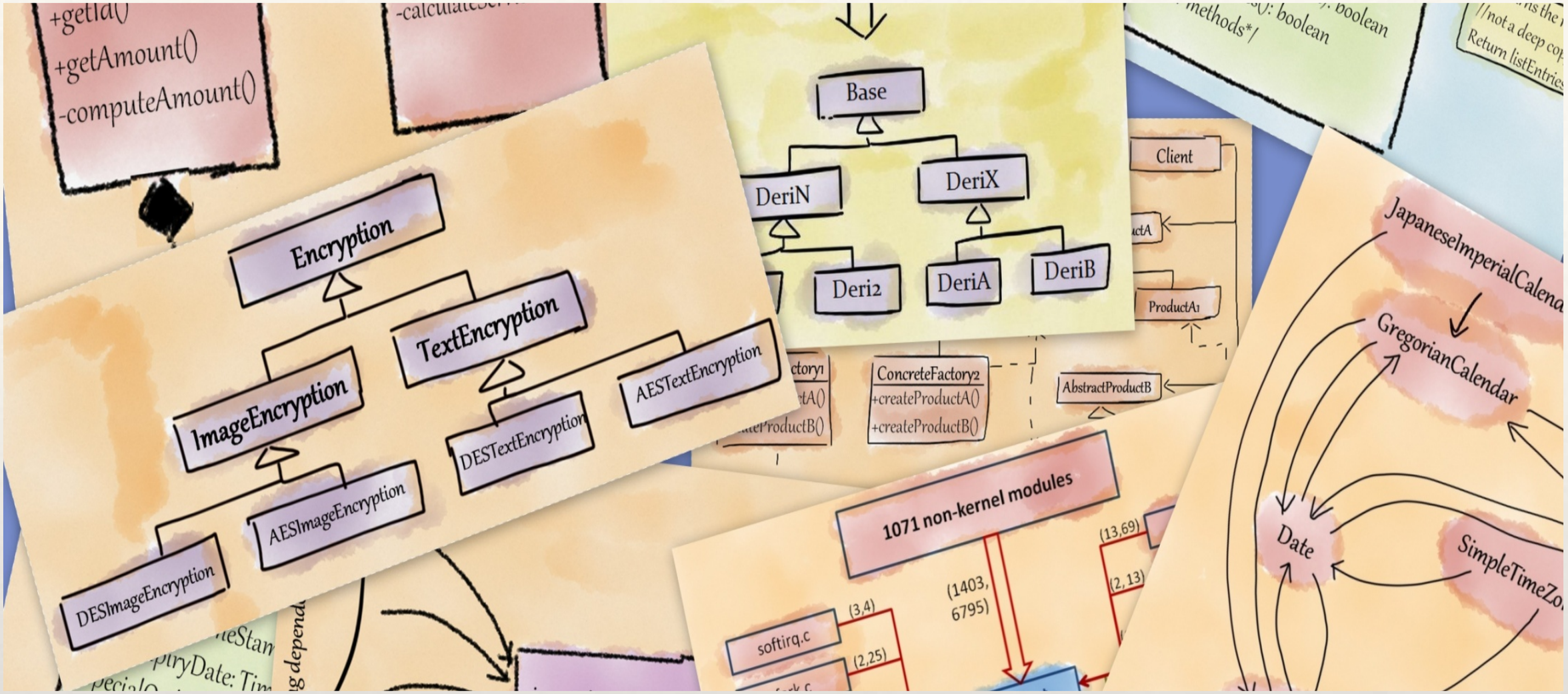


SOLID Principles & Design Patterns

Ganesh Samarthyam
ganesh@codeops.tech

"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?

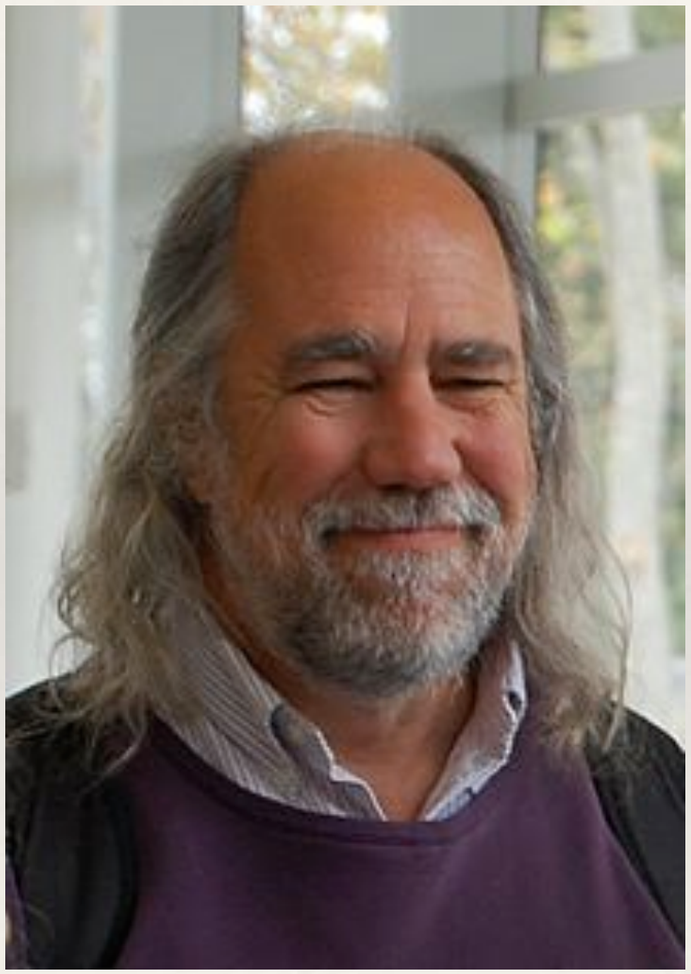
Poor software quality costs more than $150 billion per year in U.S. and greater than $500 billion per year worldwide

- Capers Jones

The city metaphor

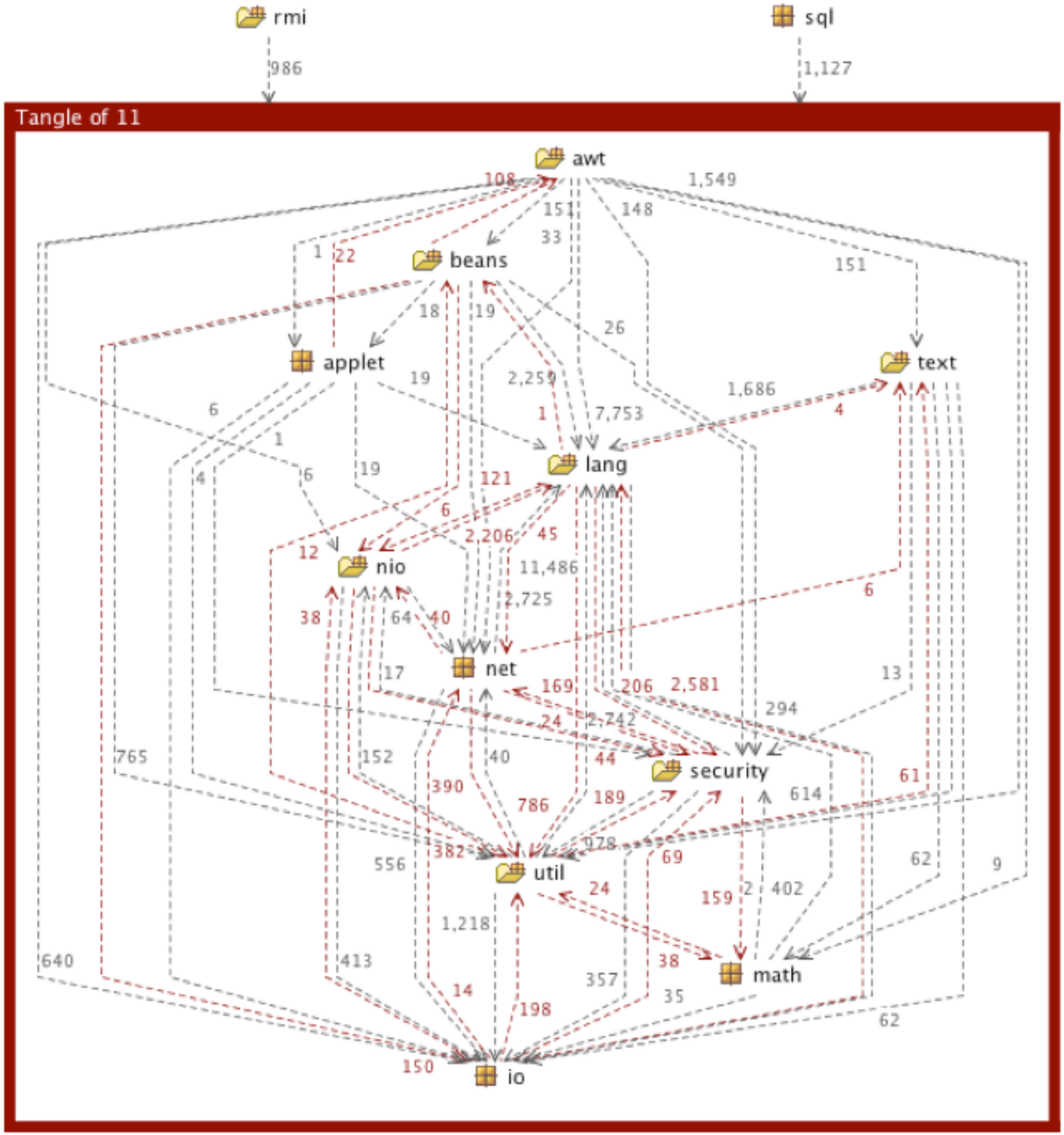
Source: http://indiatransportportal.com/wp-content/uploads/2012/04/Traffic-congestion1.jpg

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.


rmi
986
sql
1,127
Tangle of 11
awt
108
1,549
151
148
33
1
22
beans
151
18
19
26
applet
text
19
2,259
1,686
6
1
7,753
4
lang
19
121
4
6
6
2,206
45
12
nio
11,486
6
2,725
38
64
40
net
17
169
206
2,581
13
24
2,742
294
765
152
40
44
security
390
786
189
614
61
382
978
69
556
util
62
9
24
159
2
402
1,218
640
413
38
math
357
14
198
35
62
150
io

Modularisation in Java 9

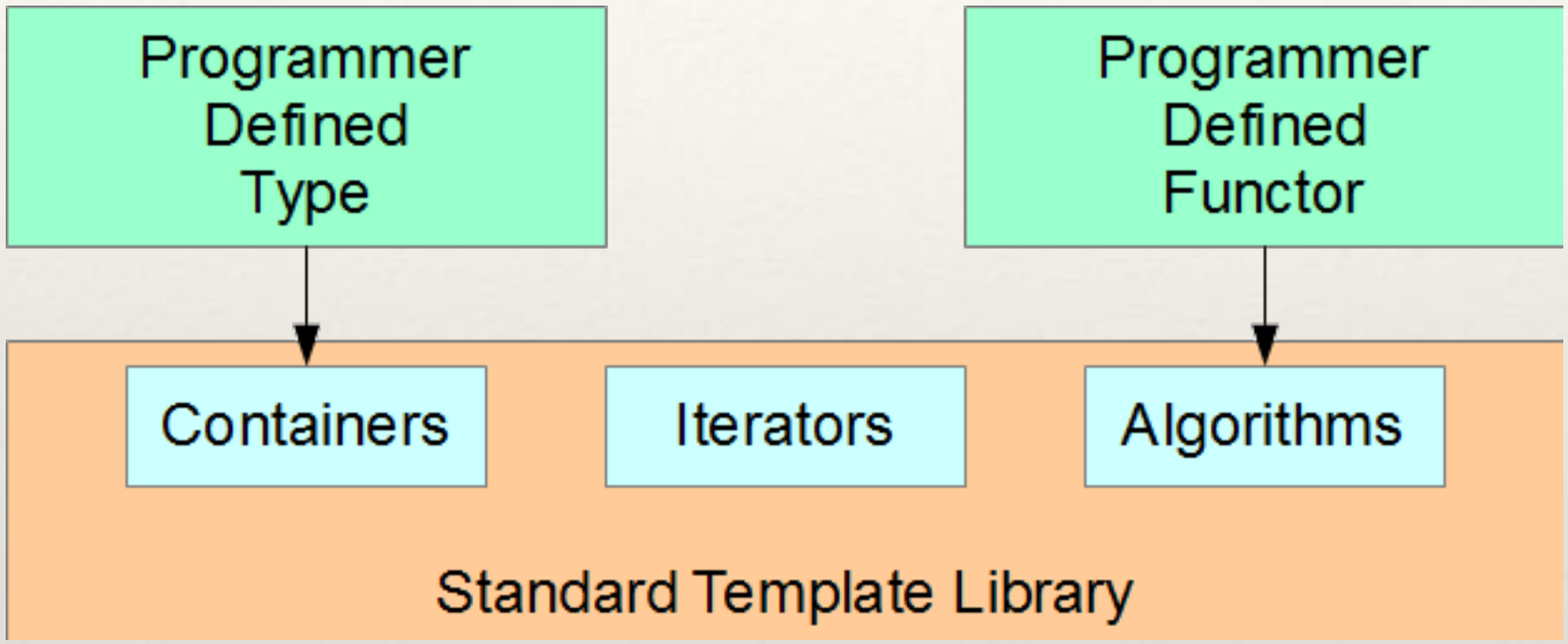
Modularize JDK & JRE

Hide platform internal details such as sun.misc

Provide a module system for Java developers

Reference: http://paulbakker.io/java/java-9-modularity/

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)

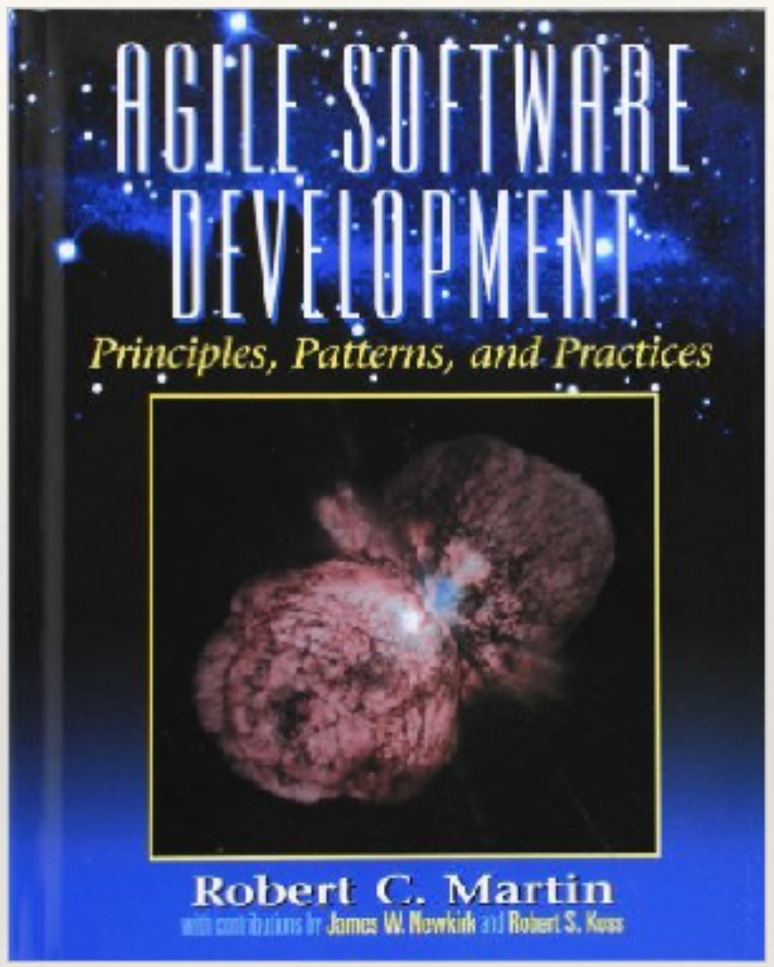
"The critical design tool for software development is a mind well educated in design principles"

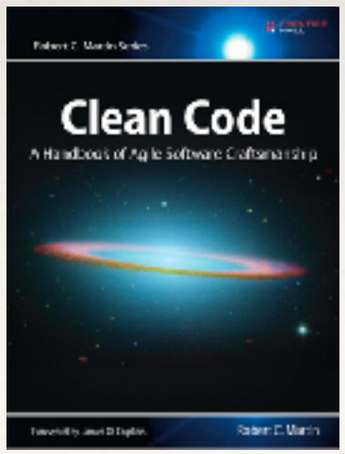
- Craig Larman

Fundamental Design Principles

Robert C. Martin

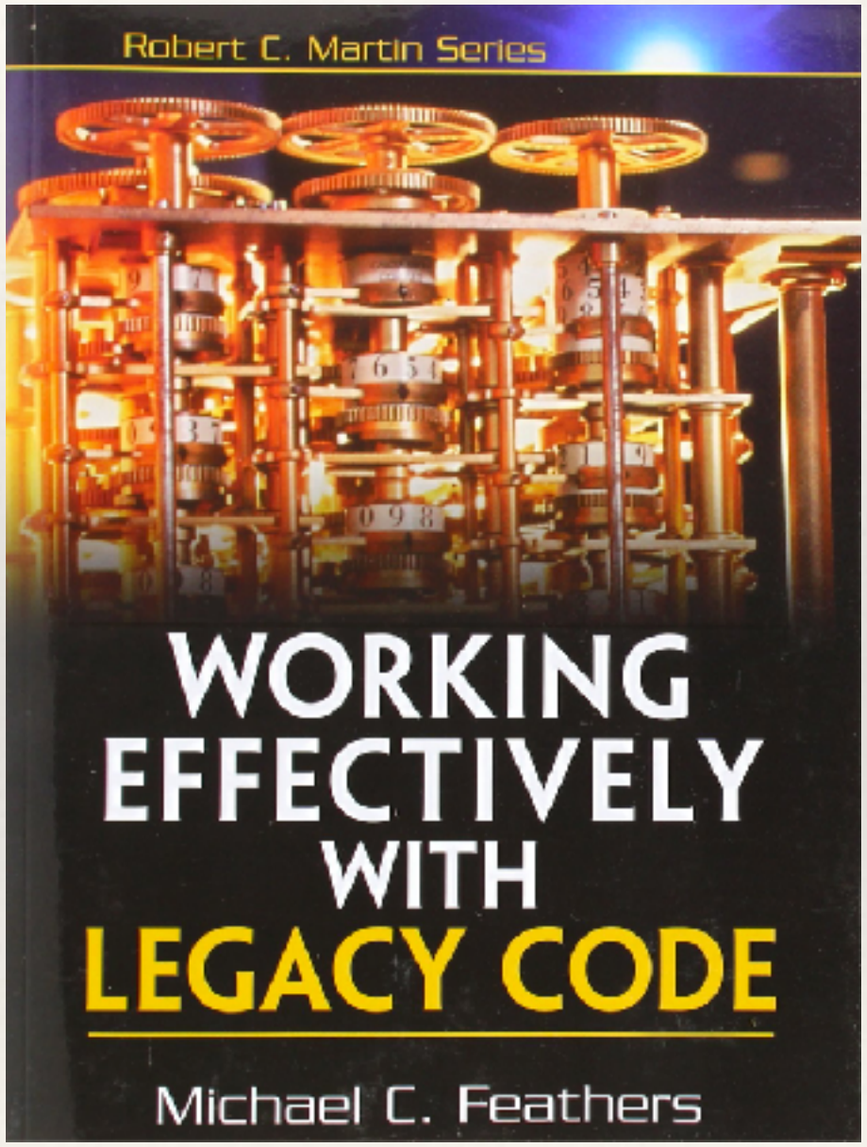
Formulated many principles and described many other important principles





