLanguage == EN ? "Refinary" : "Raffinaderij";
    }
    return string.Empty;
}
```

Solution Location: Solid-Assessment->Before->Device.cs

# Open Closed Principle (OCP)

Software entities should be open for extension, but closed for modification

Bertrand Meyer

# What's that smell?

"Refused bequest" smell from ReadOnlyDictionary.cs

```
        void IDictionary<TKey, TValue>.Add(TKey key, TValue value)
{
   throw new NotSupportedException(SR.NotSupported_ReadOnlyCollection);
}

bool IDictionary<TKey, TValue>.Remove(TKey key)
{
   throw new NotSupportedException(SR.NotSupported_ReadOnlyCollection);
}

TValue IDictionary<TKey, TValue>.this[TKey key]
{
   get
   {
      return _dictionary[key];
   }
   set
   {
      throw new NotSupportedException(SR.NotSupported_ReadOnlyCollection);
   }
}
```

# Liskov's Substitution Principle (LSP)

It should be possible to replace objects of supertype with objects of subtypes without altering the desired behavior of the program

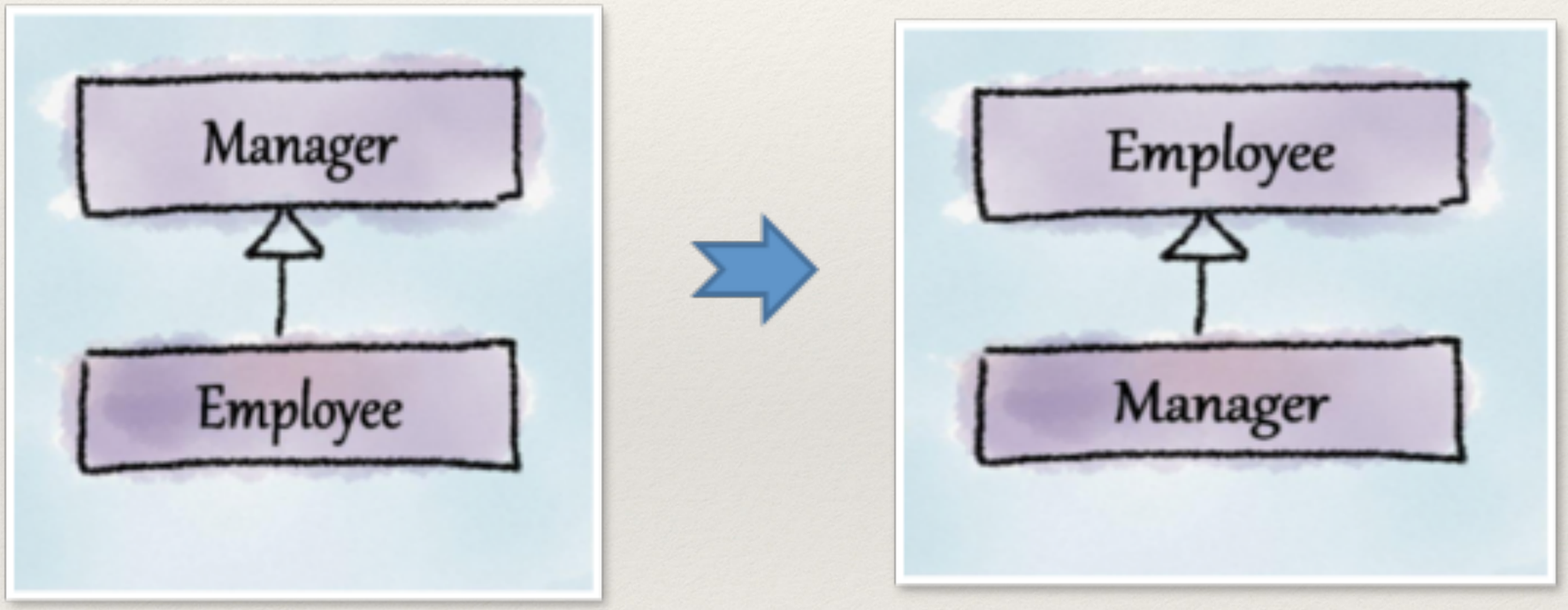
Barbara Liskov

# Refused bequest smell

A class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class

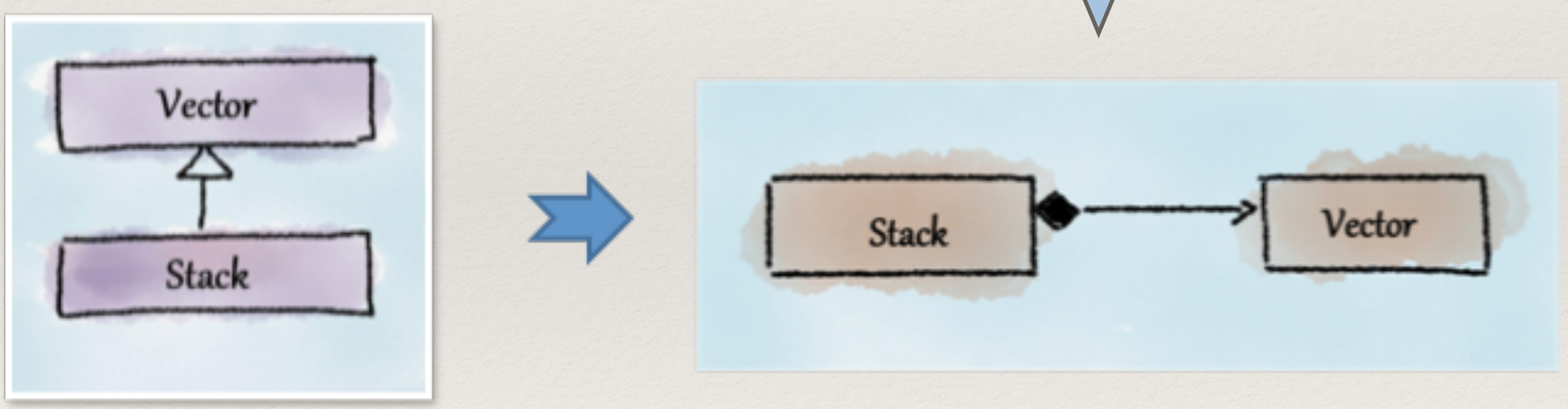
# What's that smell?

# Refactoring



# Refactoring

Replace inheritance with delegation



