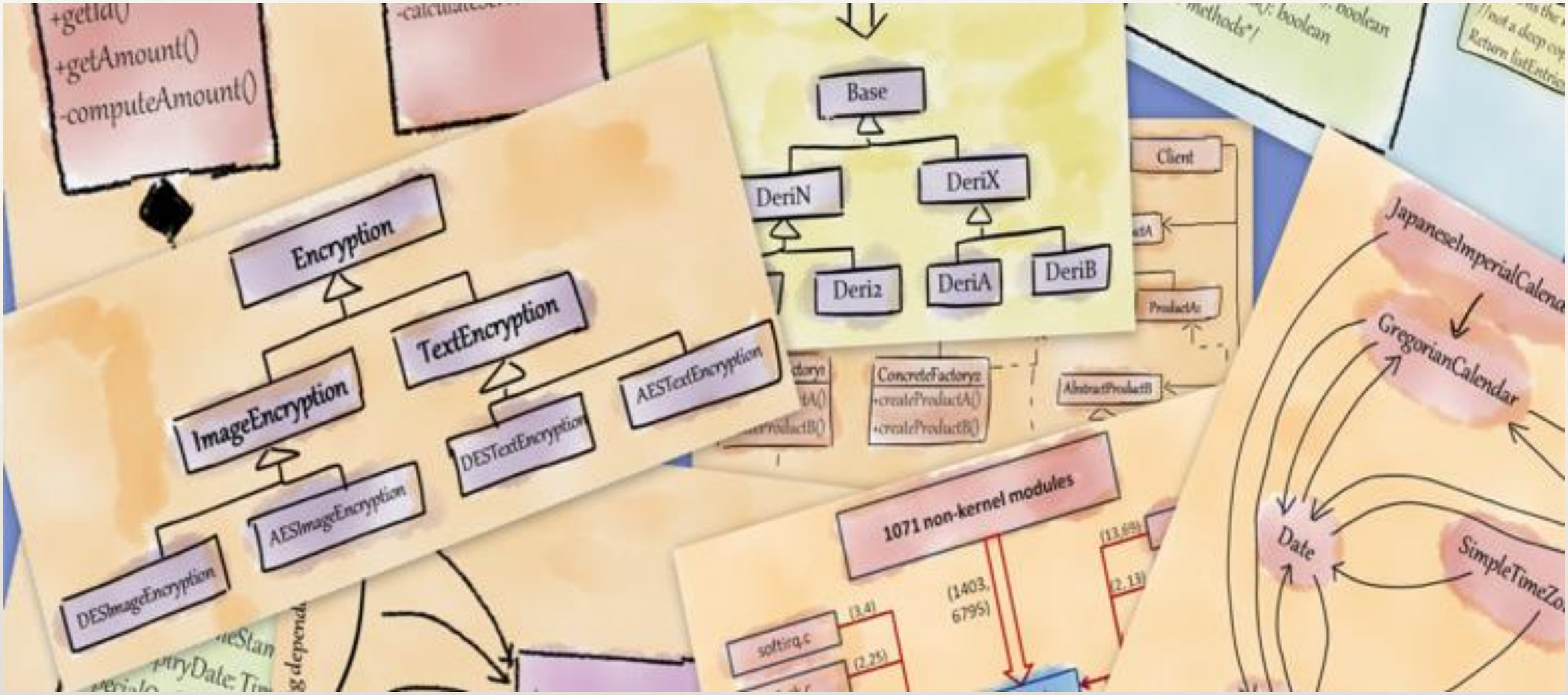


SOLID Principles & Design Patterns

Ganesh Samarthyam
Manoj Ganapathi

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

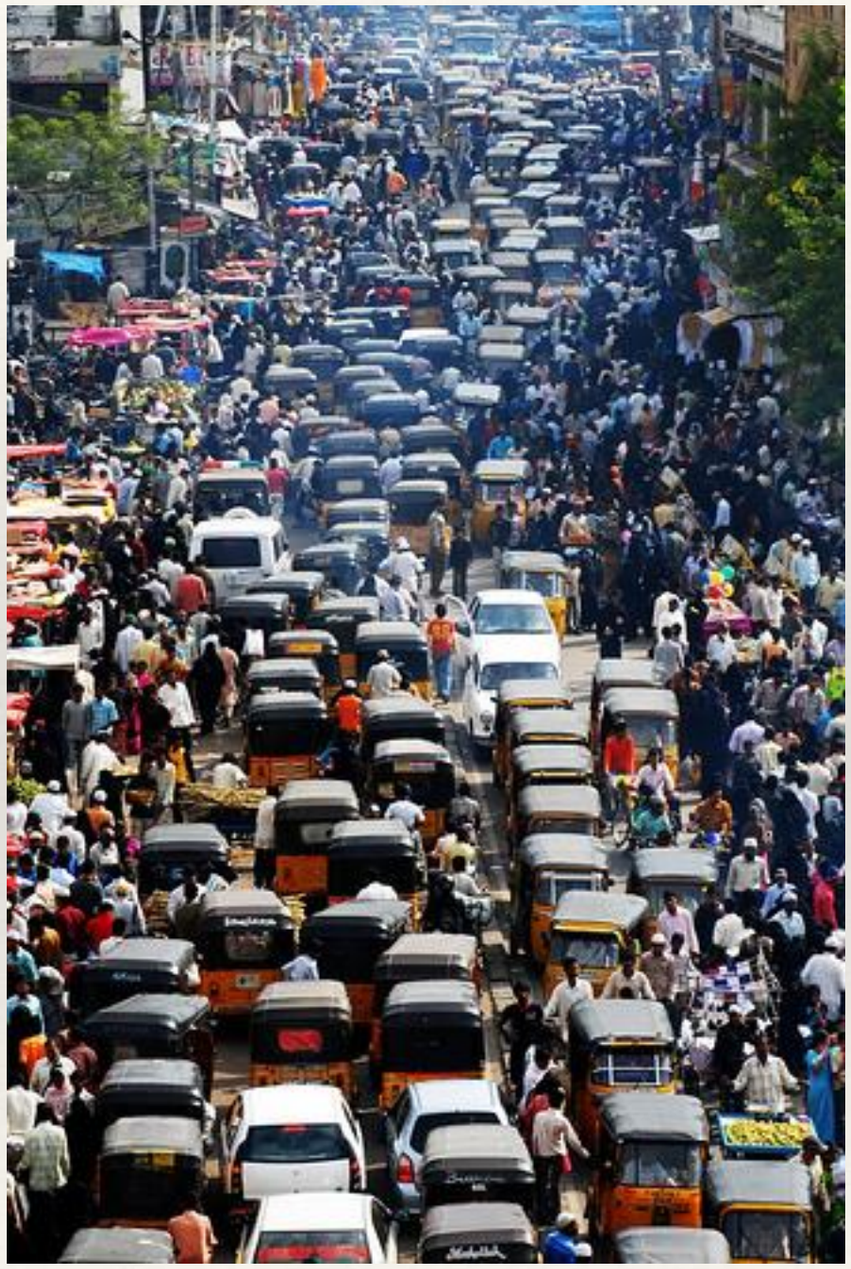
Architectural principles:
Axis, symmetry, rhythm, datum, hierarchy, transformation

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

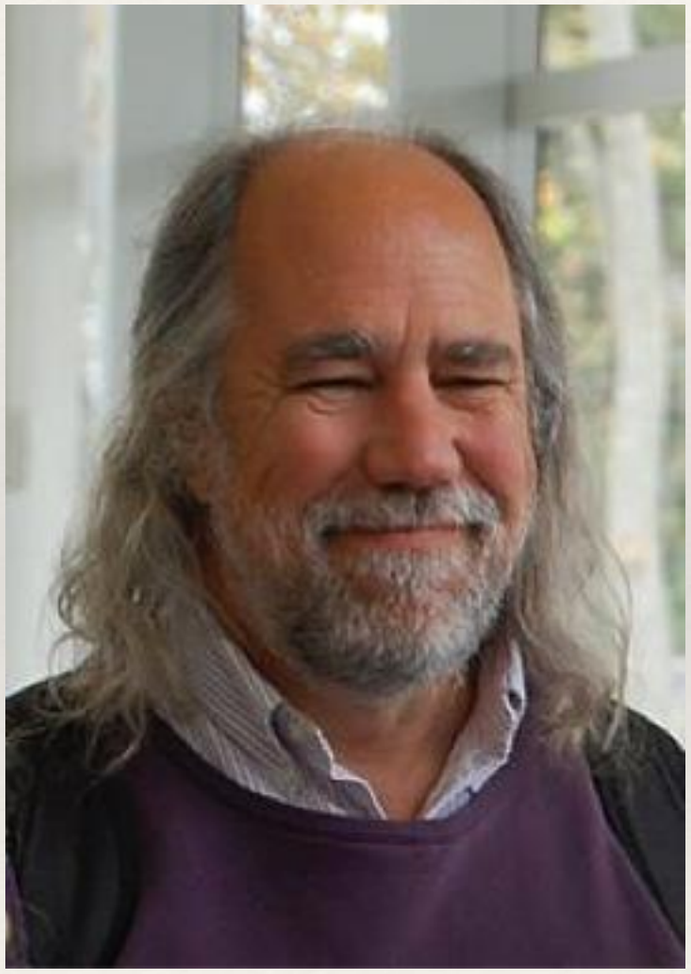
- Capers Jones

The city metaphor



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

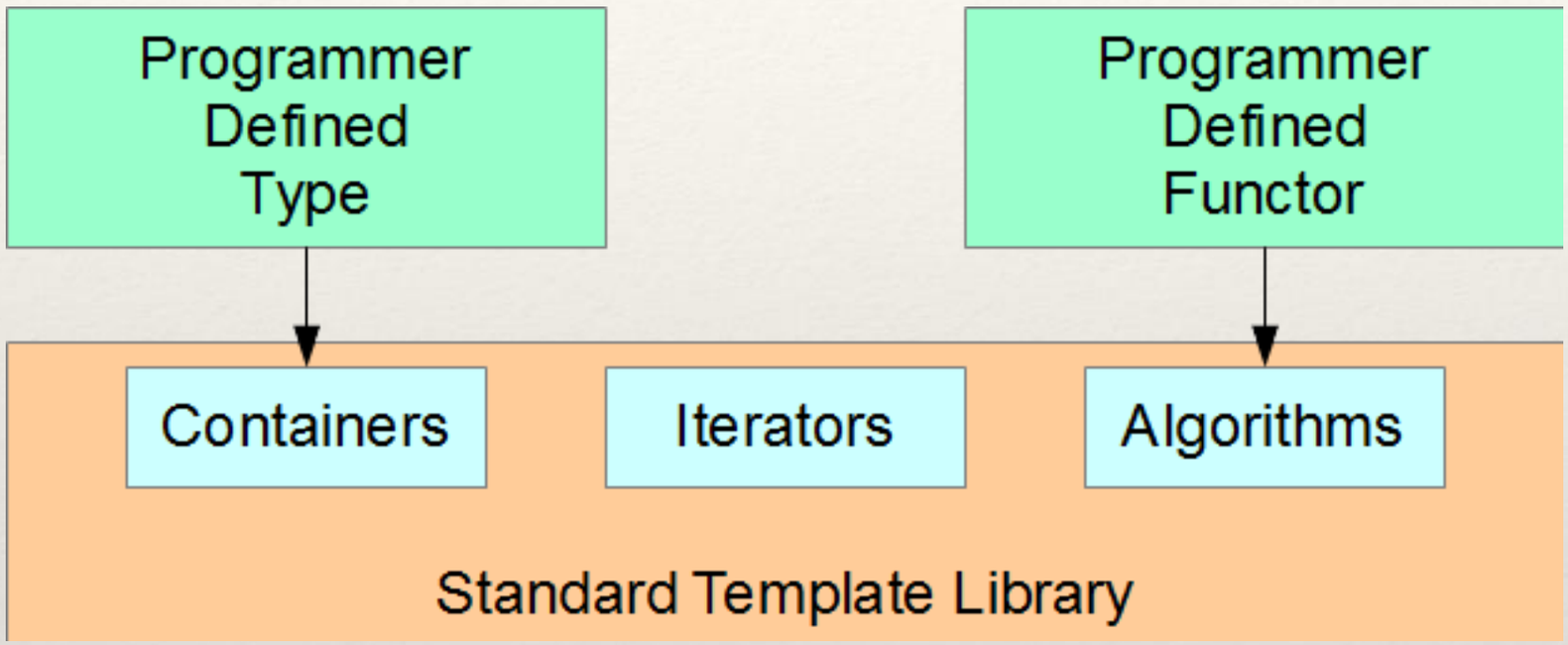
The city metaphor



"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”?

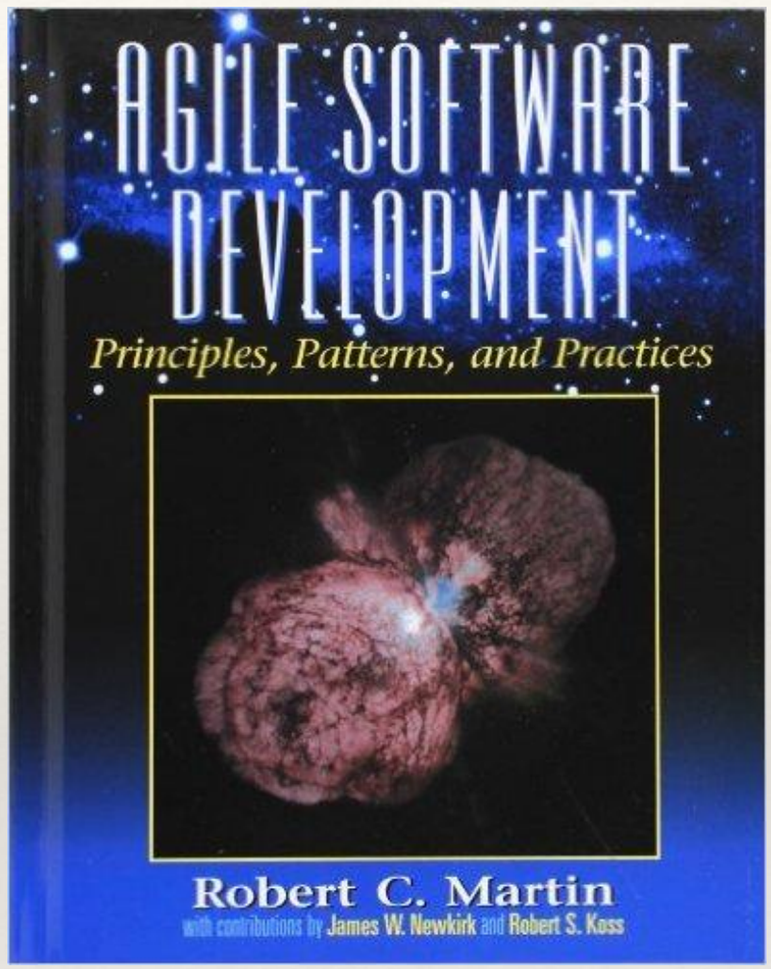
“Design principles are key notions considered fundamental to many different software design approaches and concepts.”
- SWEBOK v3 (2014)

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

Fundamental Design Principles

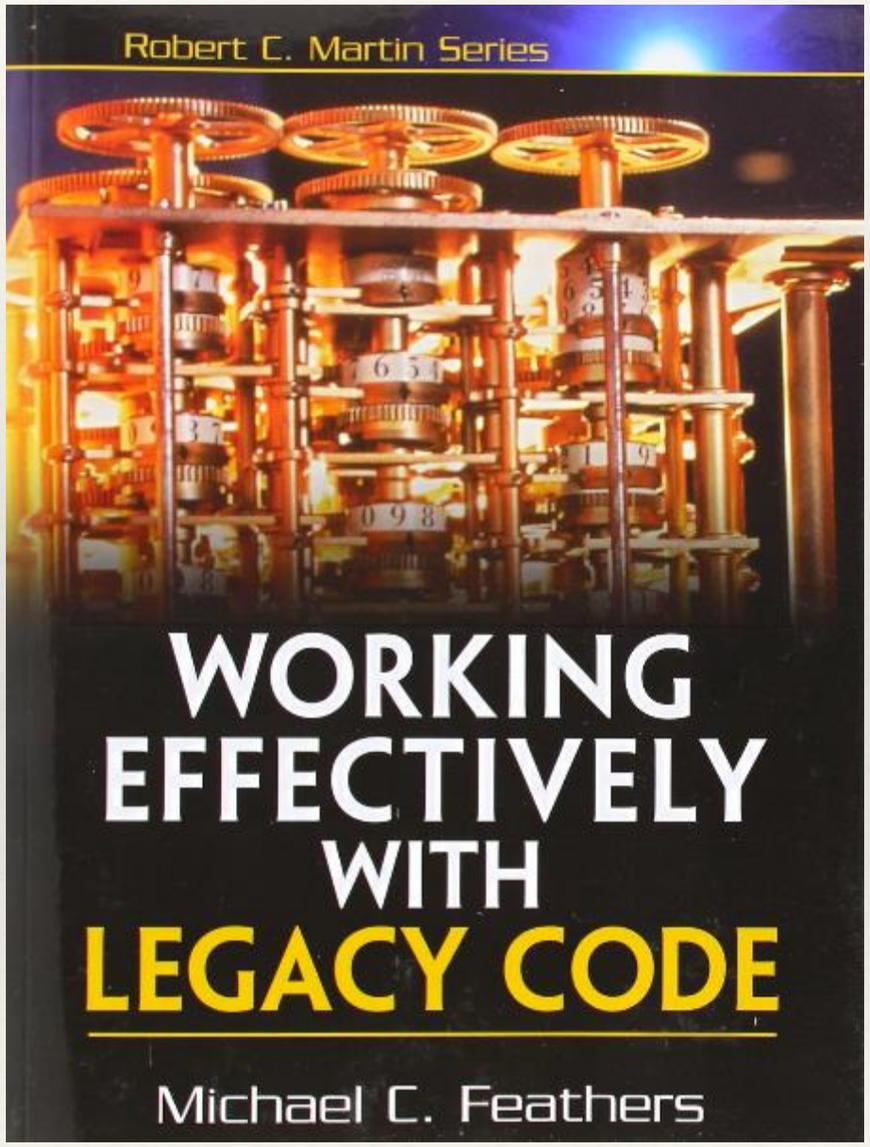
Robert C. Martin

Formulated many principles and described many other important principles



Michael Feathers

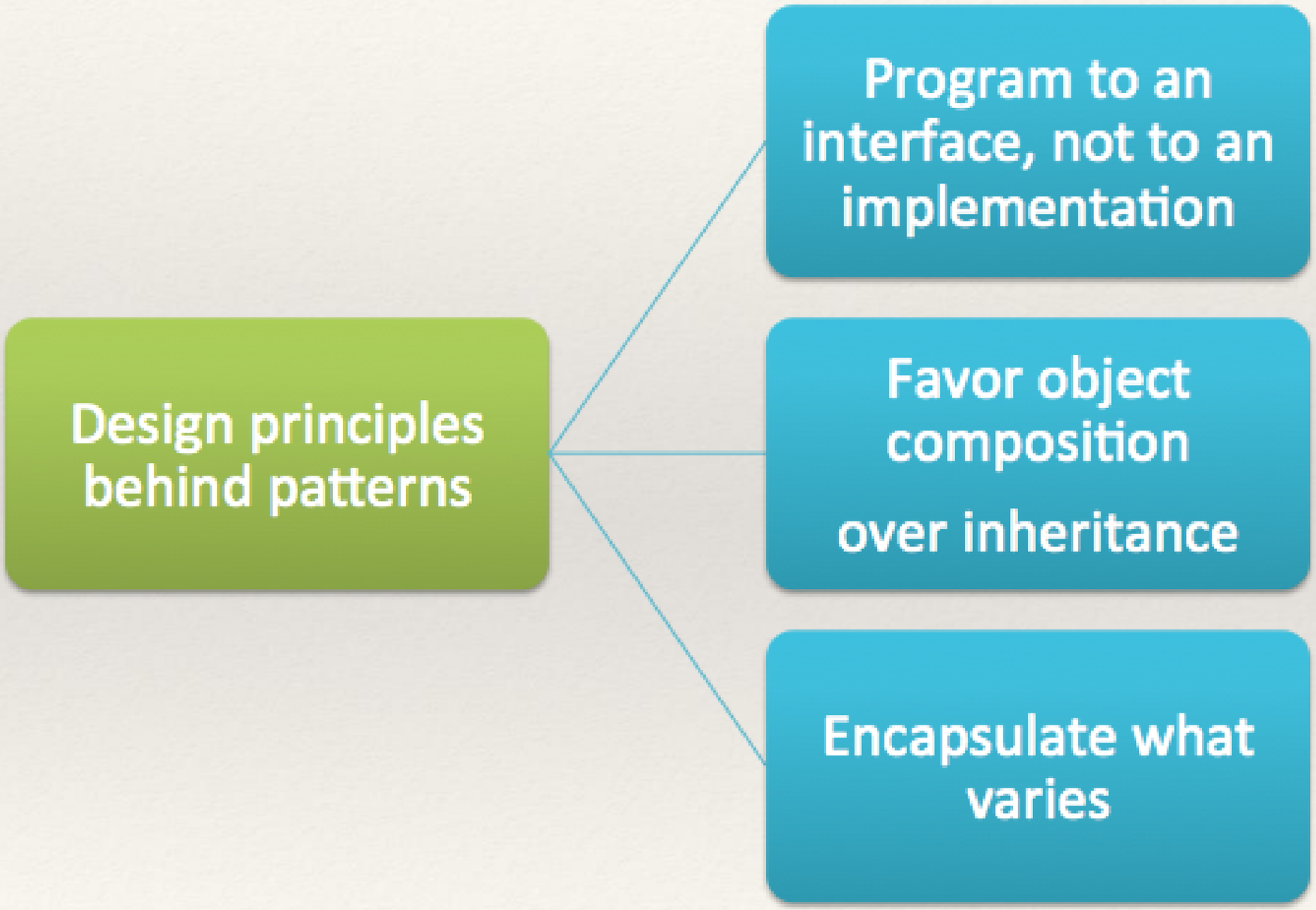
Michael Feathers coined the acronym SOLID in 2000s to remember first five of the numerous principles by Robert C. Martin



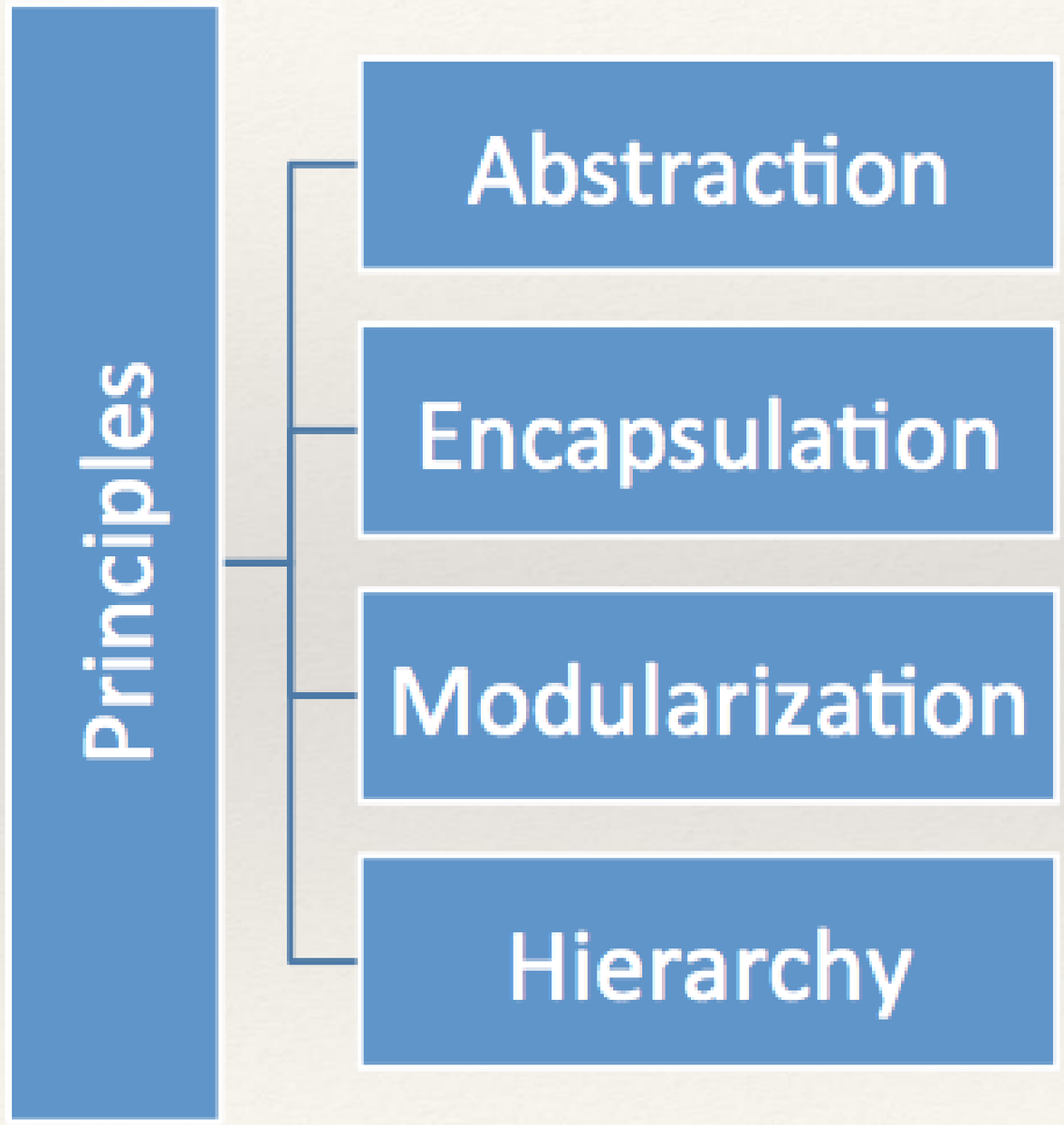
SOLID principles

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

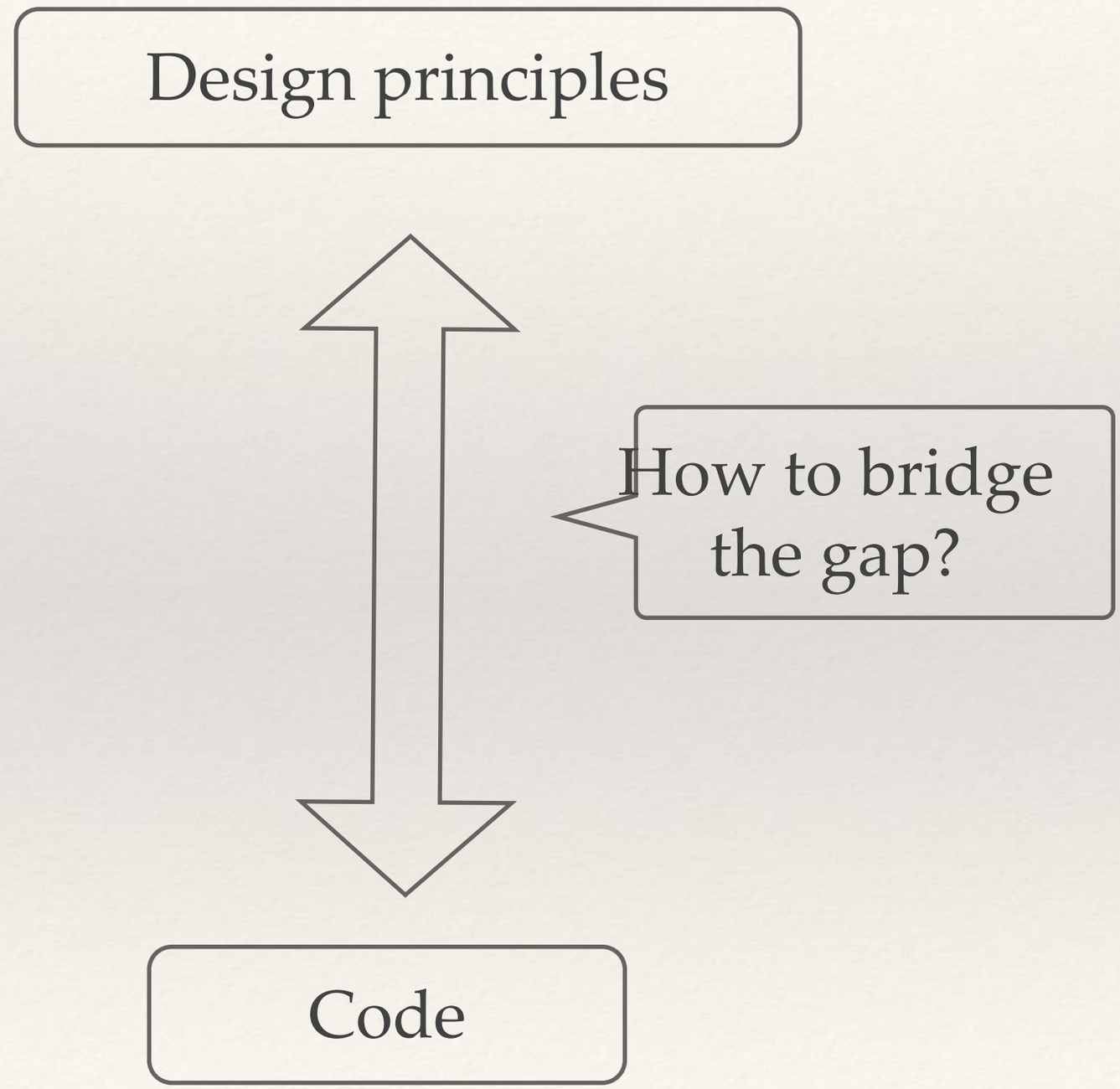
3 principles behind patterns



Booch's fundamental principles



How to apply principles in practice?



Why care about refactoring?

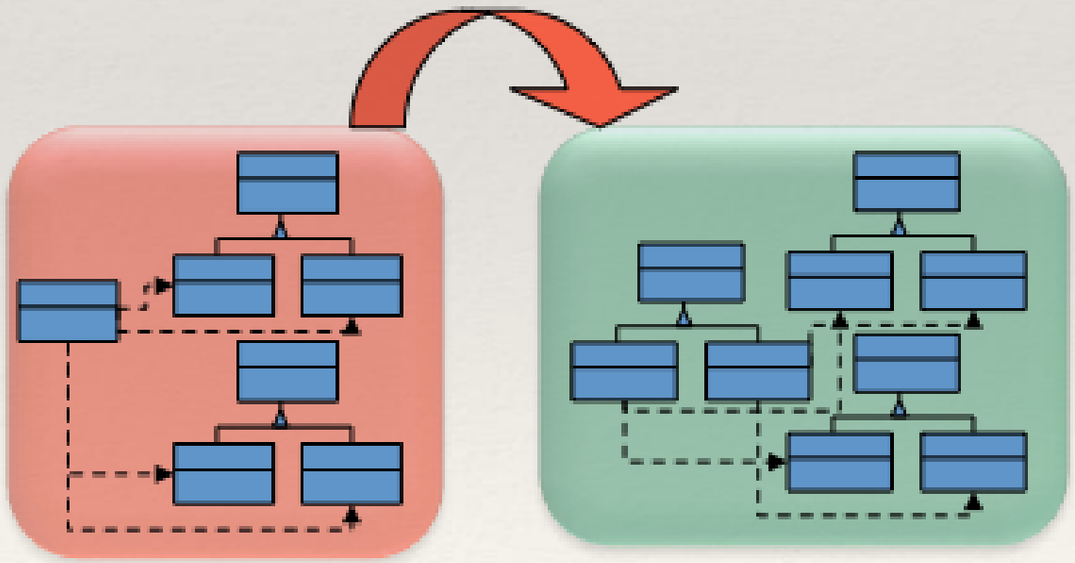
> As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it

- 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."