Michael Feathers

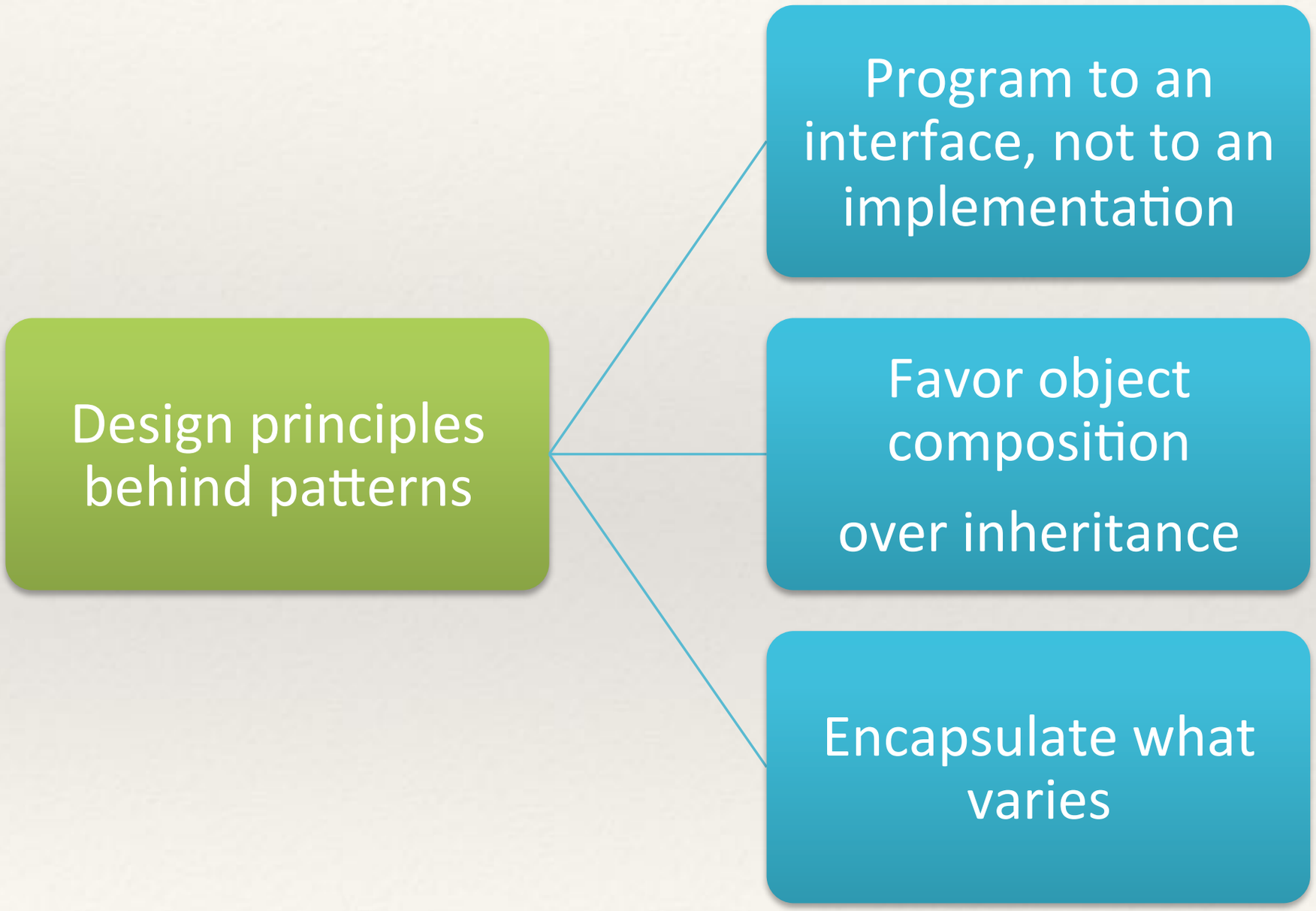
Michael Feathers coined the acronym SOLID in 2000s to remember first five of the numerous principles by Robert C. Martin



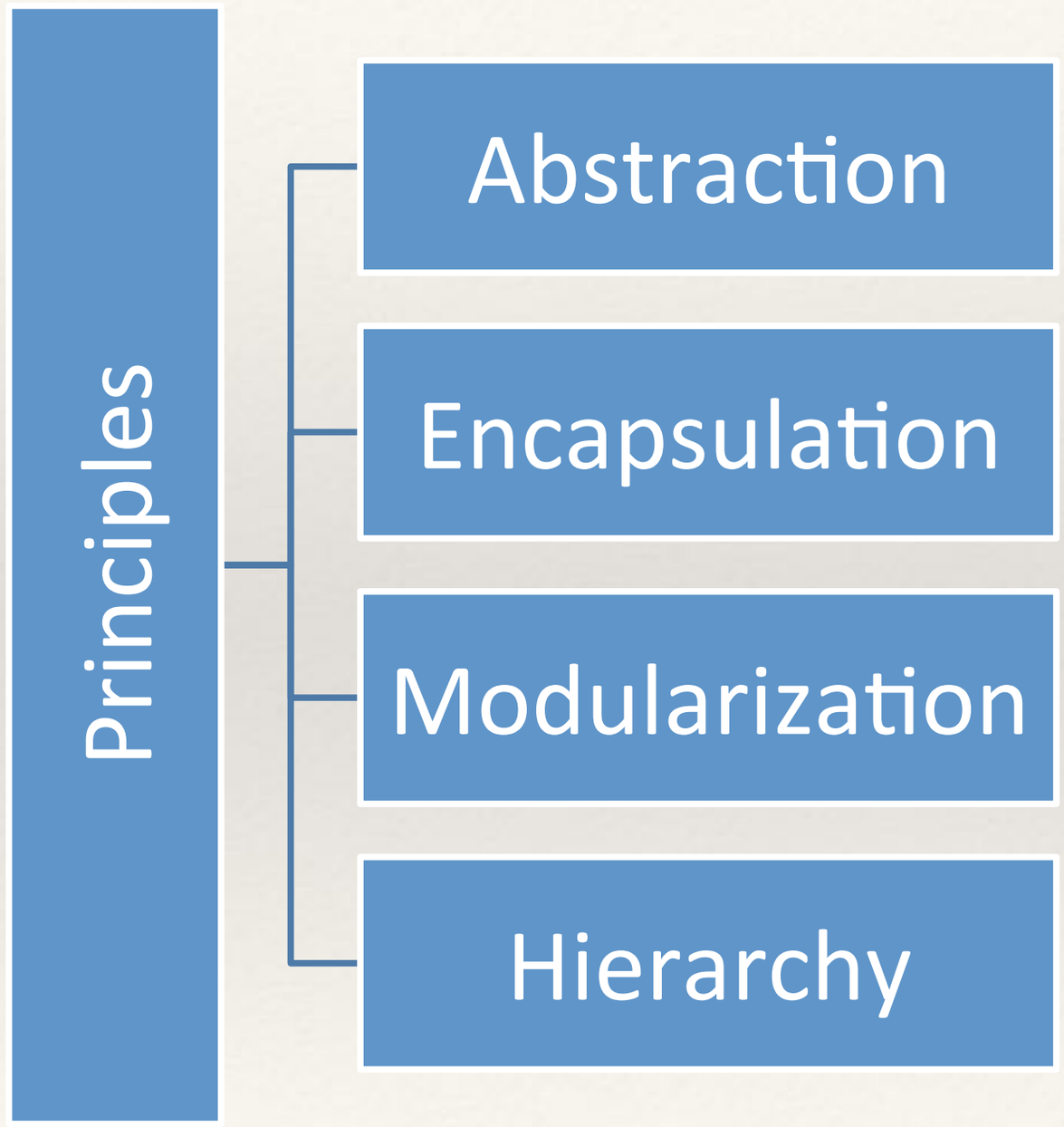
SOLID principles

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

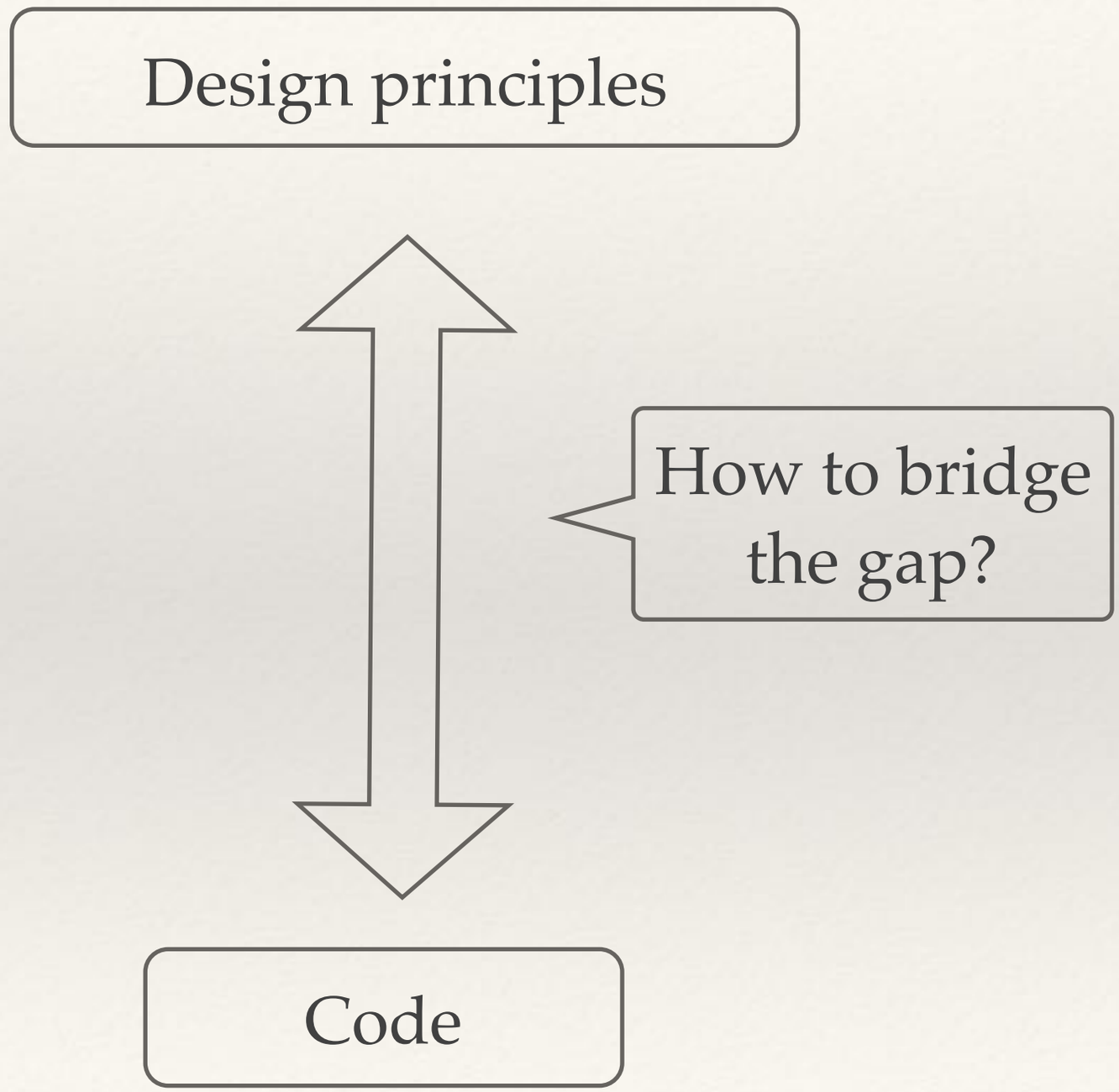
3 principles behind patterns



Booch's *fundamental* principles



How to apply principles in practice?



Why care about refactoring?

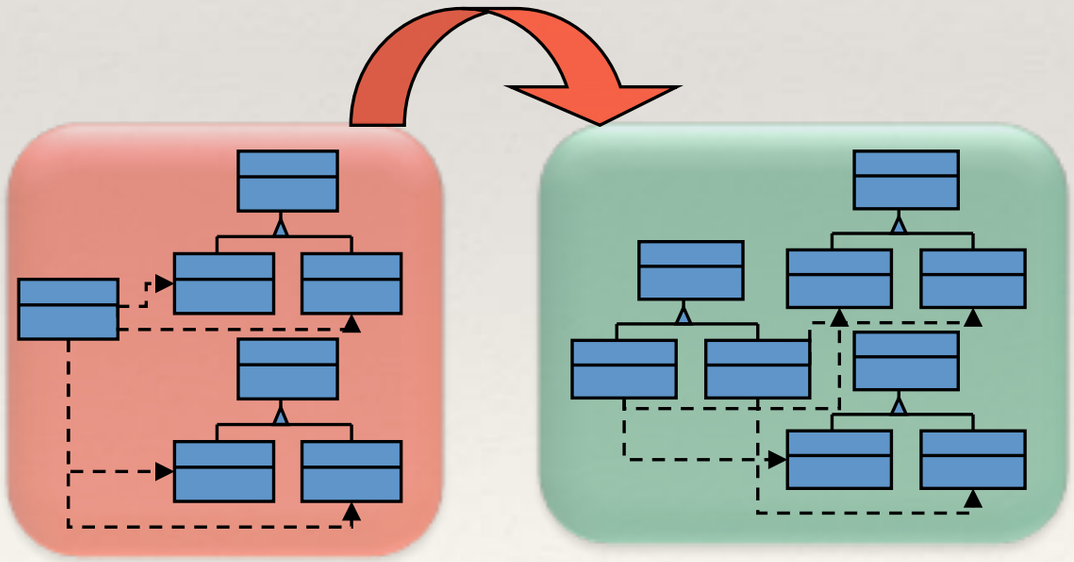
> As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it

- 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."

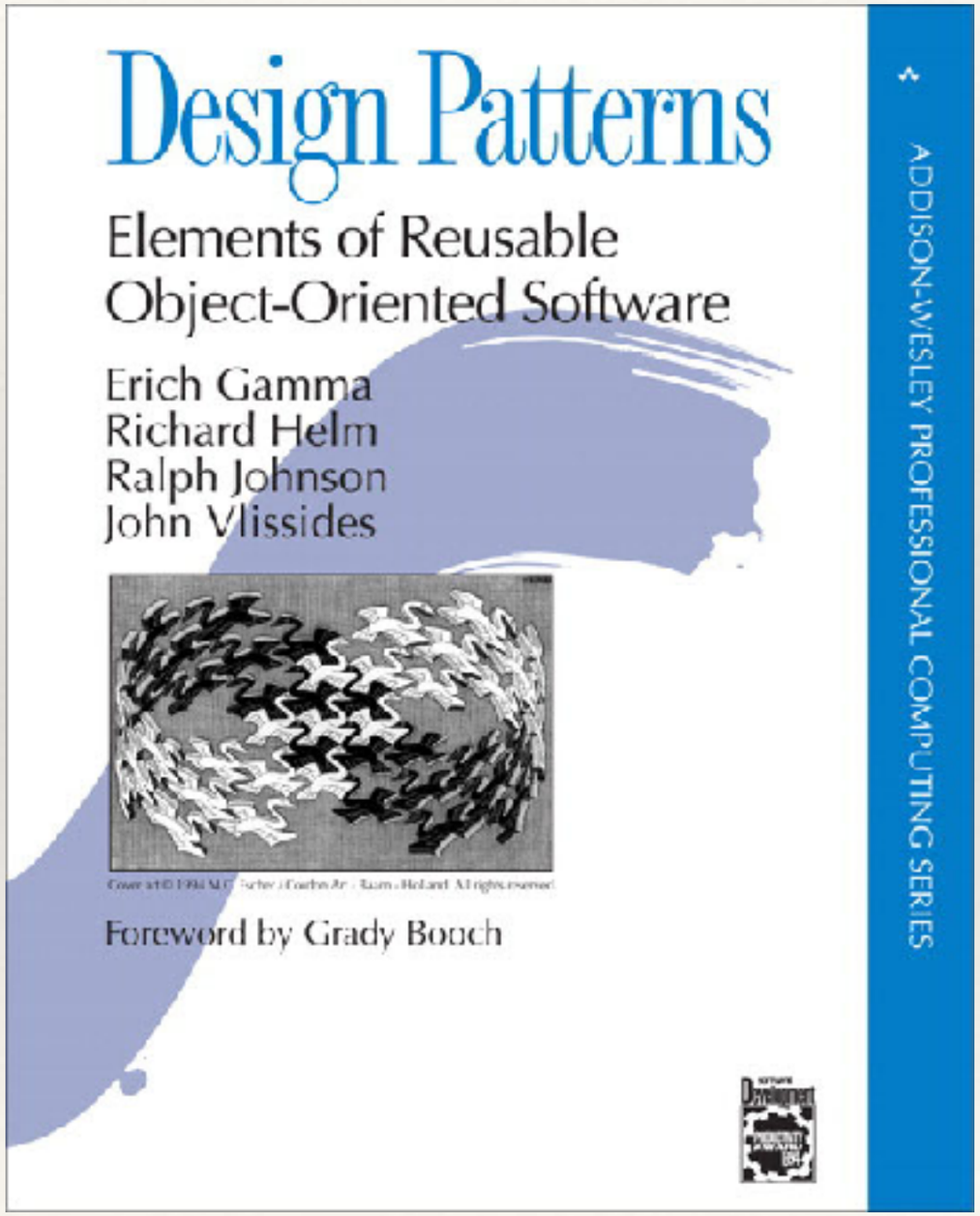
Granularity of smells

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

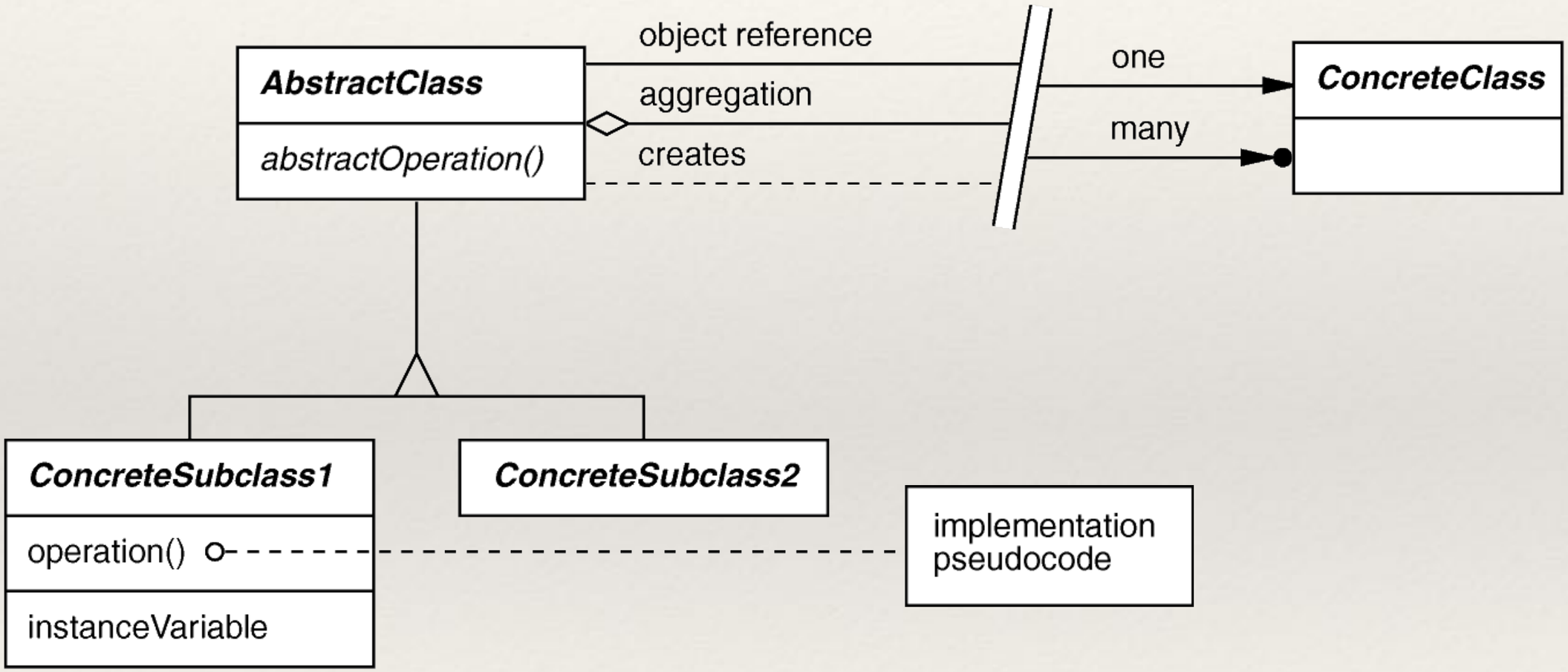
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Design pattern catalog

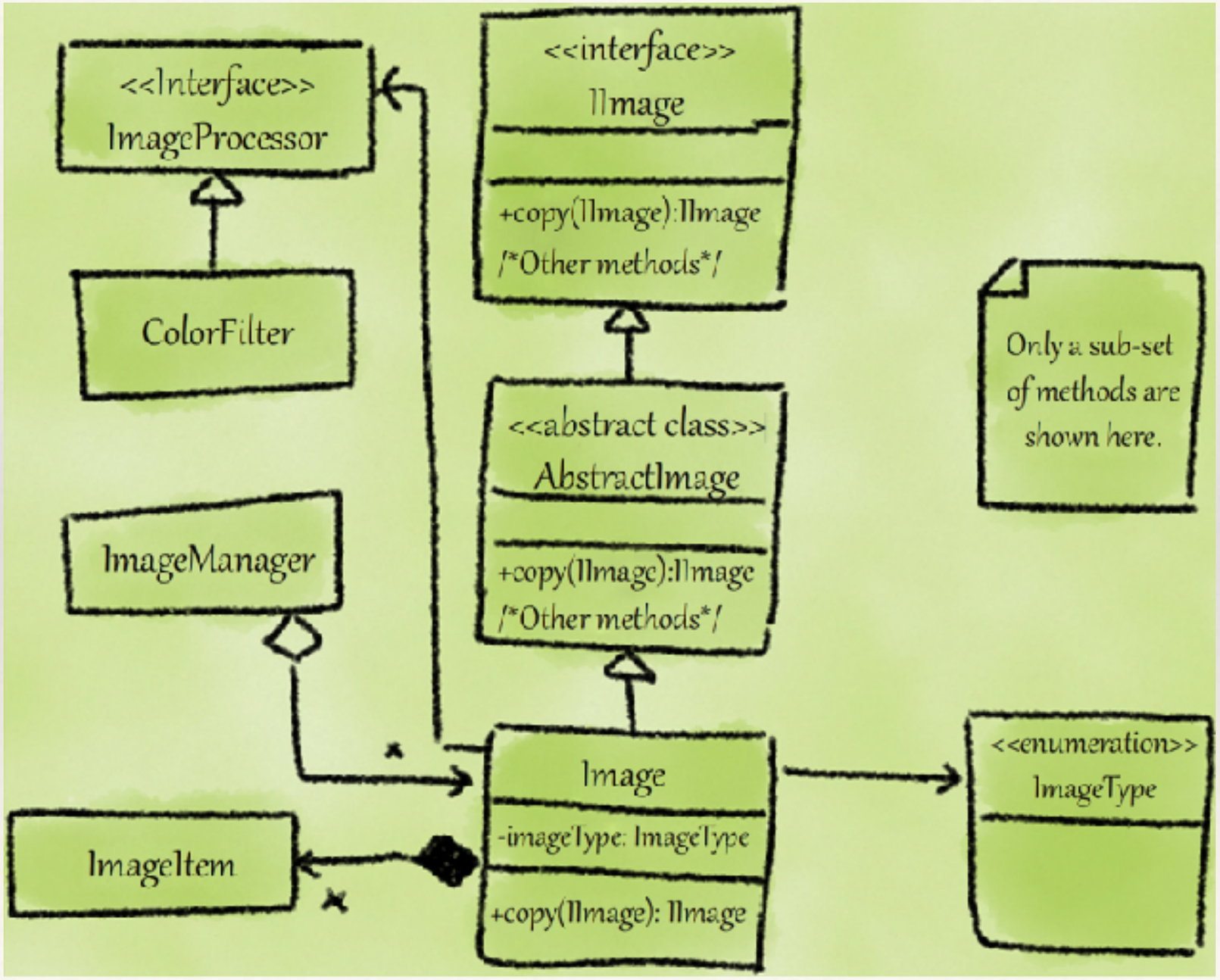
Creational	Deals with controlled object creation	*Factory method*, for example
Structural	Deals with composition of classes or objects	*Composite*, for example
Behavioral	Deals with interaction between objects / classes and distribution of responsibility	*Strategy*, for example