# Hands-on exercise

```
[TestMethod]
      public void TwentyFourfor4x6RectanglefromSquare()
      {
          Rectangle newRectangle = new Square();
          newRectangle.Height = 4;
          newRectangle.Width = 6;
          var result = AreaCalculator.CalculateArea(newRectangle);
          Assert.AreEqual(24, result);
      }
```

Is this an LSP violation?

# Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces they do not use

# Example from .NET Fx

**Our favourite Membership class in ASP.NET! Study the .NET framework class to see the violations.**

# Dependency Inversion Principle (DIP)

A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

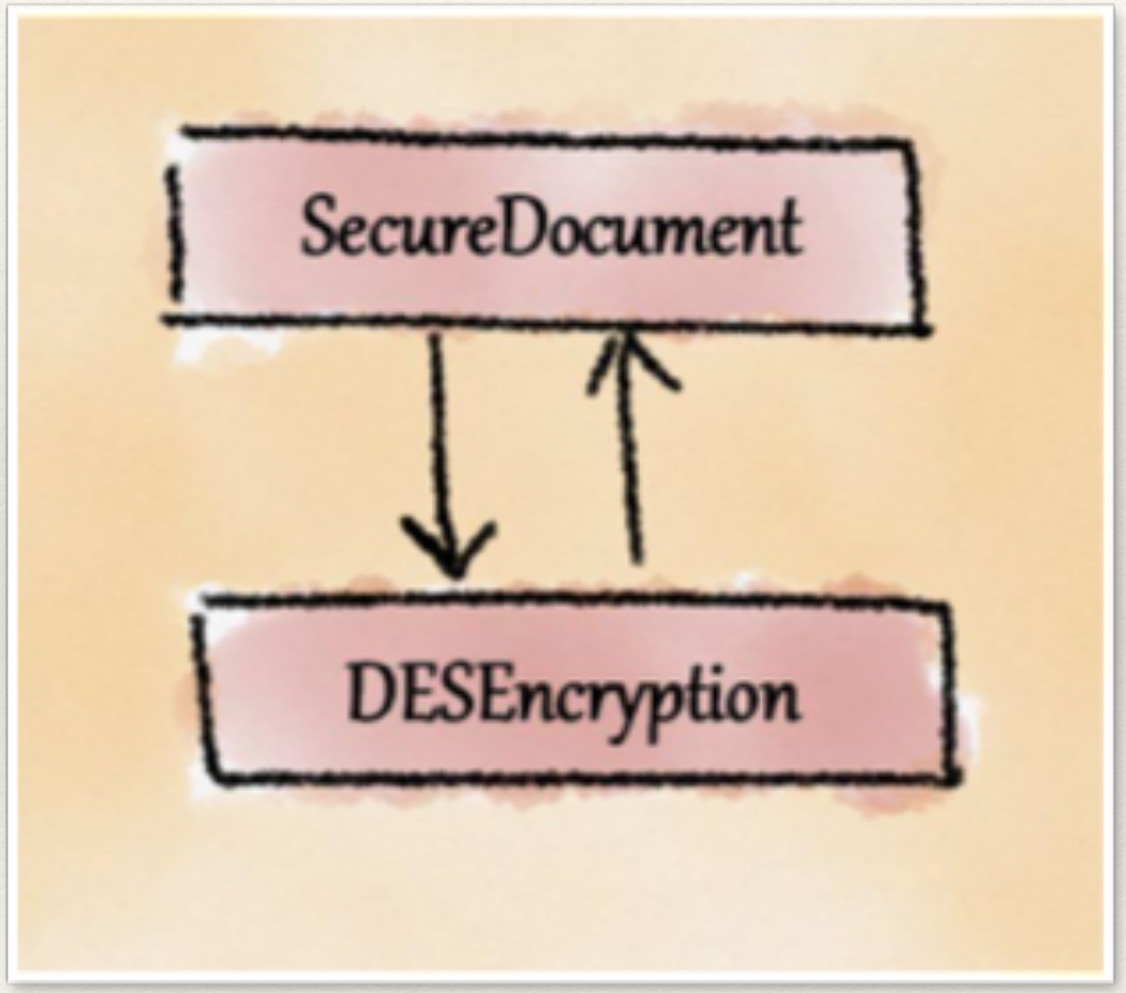
B. Abstractions should not depend upon details. Details should depend upon abstractions.

# Hands-on exercise

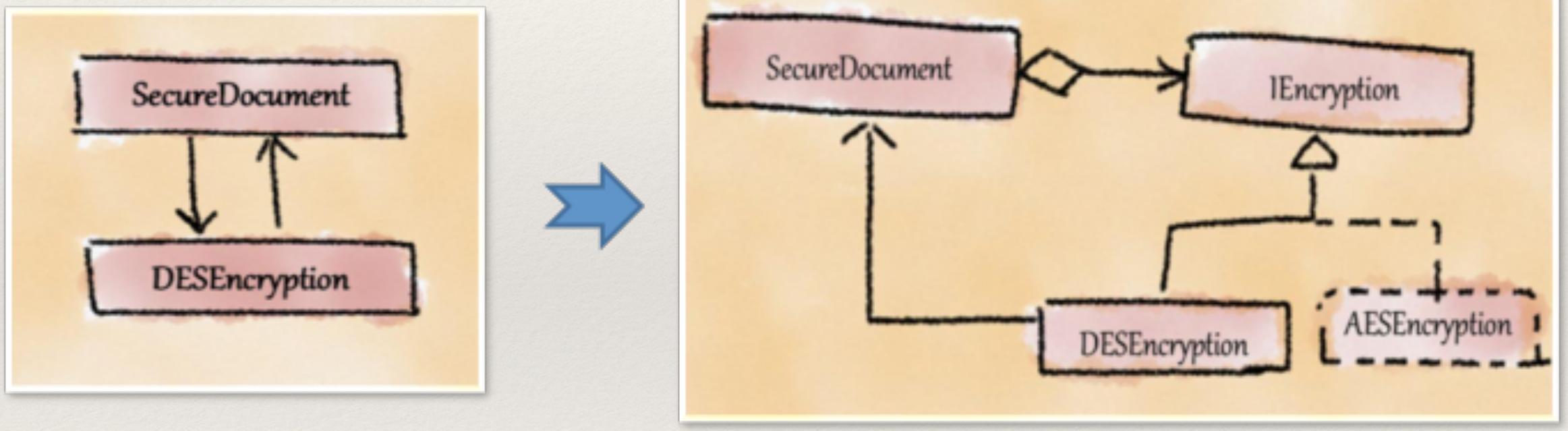
```
2 references
class EventLogWriter
{
    1 reference
    public void Write(string message)
    {
        //Write to event log here
    }
}

0 references
class ServiceWatcher
{
    // Handle to EventLog writer to write to the logs
    EventLogWriter writer = null;

    // This function will be called when the app pool has problem
    0 references
    public void Notify(string message)
    {