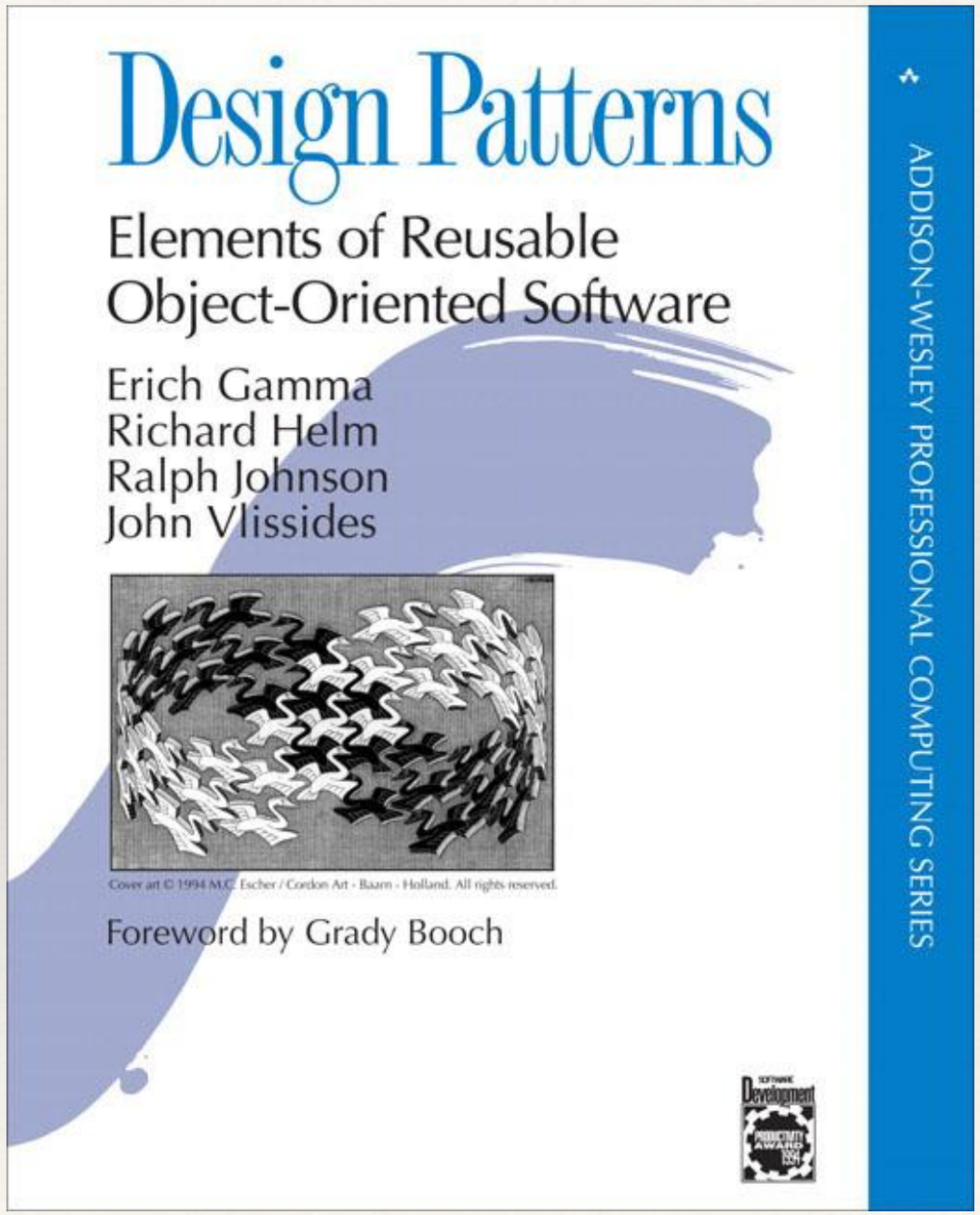
Granularity of smells

Architectural
Cyclic dependencies between modules Monolithic modules Layering violations (back layer call, skip layer call, vertical layering, etc)
Design
God class Refused bequest Cyclic dependencies between classes
Code (implementation)
Internal duplication (clones within a class) Large method 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 Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

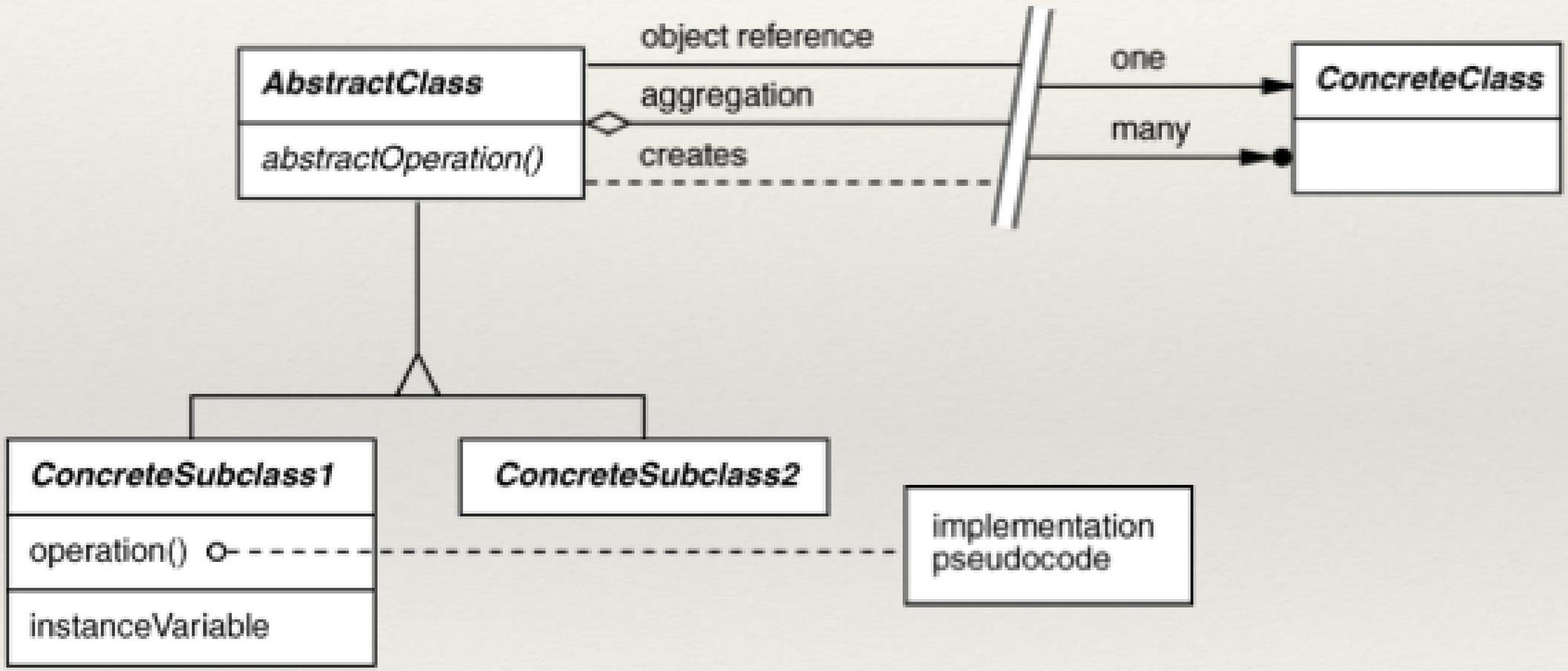
Source: "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Design pattern catalog

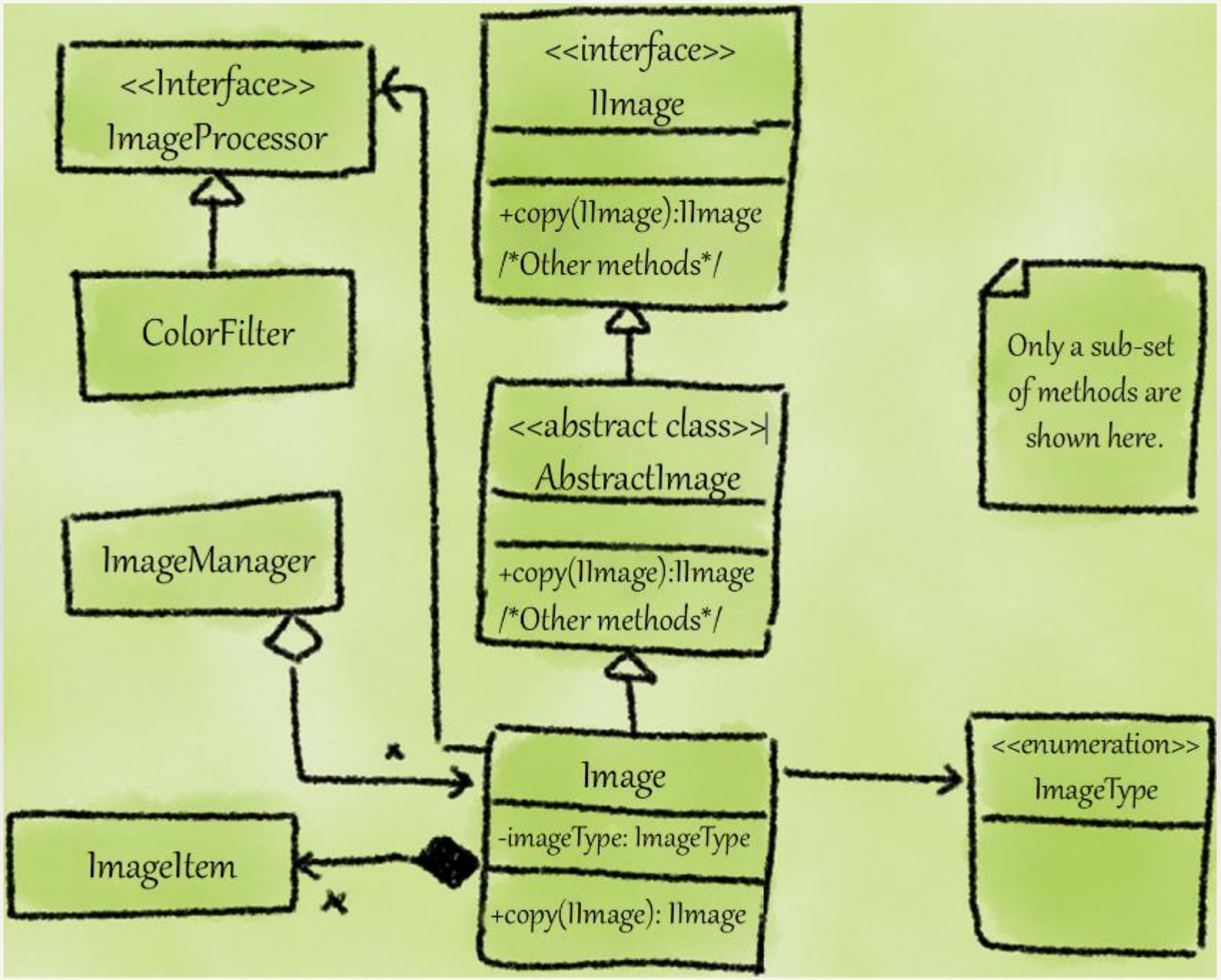
Creational	Deals with controlled object creation	*Factory method*, for example
Structural	Deals with composition of classes or objects	*Composite*, for example
Behavioral	Deals with interaction between objects/classes and distribution of responsibility	*Strategy*, for example

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

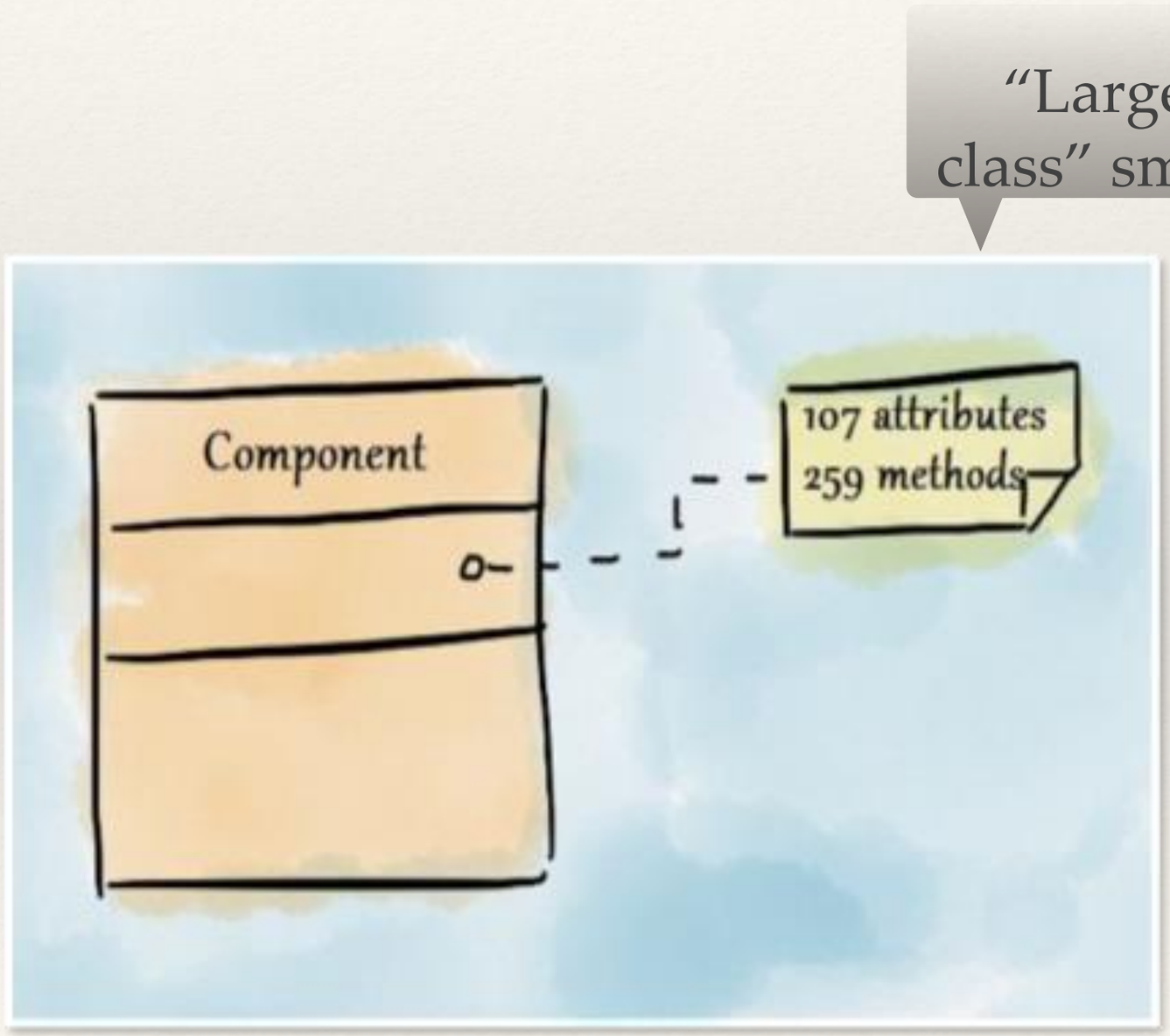


Source: "Refactoring for Software Design Smells: Managing Technical Debt", Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Morgan Kaufmann/Elsevier, 2014

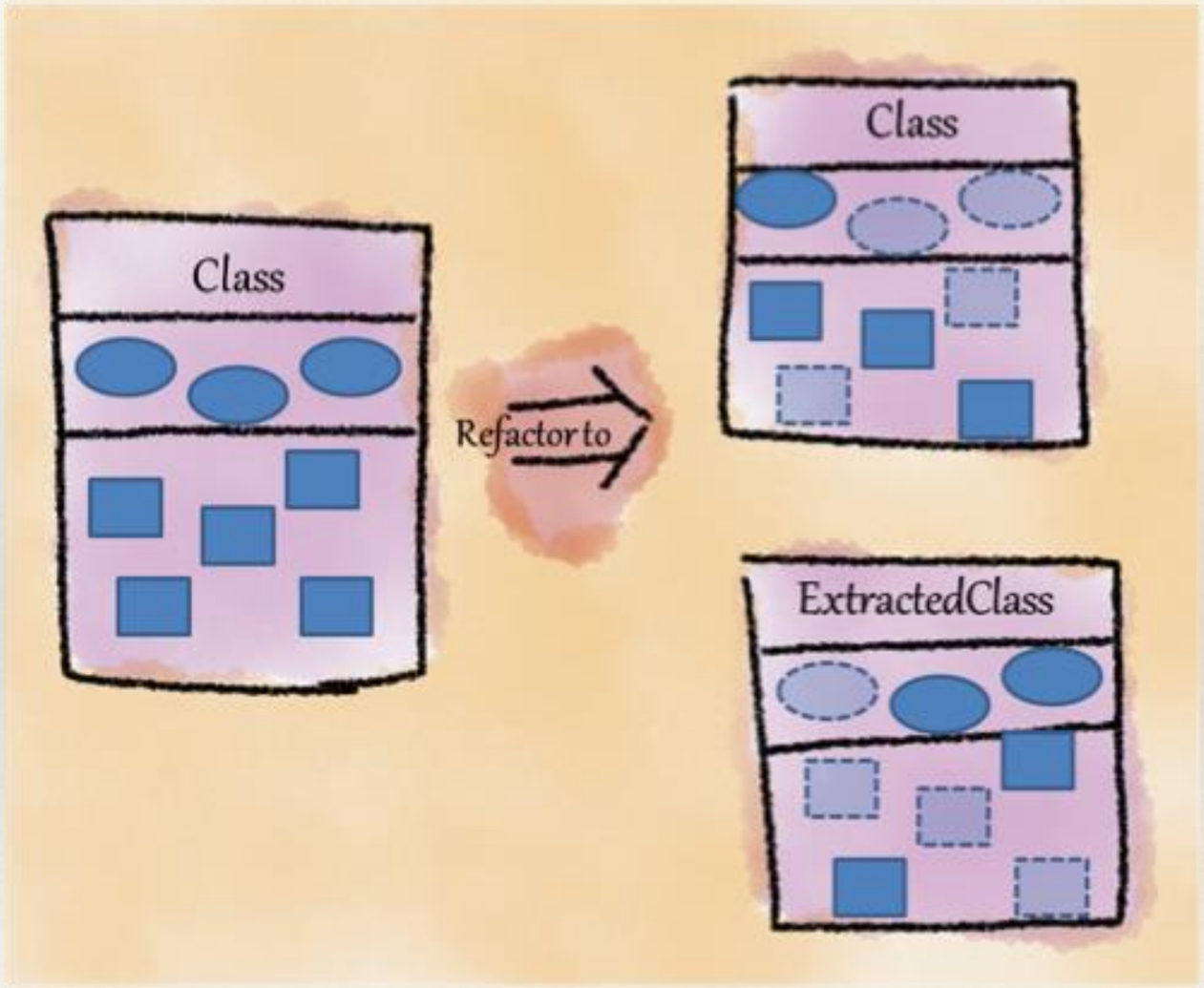
What's that smell?

```
 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  }
595
```

What's that smell?



Refactoring with “extract class”



Single Responsibility Principle

There should be only one reason for a class to change

Hands-on exercise

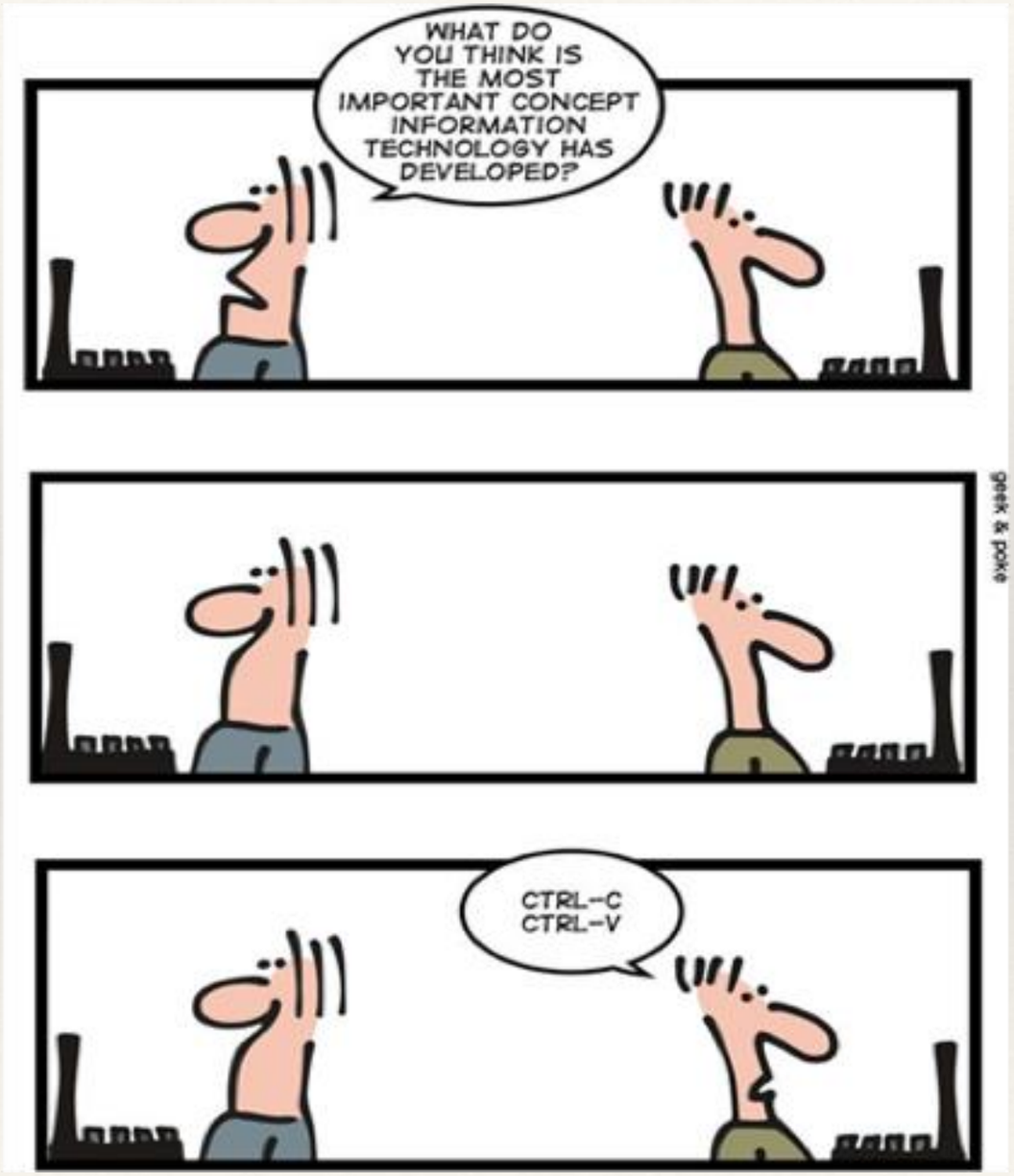
```
0 references
class DocumentProcesor
{
    0 references
    static void Main(string[] args)
    {
        var sourceFileName = Path.Combine(Environment.CurrentDirectory, "..\\..\\..\\Input Documents\\Document1.xml");
        var targetFileName = Path.Combine(Environment.CurrentDirectory, "..\\..\\..\\Output Documents\\Document1.json");

        var input = GetInput(sourceFileName);
        var doc = GetDocument(input);
        var serializedDoc = SerializeDocument(doc);
        PersistDocument(serializedDoc, targetFileName);
    }

    1 reference
    private static string GetInput(string sourceFileName)...
    1 reference
    private static Document GetDocument(string input)...
    1 reference
    private static string SerializeDocument(Document doc)...
    1 reference
    private static void PersistDocument(string serializedDoc, string targetFileName)...
}
```

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

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

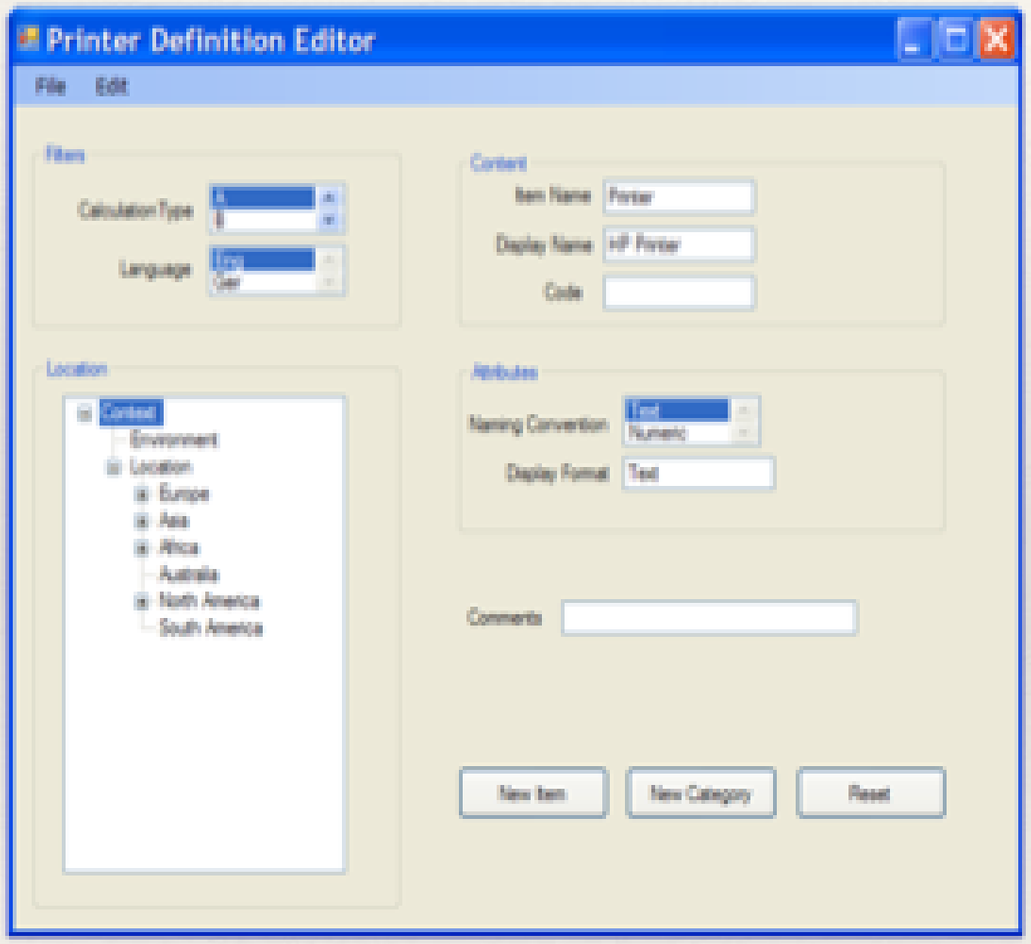
Type 3

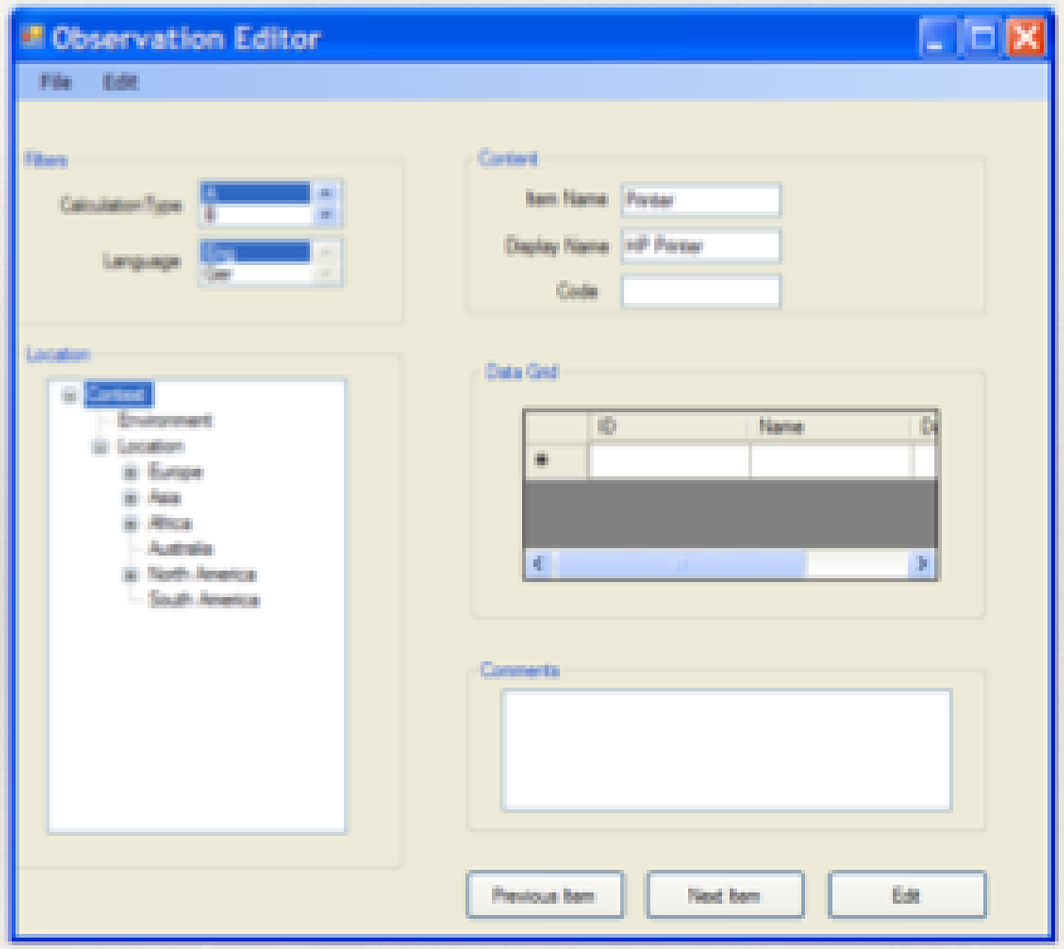
- identical except some **statements changed, added, or removed**

Type 4

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

How to deal with duplication?





What’s that smell?

```
1 reference
public class Person
{
    0 references
    public int Id { get; set; }

    0 references
    public  string Name { get; set; }

    0 references
    public string Address { get; set; }

}

0 references
public class PersonBusinessLogic
{
    0 references
    public bool Validate(Person person)
    {
        //Logic
        return true;
    }
}
```

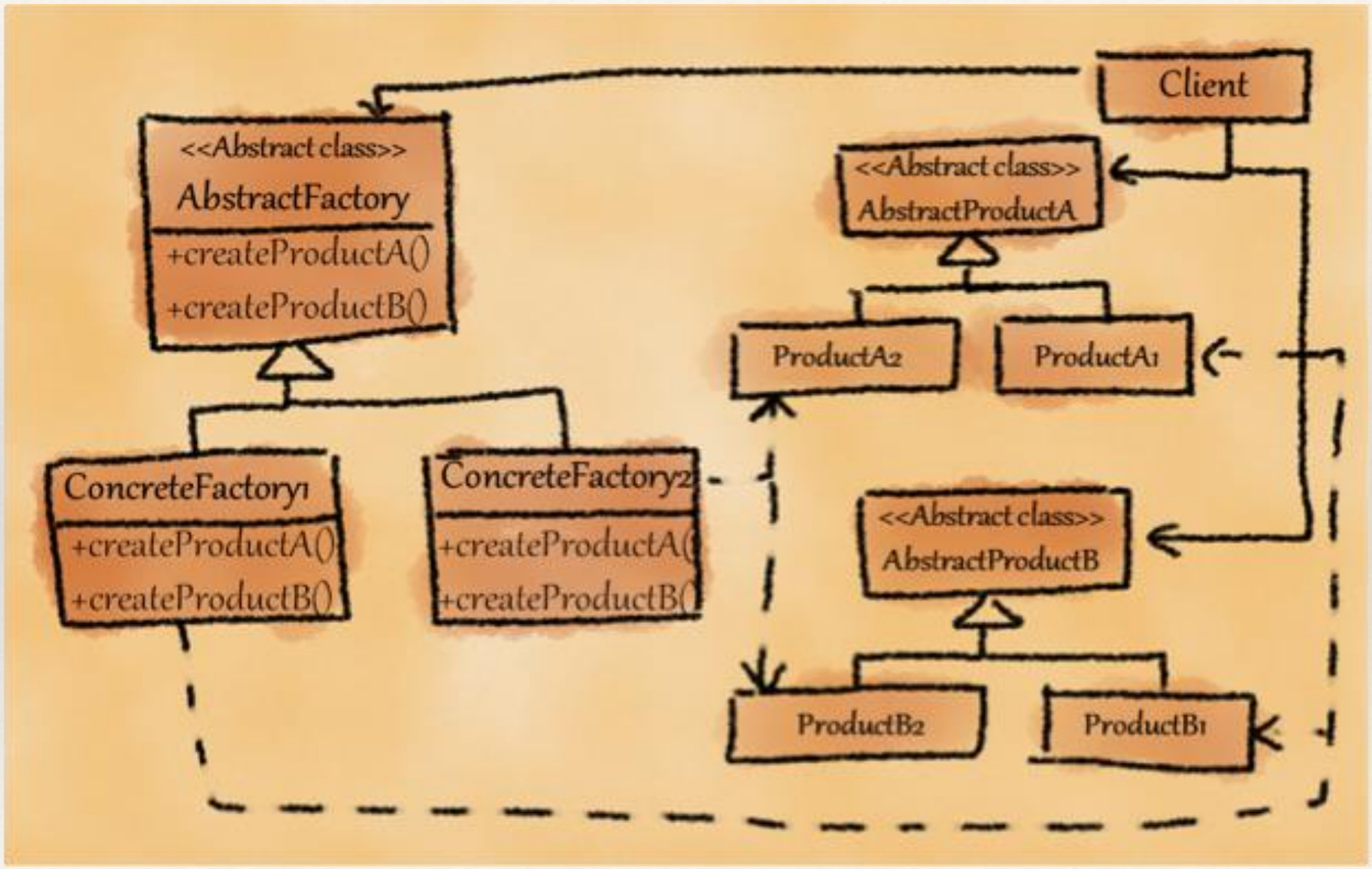
What’s that smell?

```
0 references
public class Control
{
    private bool _isEnabled = false;

    0 references
    public void Enable()
    {
        _isEnabled = true;
    }
}
```

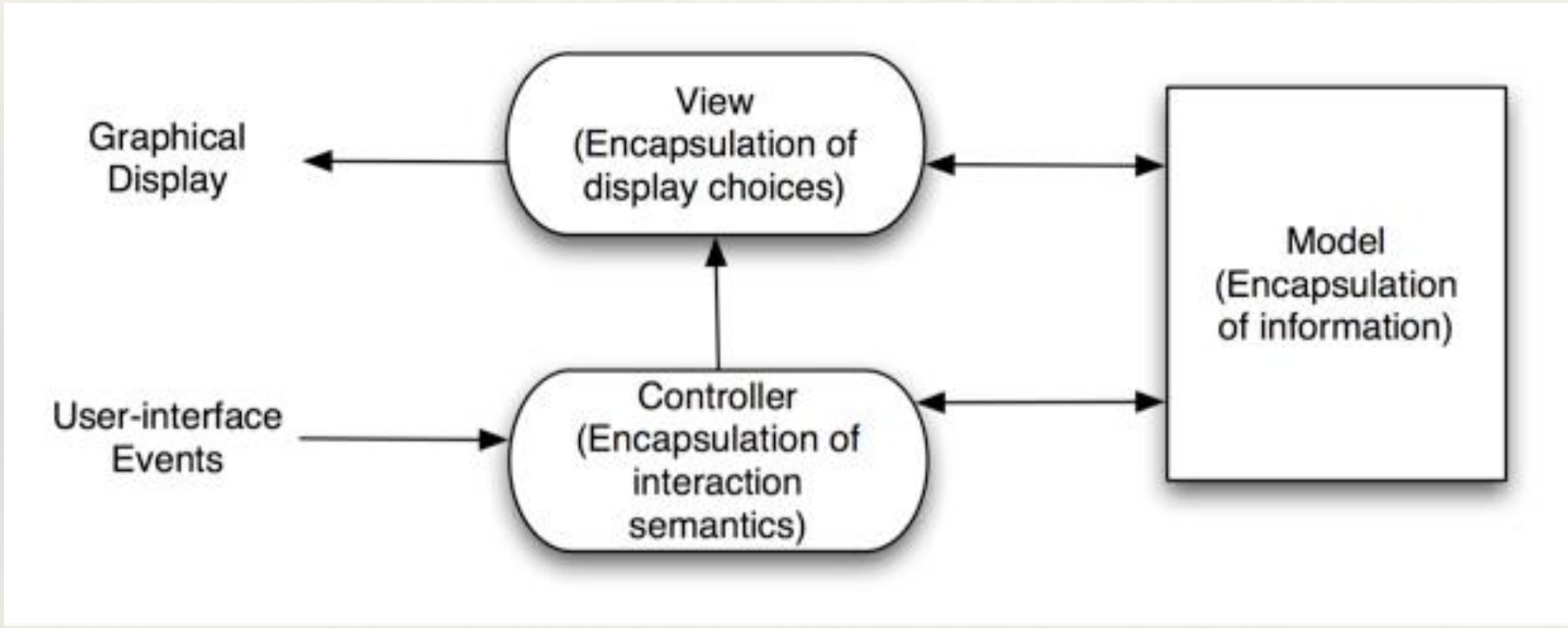
Refactoring “incomplete library classes” smell