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

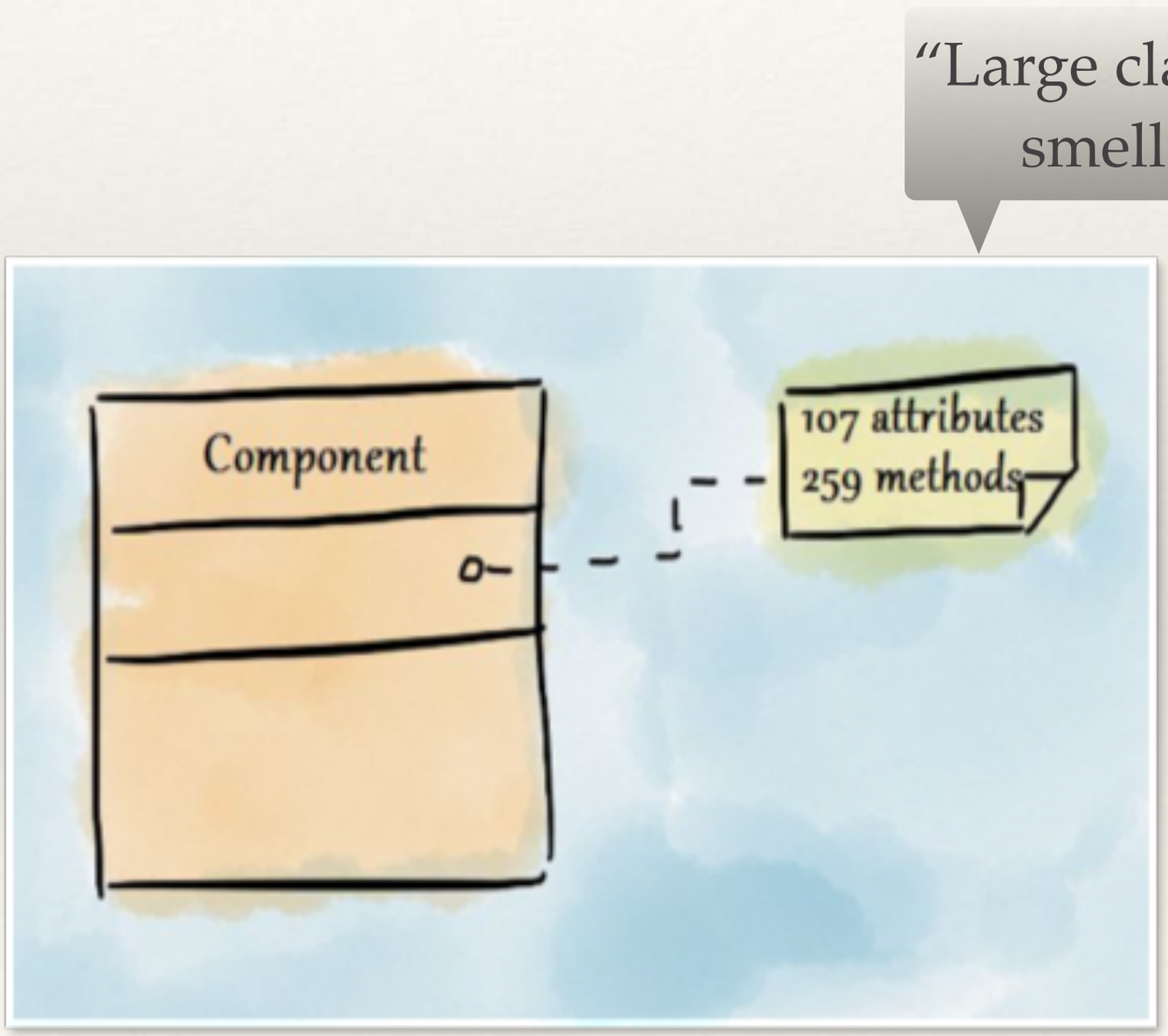


Source: "Refactoring for Software Design Smells: Managing Technical Debt", Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Morgan Kaufmann/Elsevier, 2014

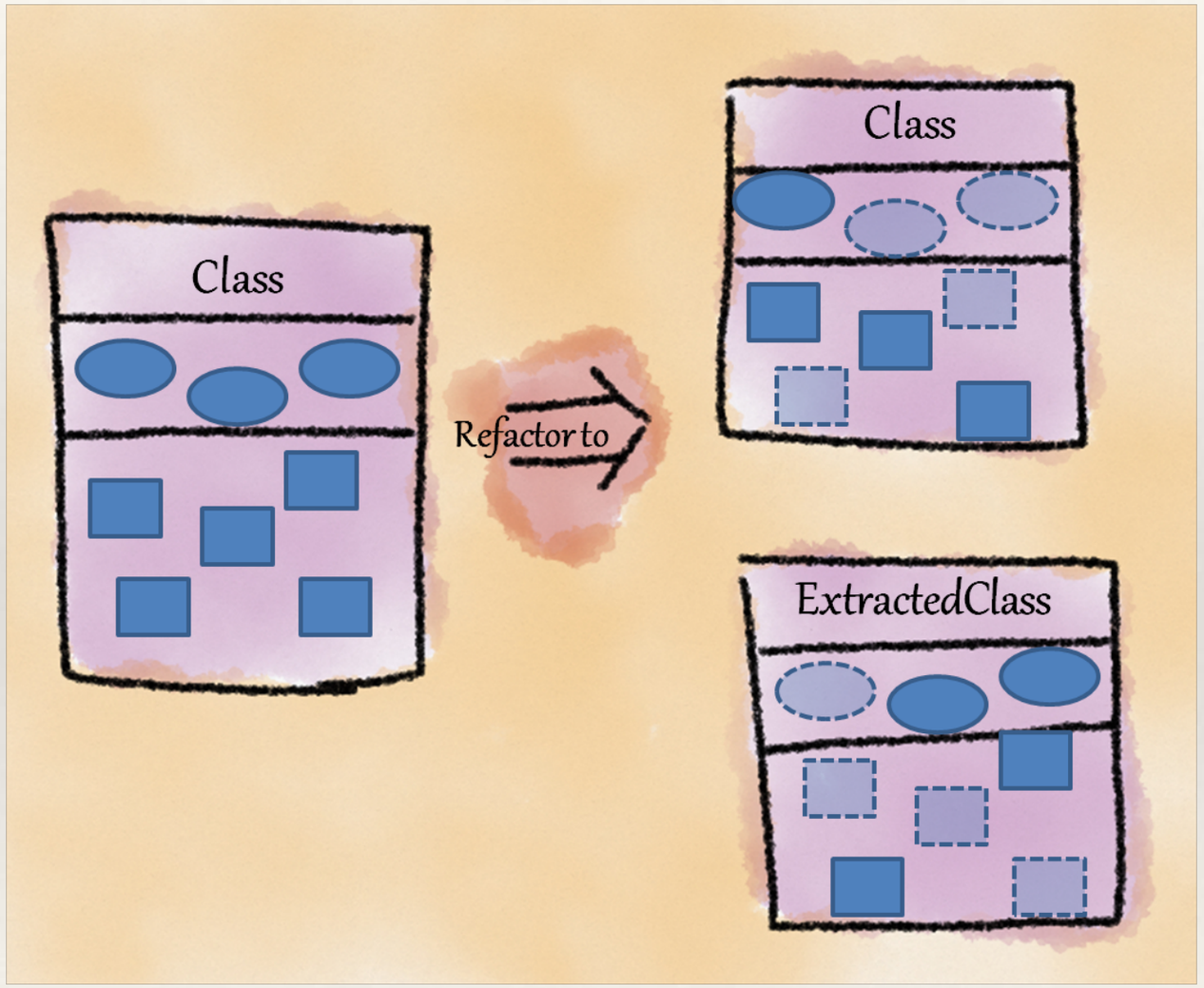
Hands-on exercise

- Refactor ZipTextFile.java

What's that smell?



Refactoring with "extract class"



Single Responsibility Principle

There should be only one reason for a class to change

What's that smell?

In addition to methods supporting dates, the class has methods for supporting time as well!

java.util.Calendar

java.time package!

What's that smell?

class GraphicsDevice

```
public void setFullScreenWindow(Window w) {
    if (w != null) {
        if (w.getShape() != null) { w.setShape(null); }
        if (w.getOpacity() < 1.0f) { w.setOpacity(1.0f); }
        if (!w.isOpaque()) {
            Color bgColor = w.getBackground();
            bgColor = new Color(bgColor.getRed(),
                                bgColor.getGreen(),
                                bgColor.getBlue(), 255);
            w.setBackground(bgColor);
        }
    }
...
}
```

This code in GraphicsDevice uses more methods of Window than calling its own methods!

"Feature envy" smell

Feature envy smell

Methods that are more interested in the data of other classes than that of their own class

What's that smell?

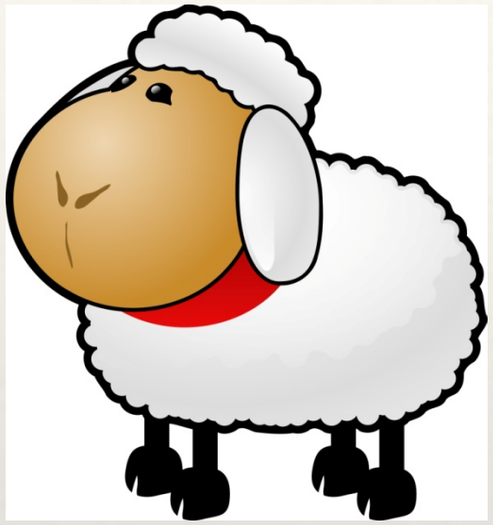
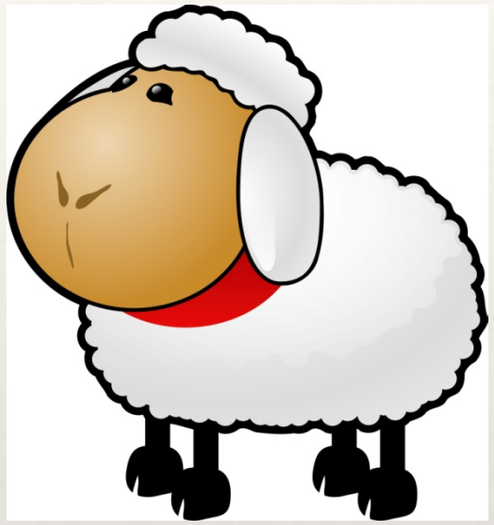
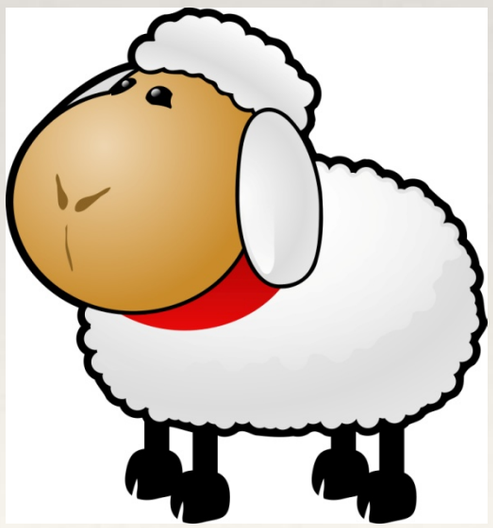
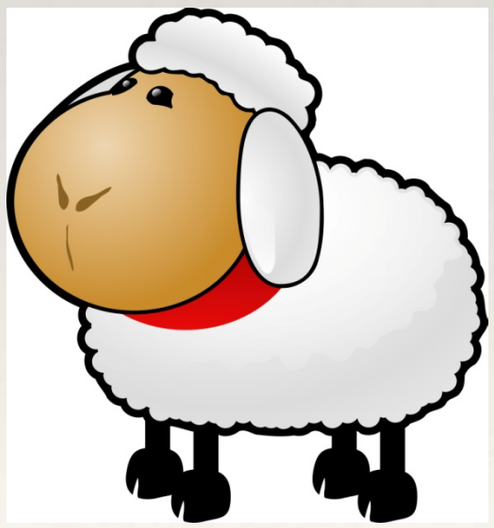
"Lazy class" smell

```
public class FormattableFlags {
        // Explicit instantiation of this class is prohibited.
        private FormattableFlags() {}
        /** Left-justifies the output.  */
        public static final int LEFT_JUSTIFY = 1<<0; // '-'
        /** Converts the output to upper case */
        public static final int UPPERCASE = 1<<1;    // 'S'
        /**Requires the output to use an alternate form. */
        public static final int ALTERNATE = 1<<2;    // '#'
}
```

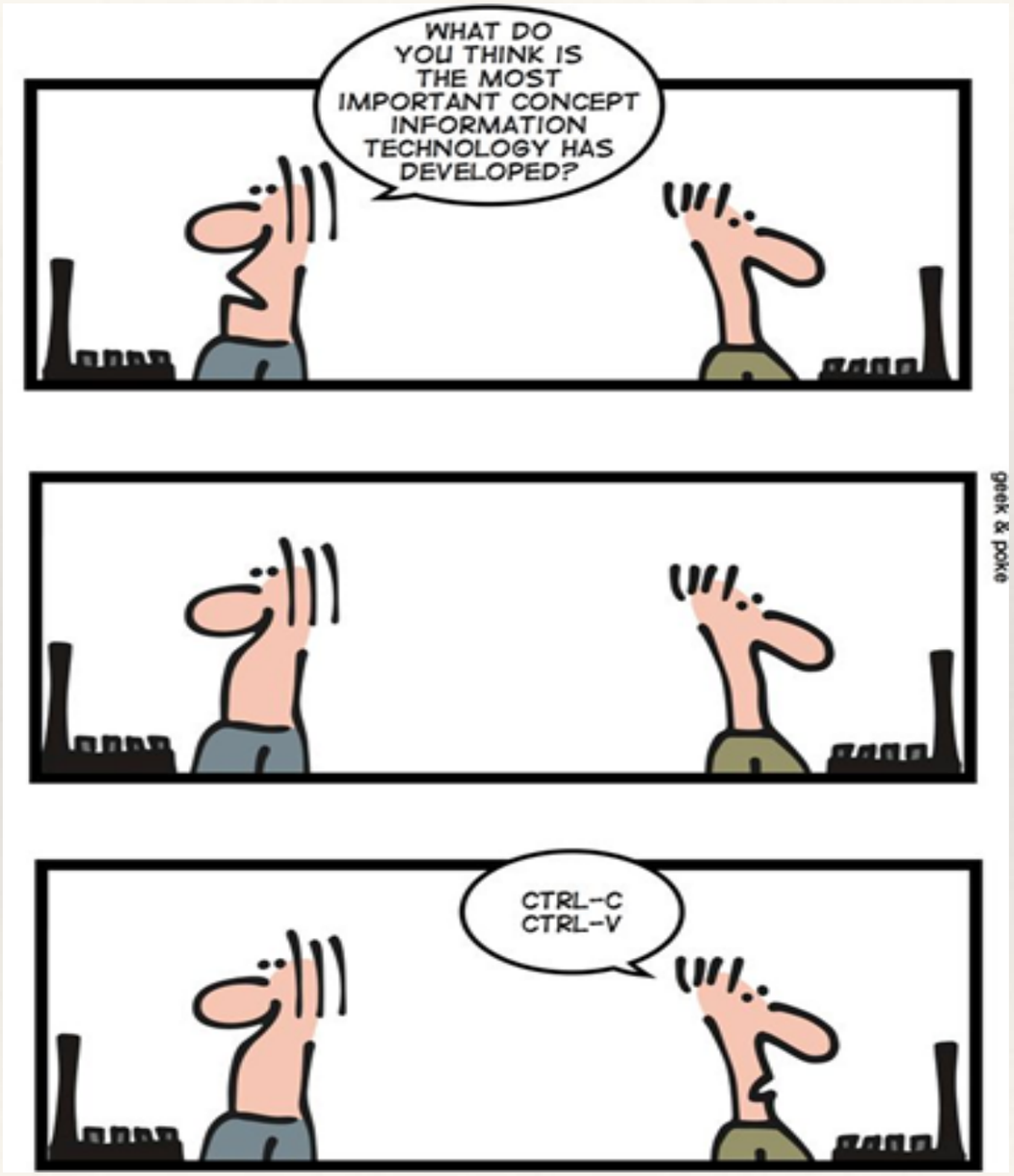
Hands-on exercise

- Refactor FormattableFlags.java

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

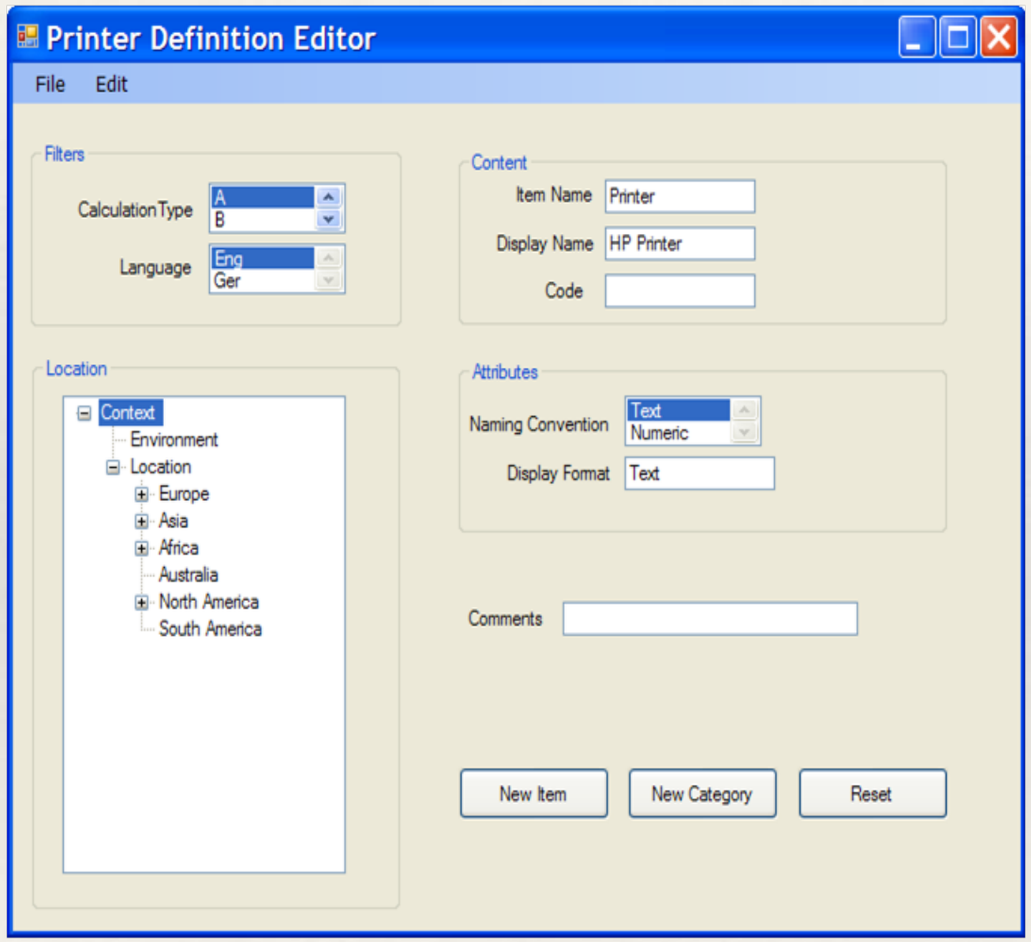
Type 3

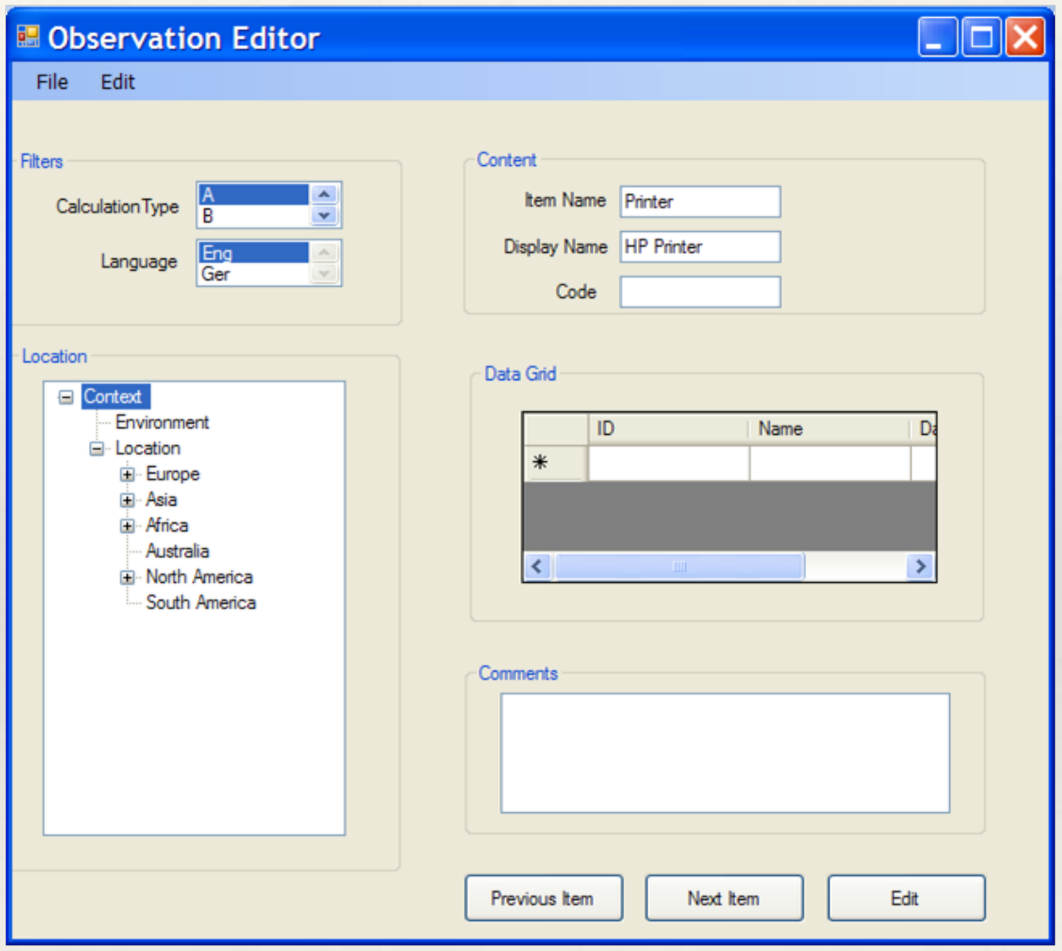
- identical except some **statements changed, added, or removed**