        if (writer == null)
        {
            writer = new EventLogWriter();
        }
        writer.Write(message);
    }
}
```

1) What is the violation here? What is the remedy?
2) Layered Architecture. What is the remedy?

WHAT'S THAT SMELL?



# Suggested refactoring for this smell



# Applying DIP in OO Design

Use references to interfaces/abstract classes as fields members, as argument types and return types

Do not derive from concrete classes

Use creational patterns such as factory method and abstract factory for instantiation

Do not have any references from base classes to its derived classes

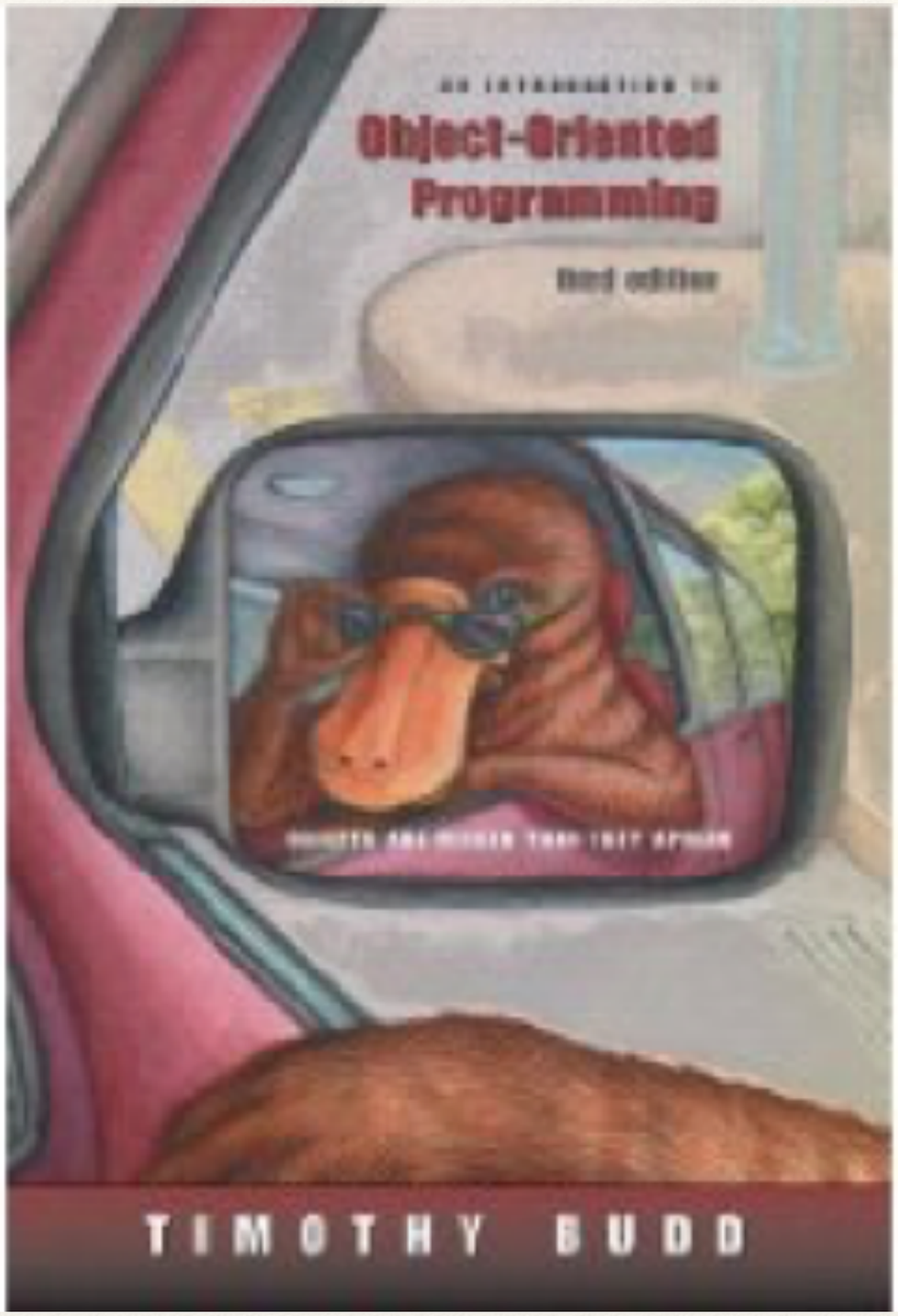