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

Abstract factory pattern

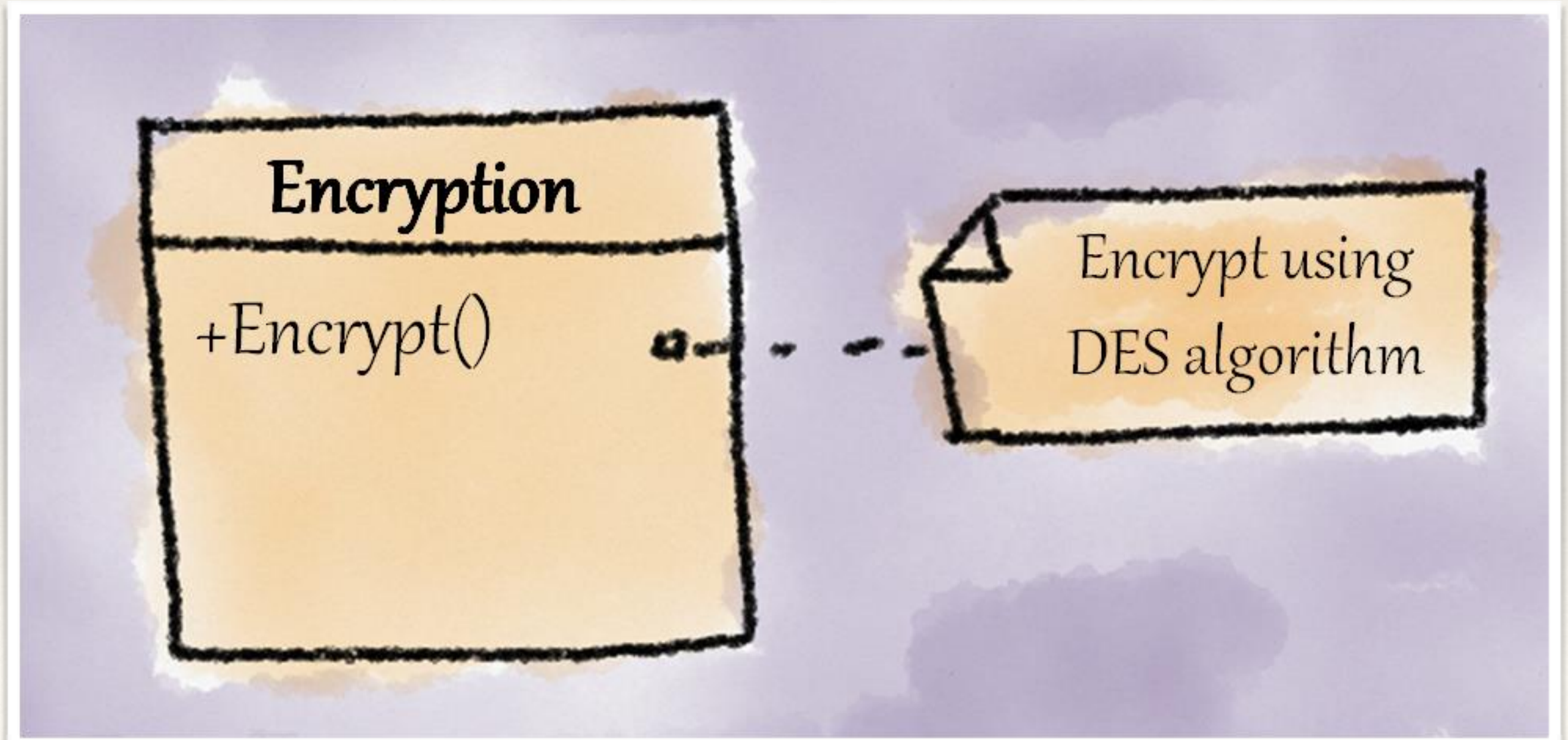


Separating concerns in MVC



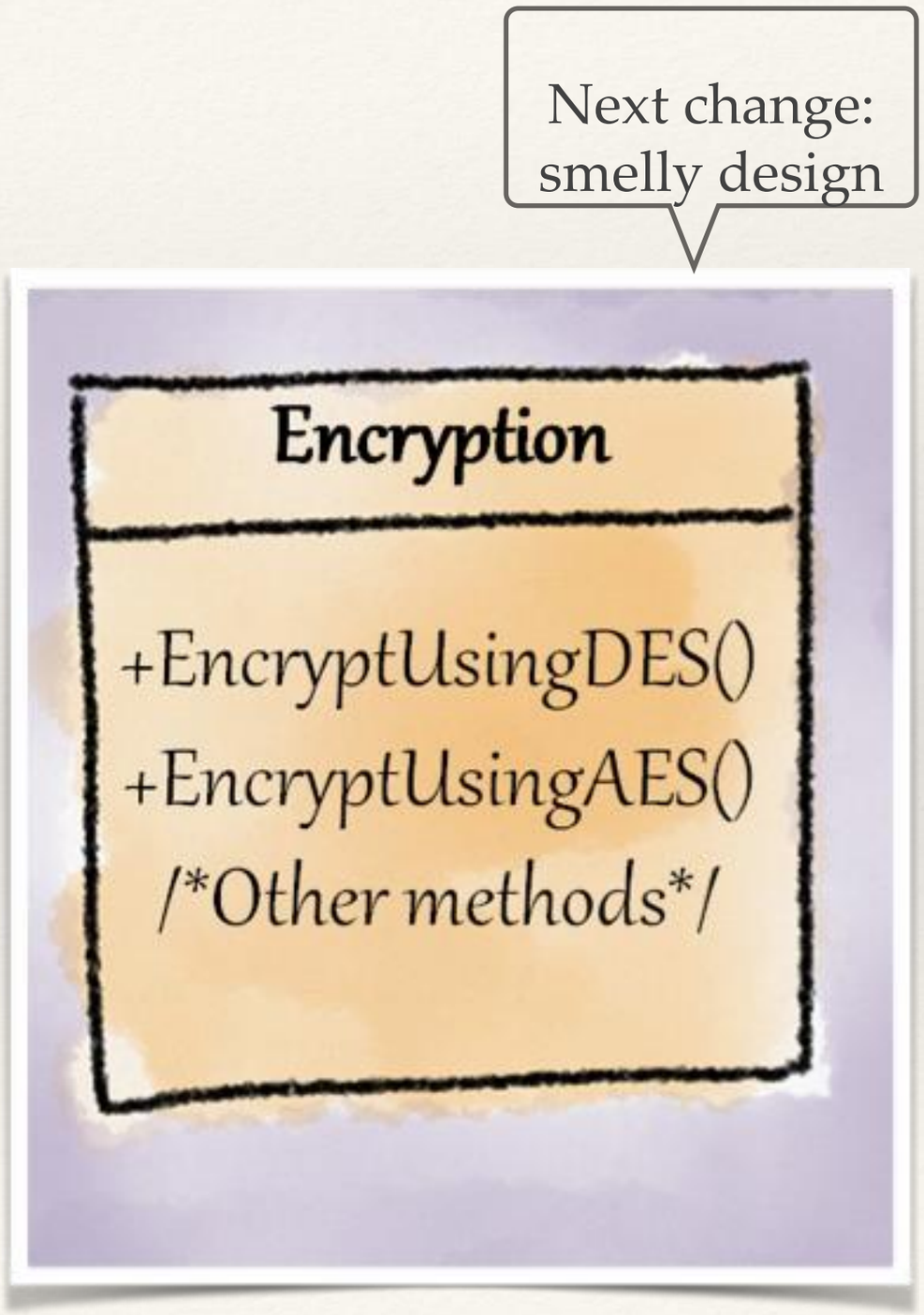
Real scenario #1

Initial design

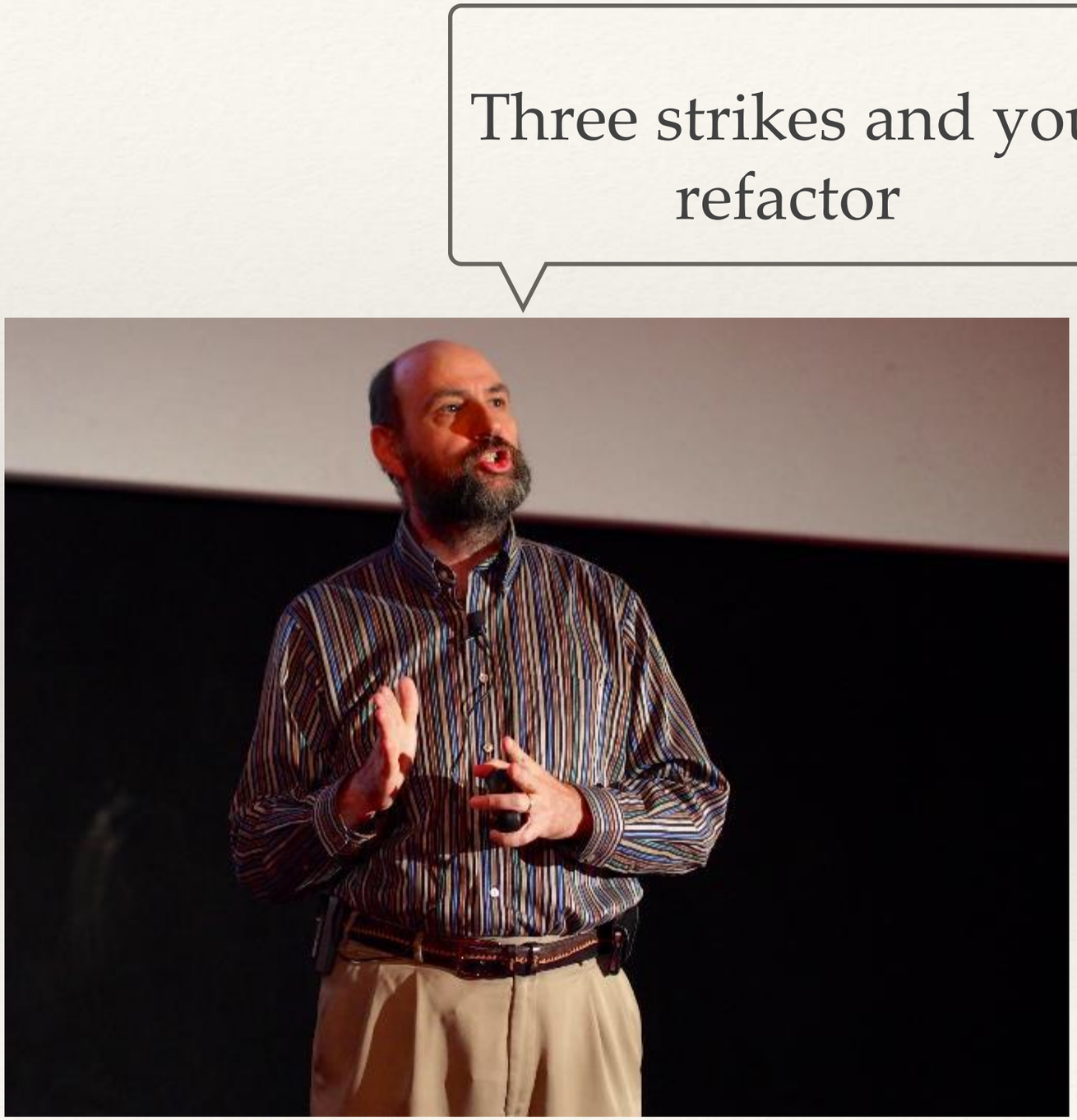


Real scenario #1

- How will you refactor such that:
  - A specific DES, AES, TDES, … can be “plugged” at runtime?
  - Reuse these algorithms in new contexts?
  - Easily add support for new algorithms in Encryption?



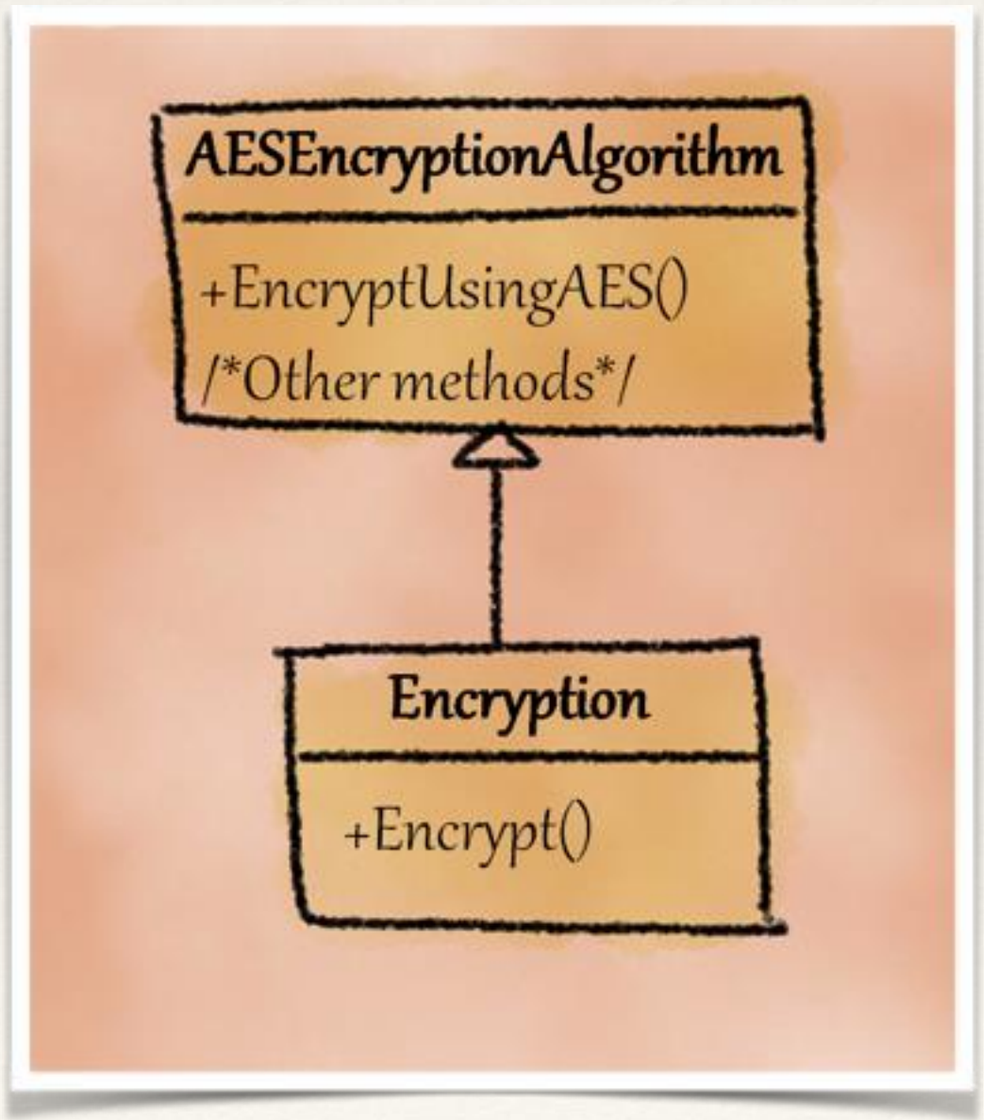
Time to refactor!



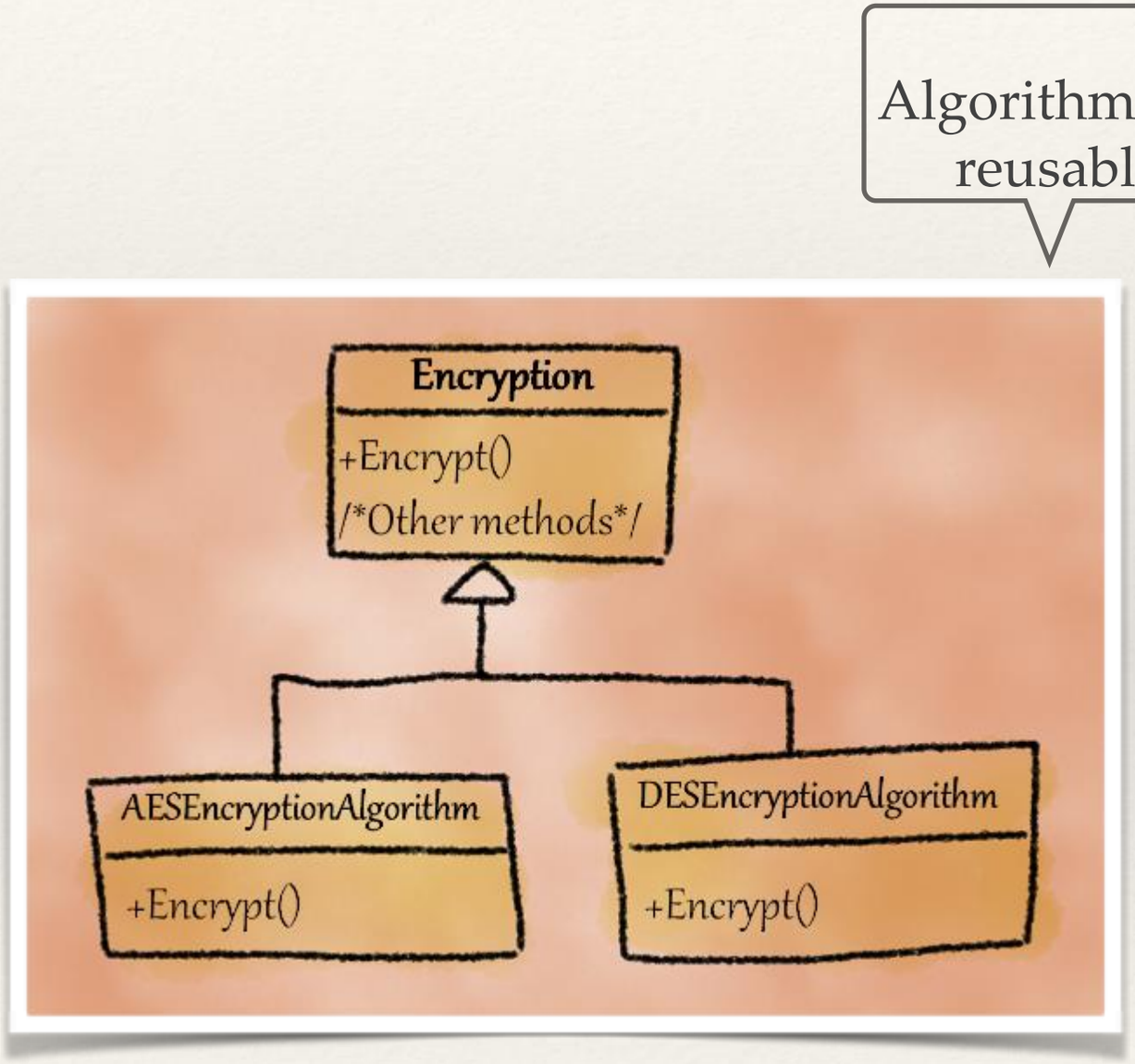
Martin Fowler

Potential solution #1?

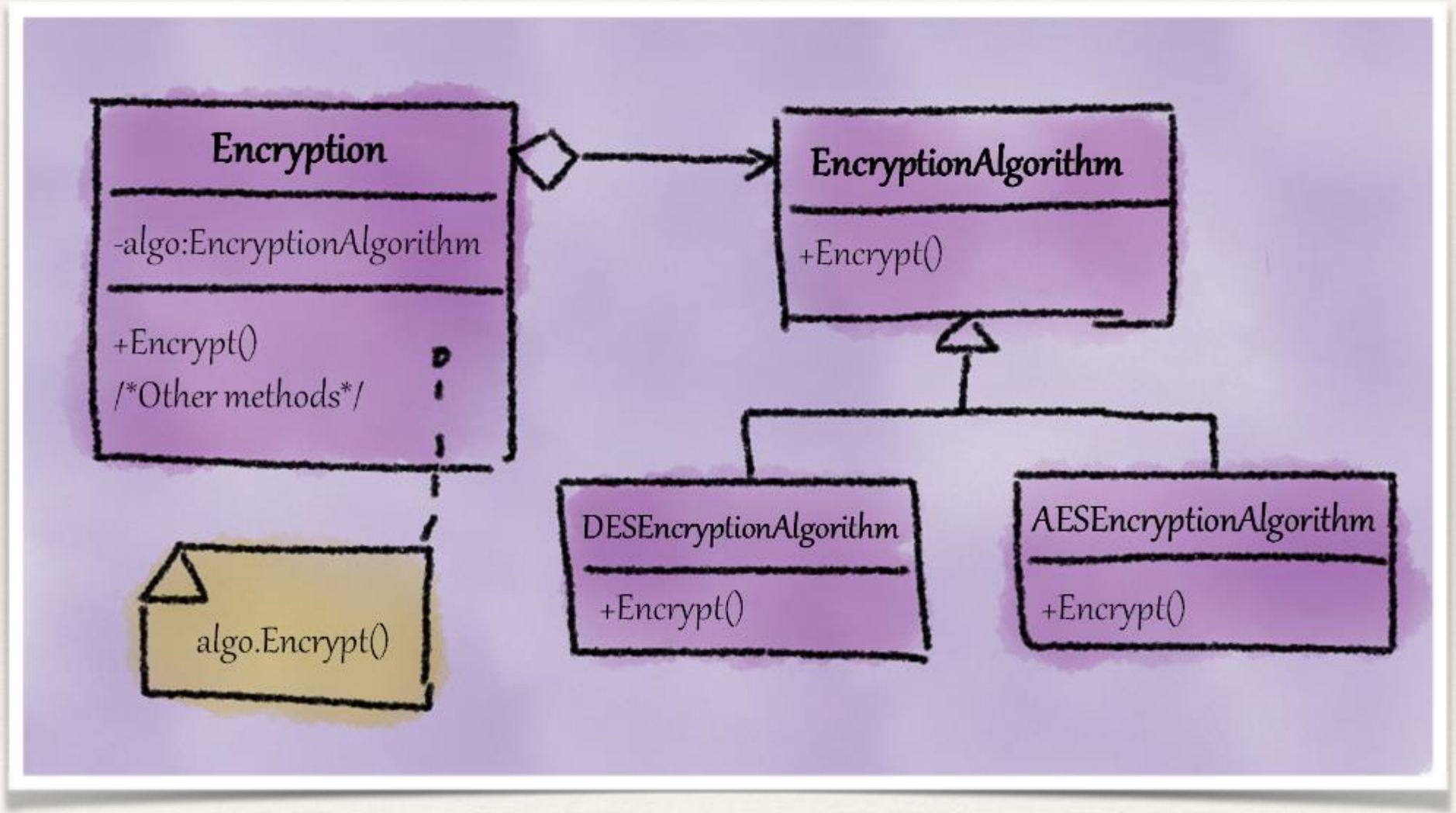
Broken hierarchy!



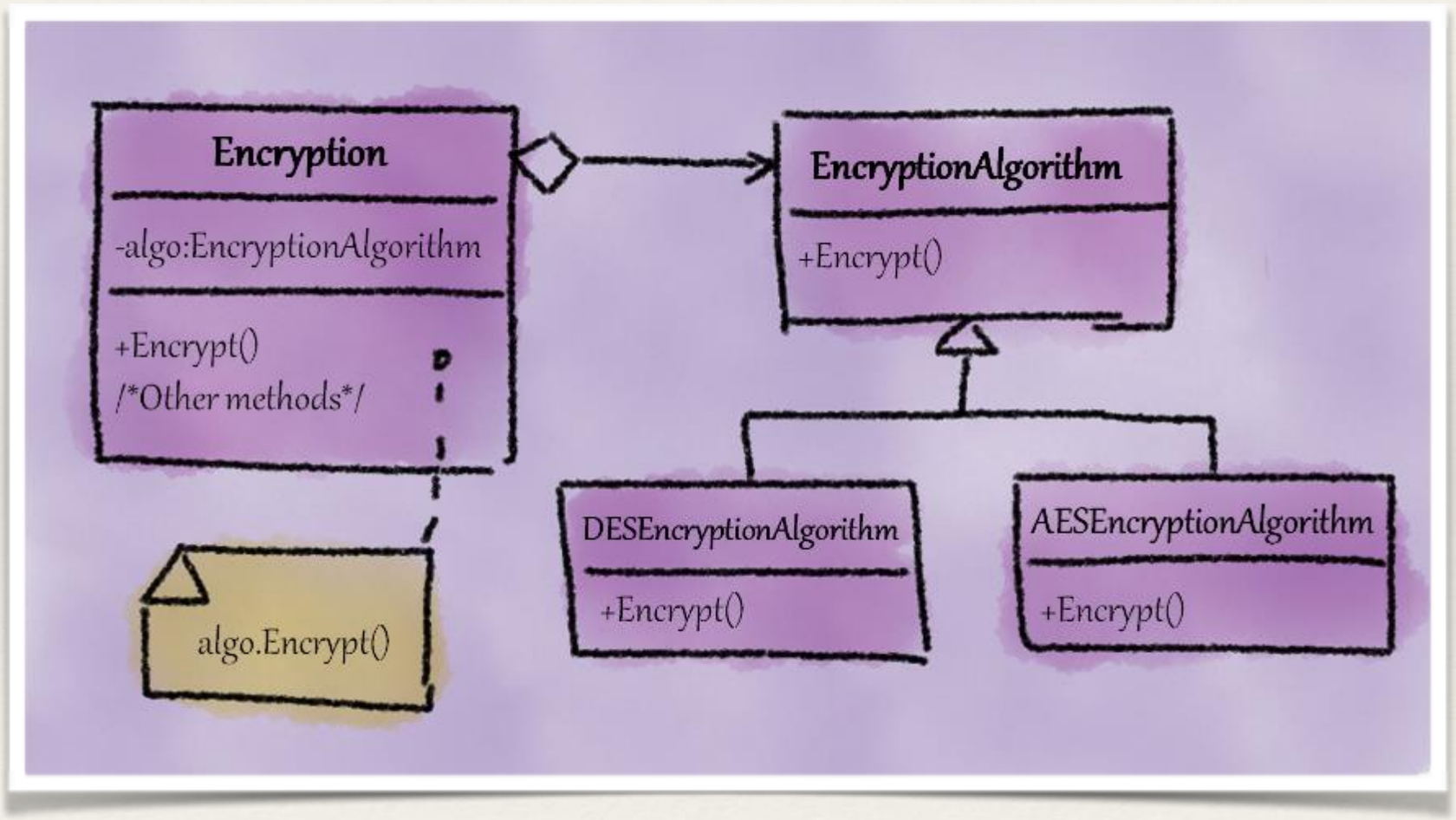
Potential solution #2?



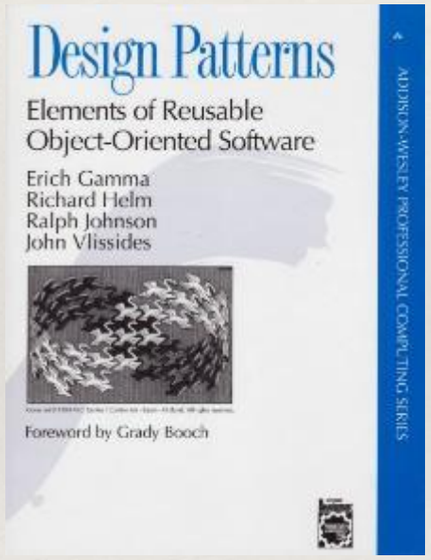
Potential solution #3?

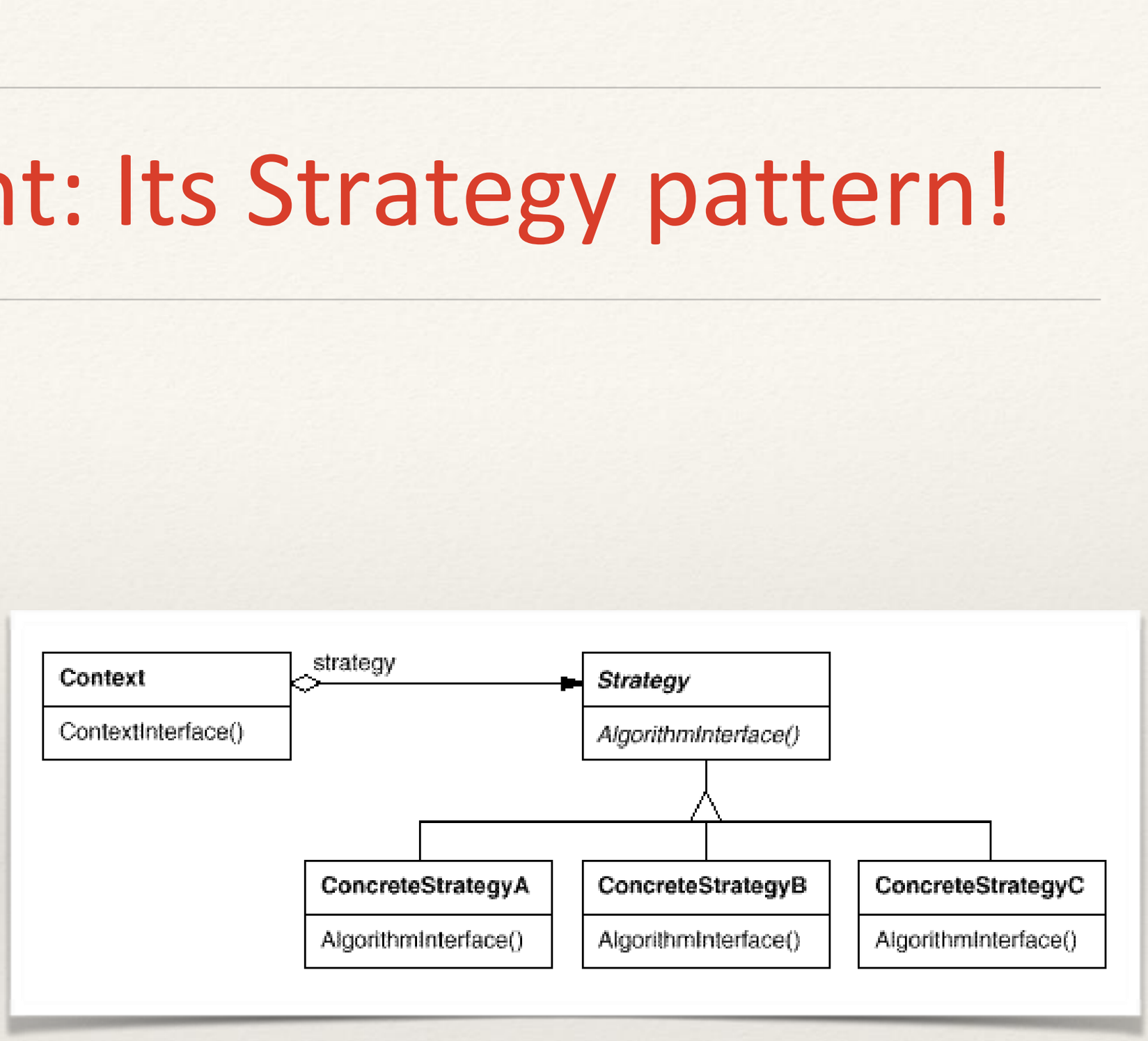


Can you identify the pattern?



You're right: Its Strategy pattern!

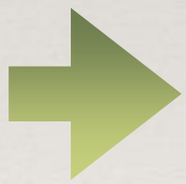




Strategy pattern: Discussion

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it

- Useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm that suits its current need



- The implementation of each of the algorithms is kept in a separate class referred to as a strategy.
- An object that uses a Strategy object is referred to as a context object.
- Changing the behavior of a Context object is a matter of changing its Strategy object to the one that implements the required algorithm

Hands-on Exercise

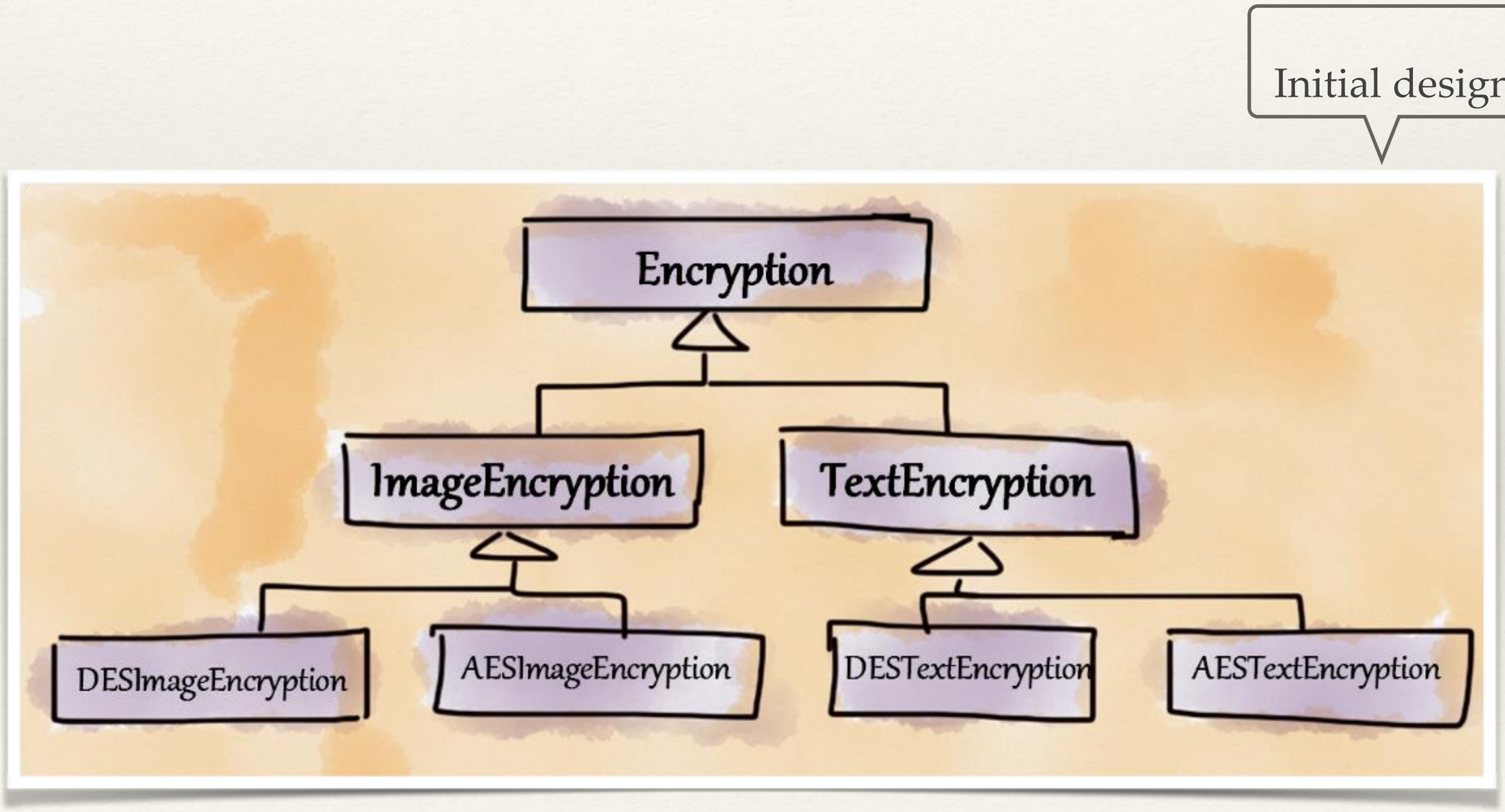
```
internal class EncryptionAlgoUtil
{
    public static readonly Action DesAlgo = () => Console.WriteLine("DES");
    public static readonly Action AesAlgo = () => Console.WriteLine("AES");
    public static readonly Action TdesAlgo = () => Console.WriteLine("TDES");
}

internal class Encryption
{
    private Action _algo = EncryptionAlgoUtil.DesAlgo;

    public Encryption()...

    public Encryption(Action algo)...

    public virtual Action EncryptionAlgorithm...

    public virtual void Encrypt()...
}
```