Type 4

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

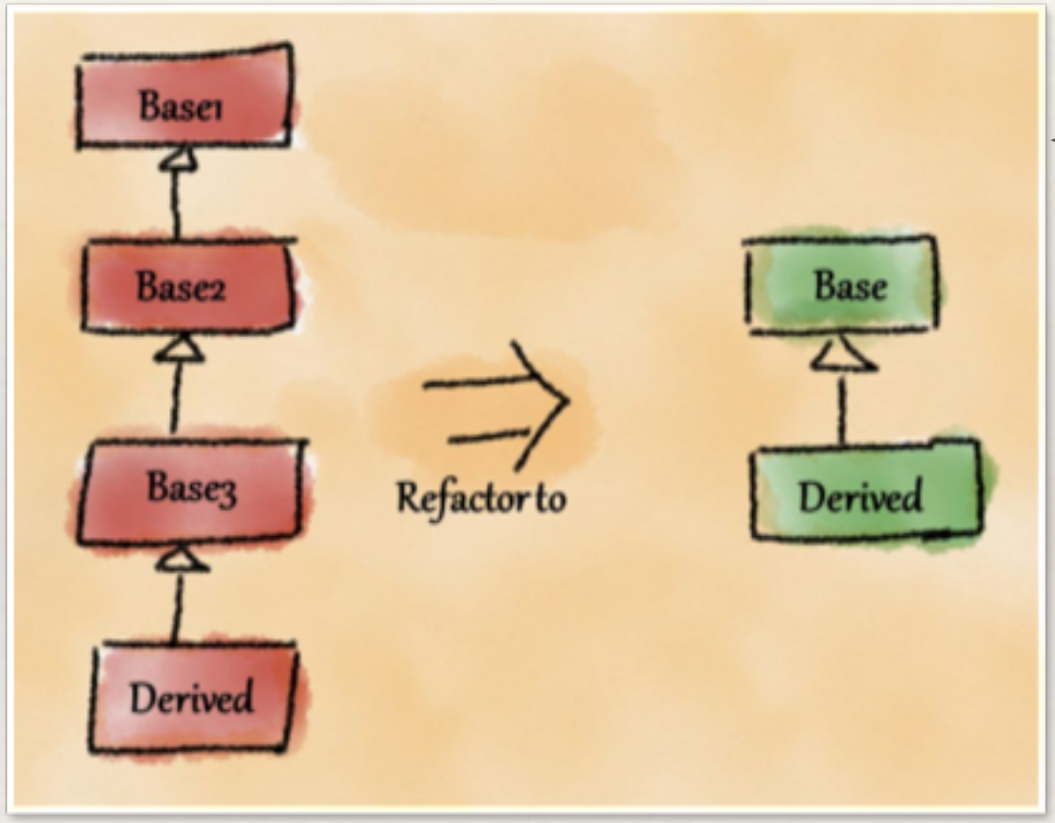
How to deal with duplication?





What's that smell?

Refactoring

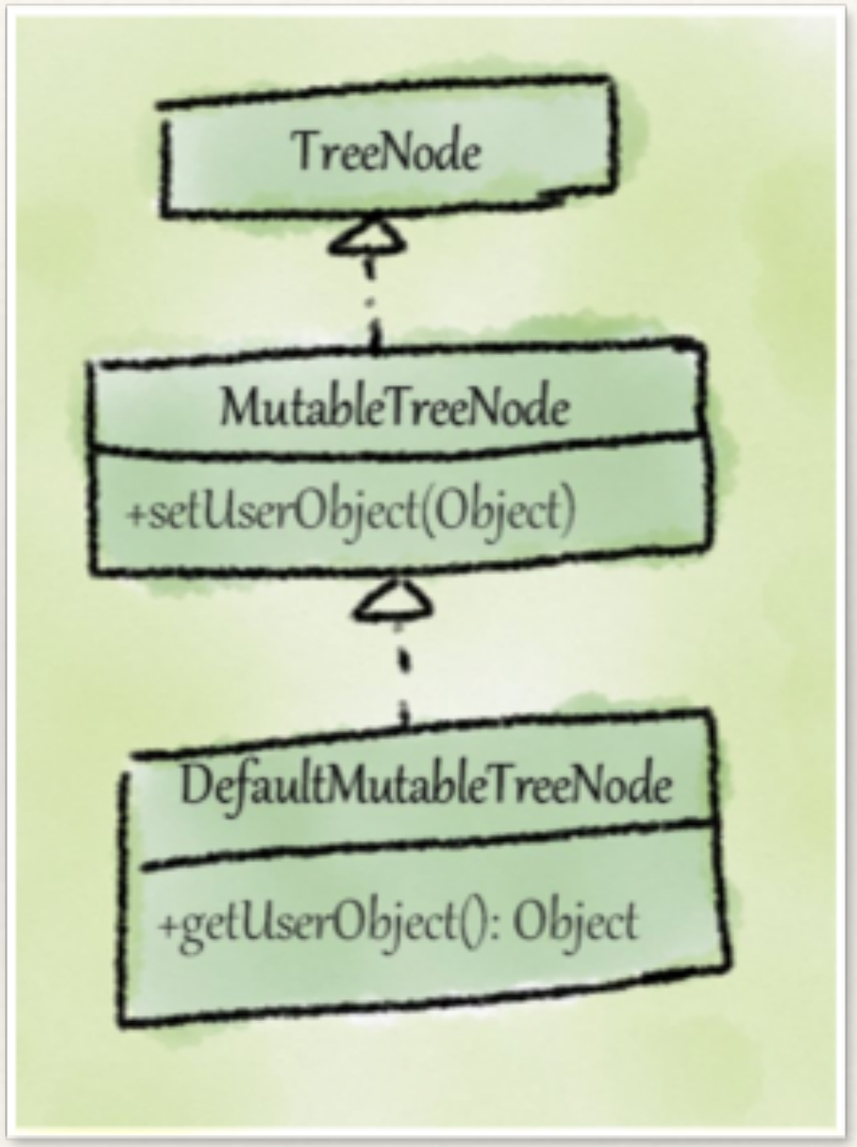


Collapse Hierarchy

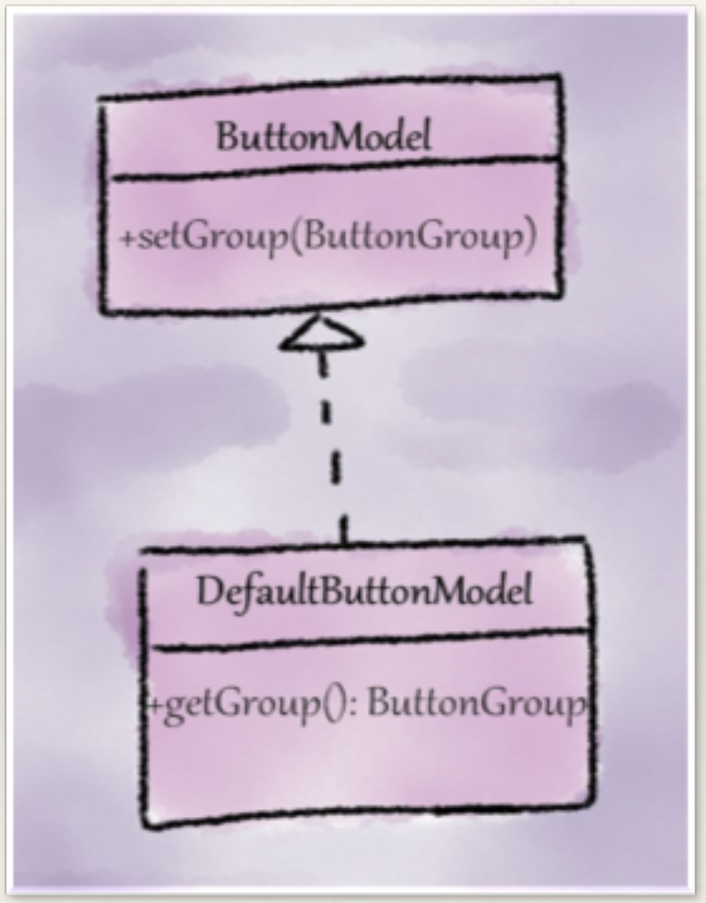
Refactoring

```
/**
 * Pluggable look and feel interface for JButton.
*/
public abstract class ButtonUI extends ComponentUI {
}
```

What's that smell?



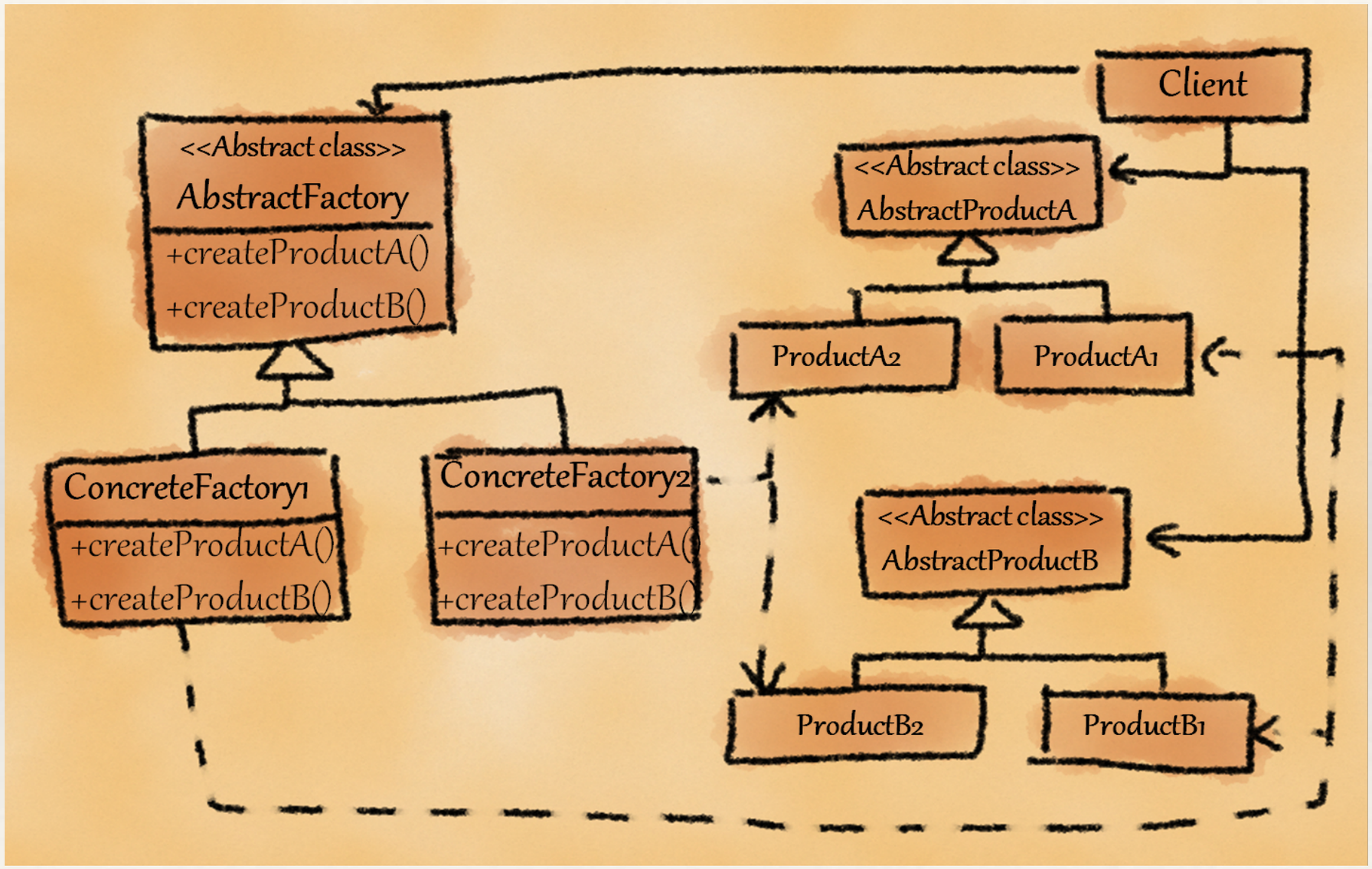
What's that smell?



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



Scenario

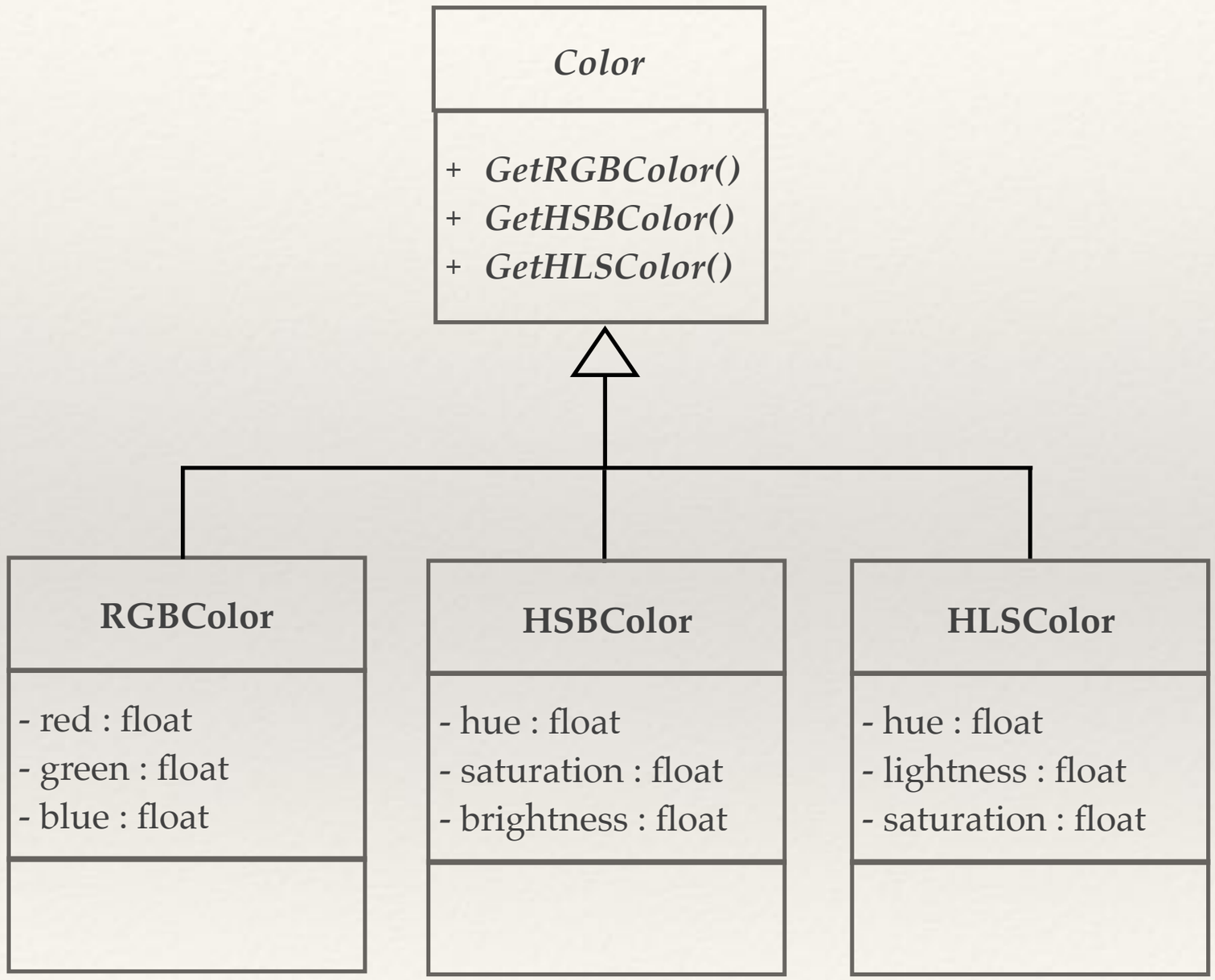
- Assume that you need to support different Color schemes in your software
  - RGB (Red, Green, Blue), HSB (Hue, Saturation, Brightness), and HLS (Hue, Lightness, and Saturation) schemes
- Overloading constructors and differentiating them using enums can become confusing
- What could be a better design?

```
enum ColorScheme { RGB, HSB, HLS, CMYK }
class Color {
        private float red, green, blue;                         // for supporting RGB scheme
        private float hue1, saturation1, brightness1;       // for supporting HSB scheme
        private float hue2, lightness2, saturation2;        // for supporting HLS scheme
        public Color(float arg1, float arg2, float arg3, ColorScheme  cs) {
                switch (cs) {
                        // initialize arg1, arg2, and arg3 based on ColorScheme value
                }
        }
}
```