# 3 principles behind patterns

Program to an interface, not to an implementation

Favor object composition over inheritance

Encapsulate what varies

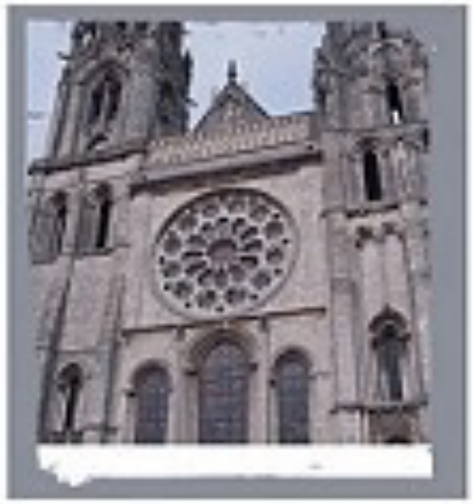
# Books to read


AN INTRODUCTION TO
Object-Oriented
Programming
third edition
TIMOTHY BUDD

WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

# PATTERN-ORIENTED SOFTWARE ARCHITECTURE

## A System of Patterns



Volume 1

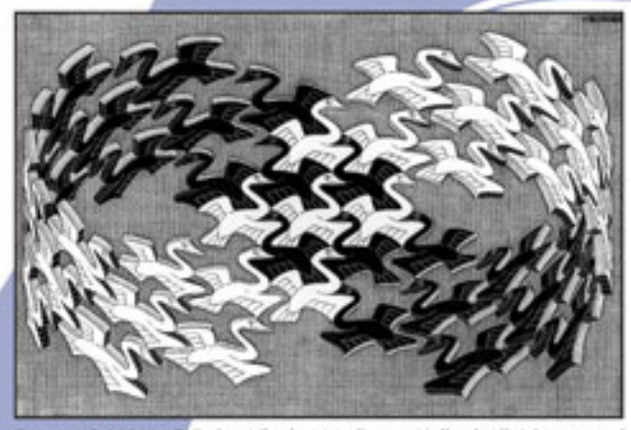
Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

WILEY

# Design Patterns

## Elements of Reusable Object-Oriented Software

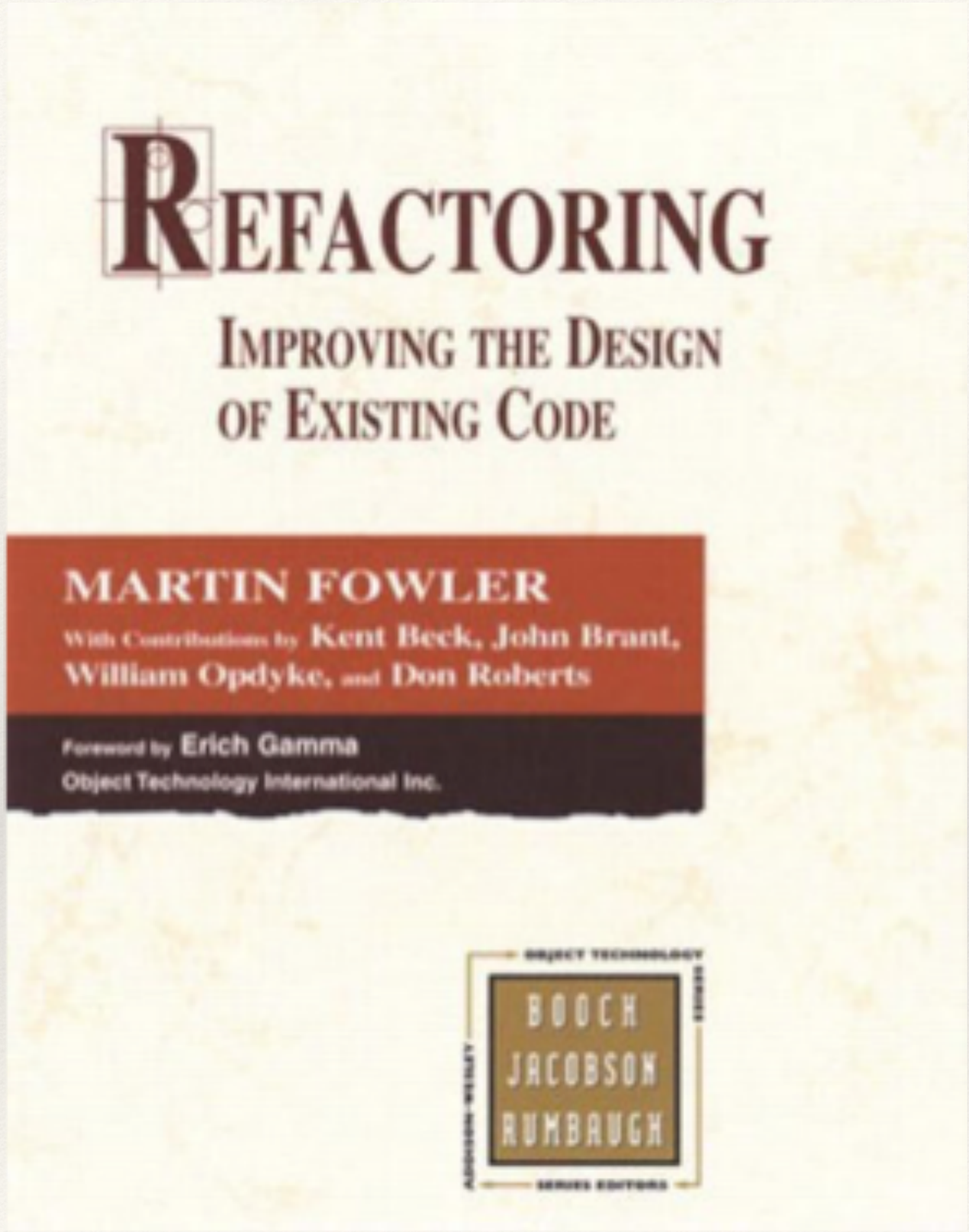