Solution Location: Solid-Master->Strategy->Encryption.cs

Hands-on exercise

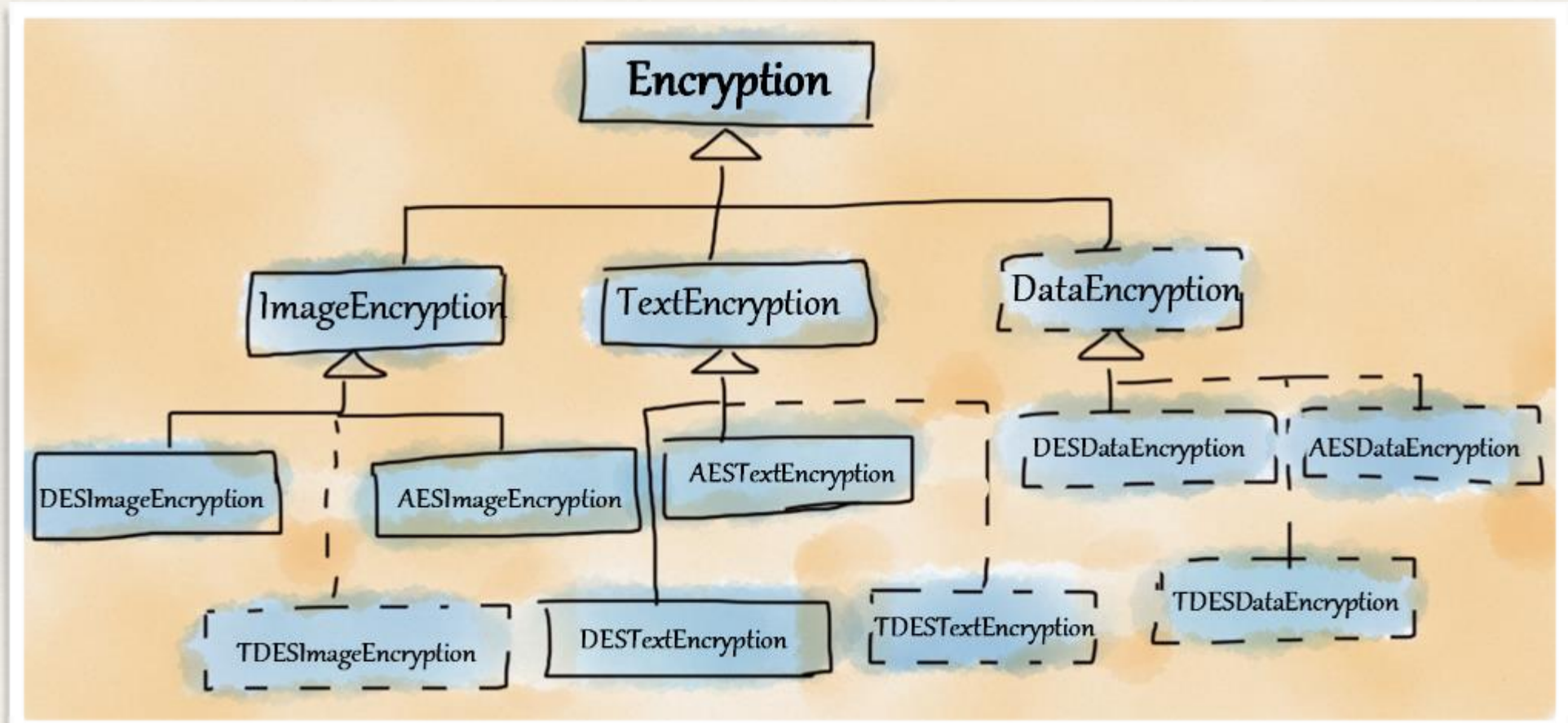
```
1 reference
internal class SortedList
{
    private readonly List<string> _list = new List<string>();
    private SortStrategy _sortstrategy;

    3 references
    public void SetSortStrategy(SortStrategy sortstrategy)
    {
        _sortstrategy = sortstrategy;
    }
```

Real scenario #2

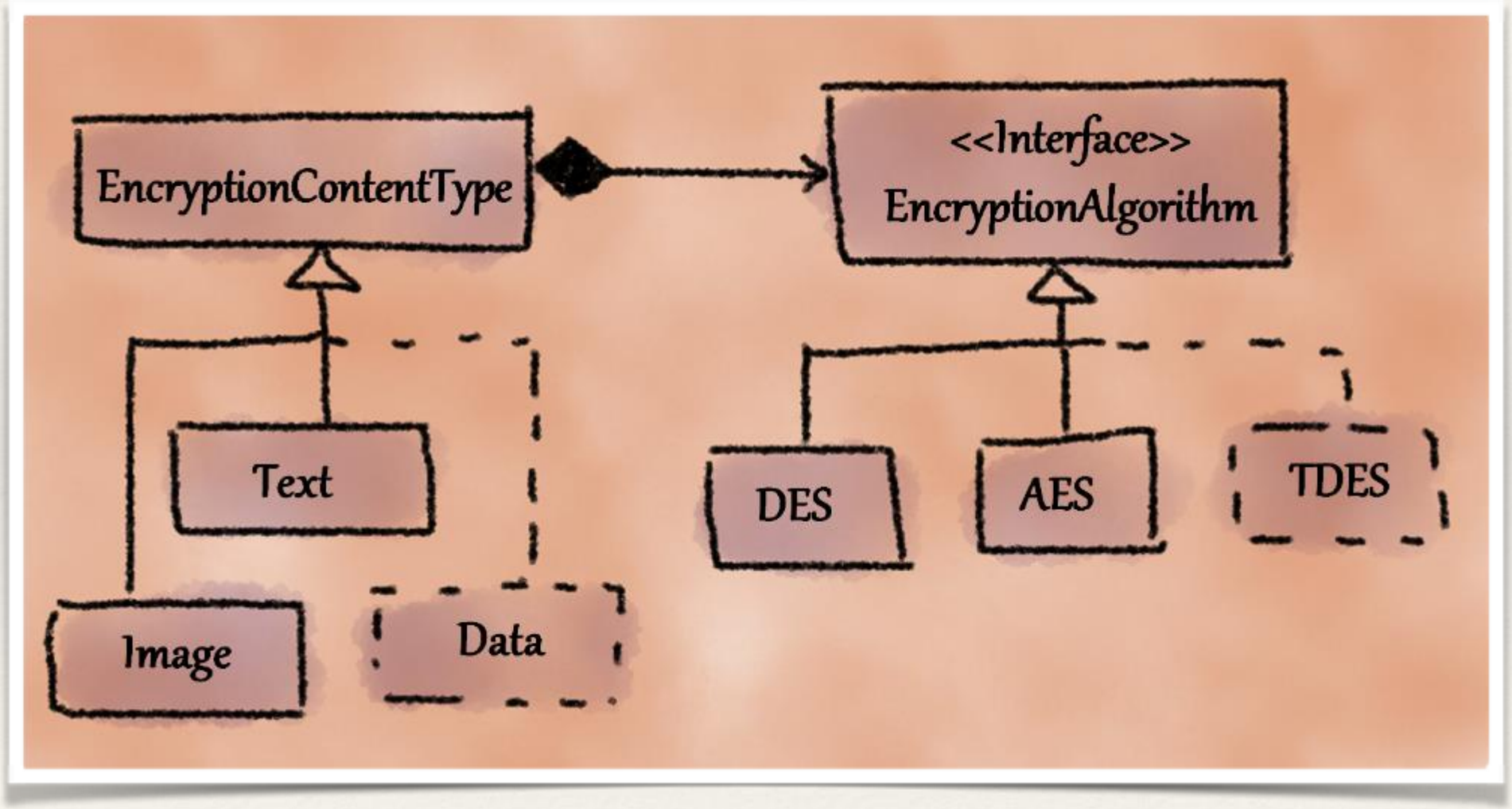


Real scenario #2

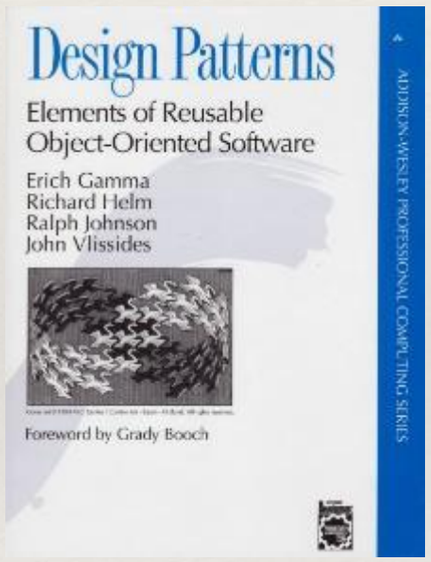
How to add support for new content types and/or algorithms?

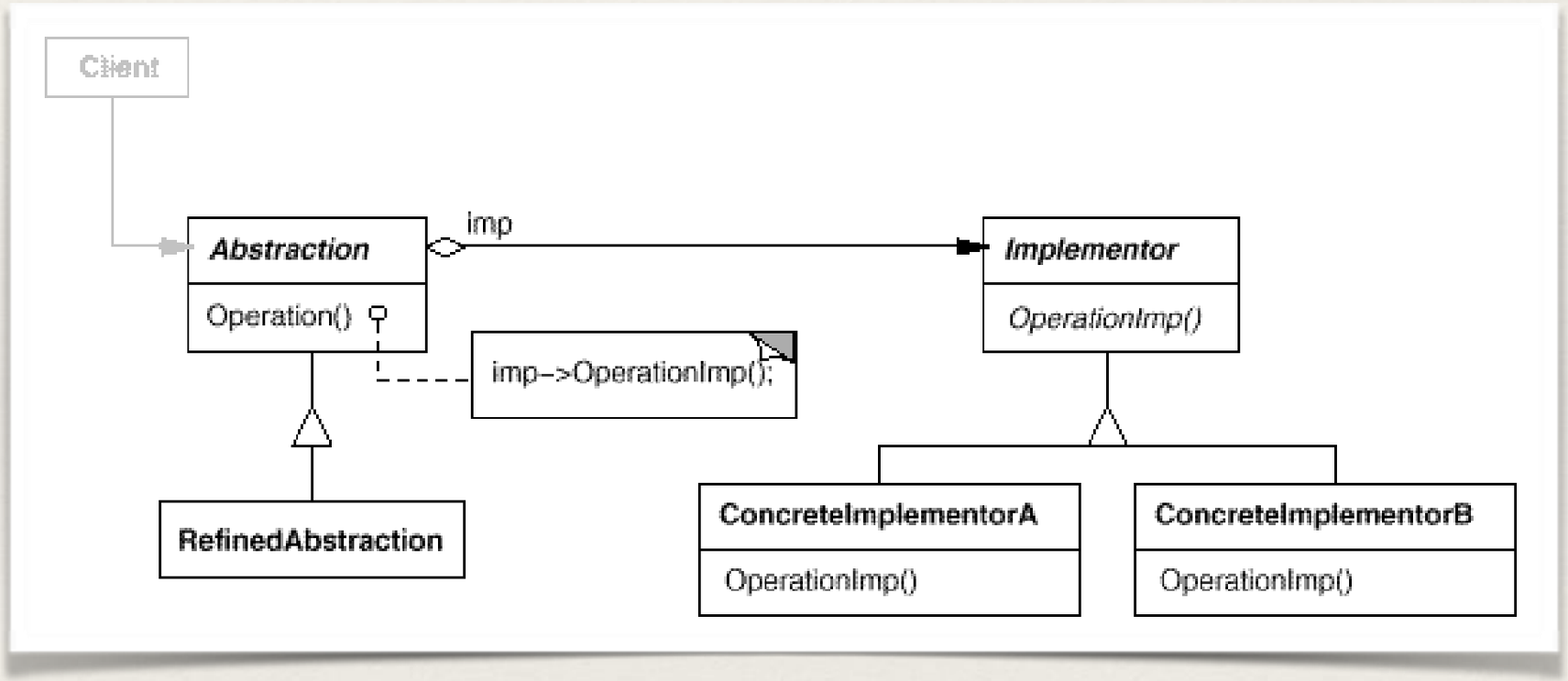


Can you identify the pattern structure?



You're right: Its Bridge pattern structure!





Hands-on Exercise

```
internal class EncryptionAlgoUtil
{
    public static readonly Action DesAlgo = () => Console.WriteLine("DES");
    public static readonly Action AesAlgo = () => Console.WriteLine("AES");
    public static readonly Action TdesAlgo = () => Console.WriteLine("TDES");
}

internal class Encryption
{
    private Action _algo = EncryptionAlgoUtil.DesAlgo;

    public Encryption()...

    public Encryption(Action algo)...

    public virtual Action EncryptionAlgorithm...

    public virtual void Encrypt()...
}
```

How about expanding this code to use Bridge structure (multiple content types)?

Open Closed Principle (OCP)

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

Bertrand Meyer

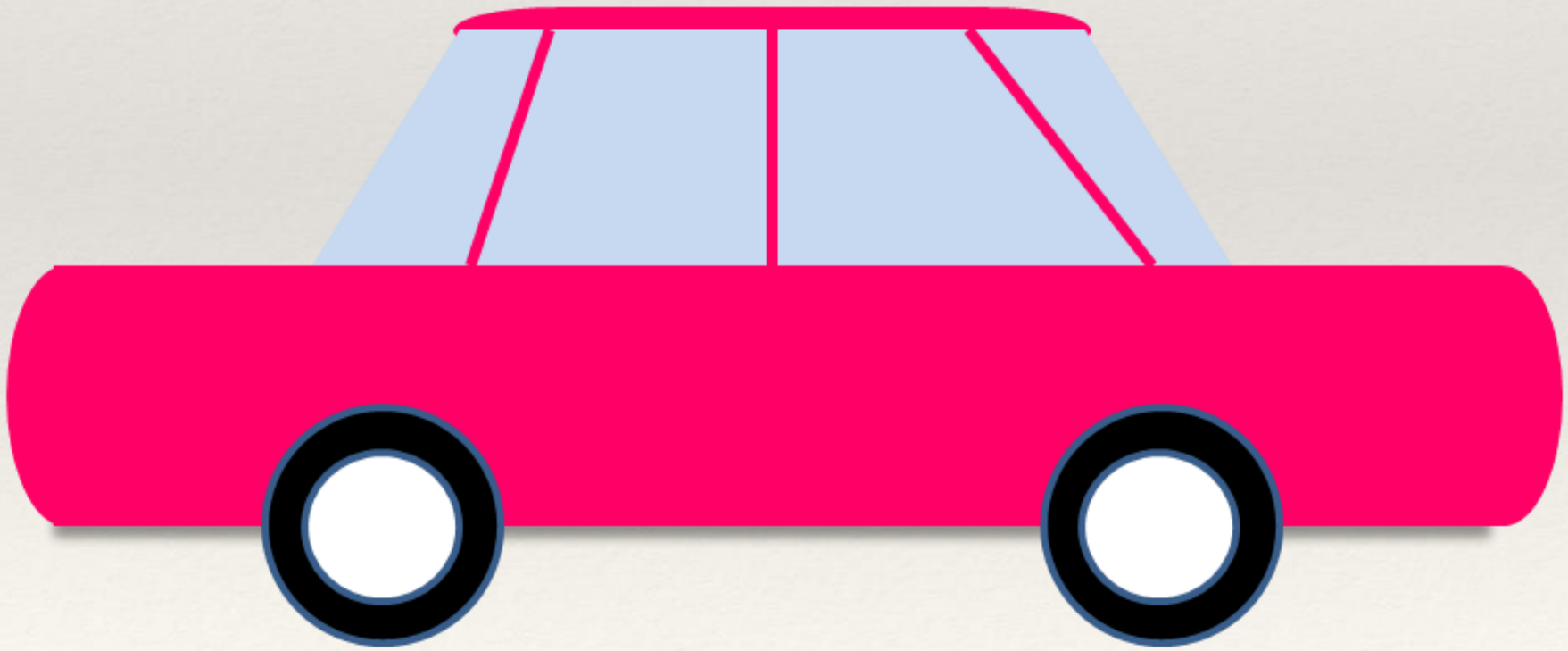
Variation Encapsulation Principle (VEP)

Encapsulate the concept that varies

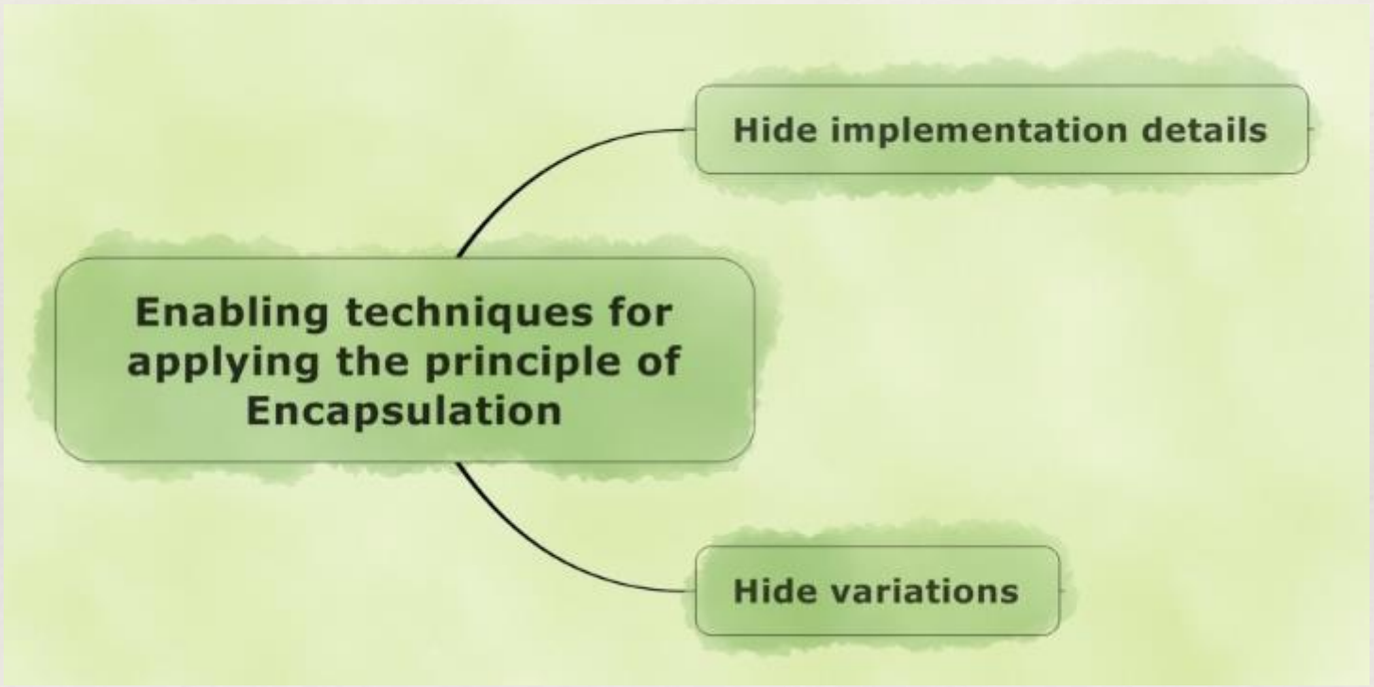
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

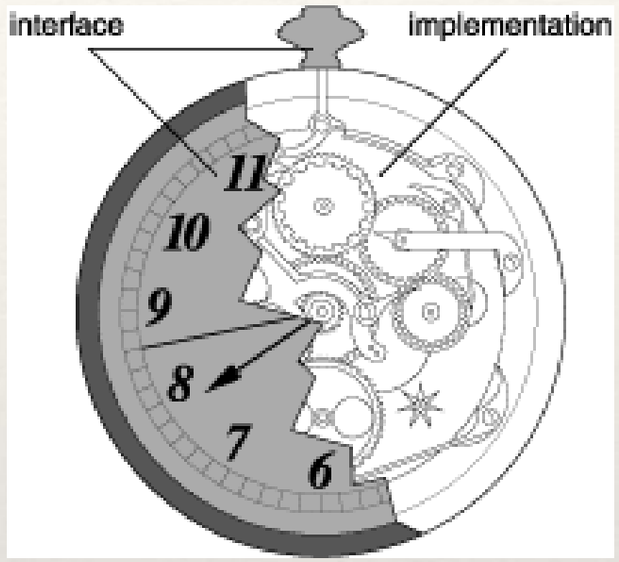
Fundamental principle: Encapsulation

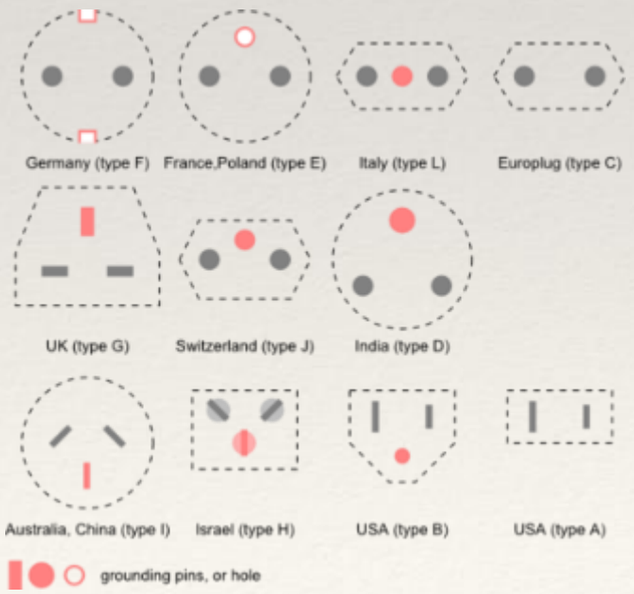
The principle of encapsulation advocates separation of concerns and information hiding through techniques such as hiding implementation details of abstractions and hiding variations



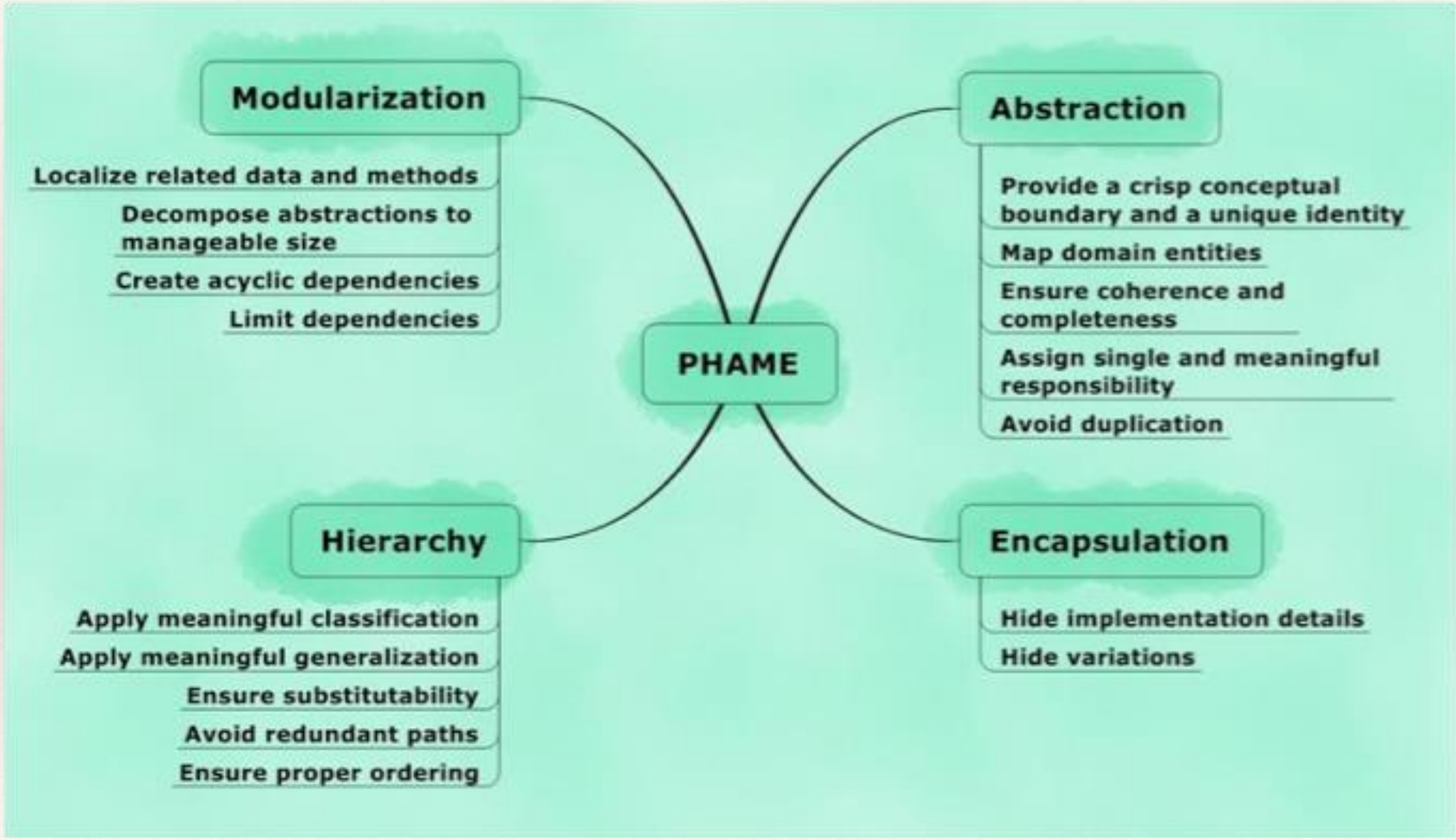
Enabling techniques for encapsulation







Design principles and enabling techniques



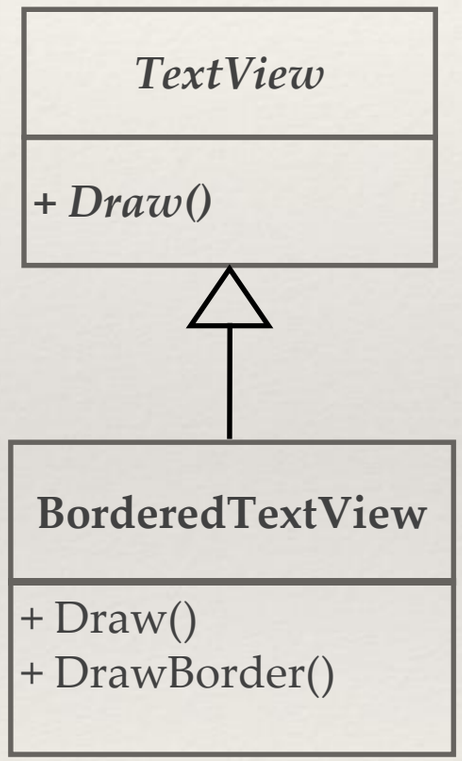
What’s that smell?

```
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;
}
```

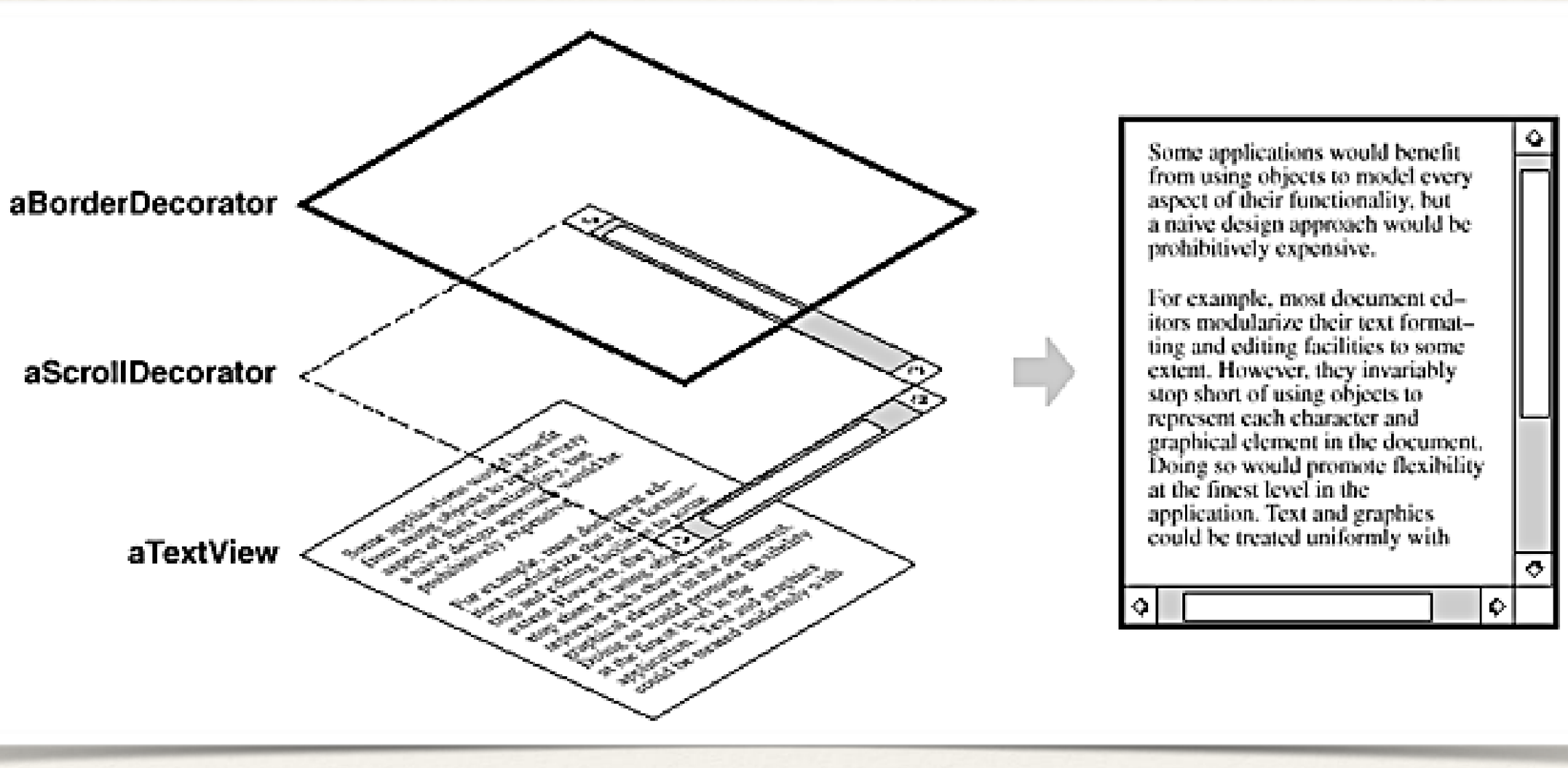
Solution Location: Solid-Assessment->Before->Device.cs

Scenario

Initial design



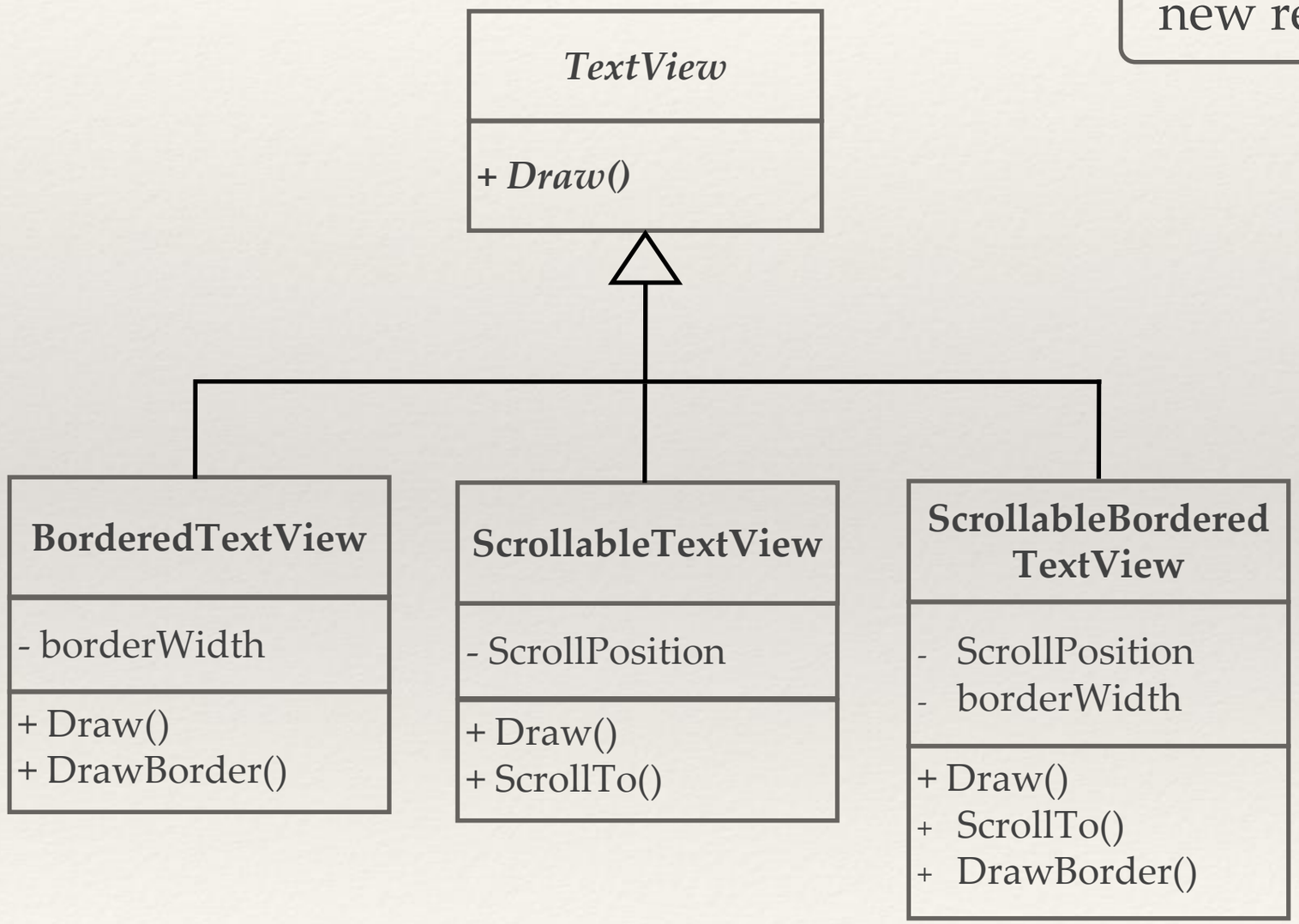
Real scenario



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

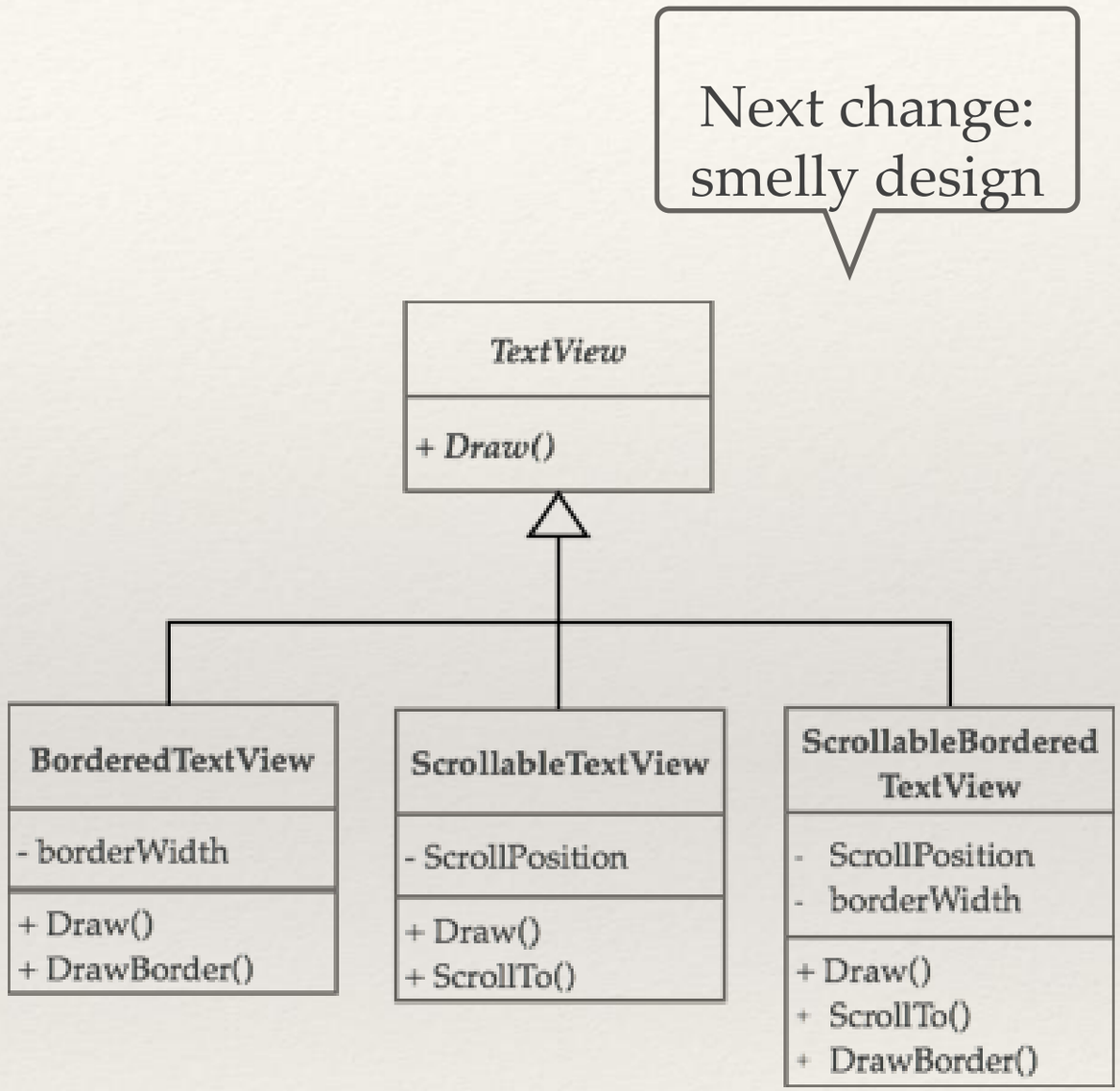
Supporting new requirements

Revised design with new requirements

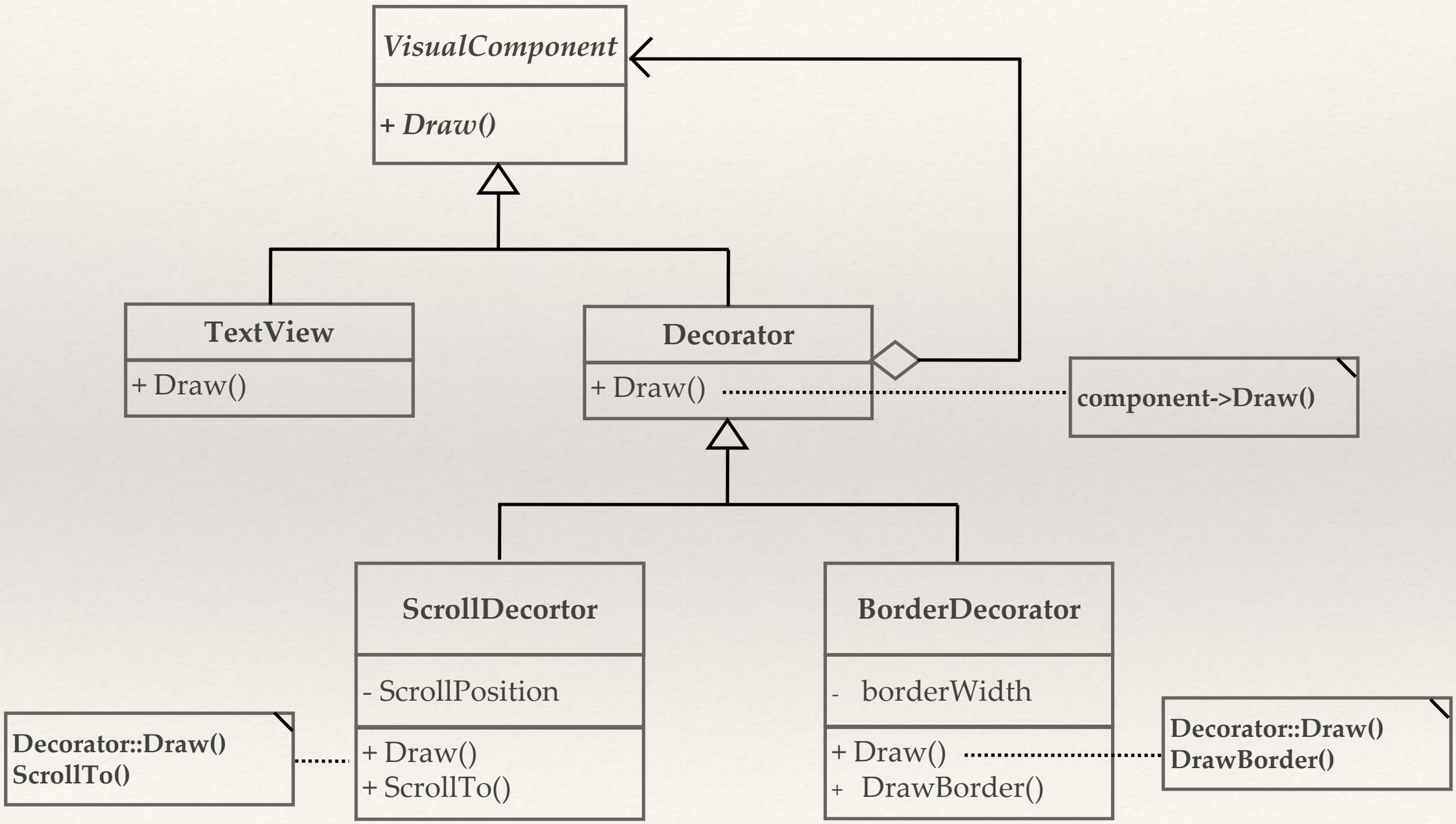


Real scenario

- How will you refactor such that:
  - You don't have to “multiply-out” sub-types? (i.e., avoid “explosion of classes”)
  - Add or remove responsibilities (e.g., scrolling) at runtime?