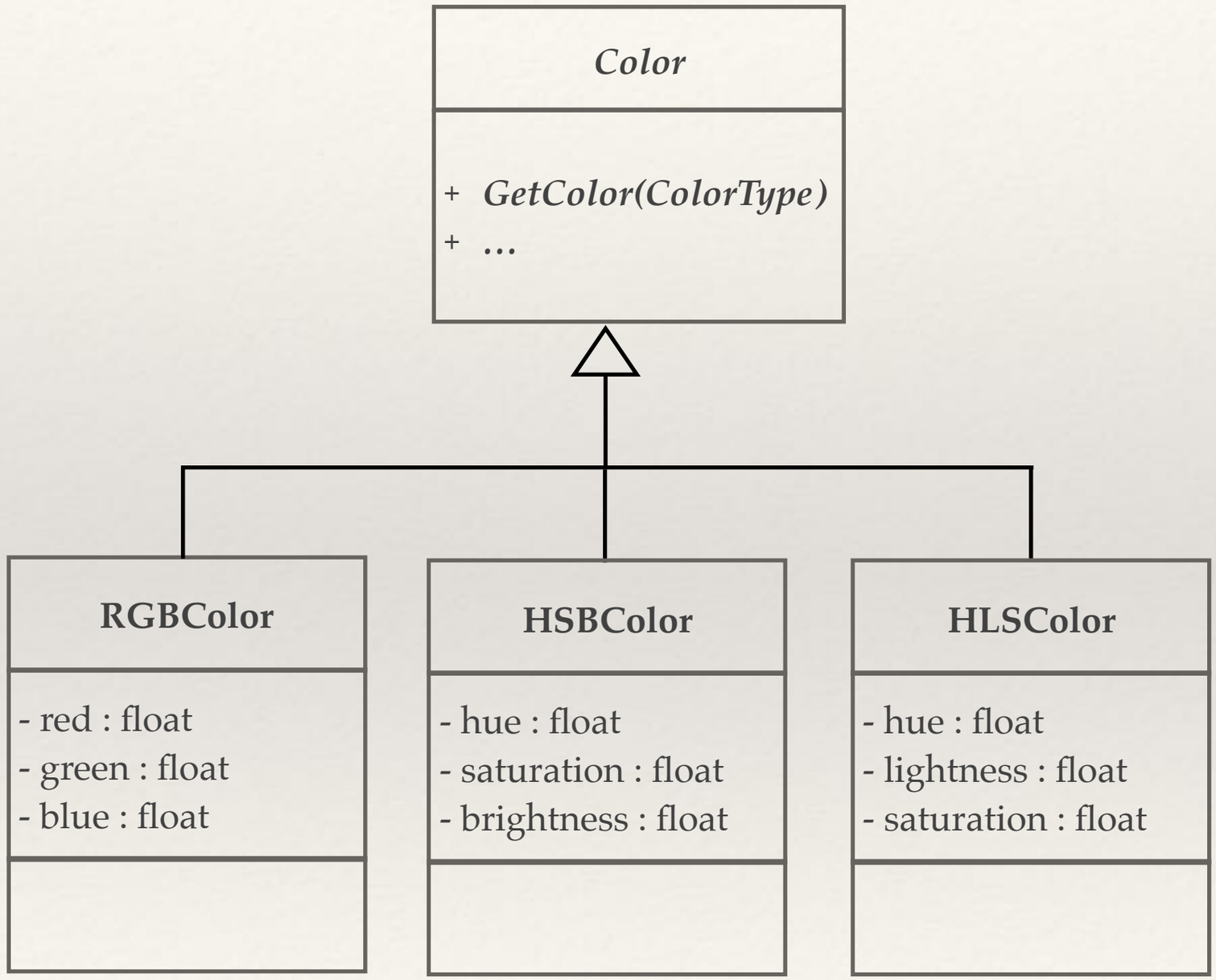
Hands-on exercise

- Refactor Color.java
- Hint: Use “factory method” design pattern

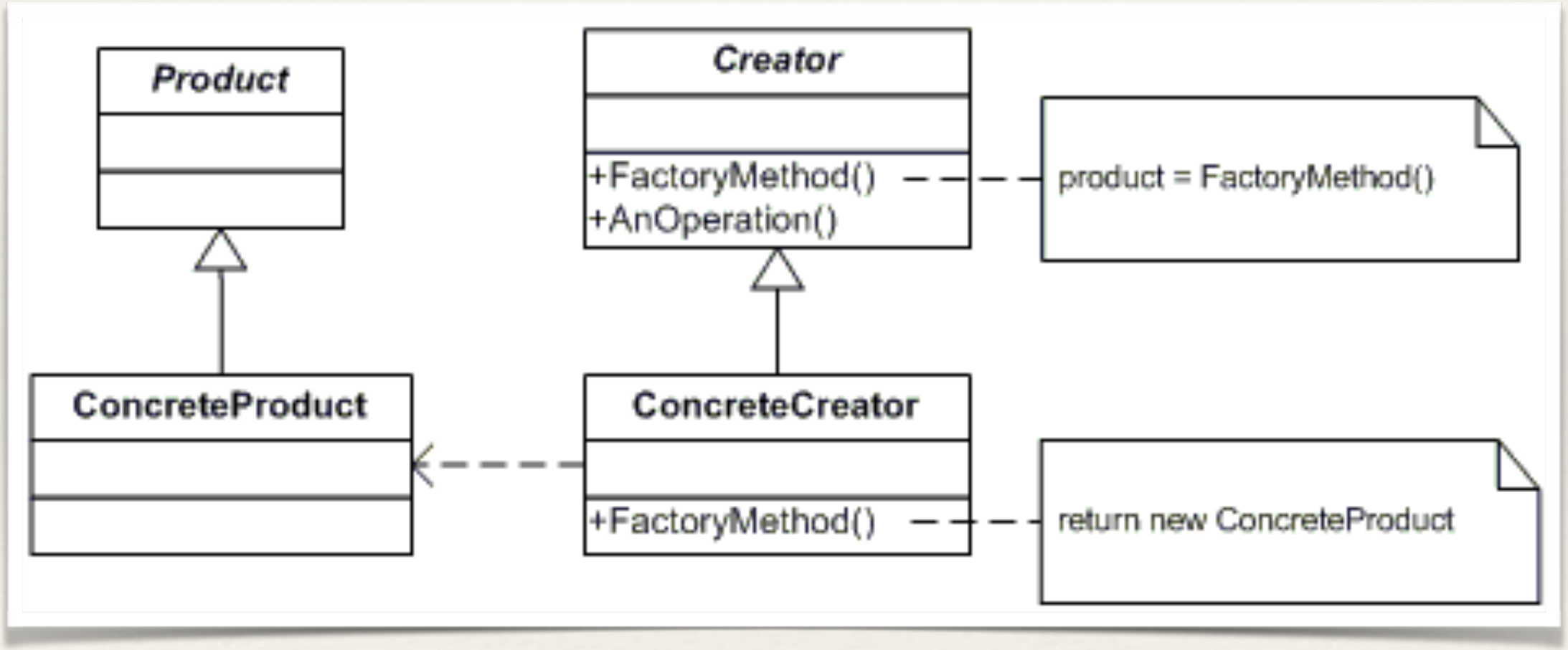
A solution using Factory Method pattern



A solution using Factory Method pattern



Factory Method pattern: Structure



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

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

Factory method: Java library example

```
Logger logger = Logger.getLogger(TestFileLogger.class.getName());
```

Factory method: Java library example

```
class SimpleThreadFactory implements ThreadFactory {
   public Thread newThread(Runnable r) {
     return new Thread(r);
   }
}
```

How to refactor?

```
public class Throwable {
        // following method is available from Java 1.0 version.
        // Prints the stack trace as a string to standard output
        // for processing a stack trace,
        // we need to write regular expressions
        public void printStackTrace();
        // other methods omitted
}
```

Extract class refactoring

```
public class Throwable {
        // following method is available from Java 1.0 version.
        // Prints the stack trace as a string to standard output
        // for processing a stack trace,
        // we need to write regular expressions
        public void printStackTrace();
        // other methods omitted
}
```

```
public class Throwable {
        public void printStackTrace();
        public StackTraceElement[] getStackTrace(); // Since 1.4
        // other methods omitted
}
```

```
public final class StackTraceElement {
        public String getFileName();
        public int getLineNumber();
        public String getClassName();
        public String getMethodName();
        public boolean isNativeMethod();
}
```

Scenario

```
class Plus extends 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?

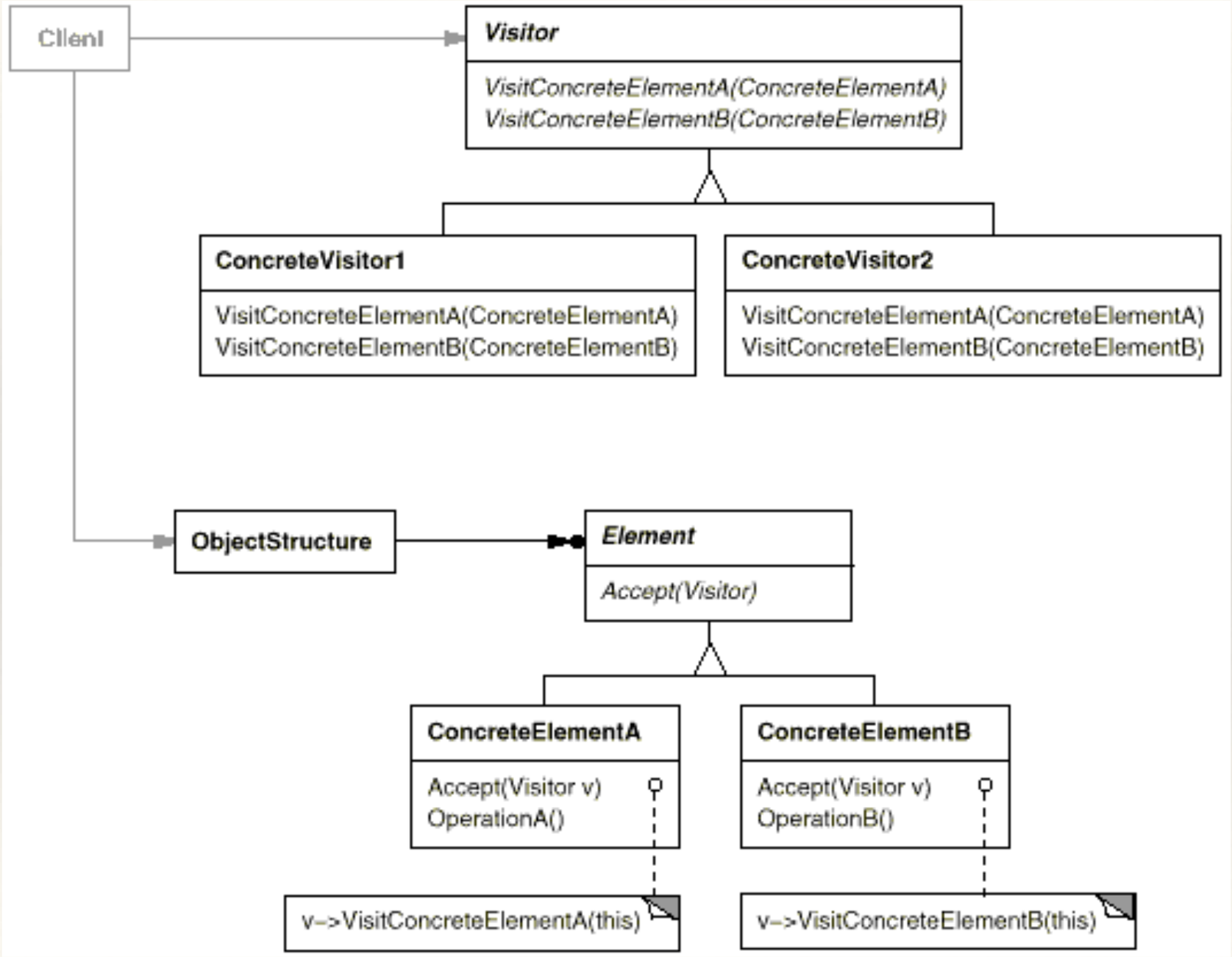
Group exercise - Understanding visitor pattern

A solution using Visitor pattern

```
class Plus extends Expr {
        private Expr left, right;
        public Plus(Expr arg1, Expr arg2) {
                left = arg1;
                right = arg2;
        }
        public Expr getLeft() {
                return left;
        }
        public Expr getRight() {
                return right;
        }
        public void accept(Visitor v) {
                v.visit(this);
        }
}
```

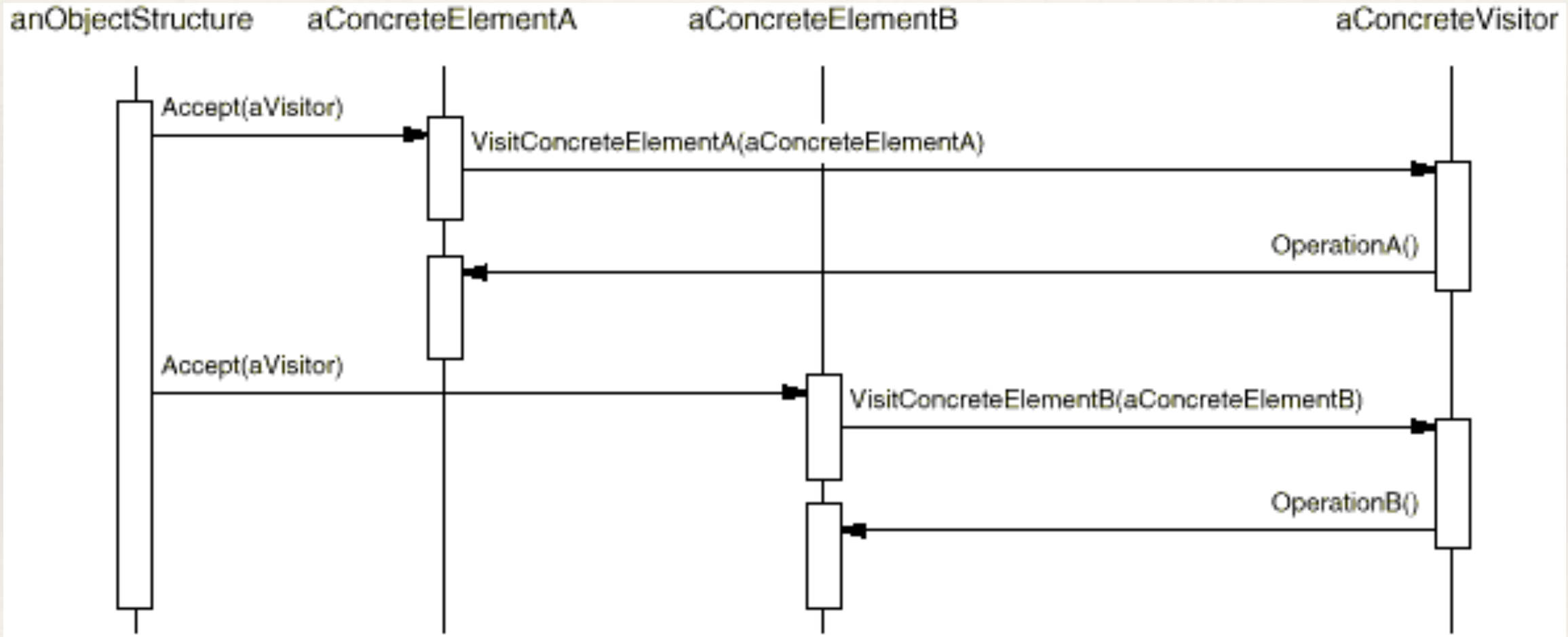
```
class DOTNETVisitor extends Visitor {
        public void visit(Constant arg) {
                System.out.println("ldarg " + arg.getVal());
        }
        public void visit(Plus plus) {
                genCode(plus.getLeft());
                genCode(plus.getRight());
                System.out.println("add");
        }
        public void visit(Sub sub) {
                genCode(sub.getLeft());
                genCode(sub.getRight());
                System.out.println("sub");
        }
        public void genCode(Expr expr) {
                expr.accept(this);
        }
}
```

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

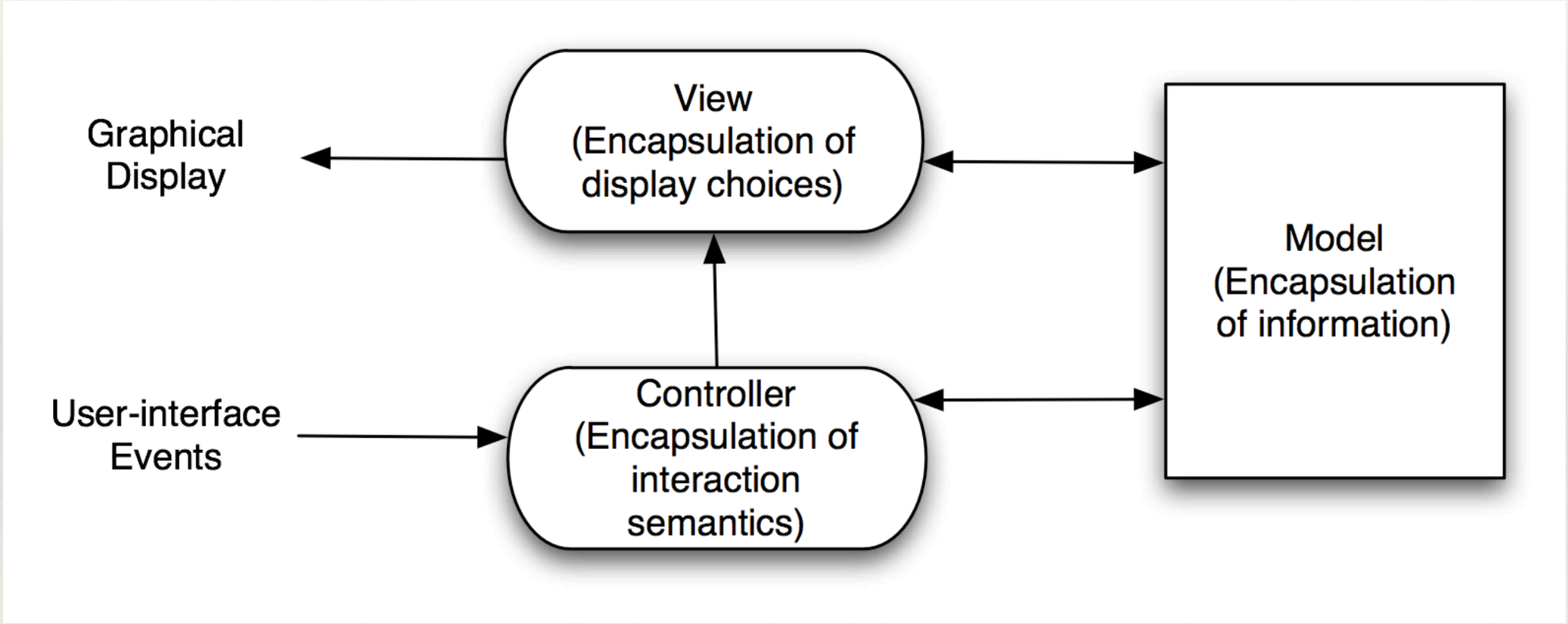
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations

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

Visitor pattern: JDK example

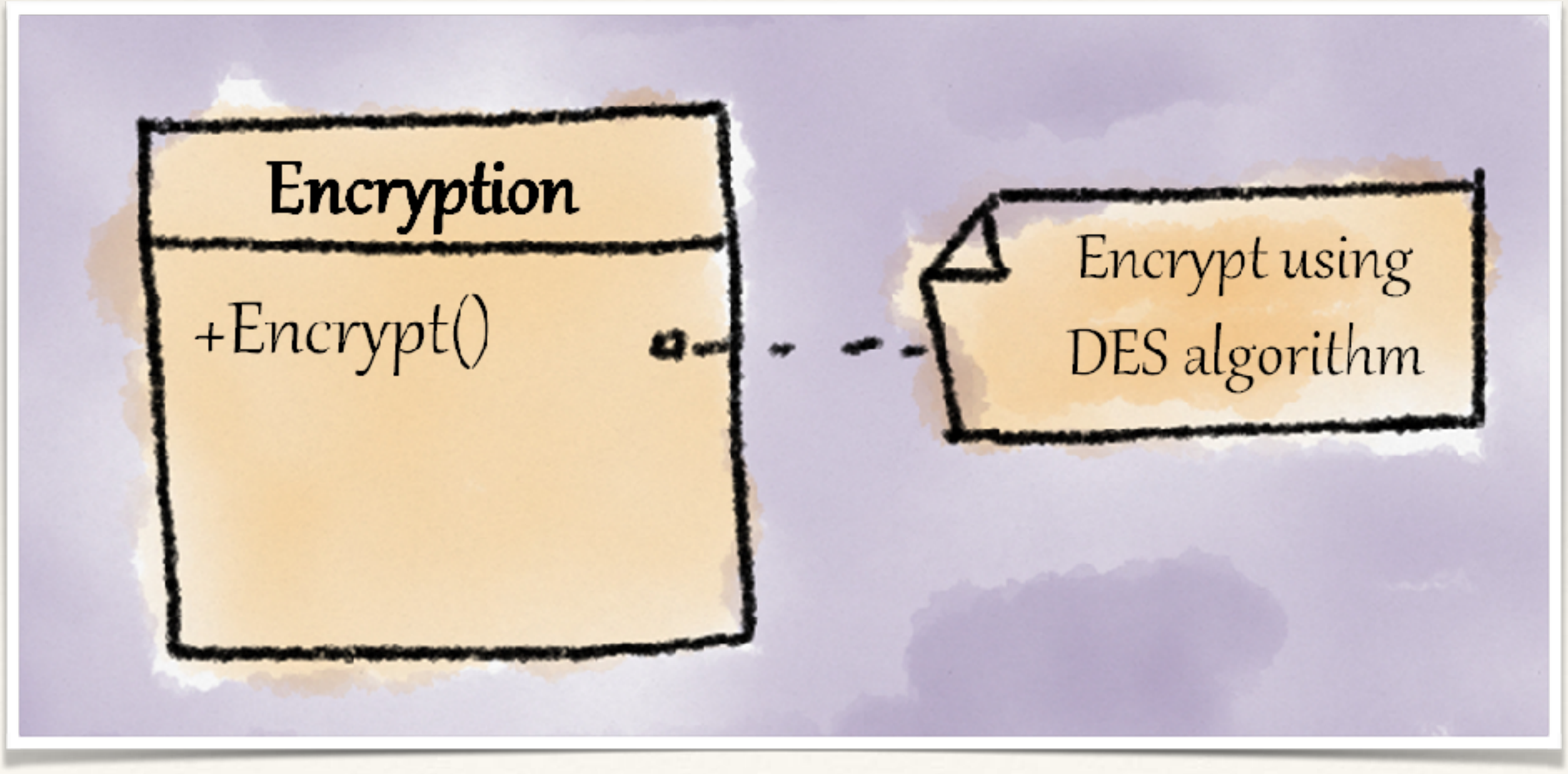
```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

class MyFileVisitor extends SimpleFileVisitor<Path> {
   public FileVisitResult visitFile(Path path, BasicFileAttributes fileAttributes){
      System.out.println("file name:" + path.getFileName());
      return FileVisitResult.CONTINUE;
   }
   public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes fileAttributes){
      System.out.println("----------Directory name:" + path + "----------");
      return FileVisitResult.CONTINUE;
   }
}

public class FileTreeWalk {
   public static void main(String[] args) {
      if(args.length != 1) {
         System.out.println("usage: FileWalkTree <source-path>");
         System.exit(-1);
      }
      Path pathSource = Paths.get(args[0]);
      try {
         Files.walkFileTree(pathSource, new MyFileVisitor());
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Separating concerns in MVC