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides





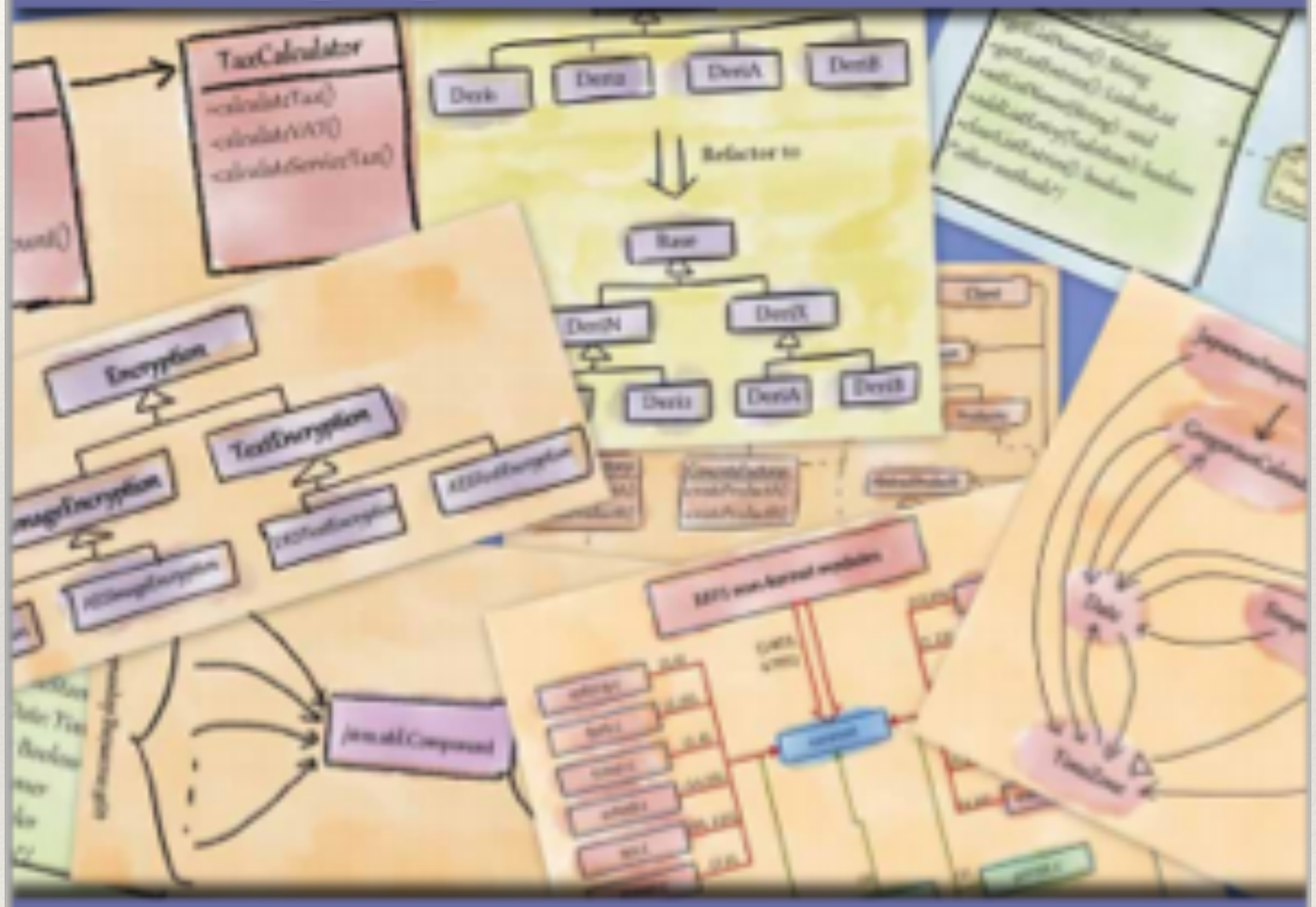
Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


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
Object Technology International Inc.
OBJECT TECHNOLOGY
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

# Refactoring for Software Design Smells

## Managing Technical Debt




Girish Suryanarayana,
Ganesh Samarthyam, Tushar Sharma

"Applying design principles is the key to creating high-quality software!"

Architectural principles:
Axis, symmetry, rhythm, datum, hierarchy, transformation