How about this solution?

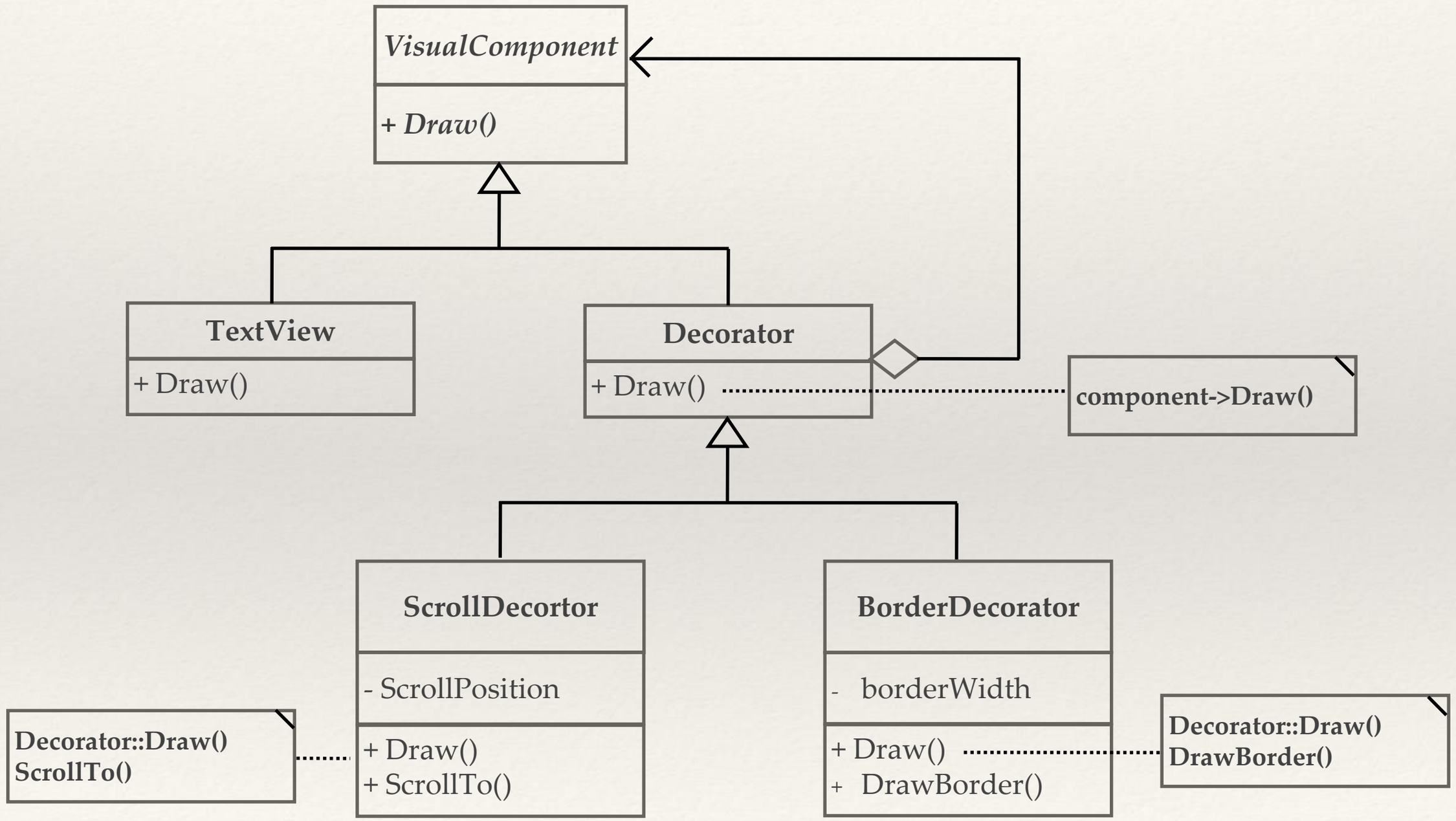


Source: "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

At runtime (object diagram)

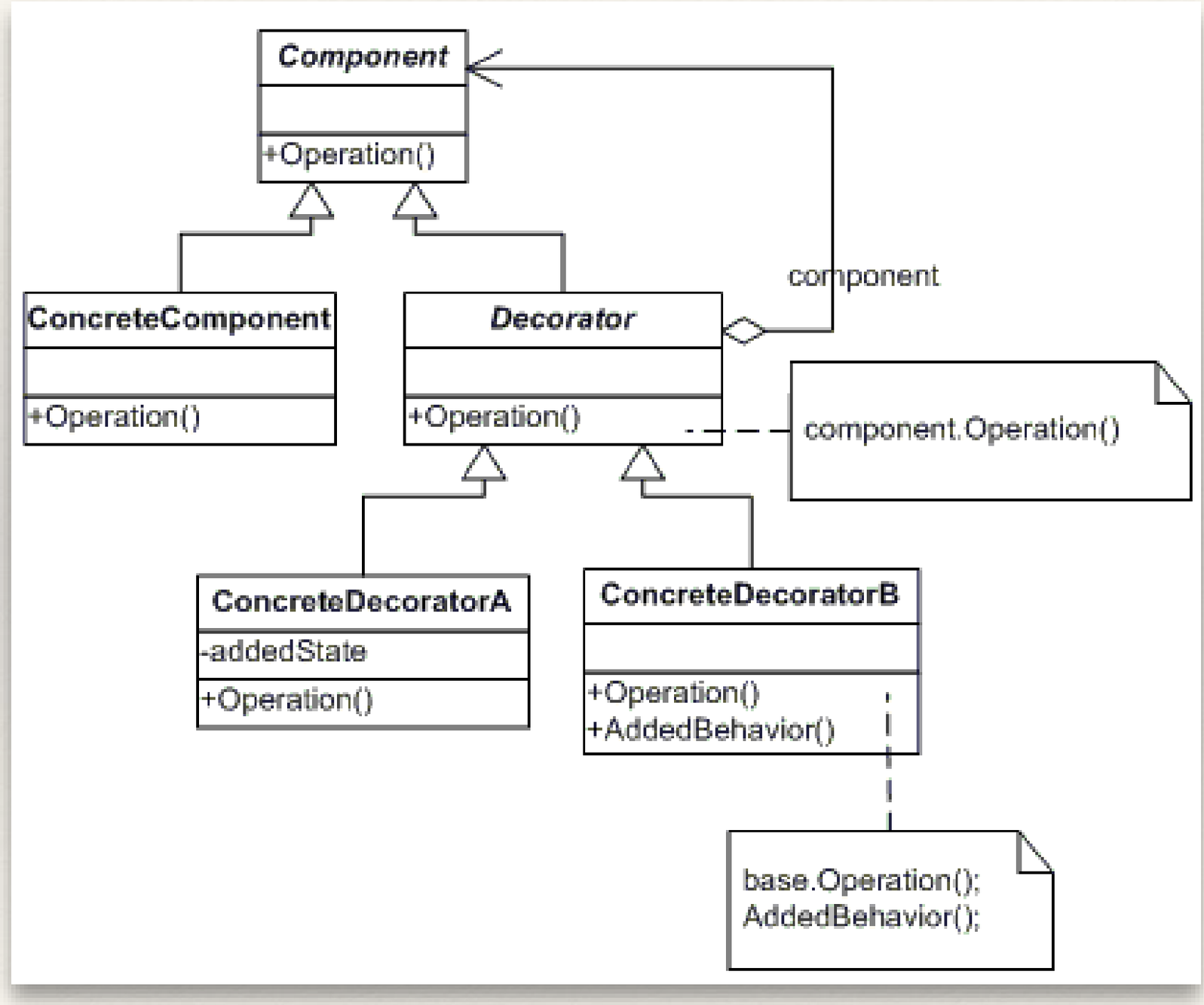
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Can you identify the pattern?



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

You're right: Its Decorator pattern!



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

Decorator pattern: Discussion

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

- Want to add responsibilities to individual objects (not an entire class)
- One way is to use inheritance
  - Inflexible; static choice
  - Hard to add and remove responsibilities dynamically

- Add responsibilities through decoration
  - in a way transparent to the clients
- Decorator forwards the requests to the contained component to perform additional actions
  - Can nest recursively
  - Can add an unlimited number of responsibilities dynamically

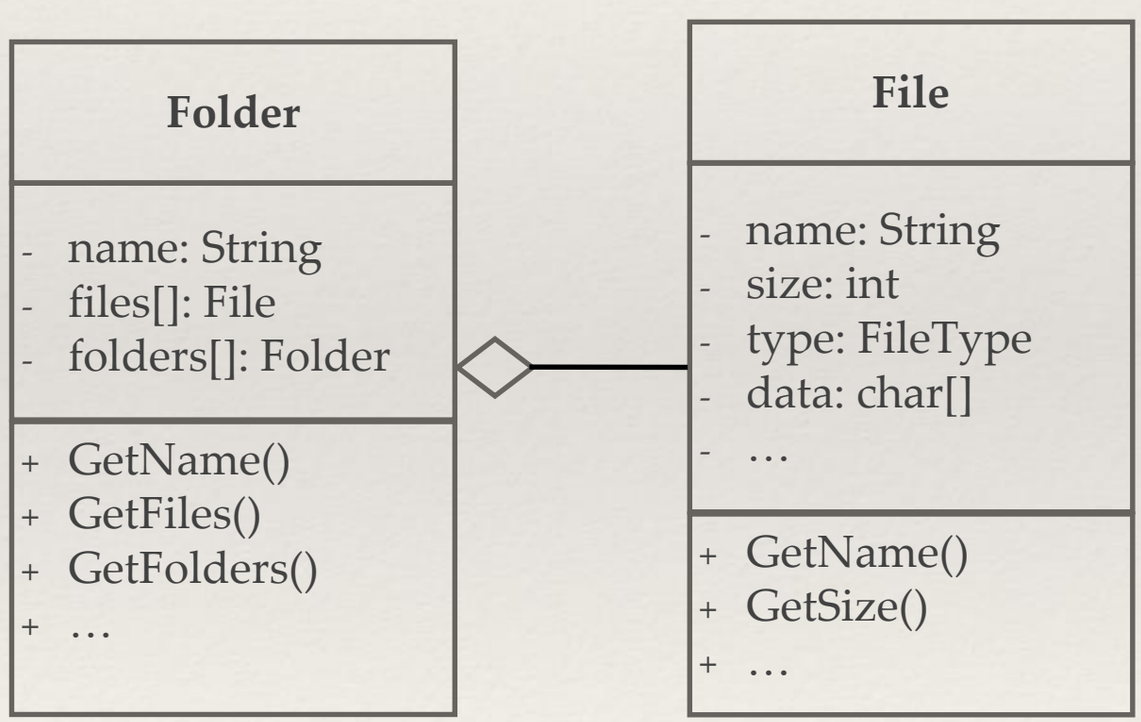
Identify pattern used in this code

```
public static void StreamReaderExample()
{
    using (var streamReader =
        new StreamReader(
            new BufferedStream(
                new FileStream("D:\\Work\\CodeOps\\SOLID-Training\\C#\\Solid-Master\\Solid-Master\\Decorator\\StreamRead.cs",
                    FileMode.Open, FileAccess.Read))))
    {
        Console.WriteLine(streamReader.ReadToEnd());
    }
    Console.ReadLine();
}
```

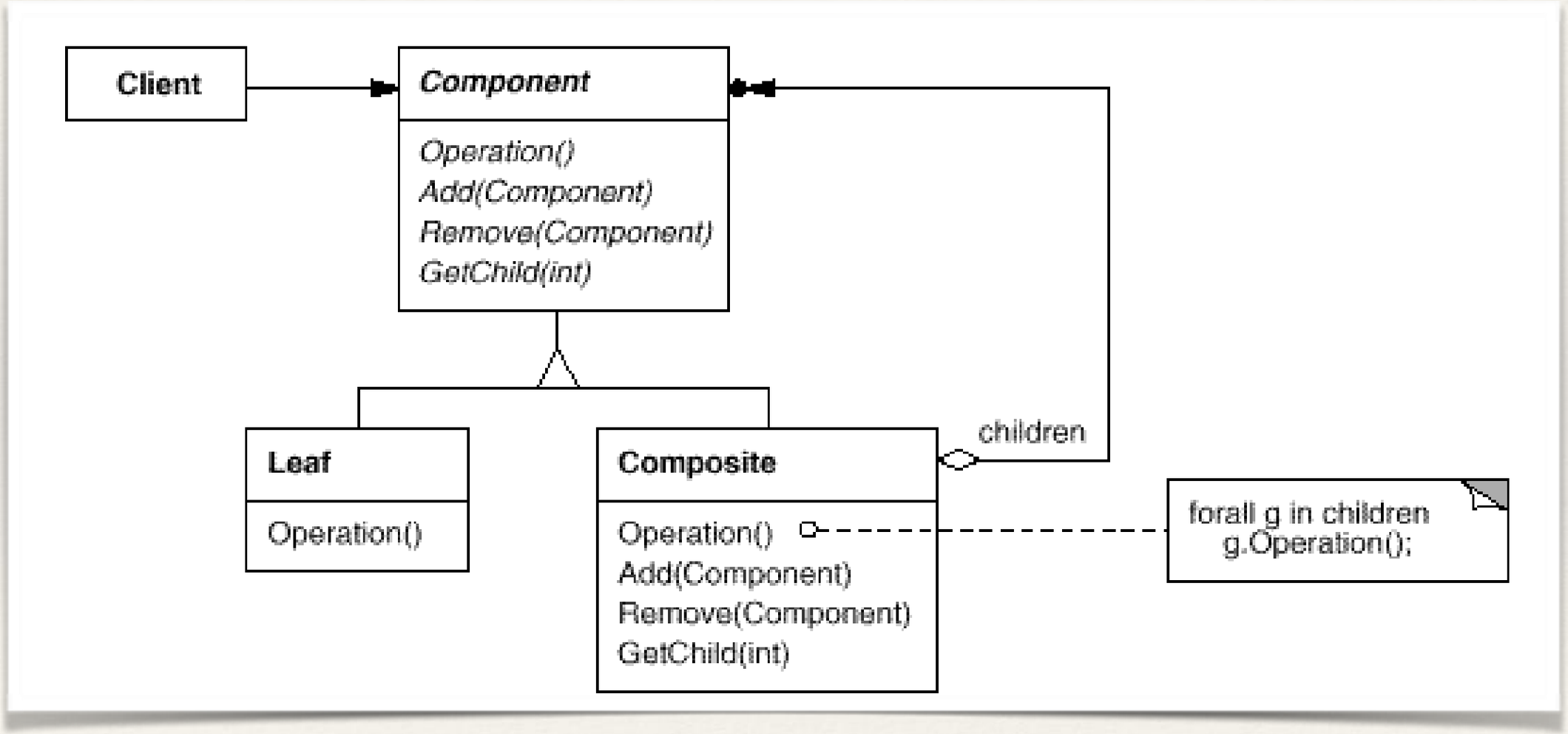
Solution Location: Solid-Master->Decorator->Steamread.cs

Scenario

a) How do we treat files and folders alike?
b) How can we handle shortcuts?



Composite pattern

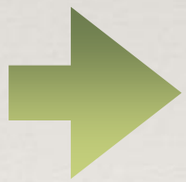


Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Composite pattern: Discussion

Compose objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

- There are many situations where a group of components form larger components
- Simplistic approach: Make container component contain primitive ones
  - Problem: Code has to treat container and primitive components differently



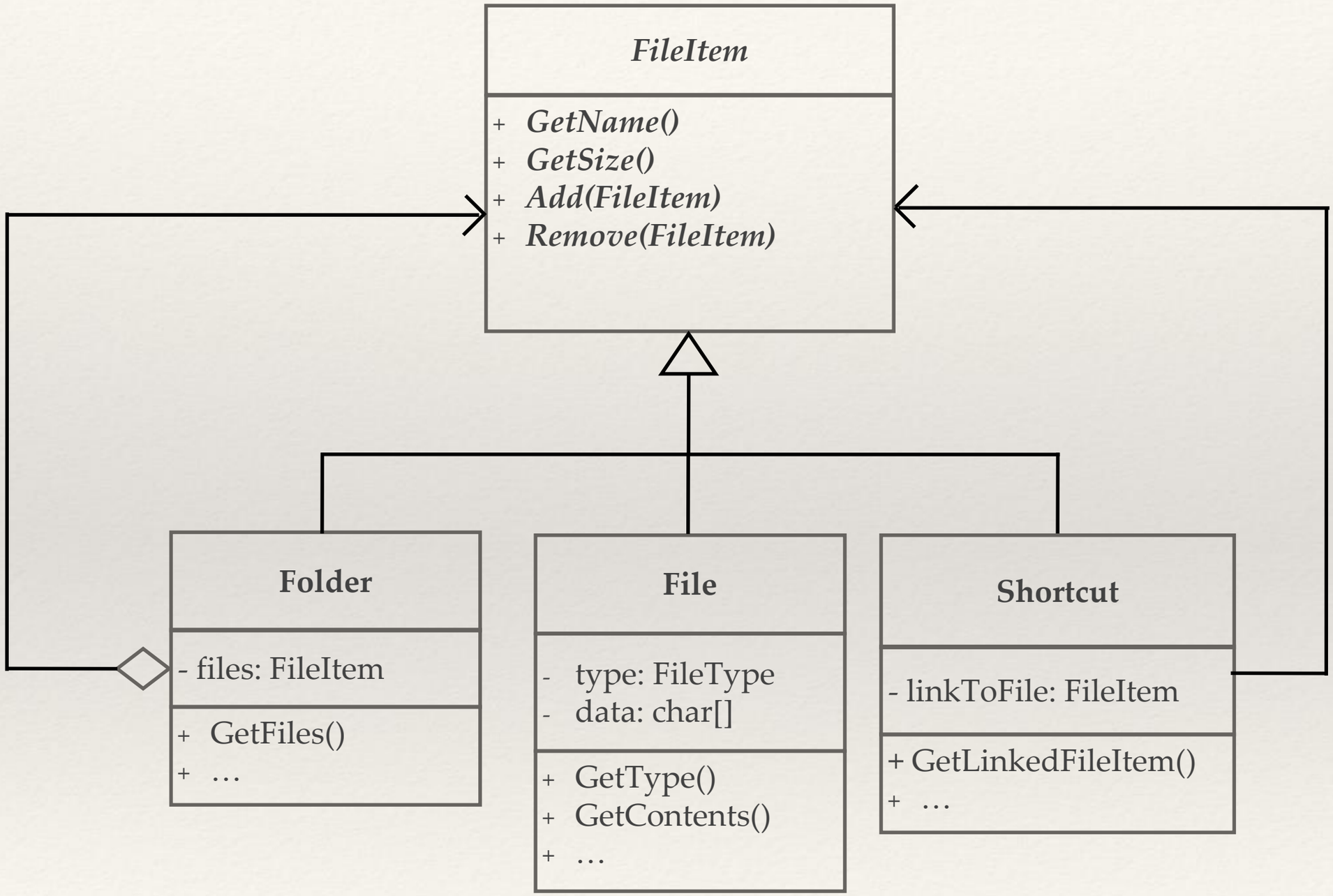
- Perform recursive composition of components
  - Clients don't have to treat container and primitive components differently

Decorator vs. Composite

Decorator and composite structure looks similar:
Decorator is a degenerate form of Composite!

Decorator	Composite
At max. one component	Can have many components
Adds responsibilities	Aggregates objects
Does not make sense to have methods such as Add(), Remove(), GetChid() etc.	Has methods such as Add(), Remove(), GetChild(), etc.

How about this solution?



Hands-on exercise

```
namespace Solid_Master.Composite
{
    internal abstract class FileItem
    {
        public virtual string Name { get; set; }

        public abstract long Size { get; }
    }

    internal class Folder ...

    internal class File ...

    internal class ShortCut ...

    public class FoldersMain...
}
```

Solution Location: Solid-Master->Composite->StreamRead.cs

What's that smell?

"Refused bequest" smell from ReadOnlyDictionary.cs

```
1   void IDictionary<TKey, TValue>.Add (TKey key, TValue value)
2   {
3       throw new NotSupportedException (SR.NotSupported_ReadOnlyCollection);
4   }
5
6   bool IDictionary<TKey, TValue>.Remove (TKey key)
7   {
8       throw new NotSupportedException (SR.NotSupported_ReadOnlyCollection);
9   }
10
11  TValue IDictionary<TKey, TValue>.this [TKey key] {
12  get
13  {
14      return _dictionary[key];
15  }
16  set
17  {
18      throw new NotSupportedException (SR.NotSupported_ReadOnlyCollection);
19  }
```

Liskov's Substitution Principle (LSP)

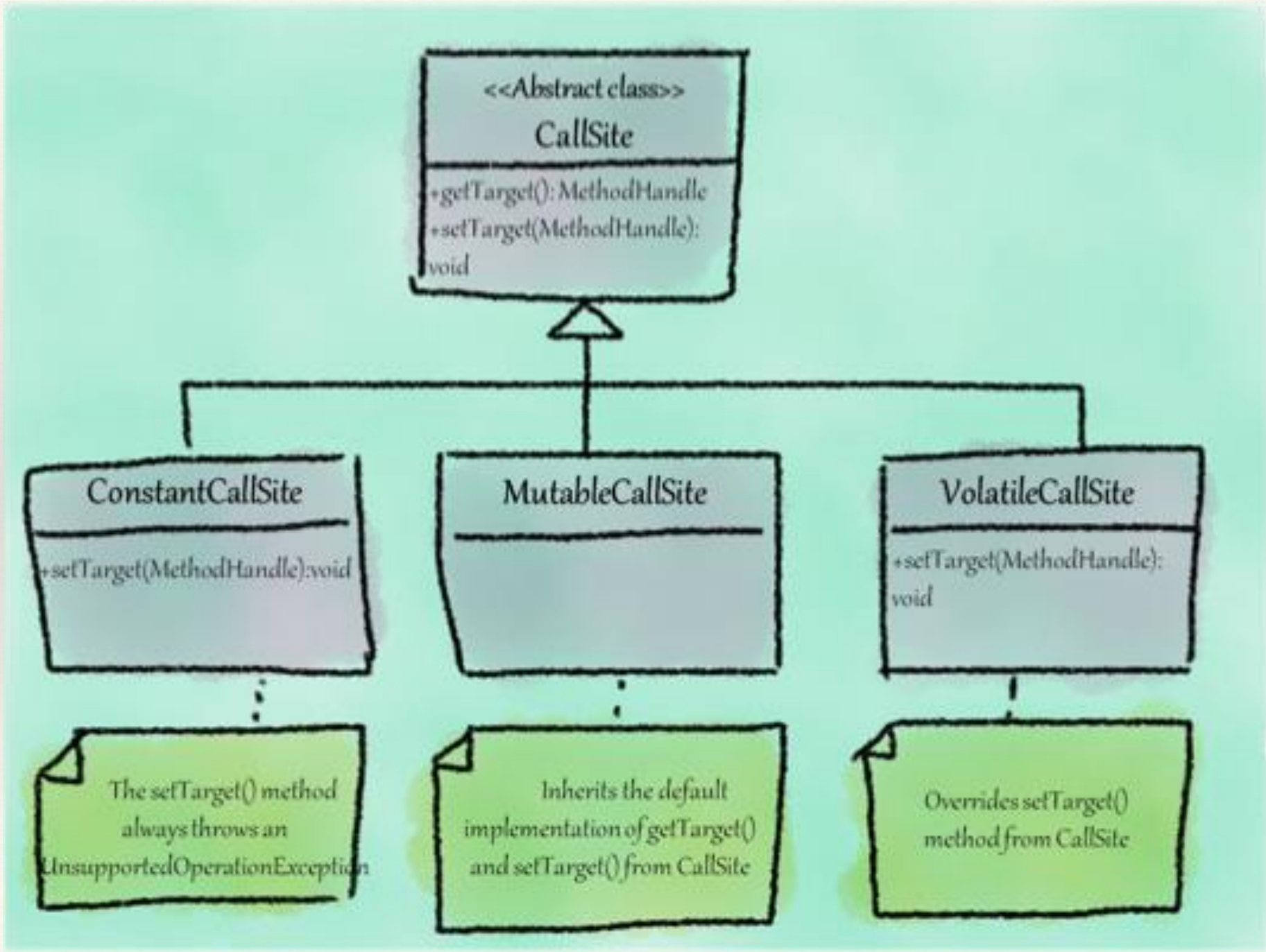
It should be possible to replace objects of supertype with objects of subtypes without altering the desired behavior of the program

Barbara Liskov

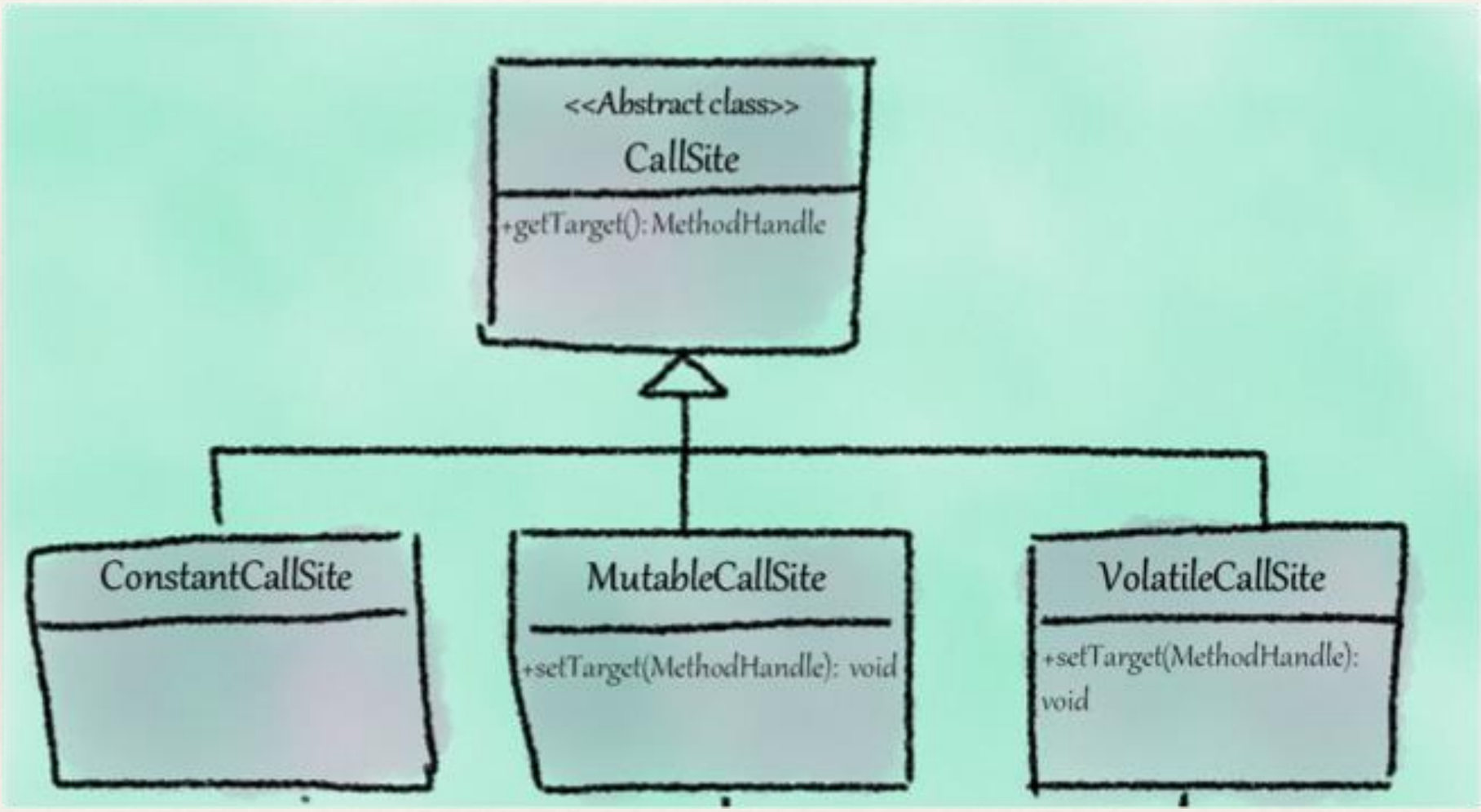
Refused bequest smell

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

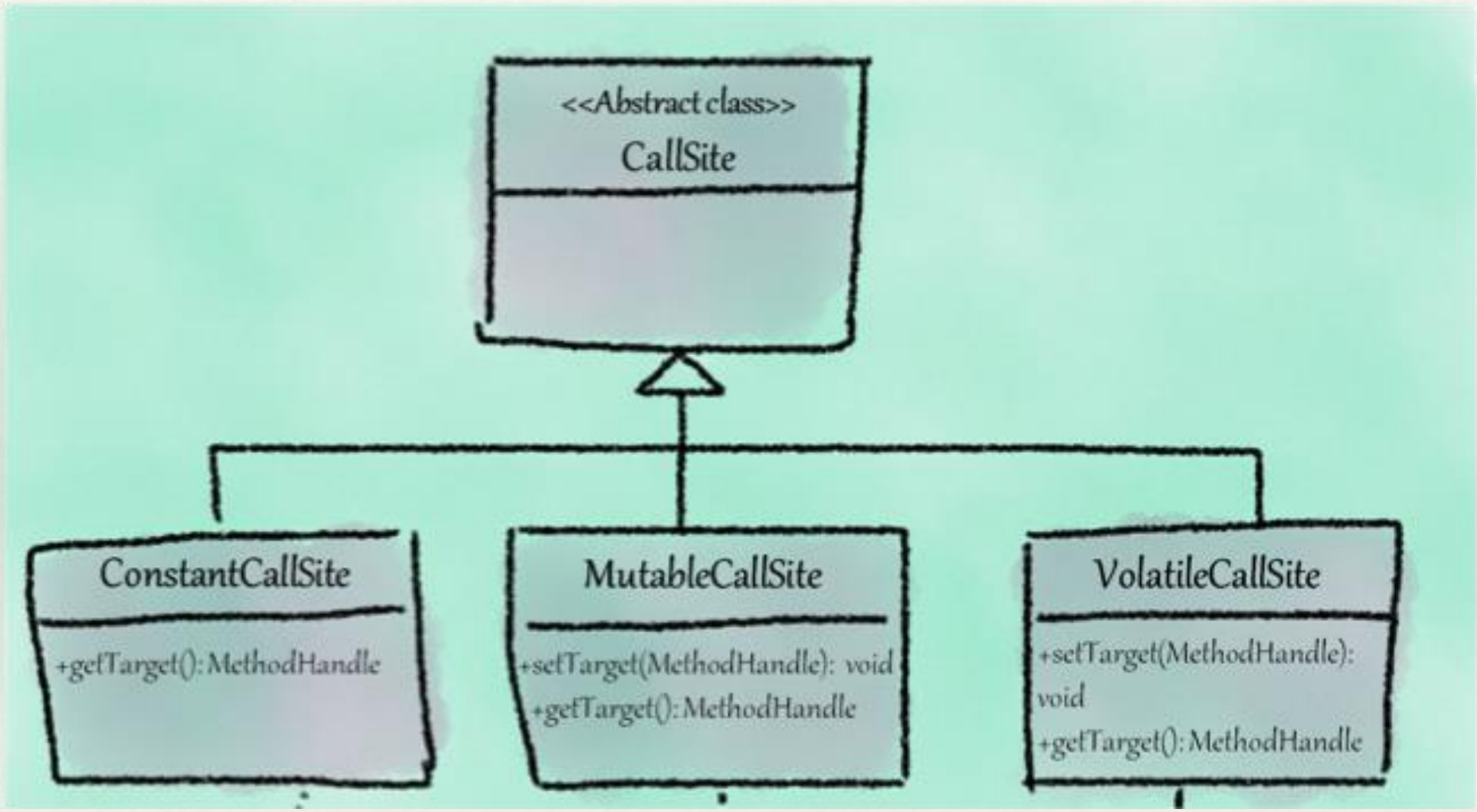
What's that smell?