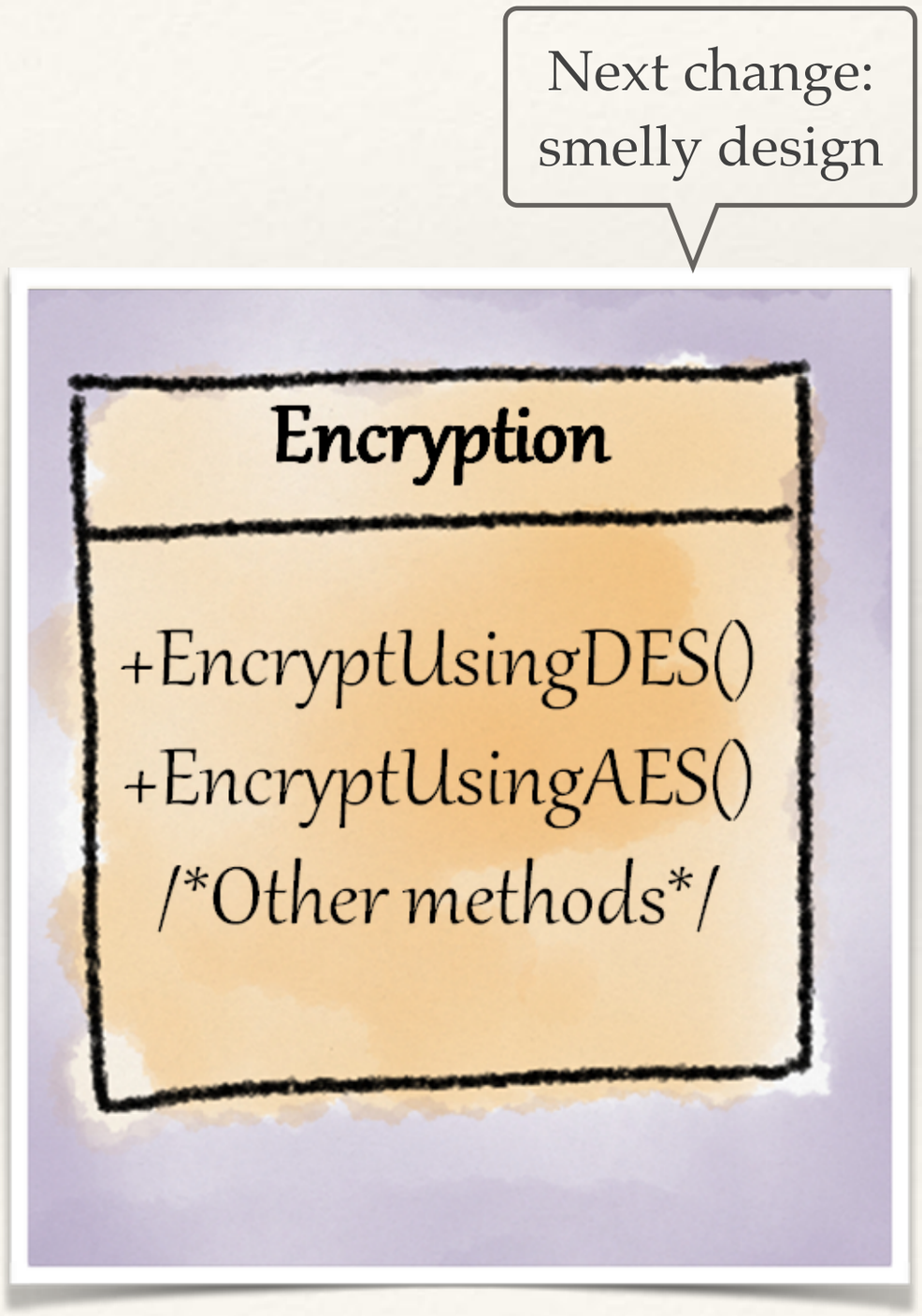
Real scenario #1

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?



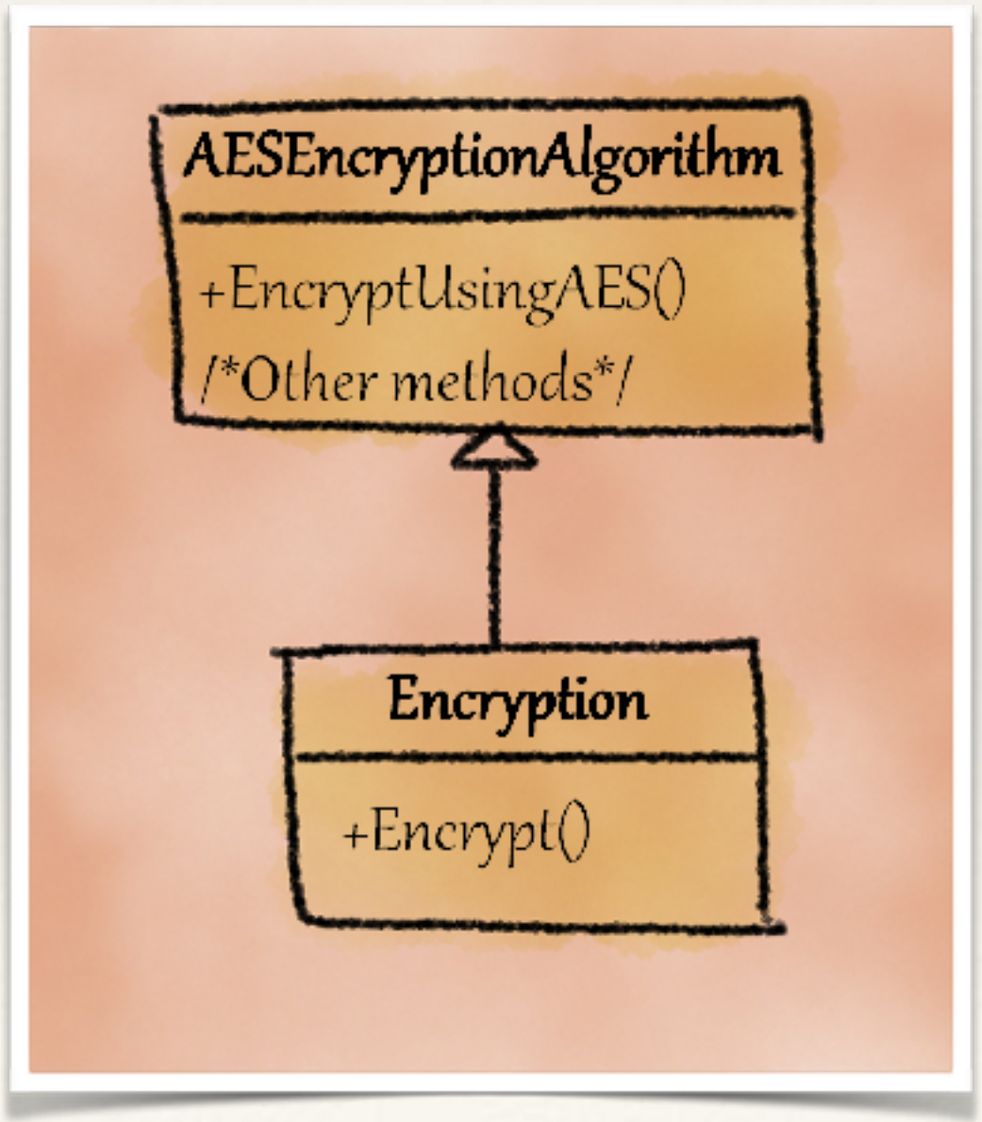
Time to refactor!

Three strikes and you refactor

Martin Fowler

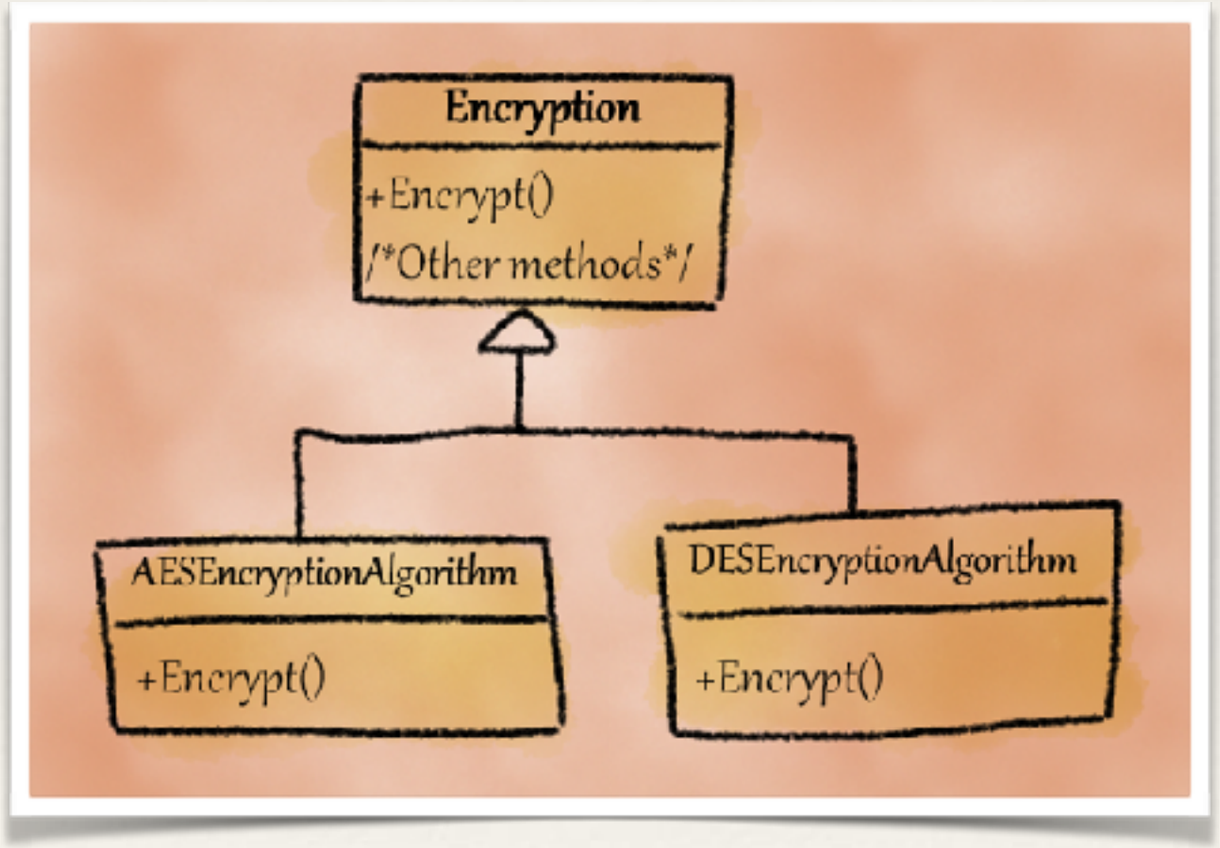
Potential solution #1?

Broken hierarchy!

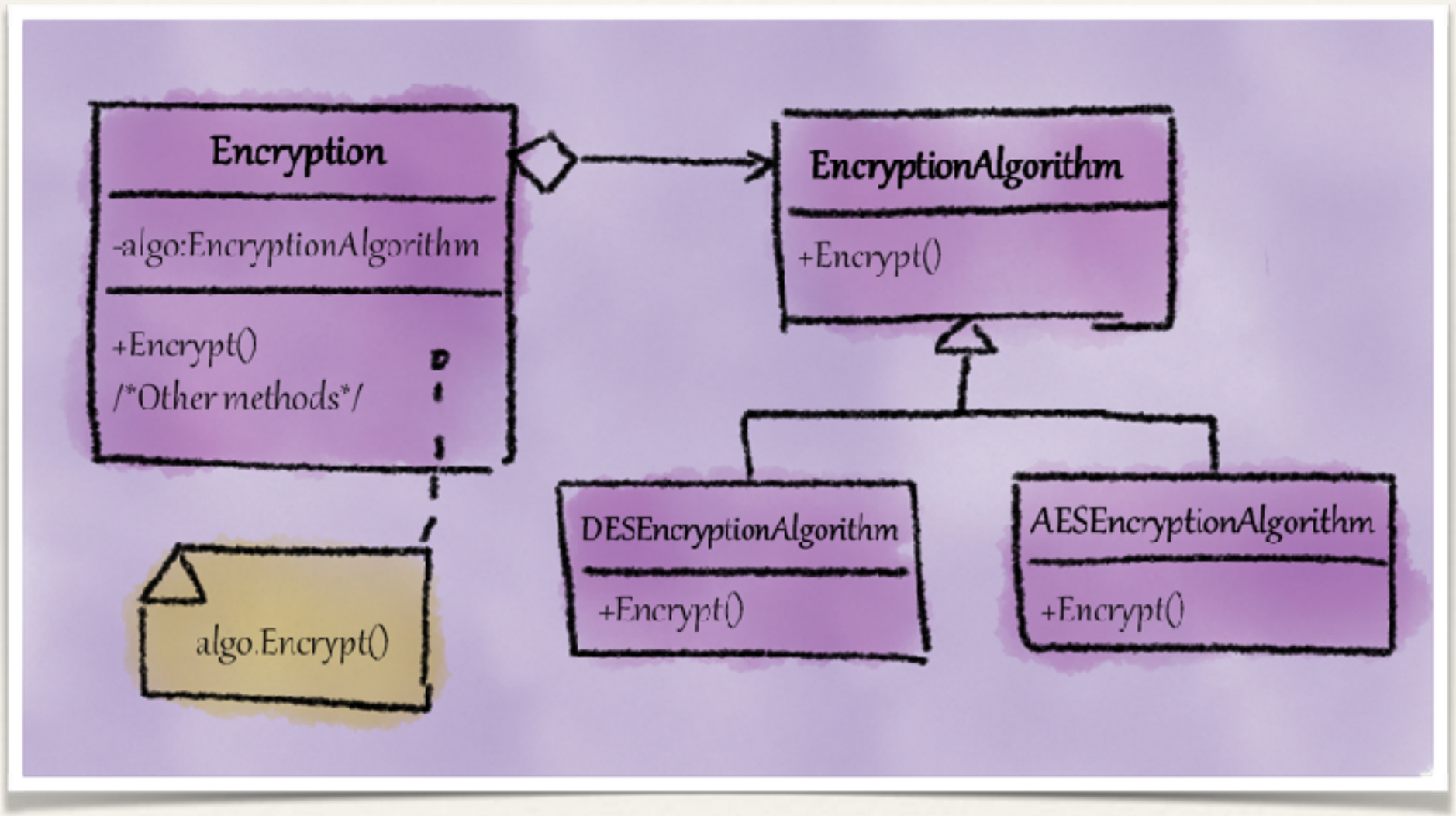


Potential solution #2?

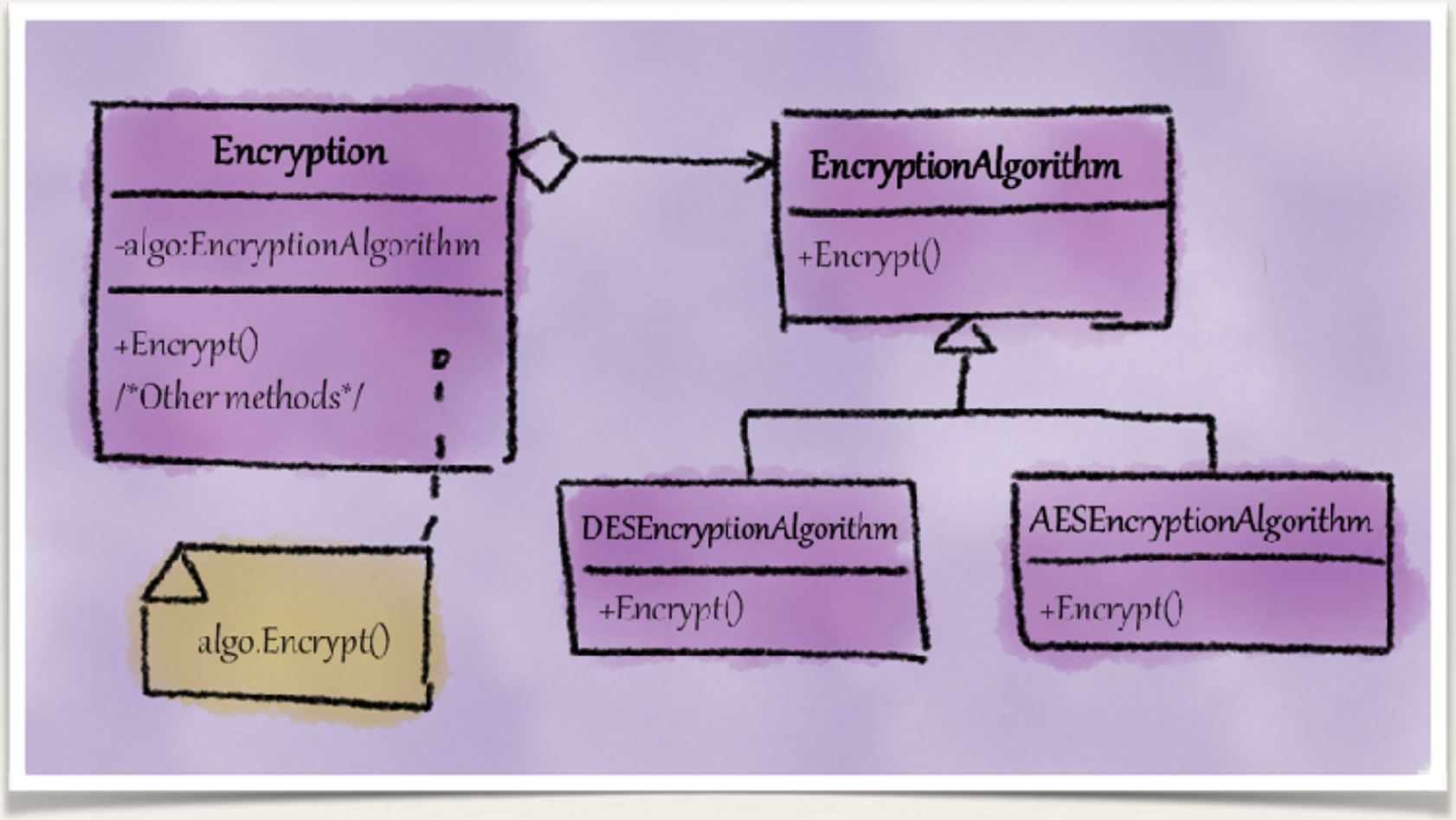
Algorithms not reusable!



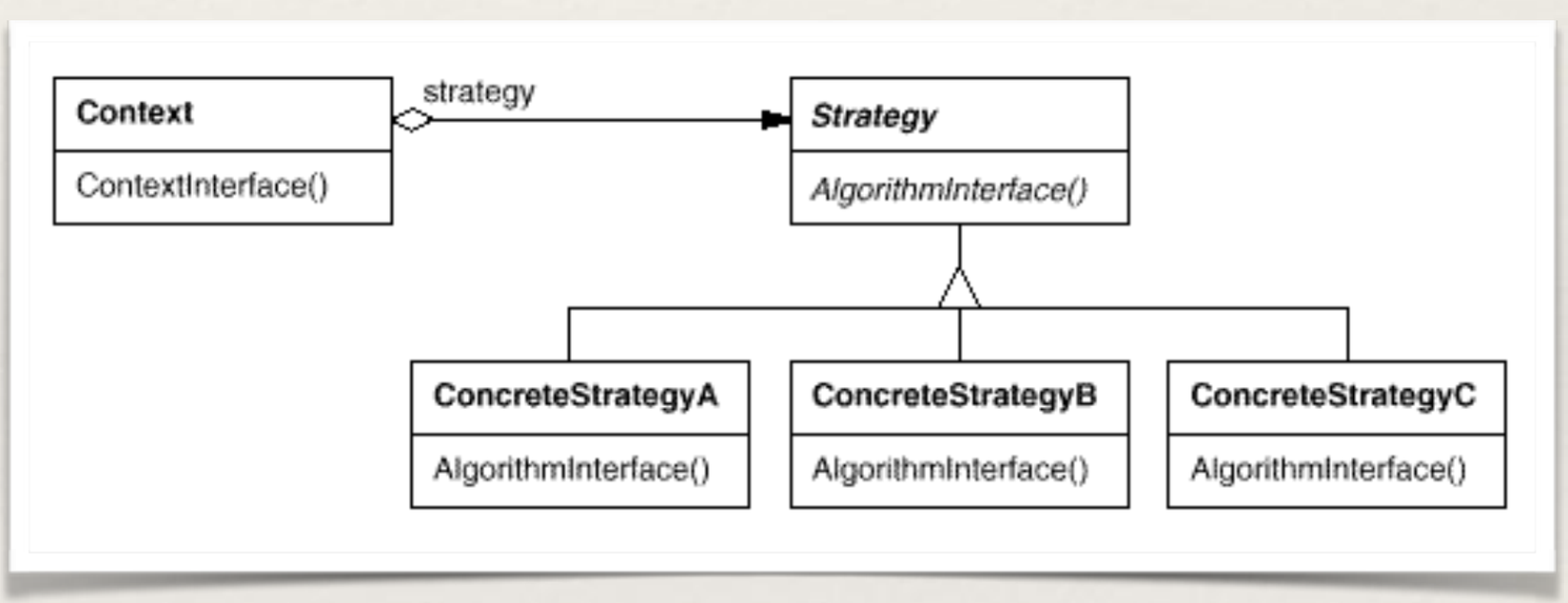
Potential solution #3?