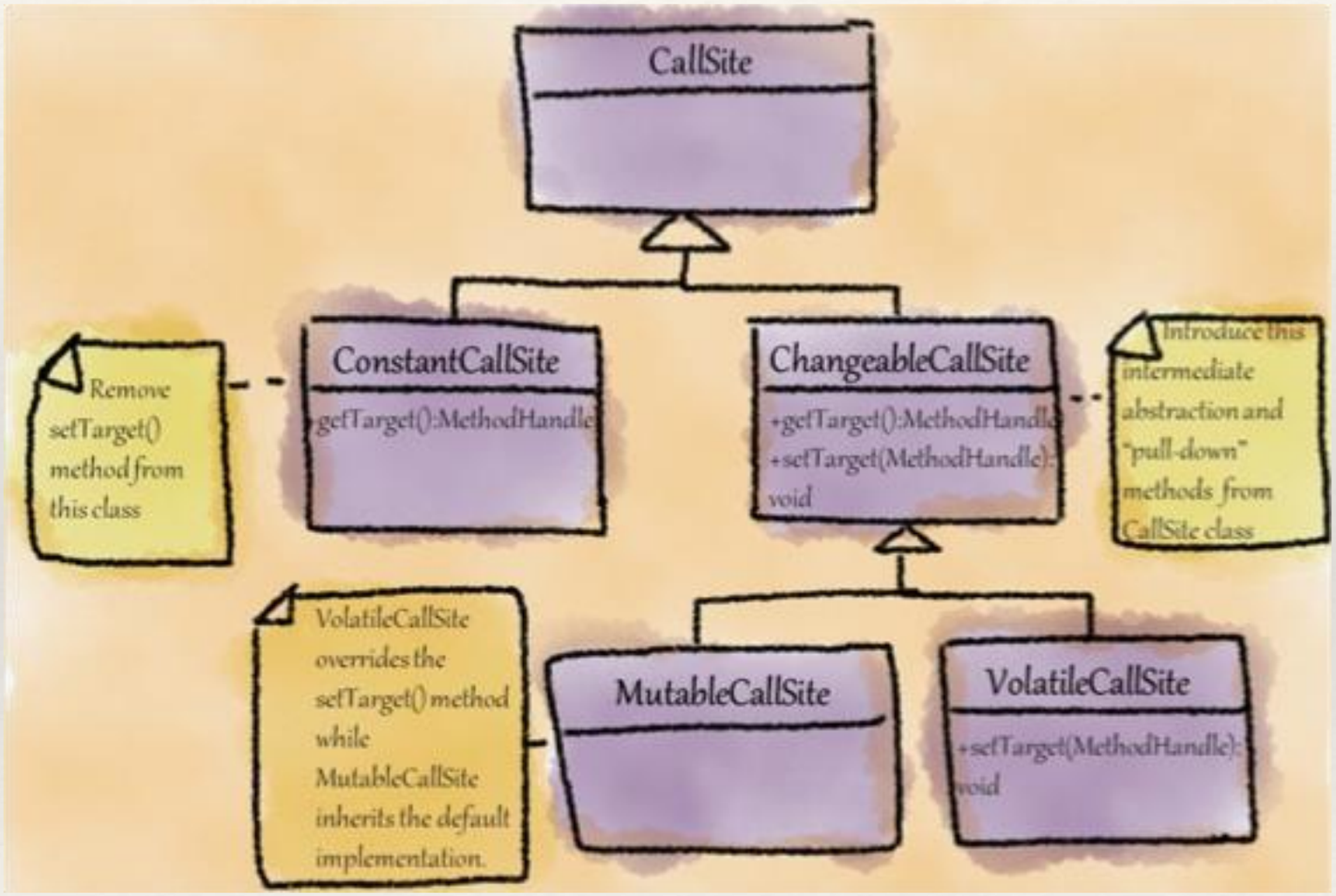
How about this refactoring?



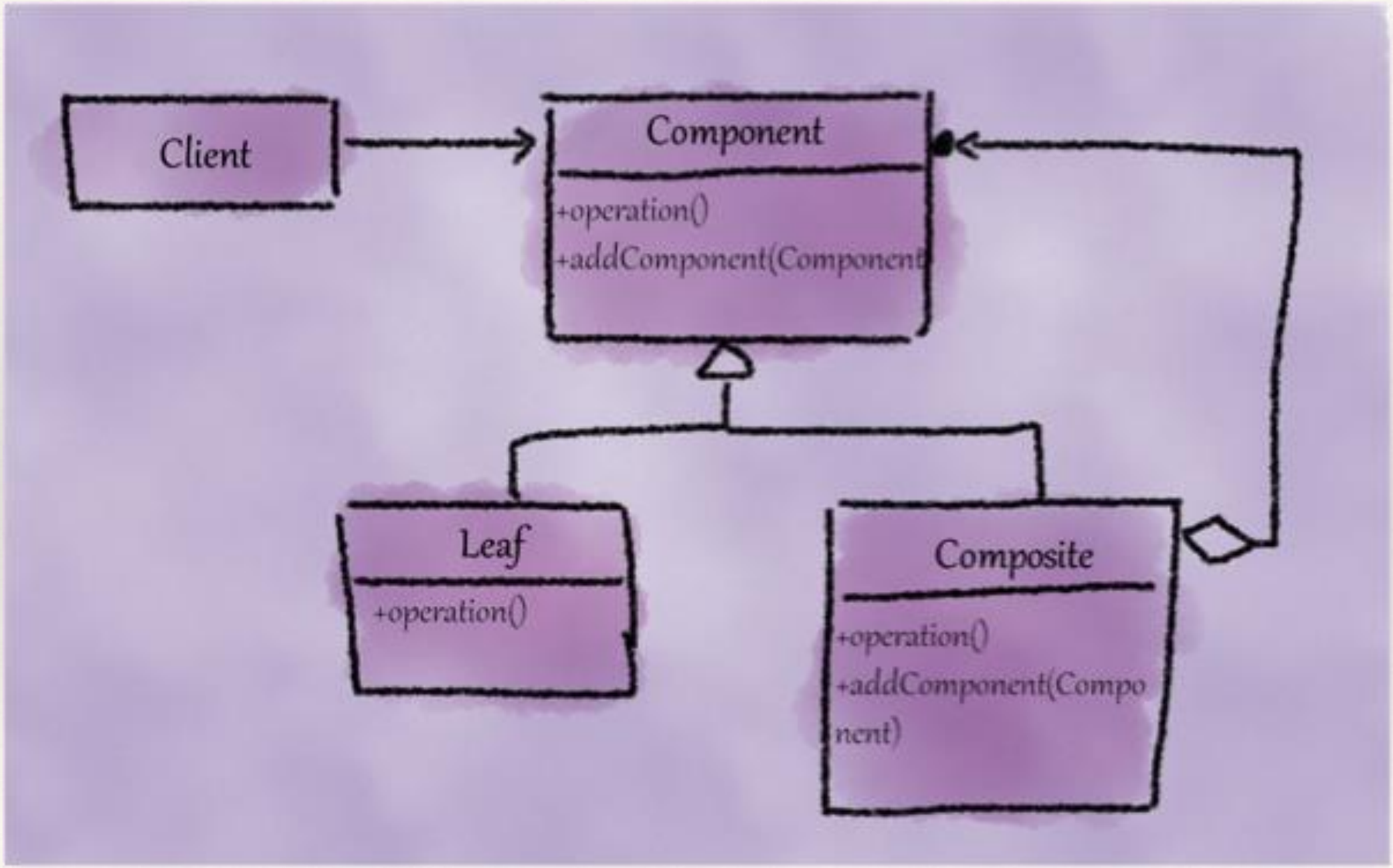
How about this refactoring?



Suggested refactoring

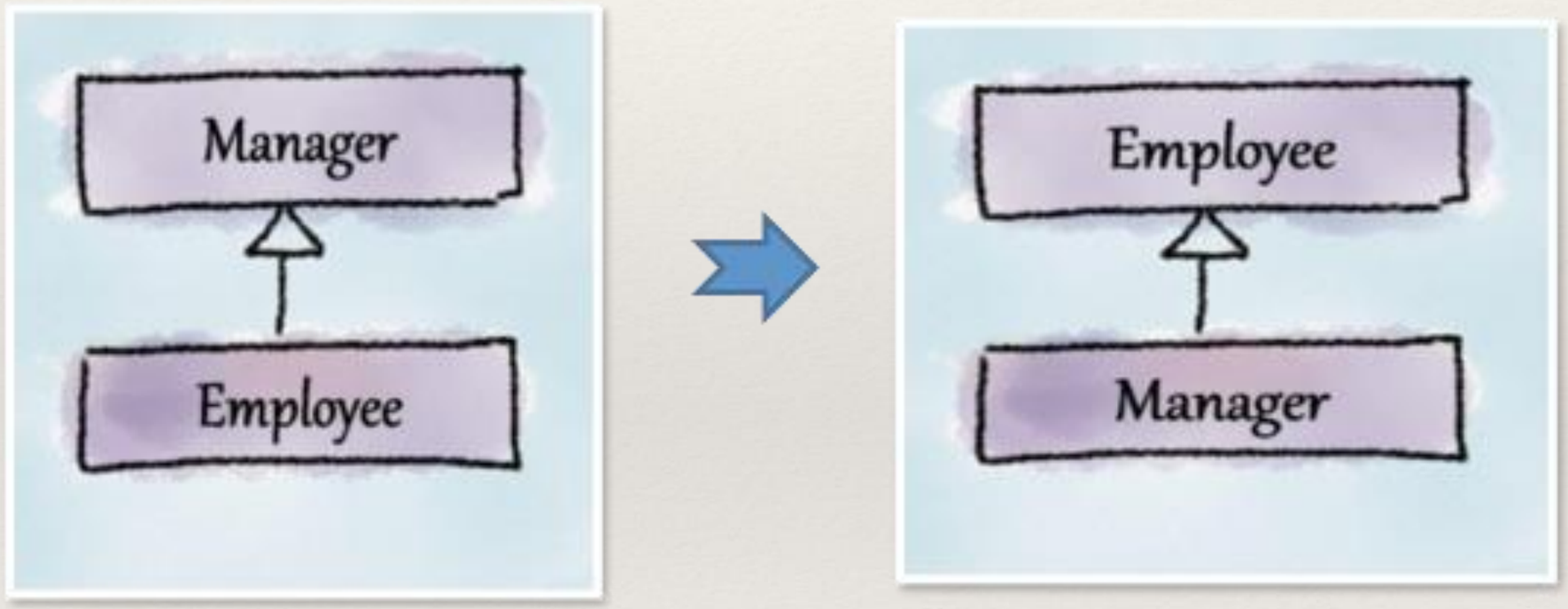


Practical considerations

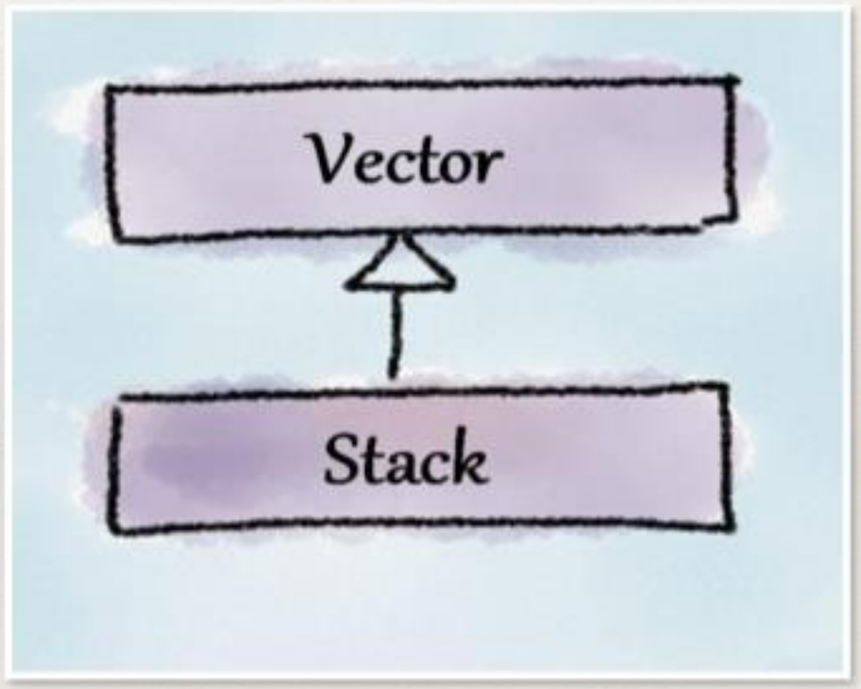


What's that smell?

Refactoring

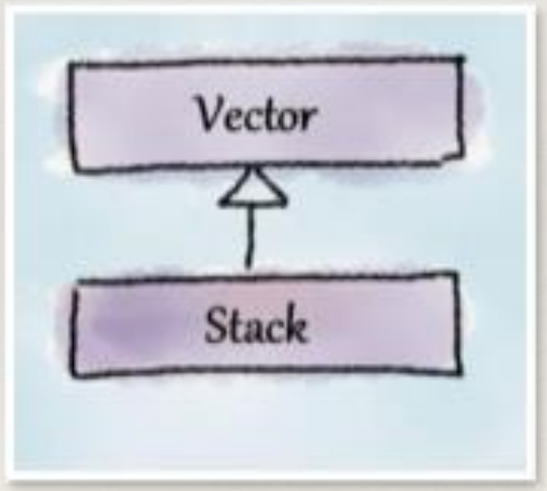


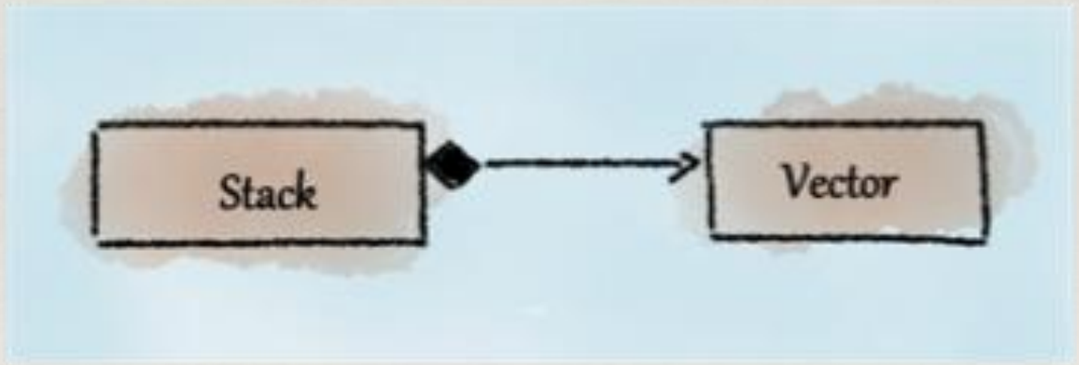
What’s that smell?



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.

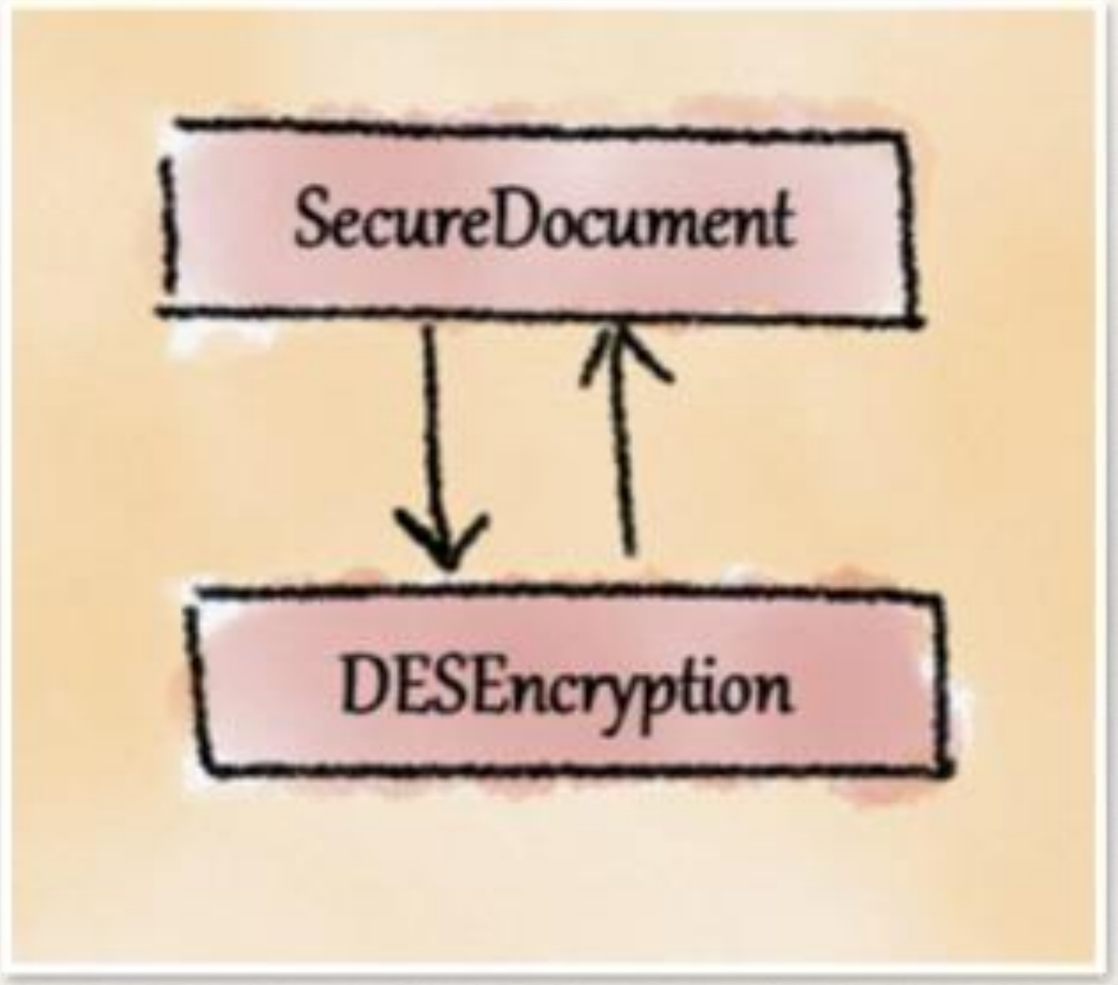
B. Abstractions should not depend upon details. Details should depend upon abstractions.

Hands-on exercise

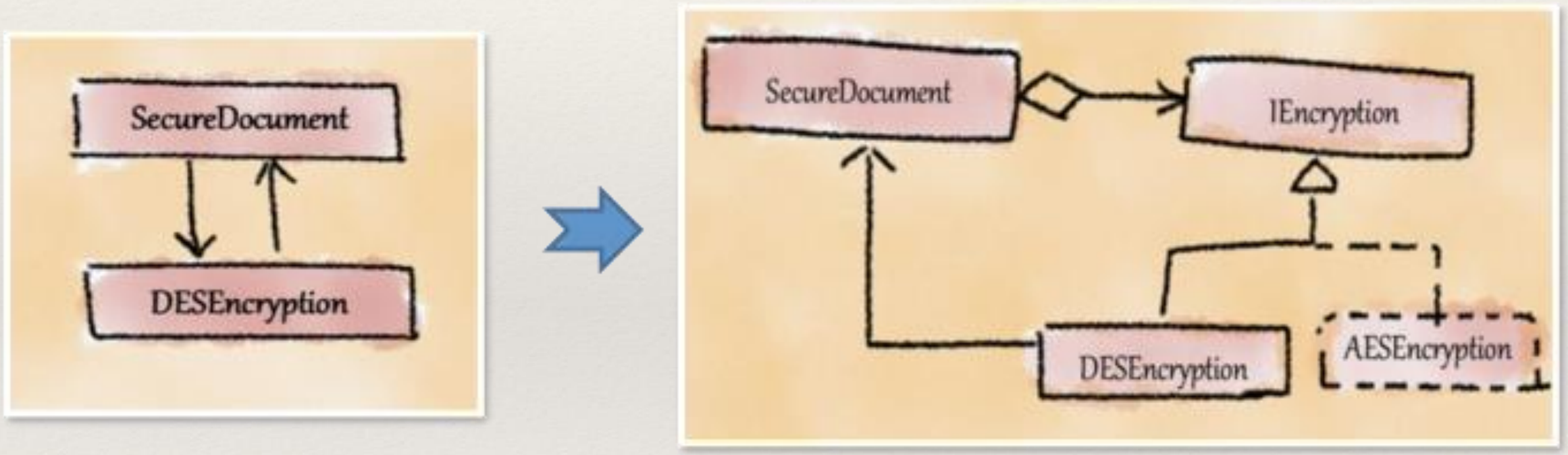
```
2 references
class EventLogWriter
{
    1 reference
    public void Write(string message)
    {
        //Write to event log here
    }
}

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

Factory method pattern: Discussion

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

- A class cannot anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates

- Delegate the responsibility to one of the several helper subclasses
- Also, localize the knowledge of which subclass is the delegate

Hands-on Exercise

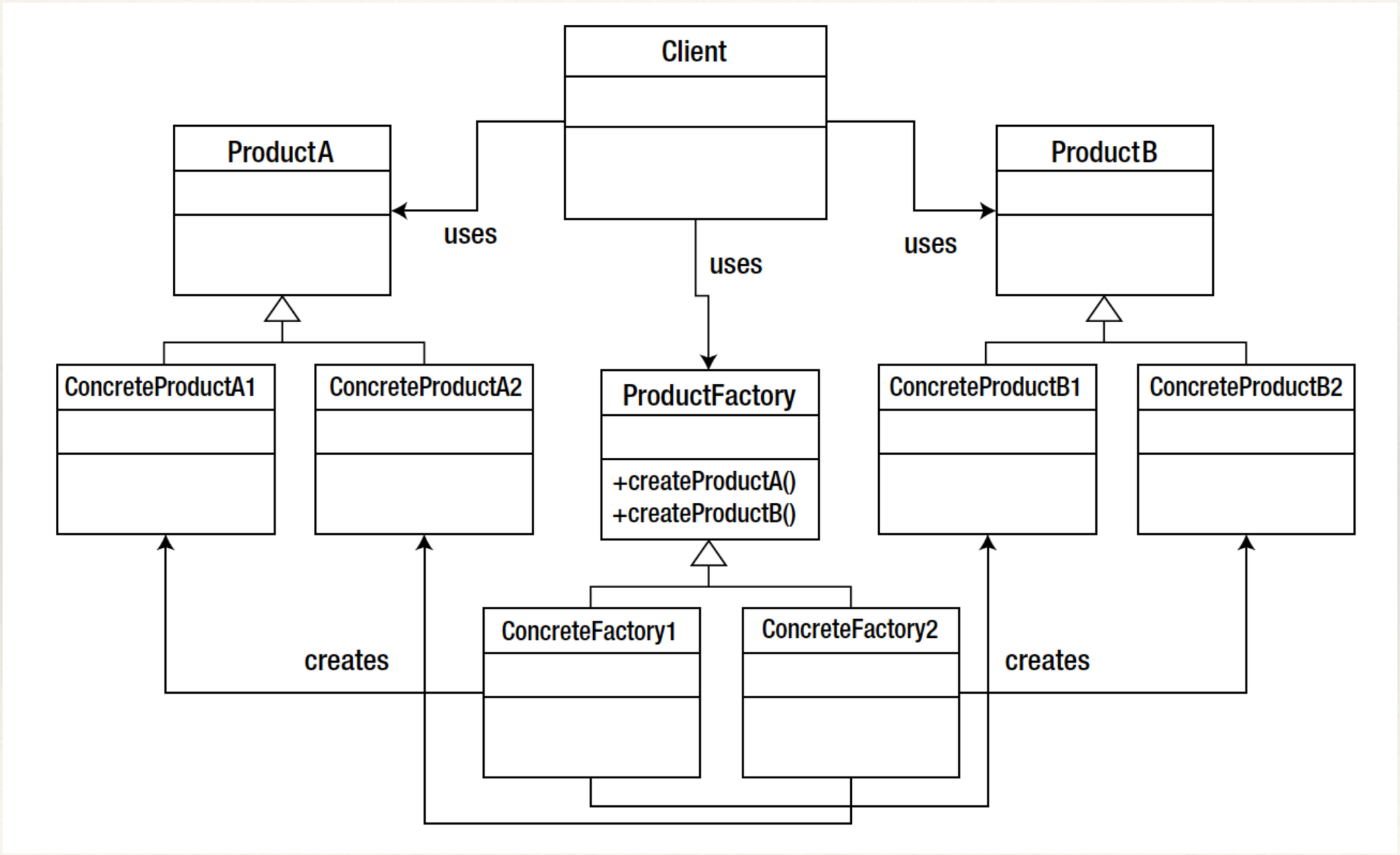
```
Uri ourUri = new Uri(url);

// Create a 'WebRequest' object with the specified url.
WebRequest myWebRequest = WebRequest.Create(url);

// Send the 'WebRequest' and wait for response.
WebResponse myWebResponse = myWebRequest.GetResponse();
```

WebRequest.Create - example of factory method
Initializes a new WebRequest instance for the specified URI scheme.
The pre-registered reserve types already registered include the following:

- http://
- https://
- ftp://
- file://

https://msdn.microsoft.com/en-us/library/bw00b1dc(v=vs.110).aspx

Abstract Factory pattern

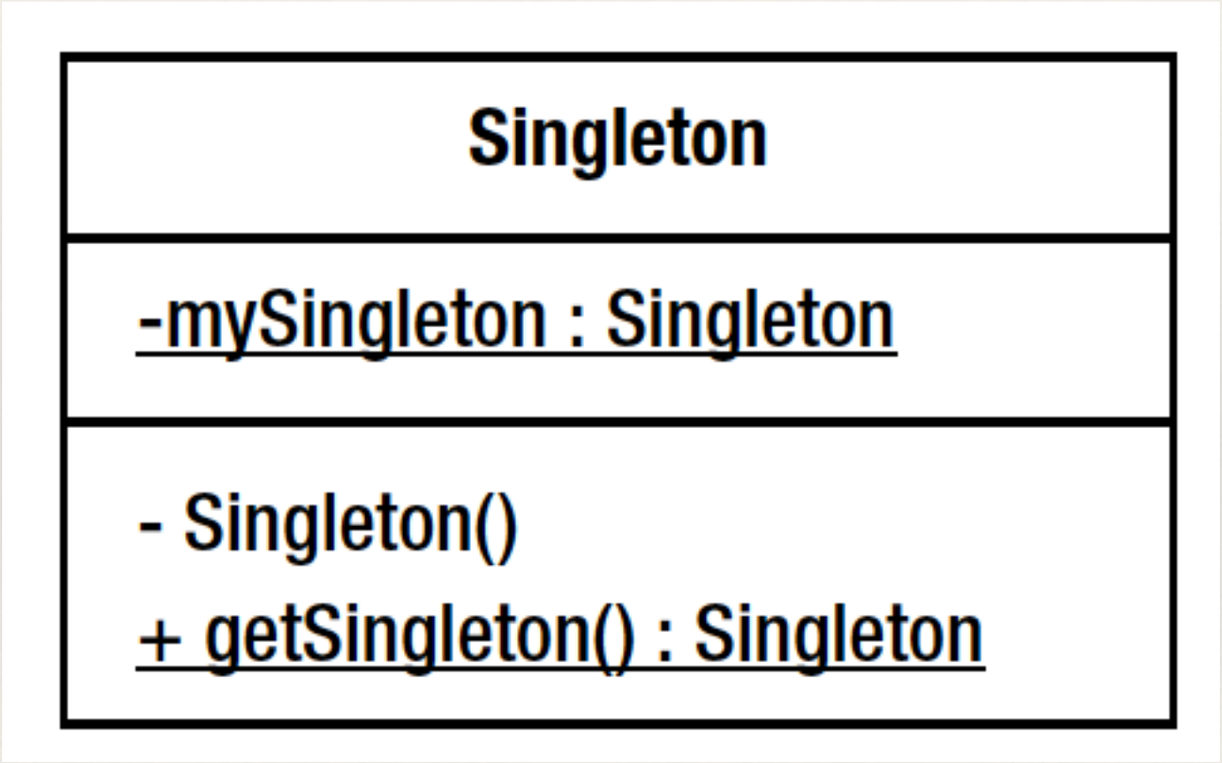


Abstract Factory in .NET framework

```
static DbConnection CreateDbConnection(
    string providerName, string connectionString)
{
    // Assume failure.
    DbConnection connection = null;

    // Create the DbProviderFactory and DbConnection.
    if (connectionString != null)
    {
        try
        {
            DbProviderFactory factory =
                DbProviderFactories.GetFactory(providerName);

            connection = factory.CreateConnection();
            connection.ConnectionString = connectionString;
        }
        catch (Exception ex)
        {
            // Set the connection to null if it was created.
            if (connection != null)
            {
                connection = null;
            }
            Console.WriteLine(ex.Message);
        }
    }
    // Return the connection.
    return connection;
}
```

https://msdn.microsoft.com/en-us/library/dd0w4a2z(v=vs.110).aspx

Singleton pattern



Singleton pattern

```
/// <summary>
///      The 'Singleton' class
/// </summary>
9 references
internal class LoadBalancer
{
    private static LoadBalancer _instance;

    // Lock synchronization object
    private static readonly object syncLock = new object();
    private readonly Random _random = new Random();
    private readonly List<string> _servers = new List<string>();

    // Constructor (protected)
    1 reference
    protected LoadBalancer()
    {
        // List of available servers
        _servers.Add("ServerI");
        _servers.Add("ServerII");

    }
```

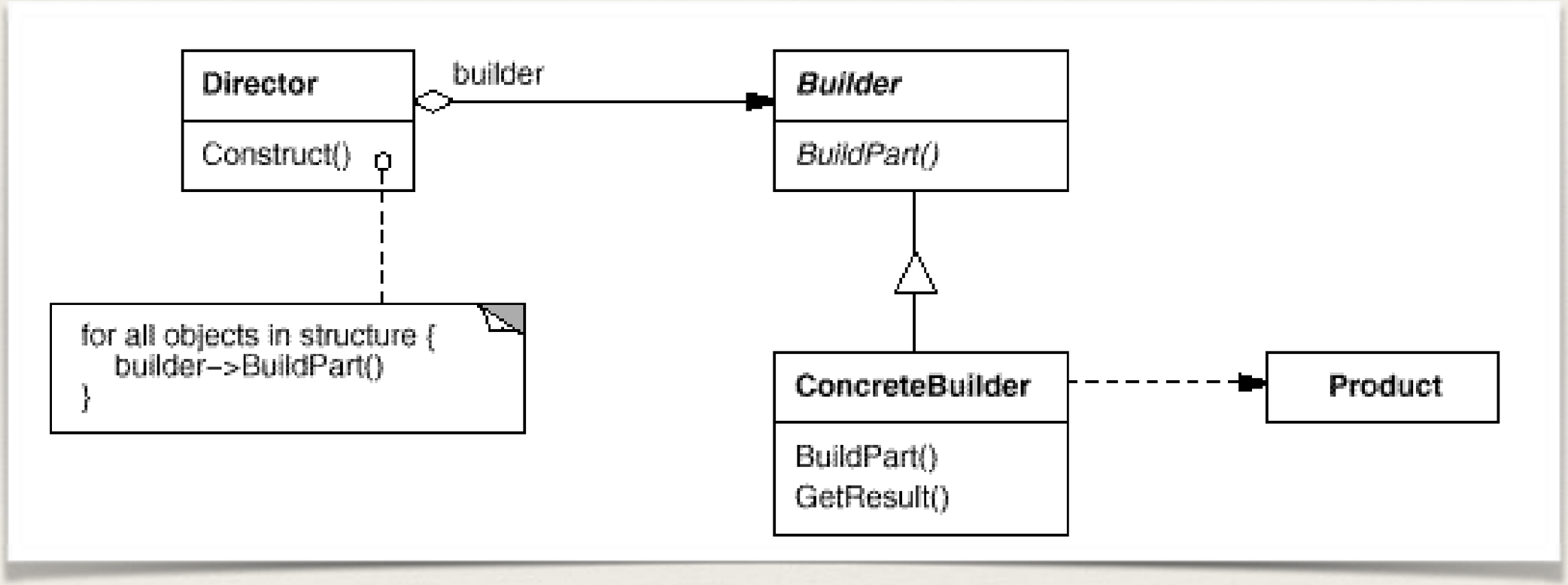
Contd..