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

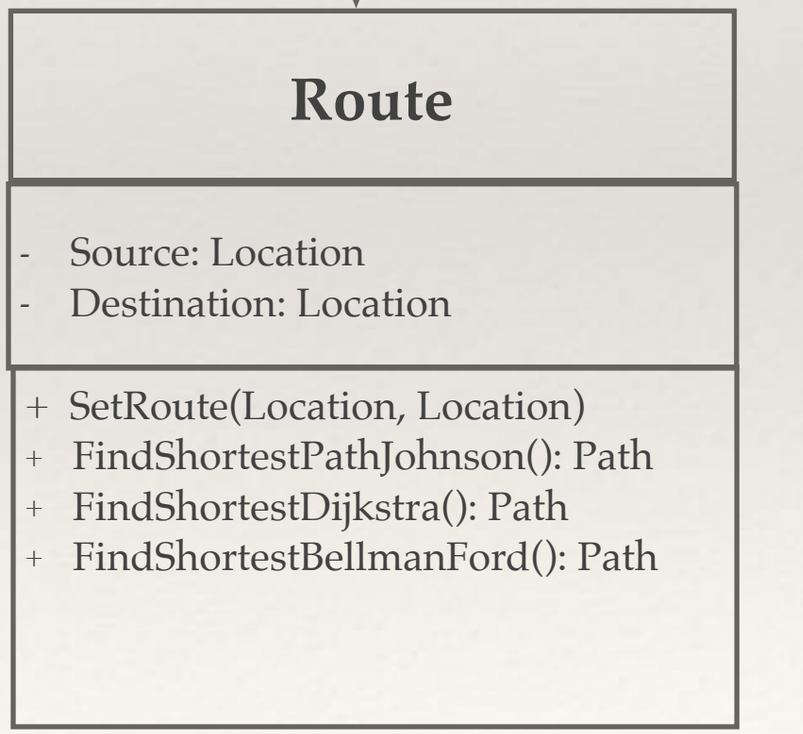
- Useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm that suits its current need

- 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

Scenario

- Consider a Route class in an application like Google Maps
- For finding shortest path from source to destination, many algorithms can be used
- The problem is that these algorithms get embedded into Route class and cannot be reused easily (smell!)

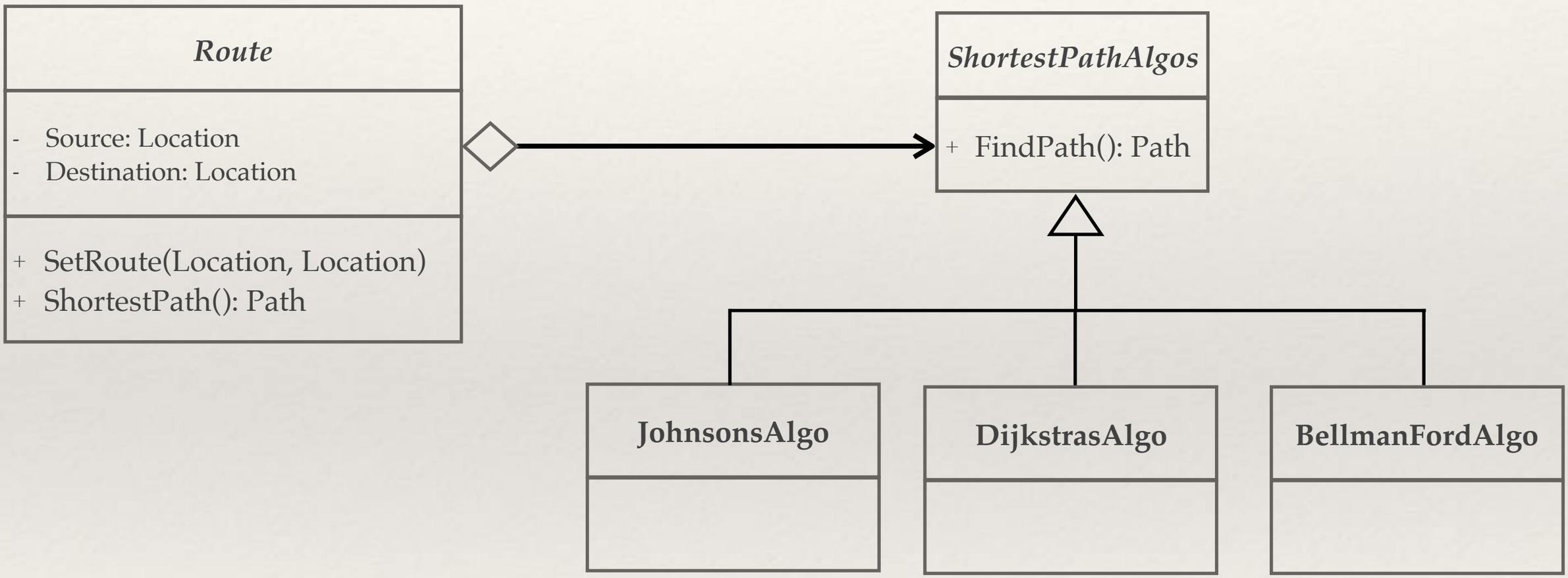
How will you refactor such that
a) Support for shortest path algorithm can be added easily?
b) Separate path finding logic from dealing with location information.



Hands-on exercise

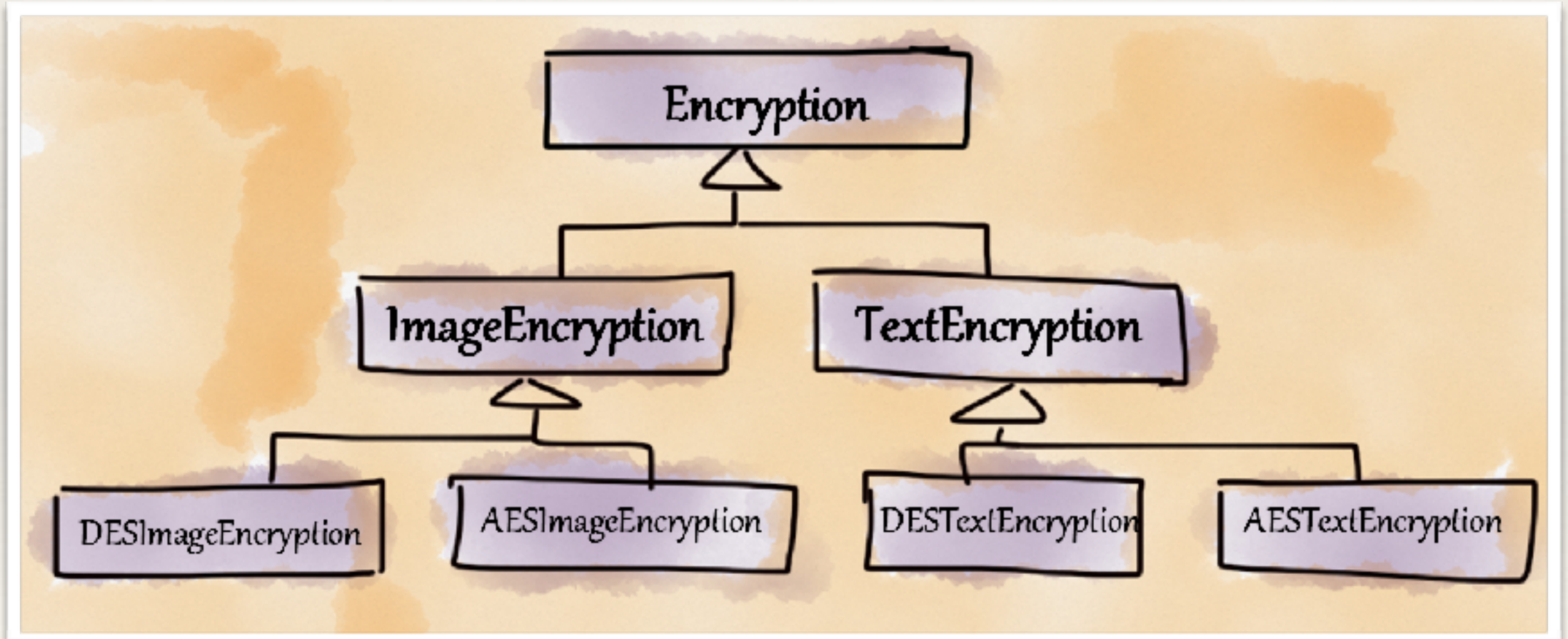
- Try out Route.java
- Refactor to eliminate switch-case statement

How about this solution?



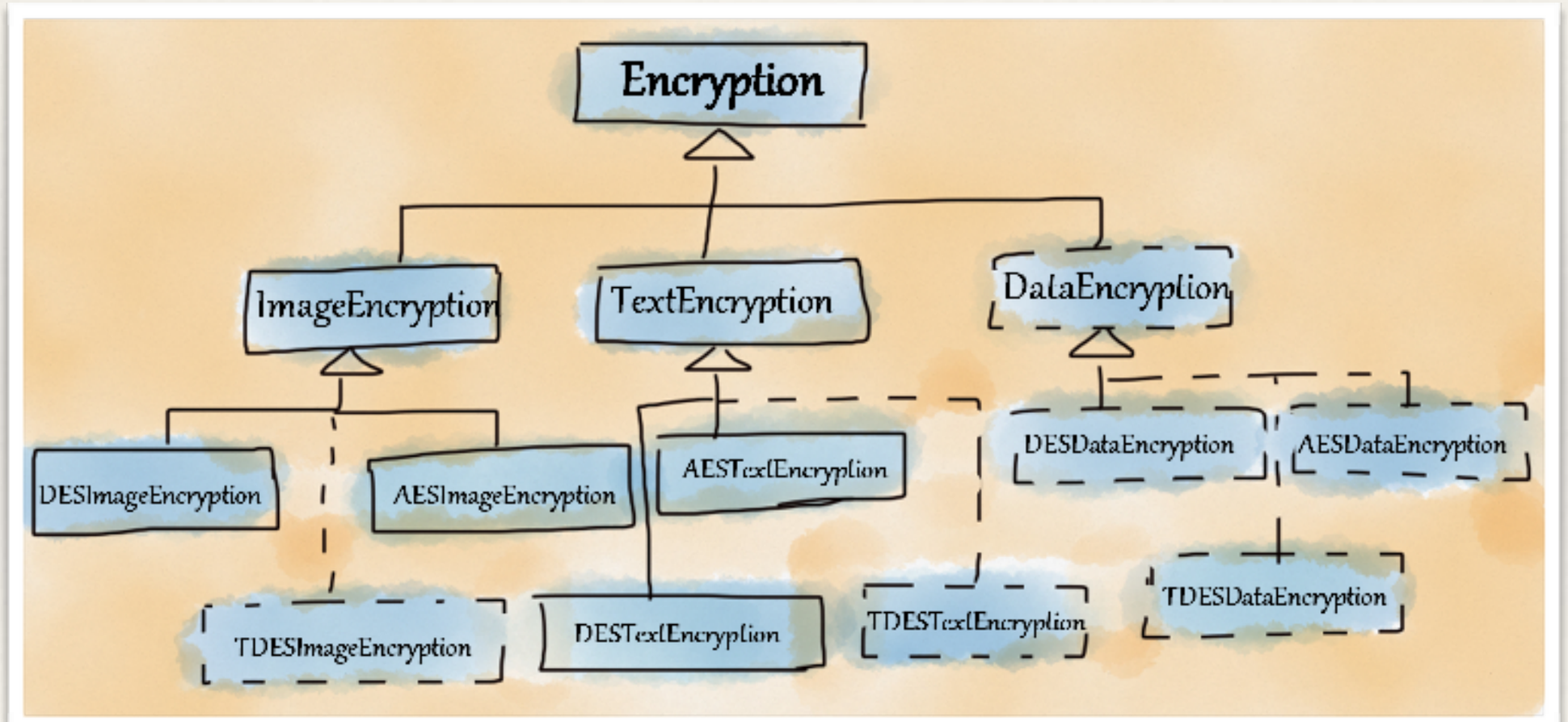
Real scenario #2

Initial design

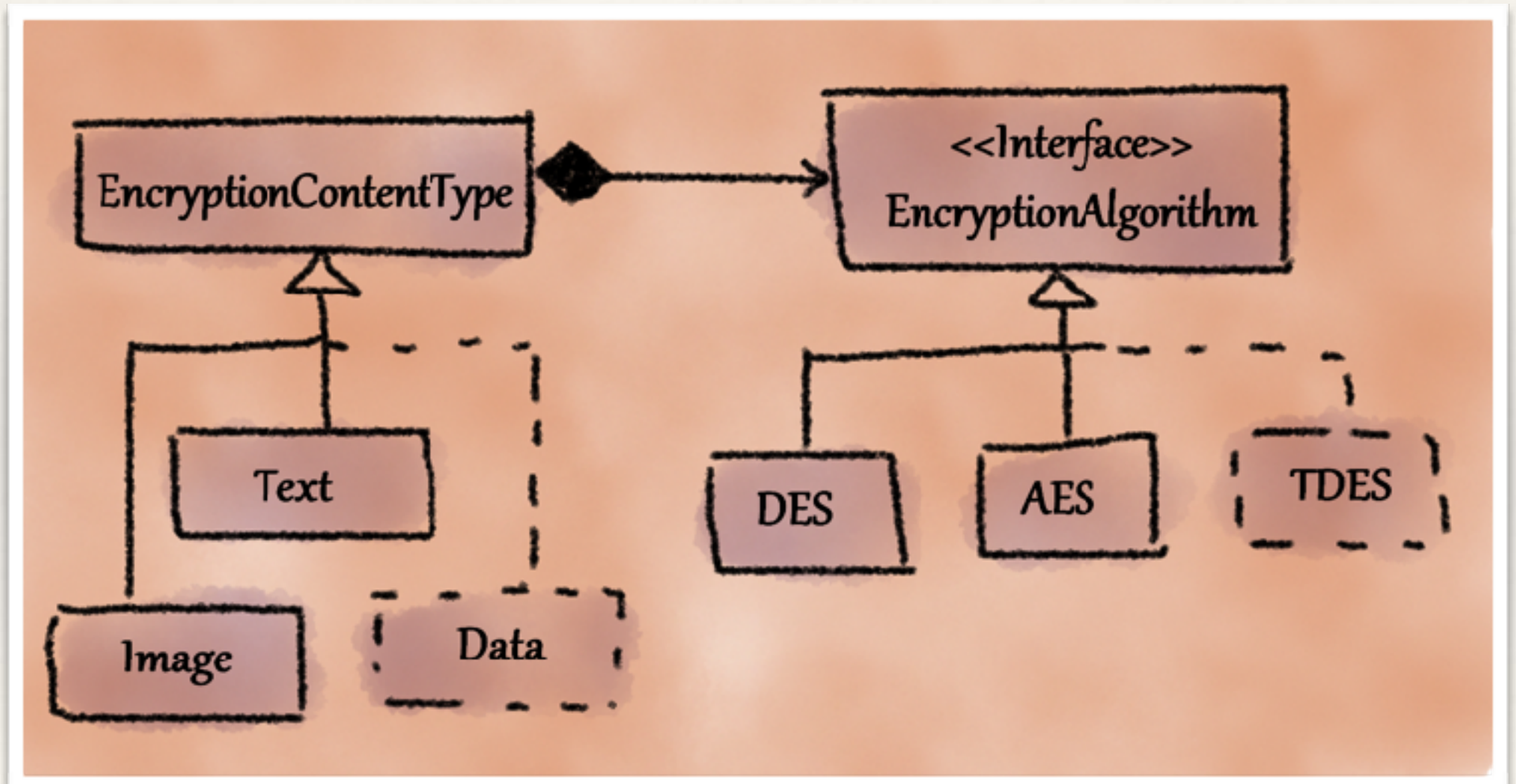


Real scenario #2

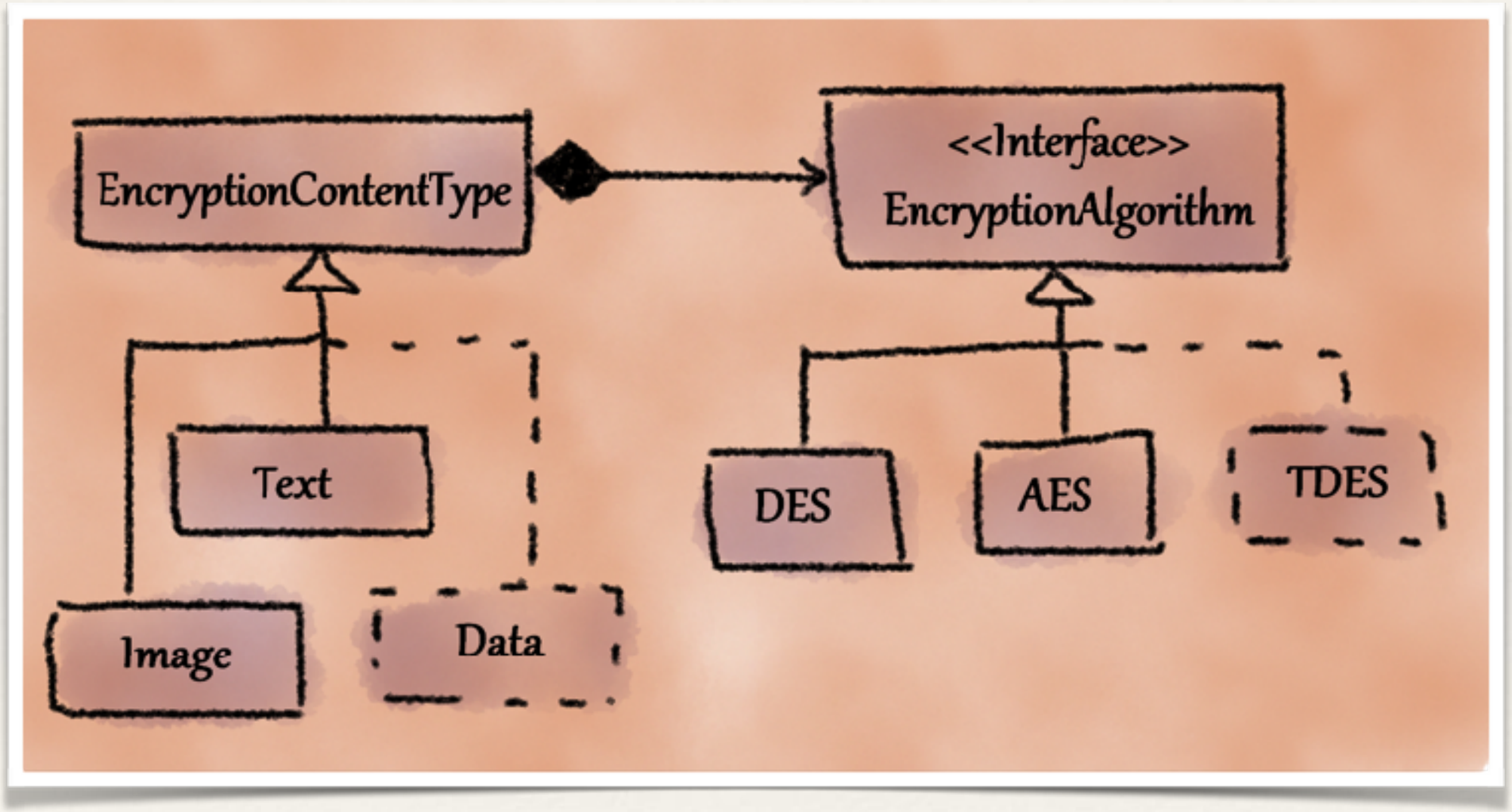
How to add support for new content types and/or algorithms?



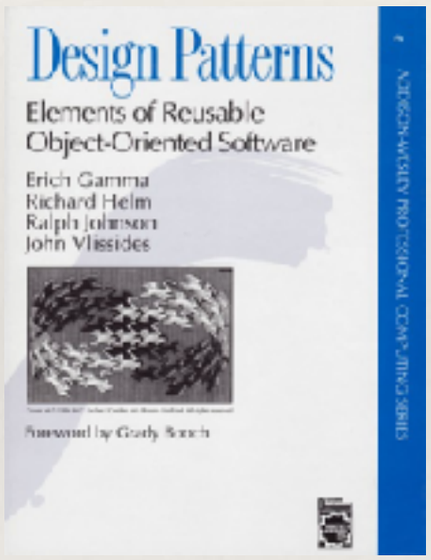
How about this solution?