```
// Simple, but effective random load balancer
1 reference
public string Server
{
    get
    {
        var r = _random.Next(_servers.Count);
        return _servers[r];
    }
}

5 references
public static LoadBalancer GetLoadBalancer()
{
    if (_instance == null)
        lock (syncLock)
        {
            if (_instance == null)
                _instance = new LoadBalancer();
        }

    return _instance;
```

Singleton pattern in .NET framework

- HttpContext.Current
- Server Activated Objects – Remoting
- Service Locators/Dependency Injection Containers (Unity)
- Other?

Builder pattern: structure



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

Builer pattern: Discussion

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- Creating or assembling a complex object can be tedious

- Make the algorithm for creating a complex object independent of parts that make up the object and how they are assembled
- The construction process allows different representations for the object that is constructed

Builders common for complex classes (1/2)

```
System.Data.SqlClient.SqlConnectionStringBuilder builder =
  new System.Data.SqlClient.SqlConnectionStringBuilder();
builder["Data Source"] = "(local)";
builder["integrated Security"] = true;
builder["Initial Catalog"] = "AdventureWorks;NewValue=Bad";
Console.WriteLine(builder.ConnectionString);
```

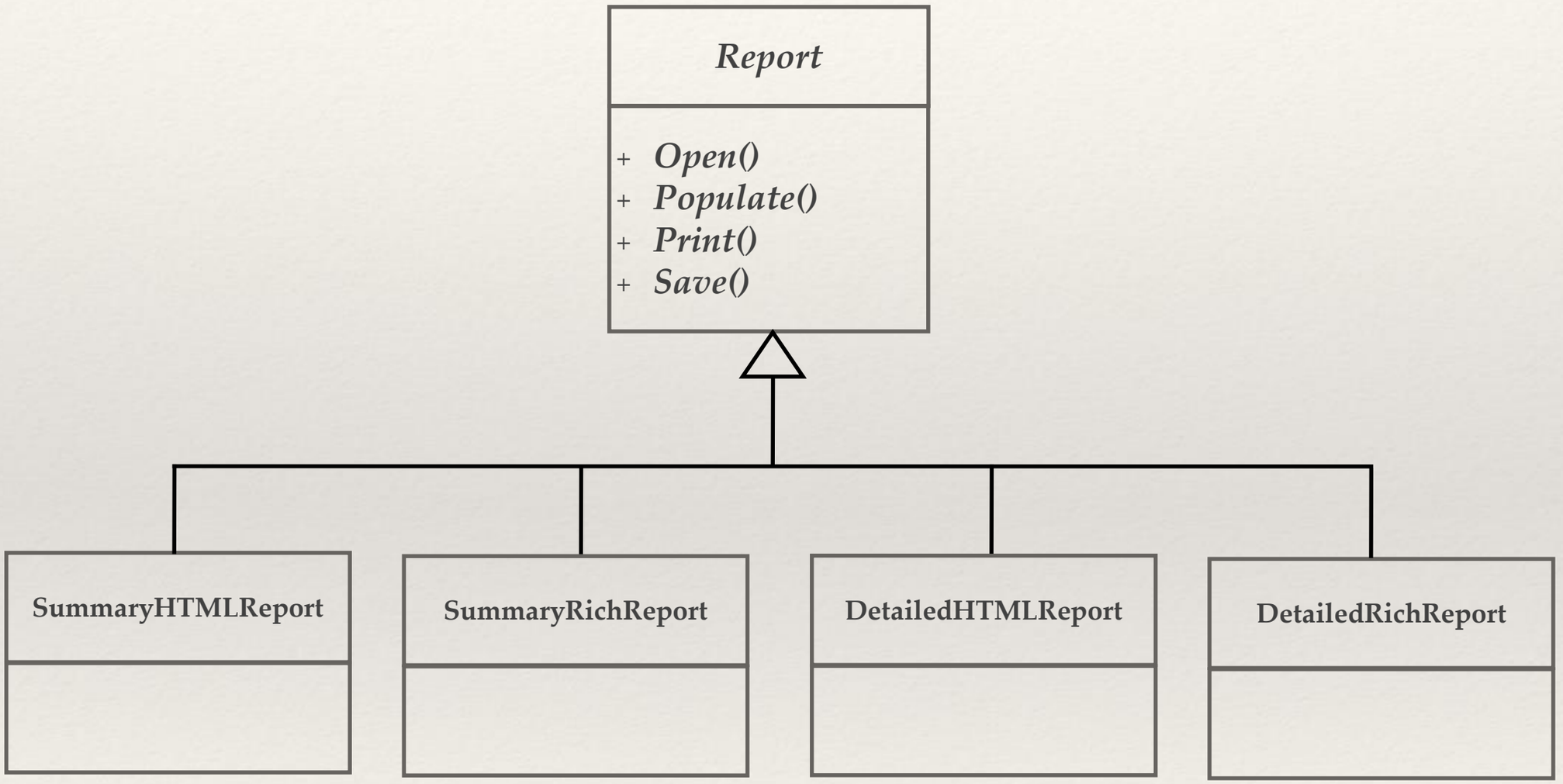
https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnectionstringbuilder(v=vs.110).aspx

Builders common for complex classes (2/2)

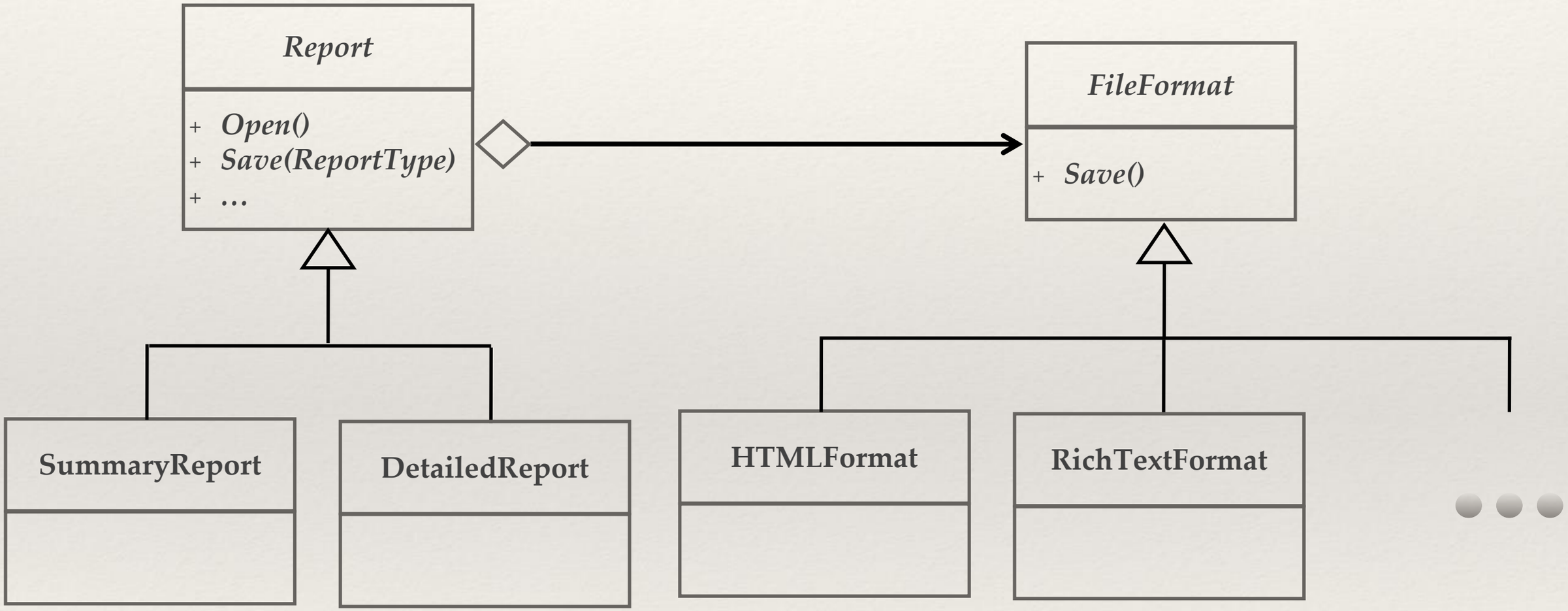
```
var uriBuilder = new UriBuilder();
uriBuilder.Scheme = "http";
uriBuilder.Host = "www.google.com";
uriBuilder.Query = "#q=uribuilder+c%23";
var uri = uriBuilder.ToString();
Console.WriteLine(uri);
```

https://msdn.microsoft.com/en-us/library/system.uribuilder(v=vs.110).aspx

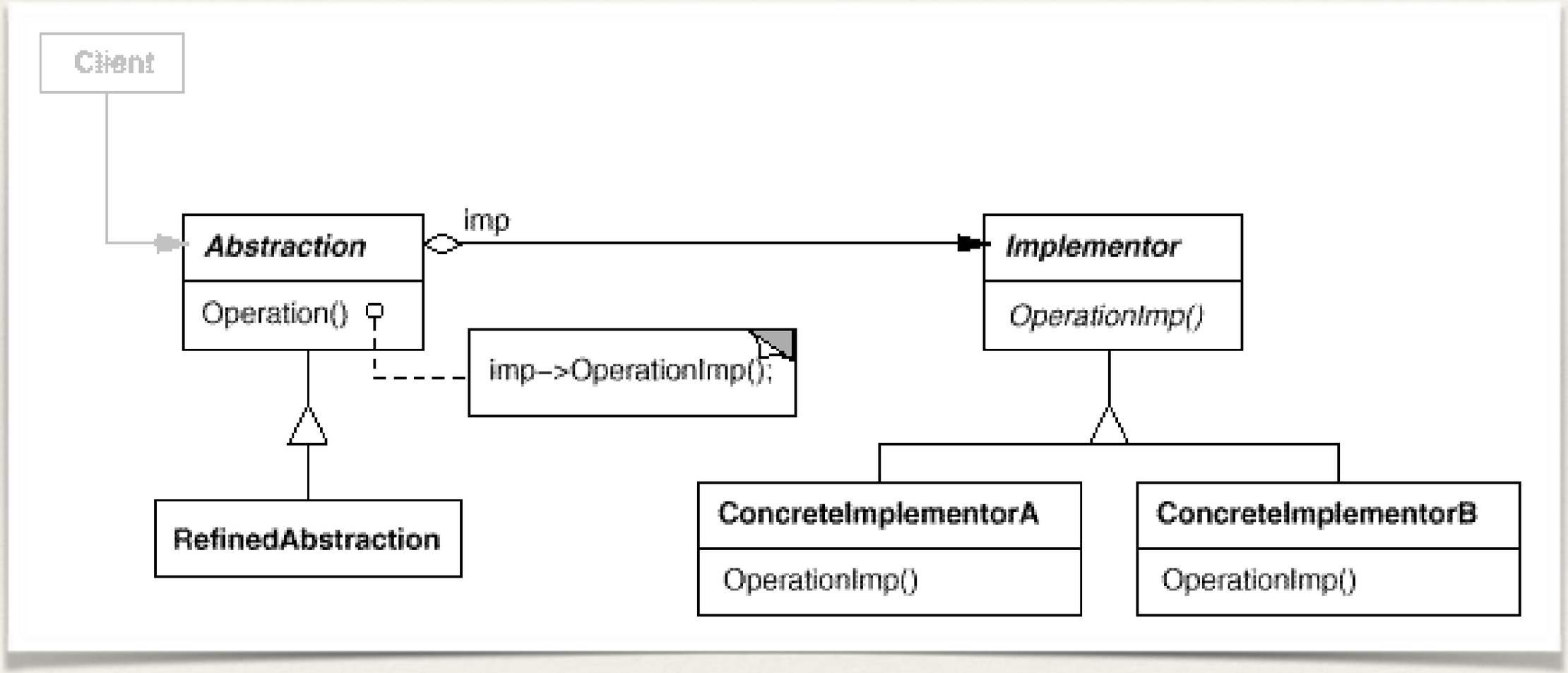
Scenario


Report
+ Open()
+ Populate()
+ Print()
+ Save()
SummaryHTMLReport
SummaryRichReport
DetailedHTMLReport
DetailedRichReport

How about this solution?

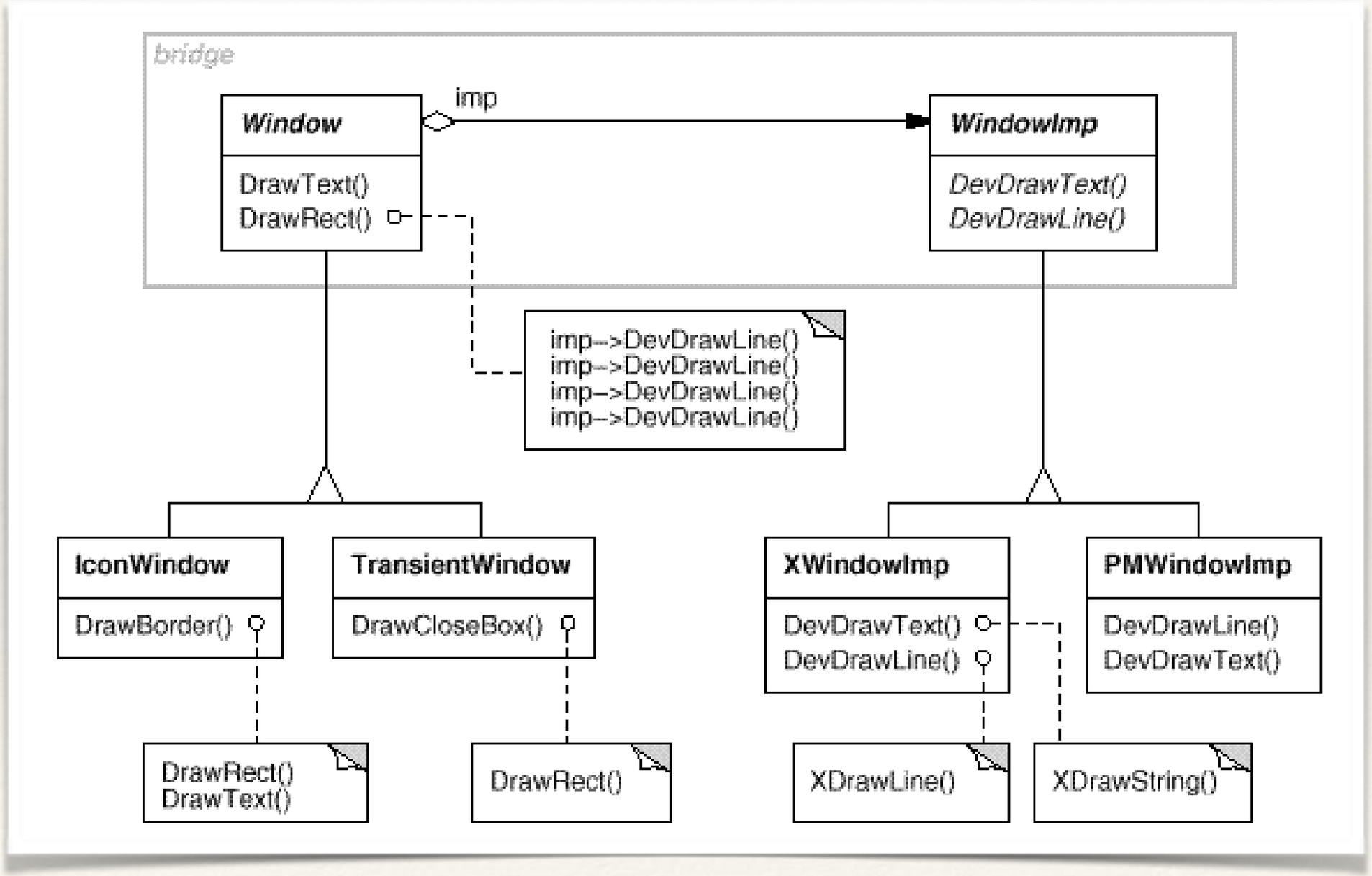


You’re right: Its Bridge pattern!



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

An example of Bridge pattern



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

Bridge pattern: Discussion

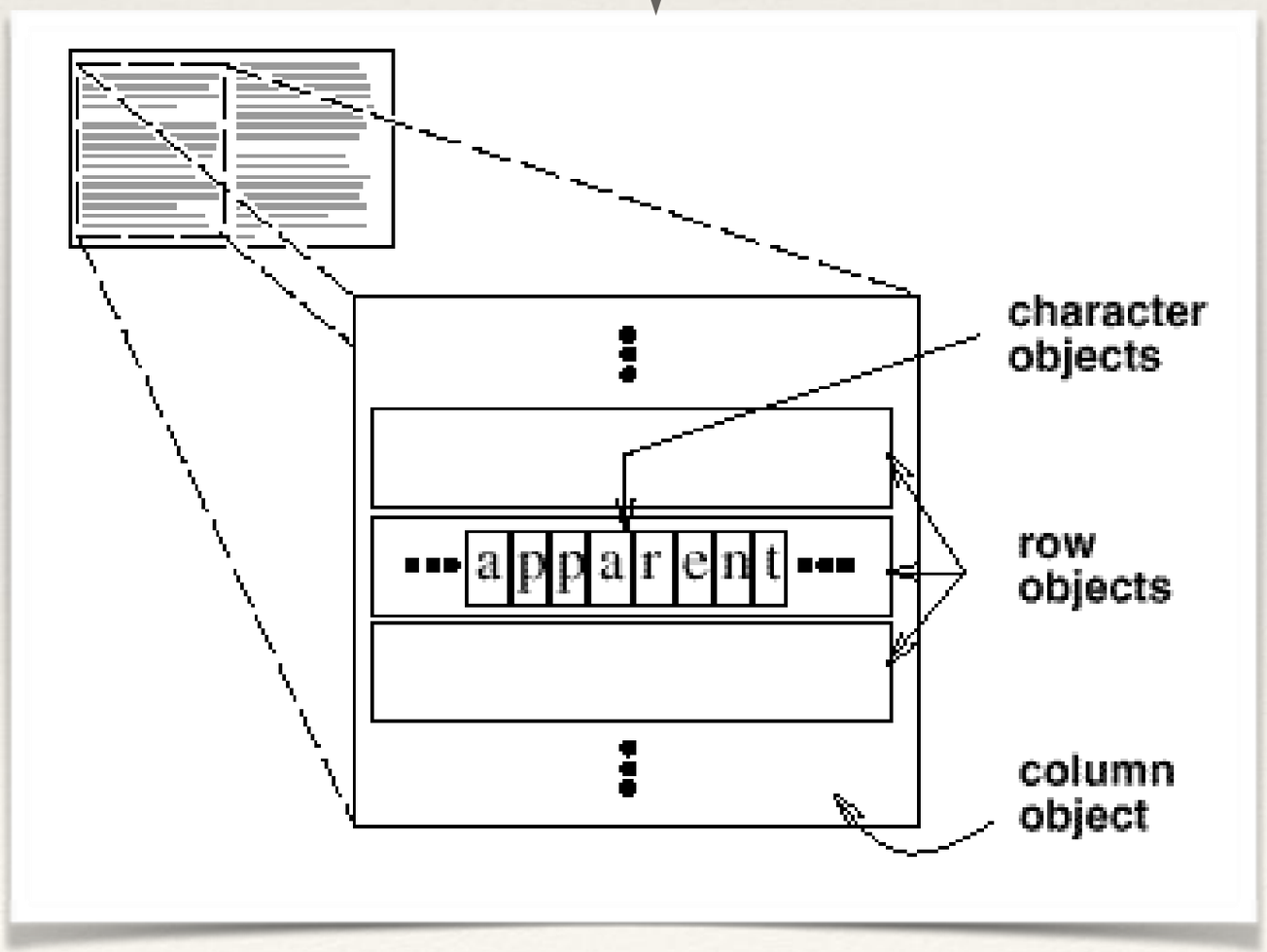
Decouples an abstraction from its implementation so that the two can vary independently

- An abstraction can be designed as an interface with one or more concrete implementers.
- When subclassing the hierarchy, it could lead to an exponential number of subclasses.
- And since both the interface and its implementation are closely tied together, they cannot be independently varied without affecting each other

- Put both the interfaces and the implementations into separate class hierarchies.
- The Abstraction maintains an object reference of the Implementer type.
- A client application can choose a desired abstraction type from the Abstraction class hierarchy.
- The abstraction object can then be configured with an instance of an appropriate implementer from the Implementer class hierarchy

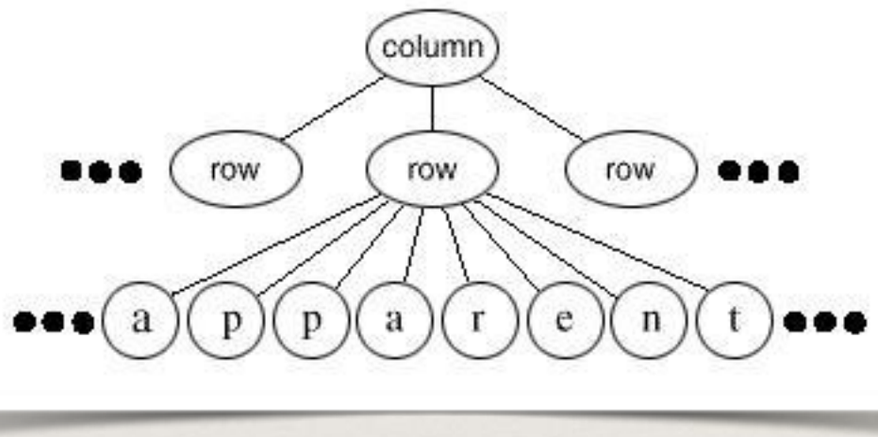
Scenario

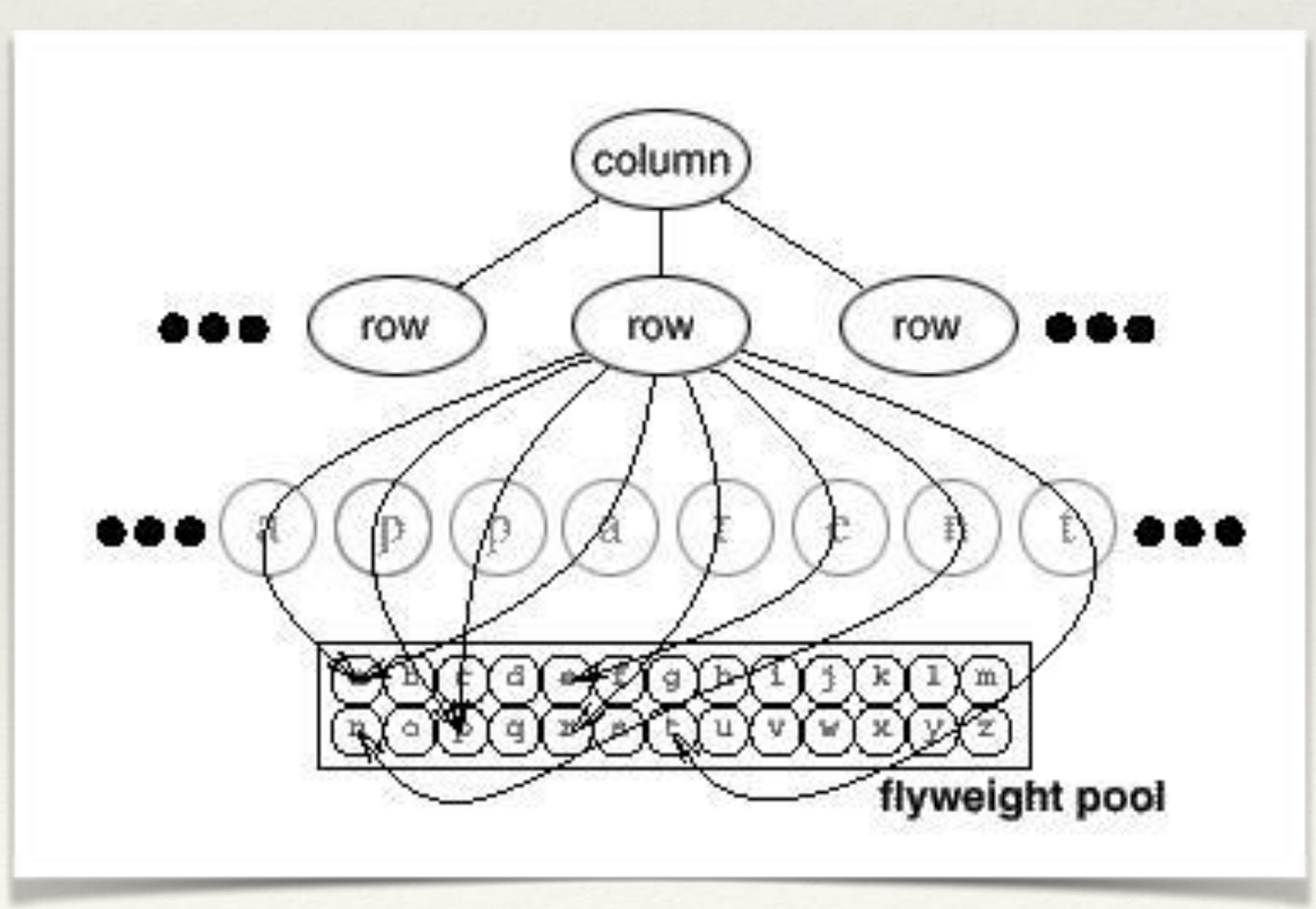
How will you design to share the characters to save space?



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

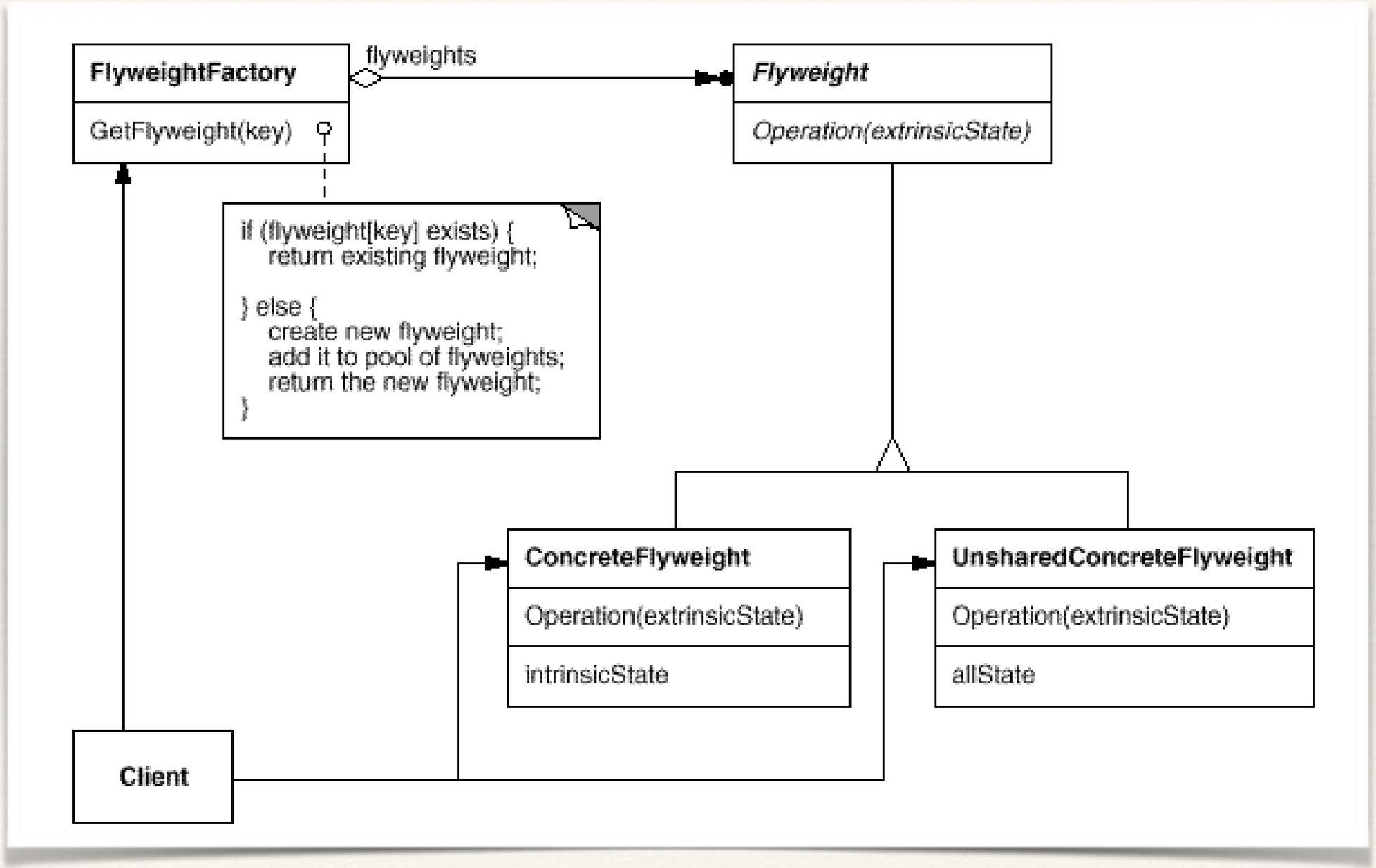
Flyweight as a solution





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

Flyweight pattern: structure



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

Flyweight pattern: Discussion

Use sharing to support large numbers of fine-grained objects efficiently

- When an application uses a large number of small objects, it can be expensive
- How to share objects at granular level without prohibitive cost?

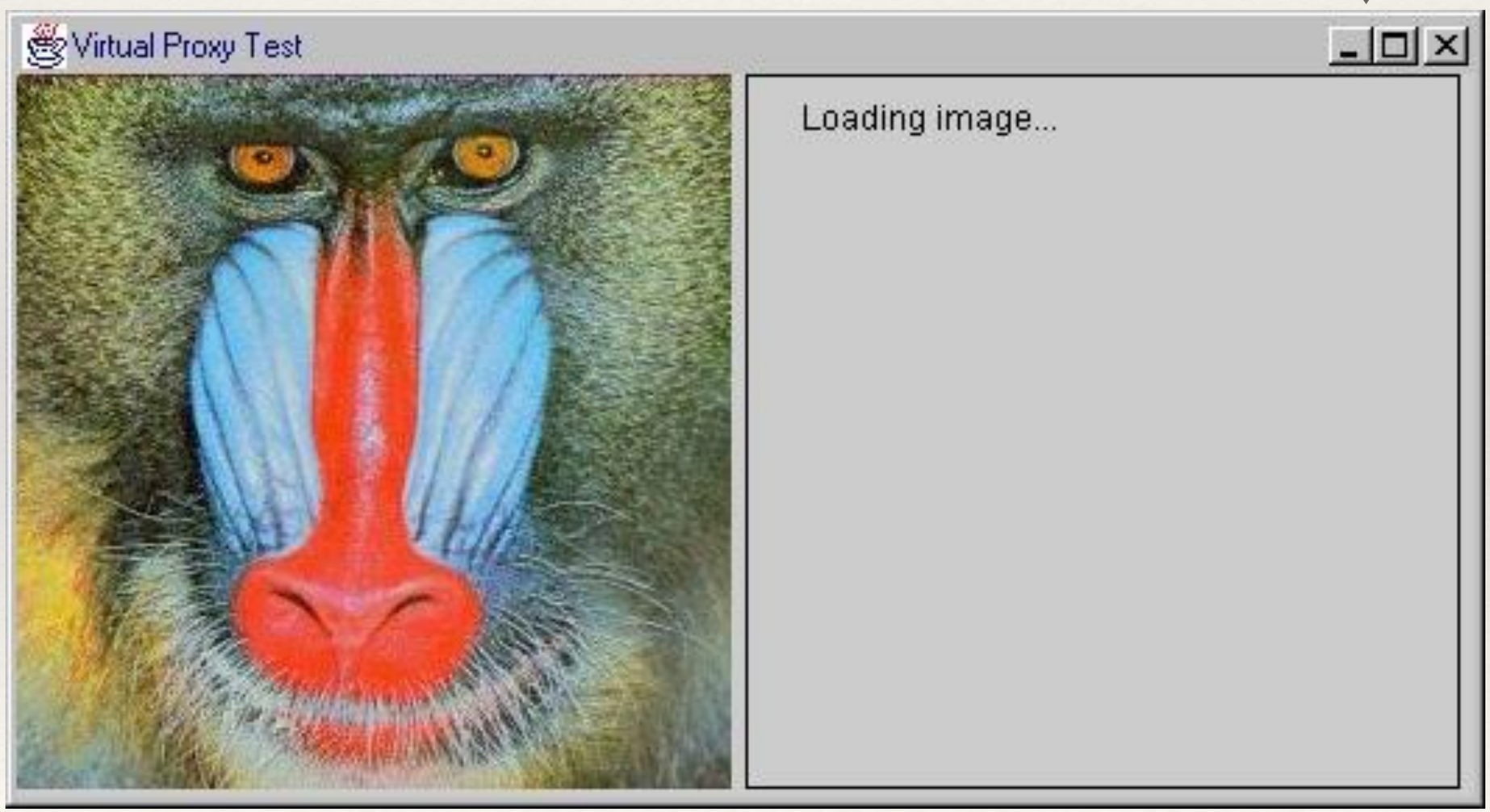
- When it is possible to share objects (i.e., objects don't depend on identity)
  - When object's value remain the same irrespective of the contexts - they can be shared
- Share the commonly used objects in a pool

Flyweight in .NET

- String Interning – all literal strings are shared
- ThreadPool – is it Flyweight?

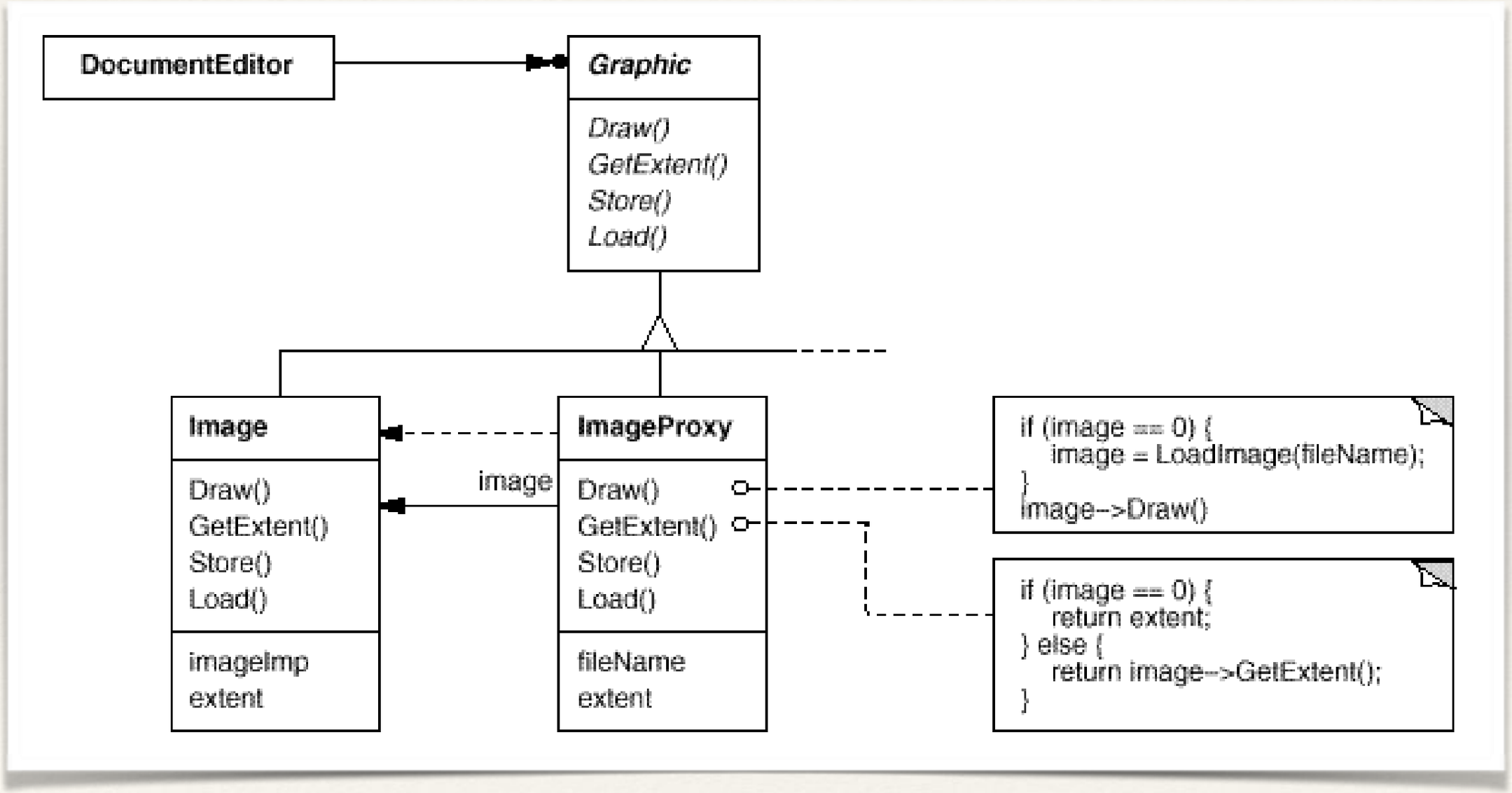
Scenario

How to load objects like large images efficiently "on-demand?"



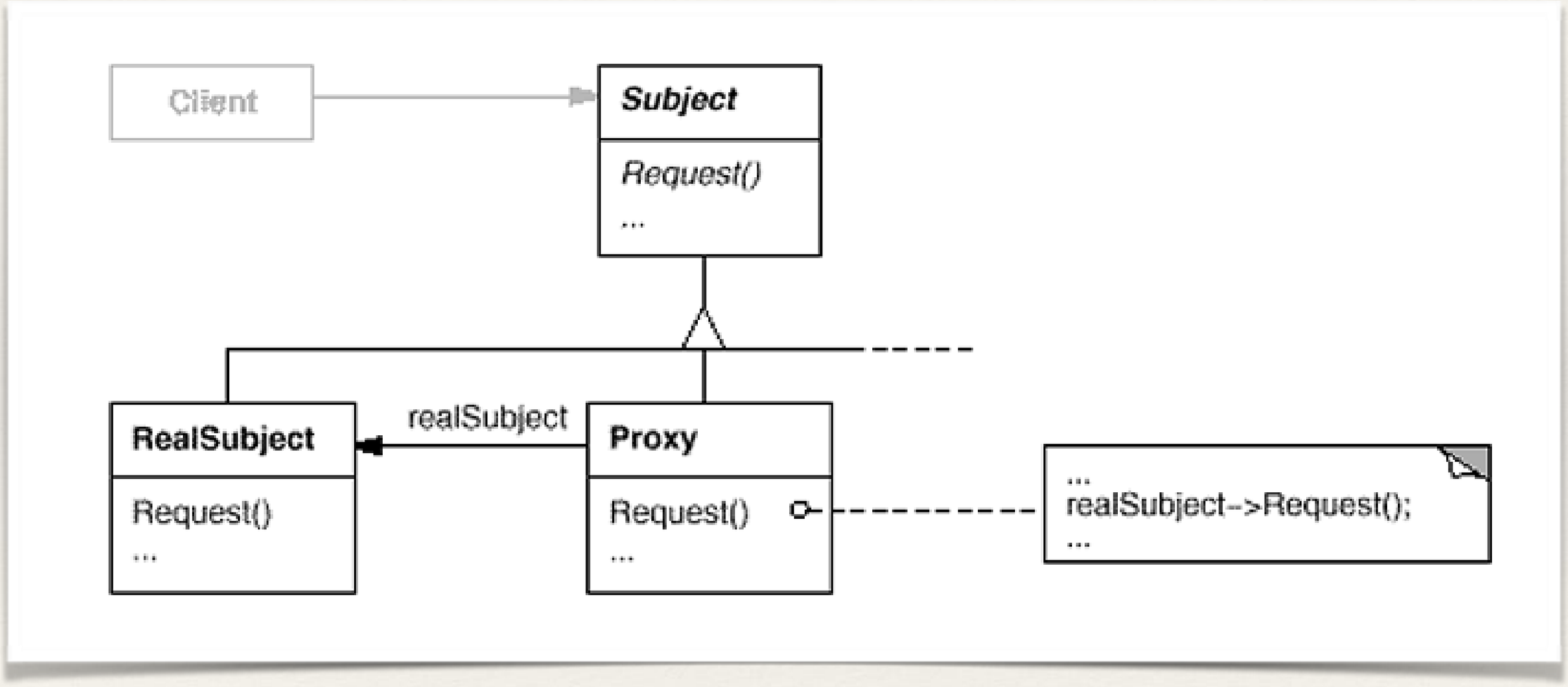
Source: http://www.javaworld.com/article/2074068/learn-java/take-control-with-the-proxy-design-pattern.html

Proxy pattern: example



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

Proxy pattern: structure

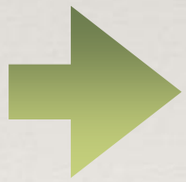


Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Proxy pattern: Discussion

Provide a surrogate or placeholder for another object to control access to it.

- How to defect the cost of full creation and initialization until we really need it?



- When it is possible to share objects
  - When object's value remain the same irrespective of the contexts - they can be shared
- Share the commonly used objects in a pool

Proxy in .NET Fx

Service References

- WCF (ClientBase<T>)
- SOAP

Scenario

```
Class Plus : Expr {
        private Expr left, right;
        public Plus(Expr arg1, Expr arg2) {
                left = arg1;
                right = arg2;
        }
        public void genCode() {
                left.genCode();
                right.genCode();
                if(t == Target.JVM) {
                        System.out.println("iadd");
                }
                else { // DOTNET
                        System.out.println("add");
                }
        }
}
```

How to separate:

a) code generation logic from node types?
b) how to support different target types?

Group exercise -

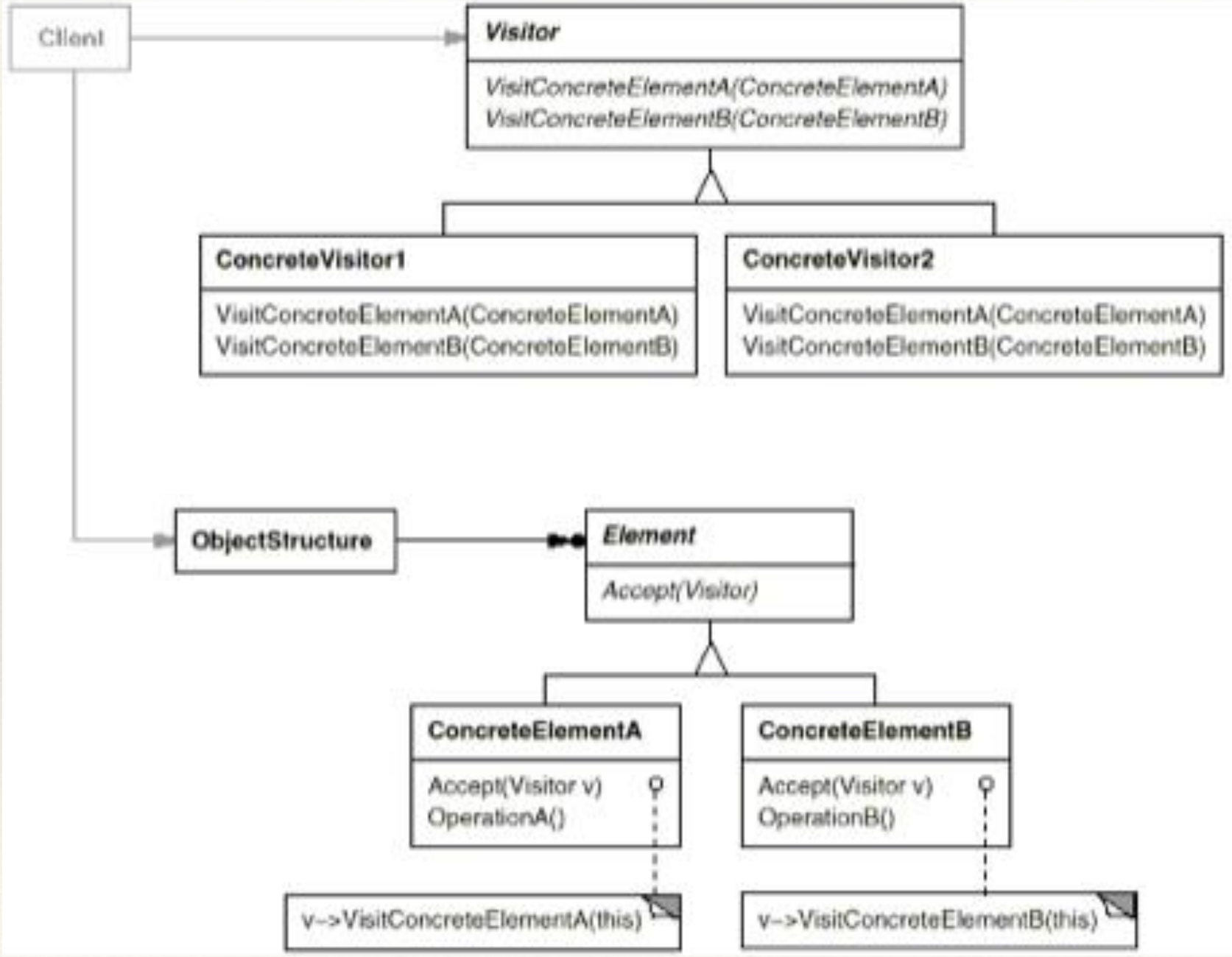
Understanding visitor pattern

A solution using Visitor pattern

```
namespace Solid_Master.Visitor.Sample2
{
    internal abstract class Expr...

    internal class Constant ...

    internal class Plus ...

    internal class Sub ...

    internal abstract class Visitor...

    internal class JvmVisitor ...

    internal class DotnetVisitor ...

    internal class ExprEvalVisitor...
}
```

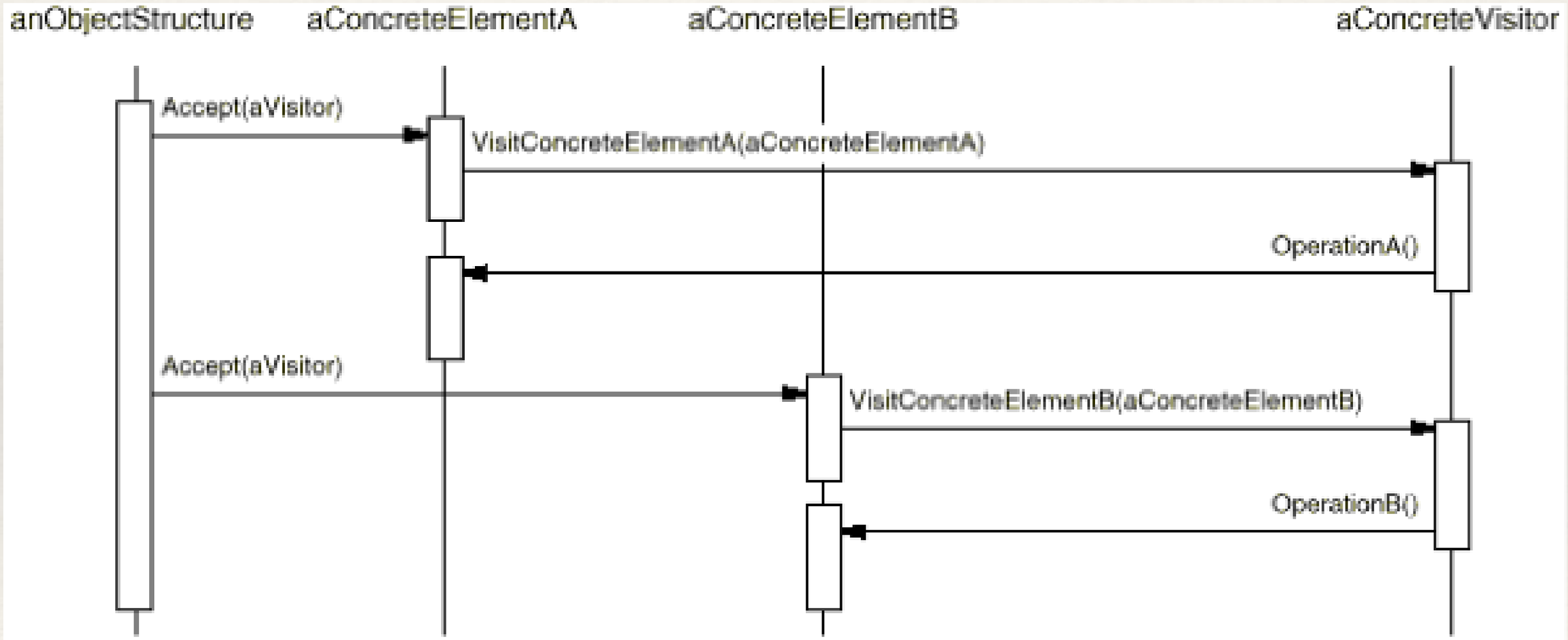
Solution Location: Solid-Master->Visitor->ExprEvalVisitor.cs

Visitor pattern: structure



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

Visitor pattern: call sequence



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

Visitor pattern: Discussion

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operations

- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations

- Create two class hierarchies:
  - One for the elements being operated on
  - One for the visitors that define operations on the elements

Visitor pattern: .NET example

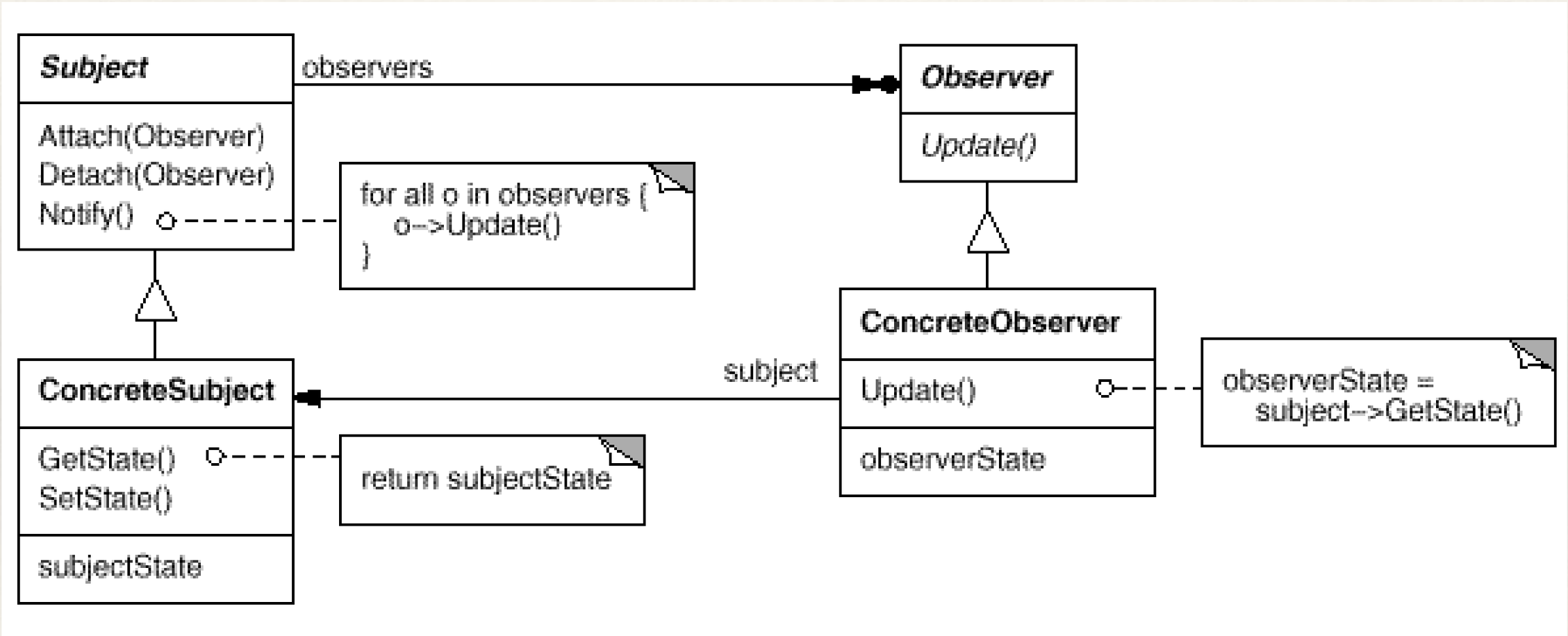
- LINQ ExpressionVisitors

Scenario

In an application similar to MS Paint, assume that a class (say ShapeArchiver) is responsible for archiving information about all the drawn shapes. Similarly, another class (say Canvas) is responsible for displaying all drawn shapes. Whenever any change in shapes takes place, you need to inform these two classes as to the changed information.

So, how you would like to implement this notification?

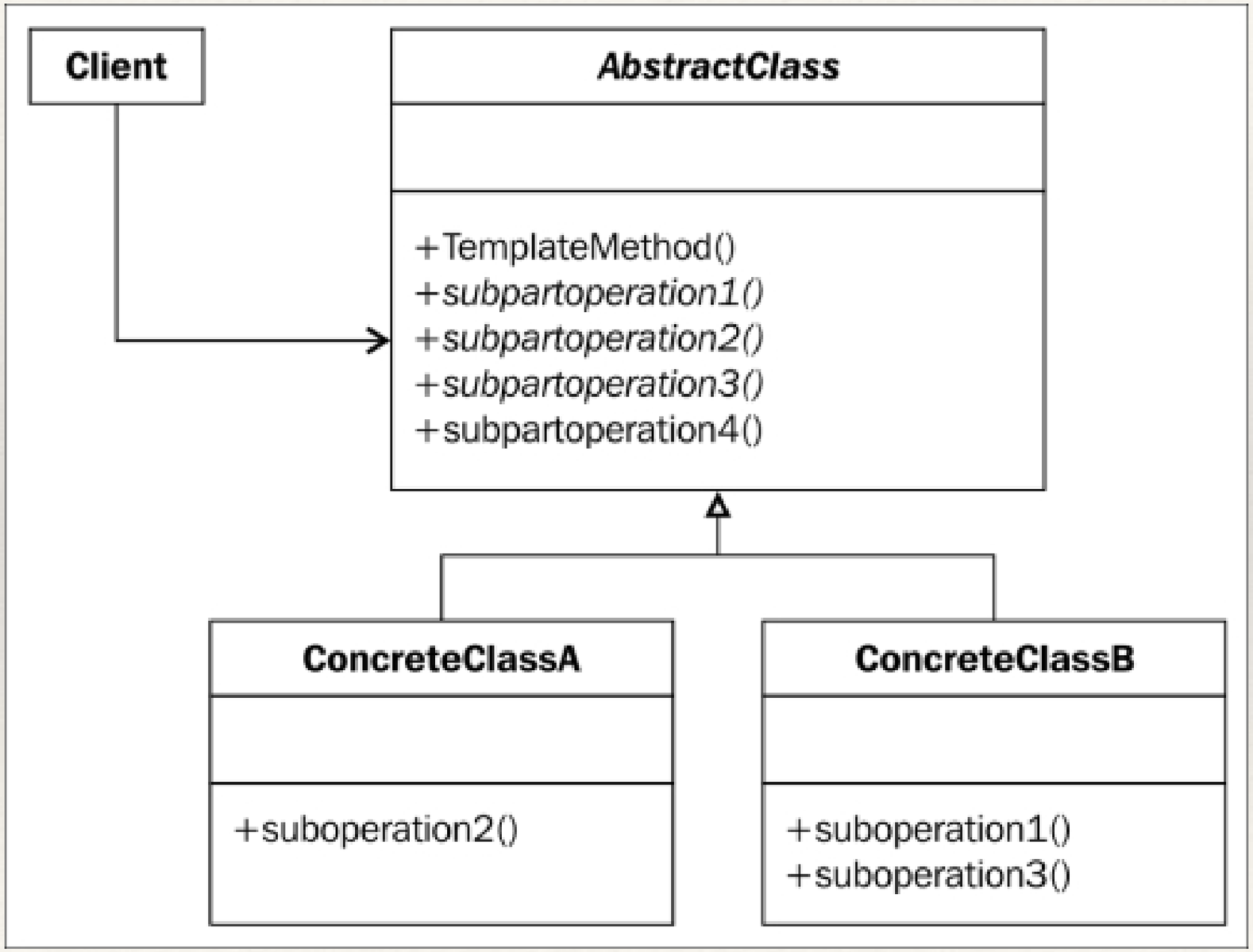
Observer pattern



Consideration in .NET Fx

- Events & Delegates
- IObservable<T>, IObserver<T>

Template Method: Structure



Source: https://www.safaribooksonline.com/library/view/swift-2-design/9781785887611/graphics/4582_05_07.jpg

Template Method Pattern: Discussion

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Hands-on exercise

```
4 references
internal abstract class DataAccessObject
{
    protected string connectionString;
    protected DataSet dataSet;

    1 reference
    public virtual void Connect()...

    3 references
    public abstract void Select();
    3 references
    public abstract void Process();

    1 reference
    public virtual void Disconnect()...

    // The 'Template Method'
    2 references
    public void Run()
    {
        Connect();
        Select();
        Process();
        Disconnect();
    }
}
```

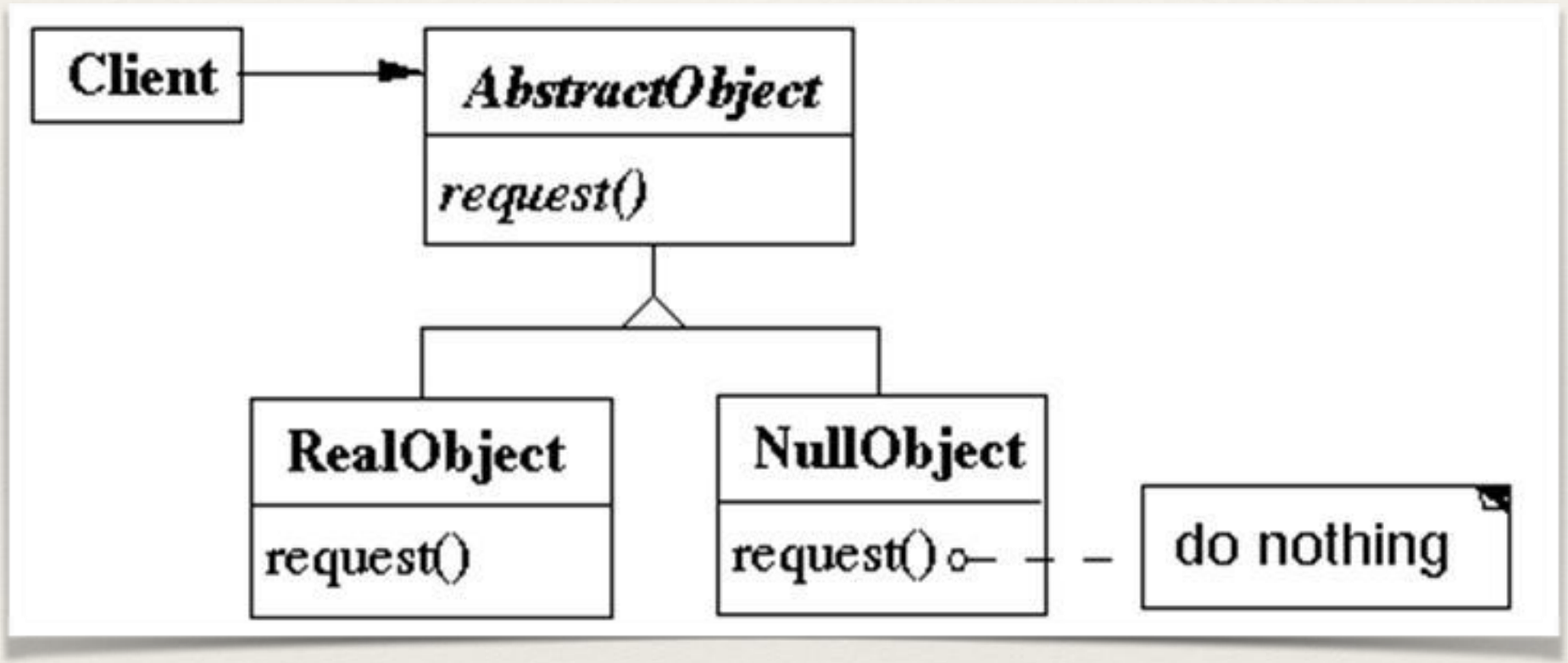
Template Method in .NET

- ASP.NET Page Model

Patterns discussed so far

- ✓ Builder pattern
- ✓ Factory method pattern
- ✓ Strategy pattern
- ✓ Bridge pattern
- ✓ Decorator pattern
- ✓ Observer pattern
- ✓ Composite pattern
- ✓ Flyweight pattern
- ✓ Proxy pattern
- ✓ Visitor pattern
- ✓ Template method pattern
- ✓ Interpreter pattern

Null Object pattern: structure



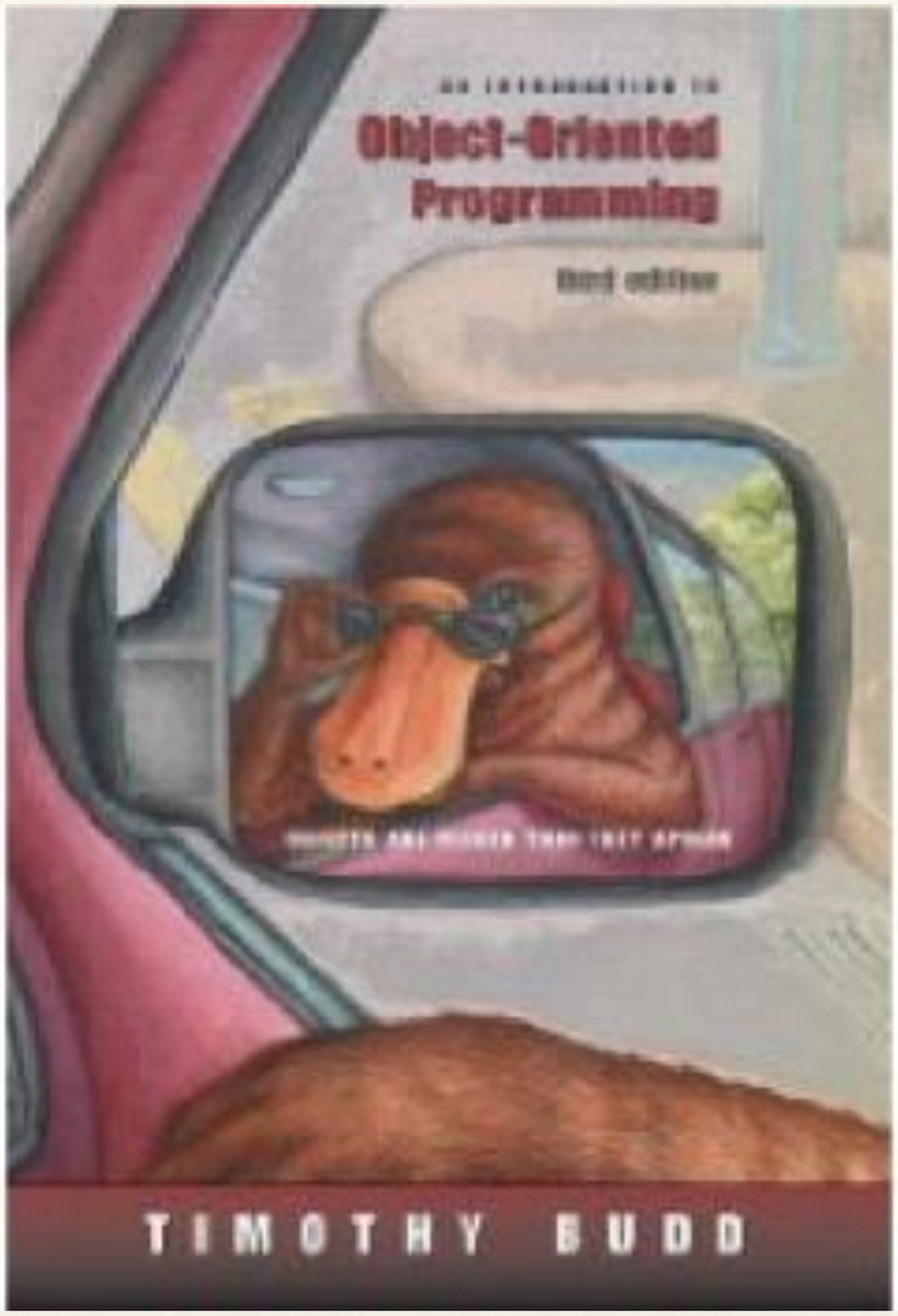
From .NET library

- DbNull – Represents a NULL value in the database
- String.Empty
- Nullable<T>? Is Null object pattern still relevant?

Beware of “patterns mania”!

What are your takeaways?

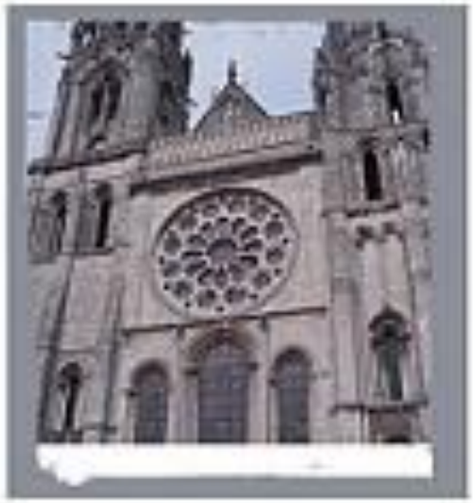
Books to read


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

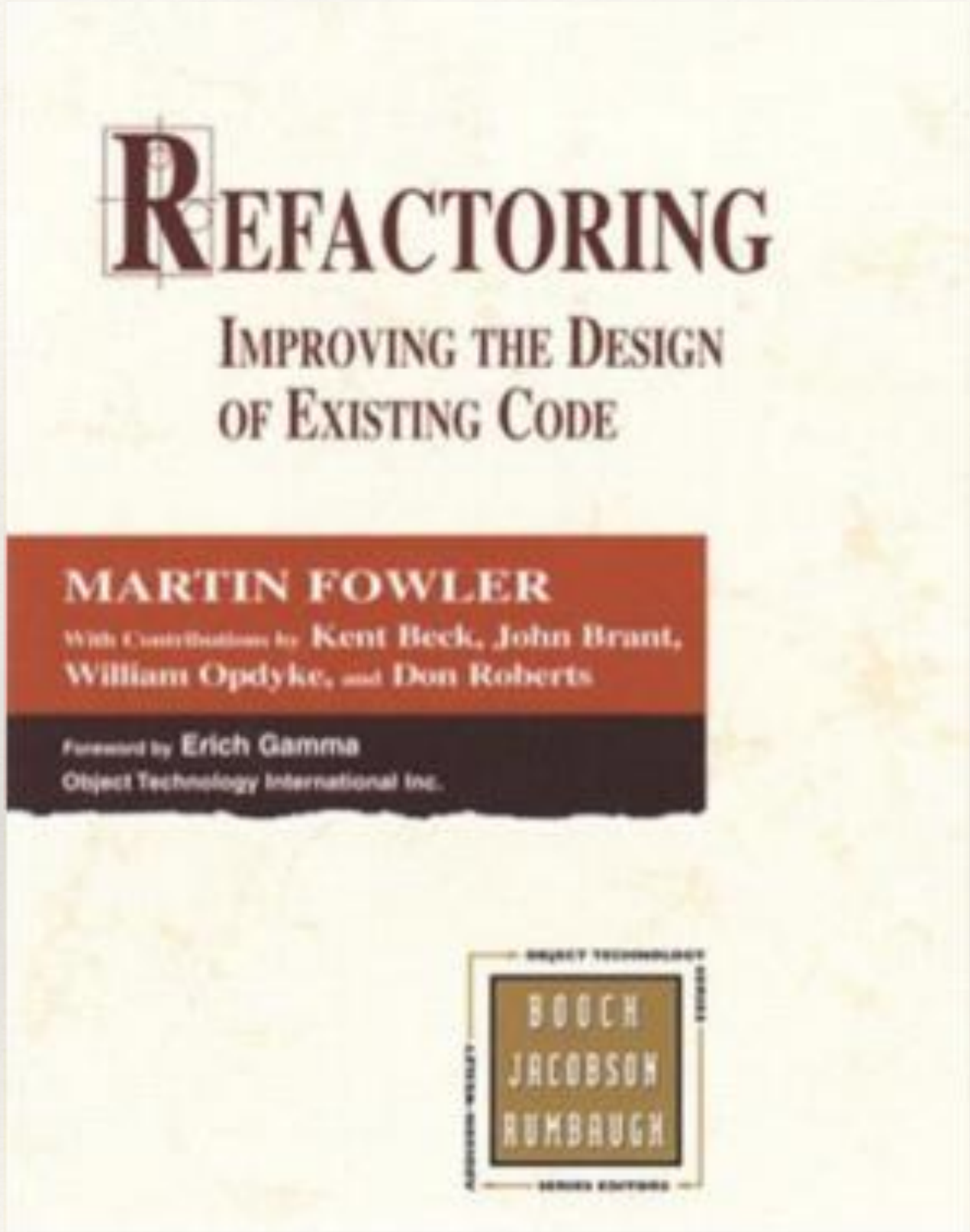
Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

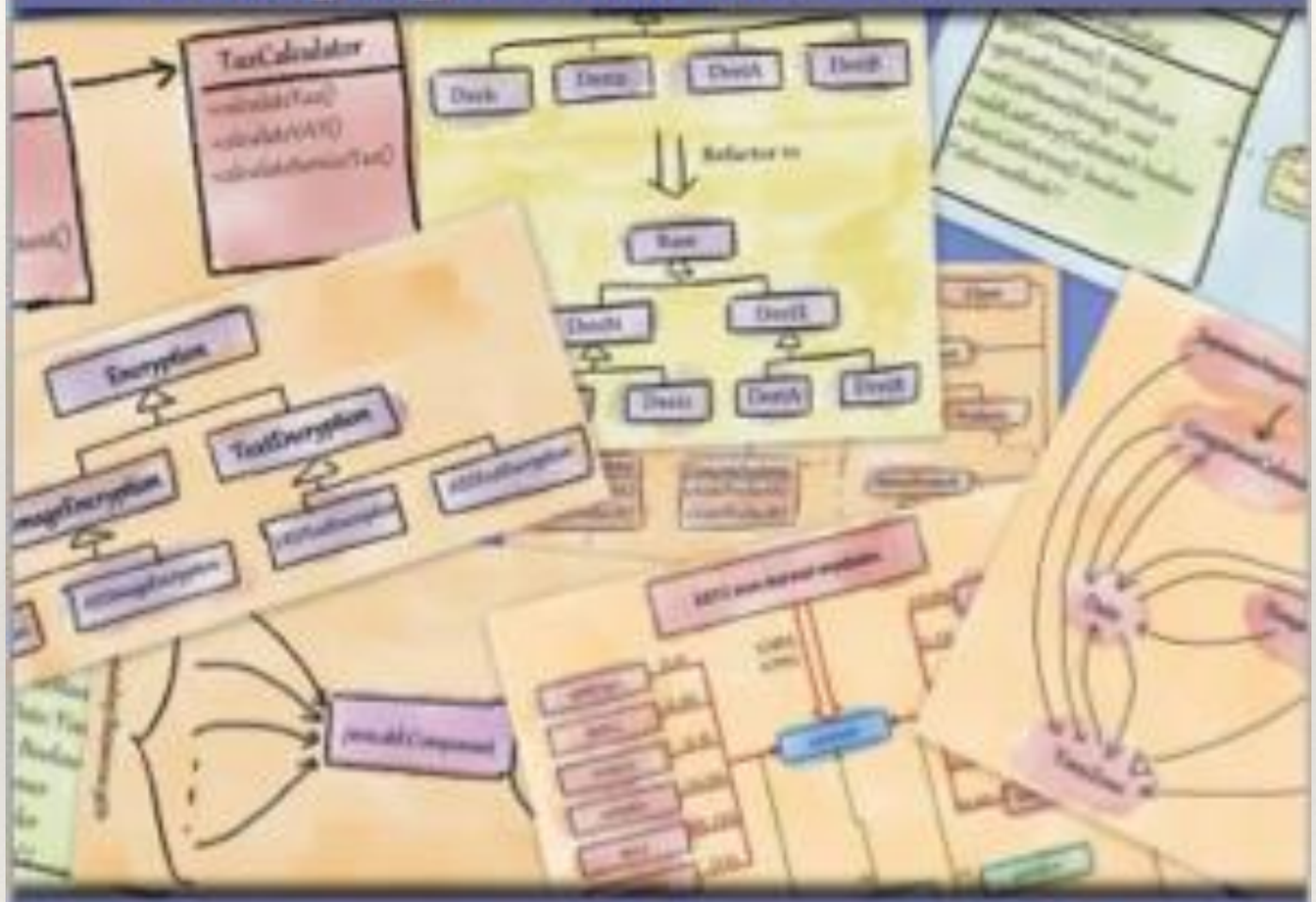
Foreword by Grady Booch

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.
BOOCH
JACOBSON
RUMBAUGH

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