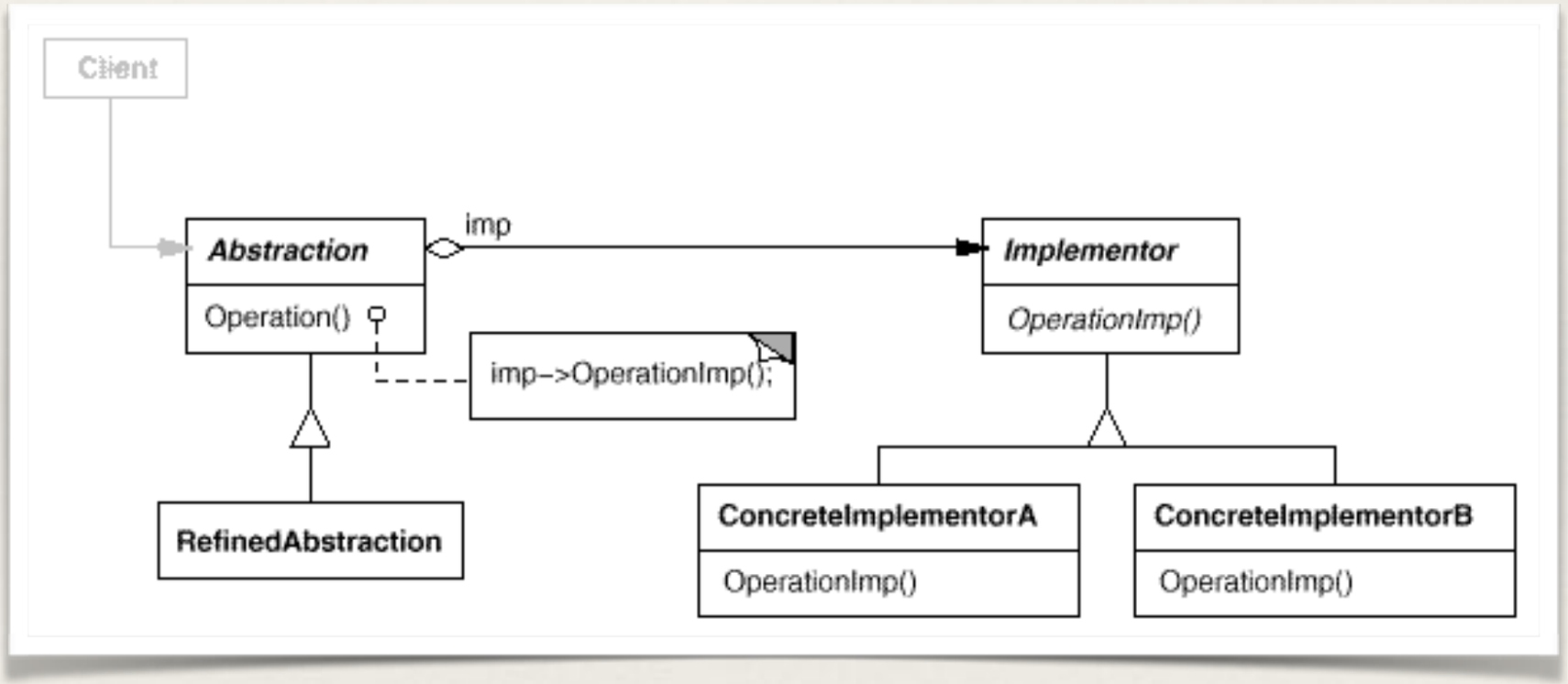


Can you identify the pattern structure?



You're right: Its Bridge pattern structure!





More design patterns to explore

Creational	Structural	Behavioral
• Abstract factory • Builder • Factory method • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of responsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • Template method • Visitor
deals with object creation	deals with composition of classes or objects	deals with interaction between objects/ classes and distribution of responsibility

Wait, what principle did we apply?

Open Closed Principle (OCP)

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

Bertrand Meyer

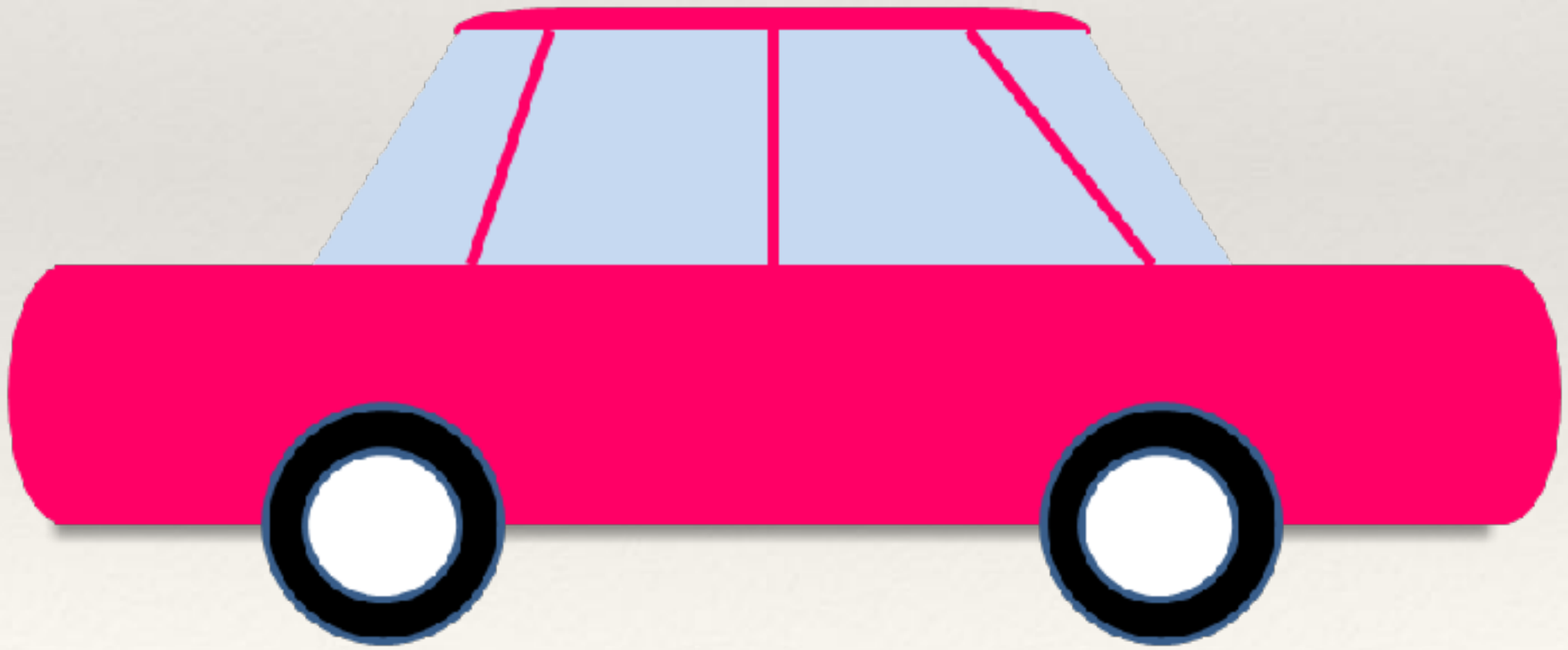
Variation Encapsulation Principle (VEP)

Encapsulate the concept that varies

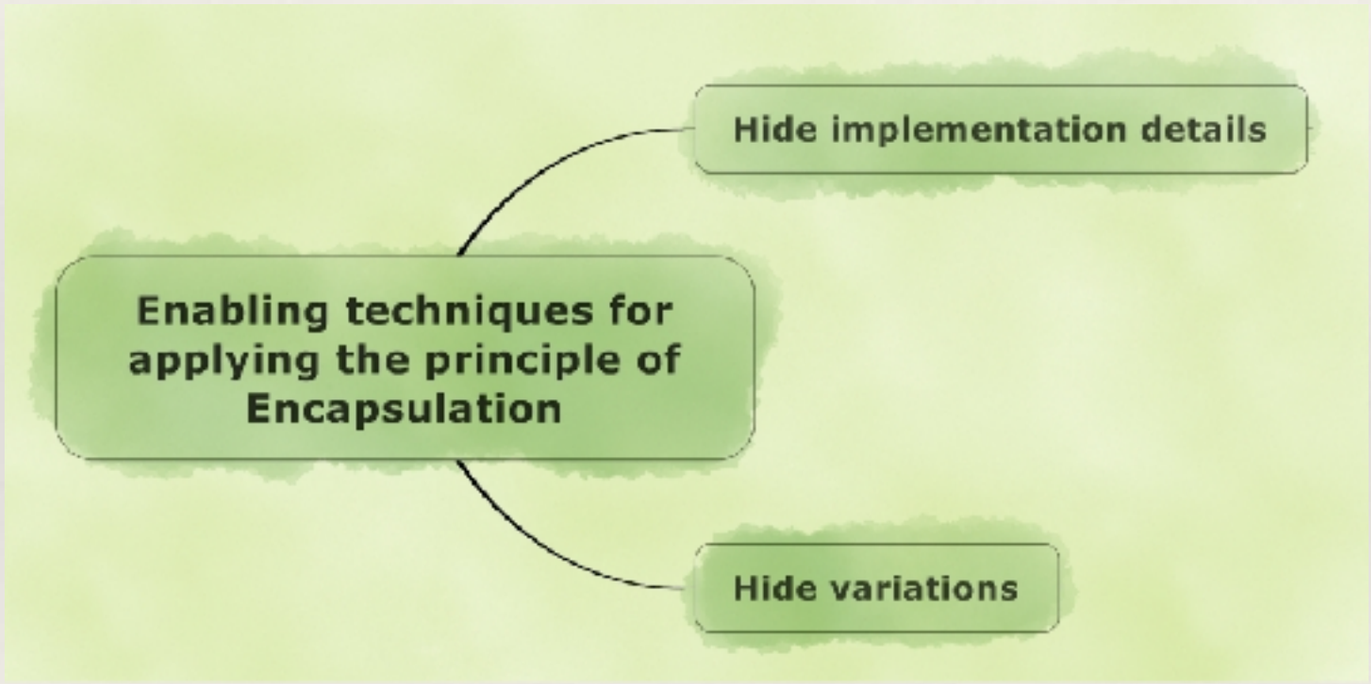
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

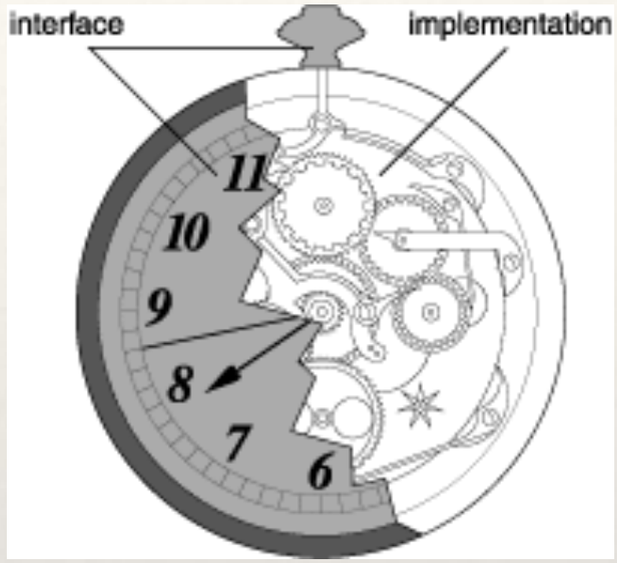
Fundamental principle: *Encapsulation*

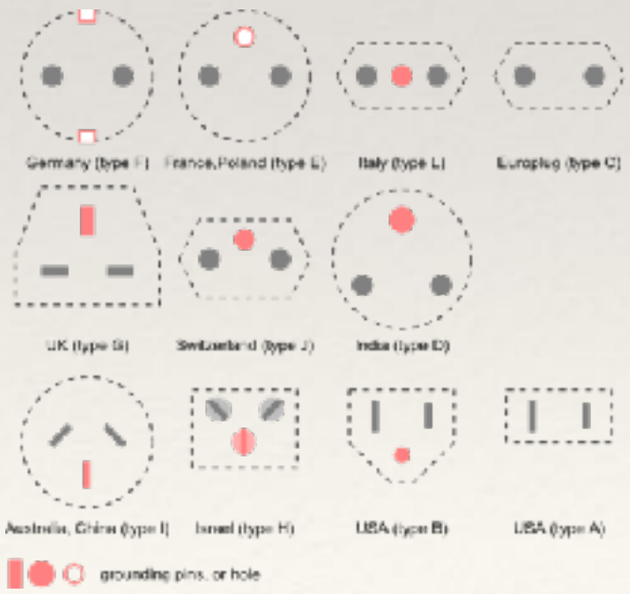
The principle of encapsulation advocates separation of concerns and information hiding through techniques such as hiding implementation details of abstractions and hiding variations



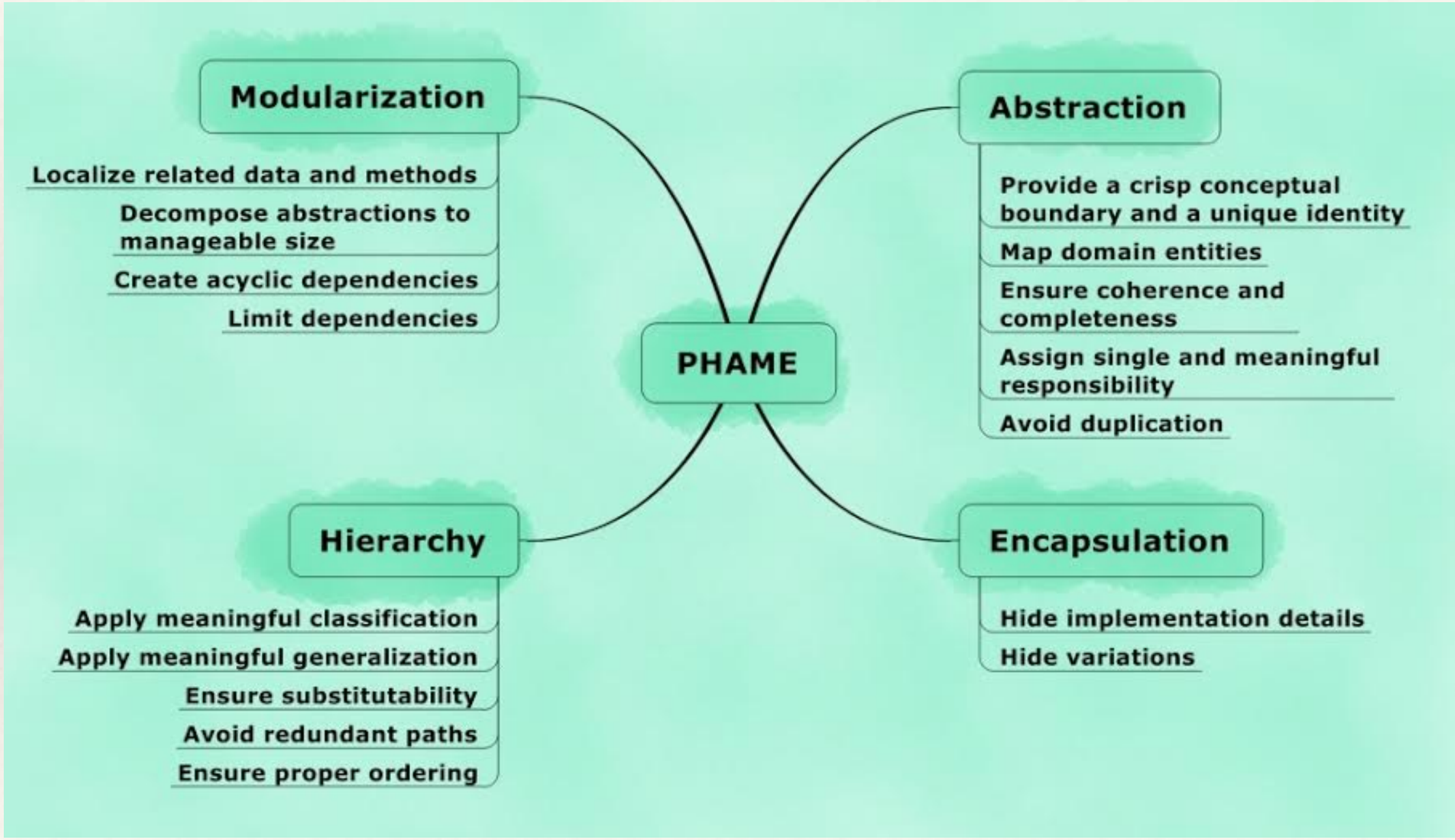
Enabling techniques for encapsulation







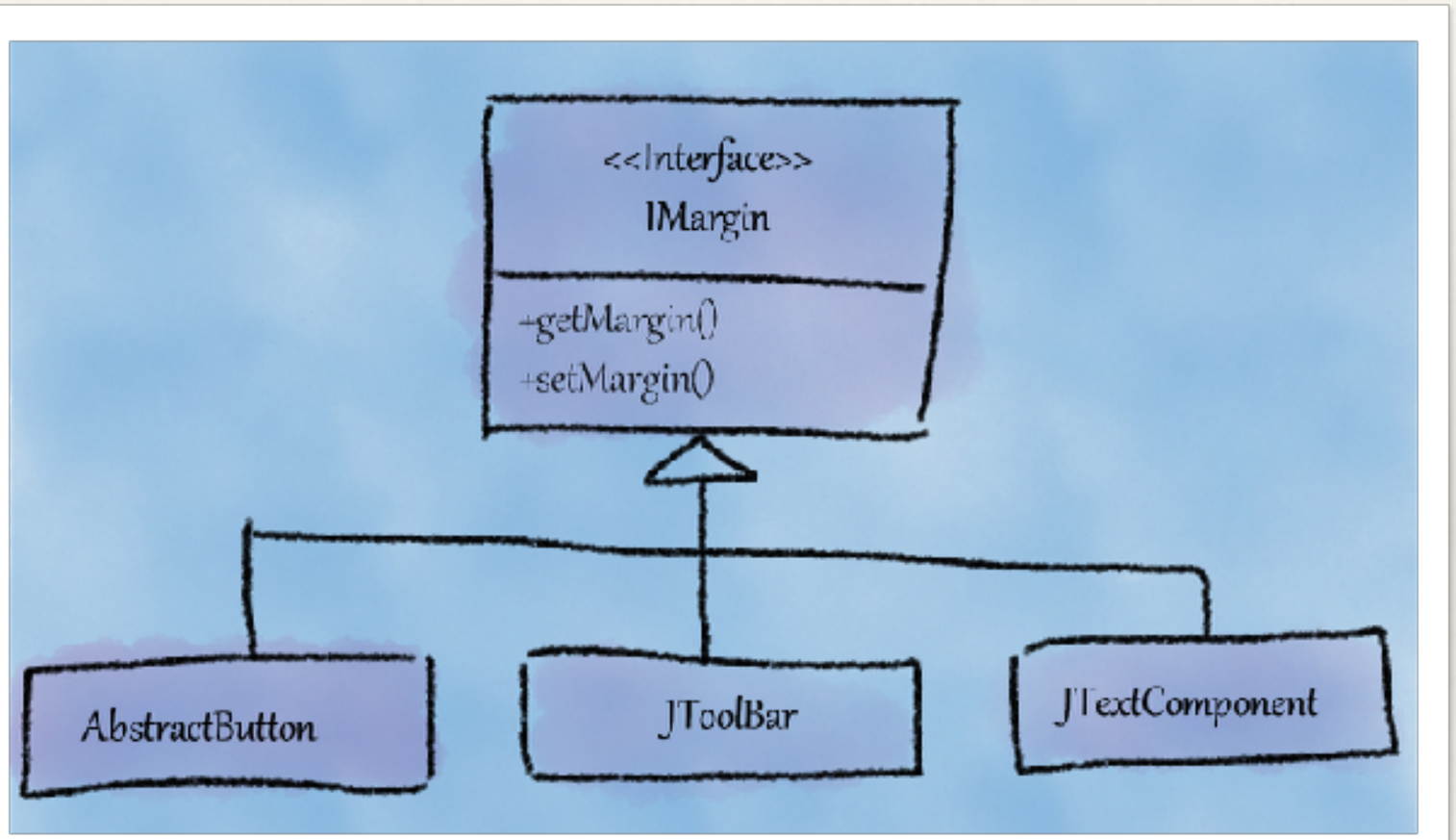
Design principles and enabling techniques



What's that smell?

```
public Insets getBorderInsets(Component c, Insets insets){
        Insets margin = null;
         // Ideally we'd have an interface defined for classes which
        // support margins (to avoid this hackery), but we've
        // decided against it for simplicity
        //
        if (c instanceof AbstractButton) {
                margin = ((AbstractButton)c).getMargin();
        } else if (c instanceof JToolBar) {
                margin = ((JToolBar)c).getMargin();
        } else if (c instanceof JTextComponent) {
                margin = ((JTextComponent)c).getMargin();
        }
        // rest of the code omitted …
```

Refactoring



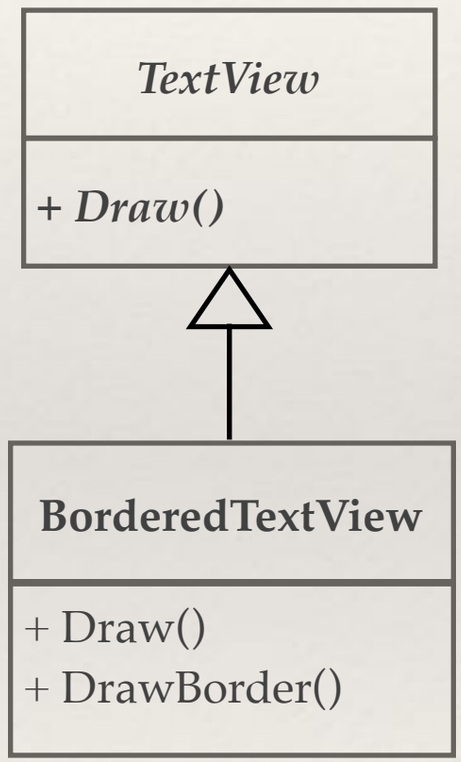
Refactoring

```
if (c instanceof AbstractButton) {
        margin = ((AbstractButton)c).getMargin();
} else if (c instanceof JToolBar) {
        margin = ((JToolBar)c).getMargin();
} else if (c instanceof JTextComponent) {
        margin = ((JTextComponent)c).getMargin();
}
```

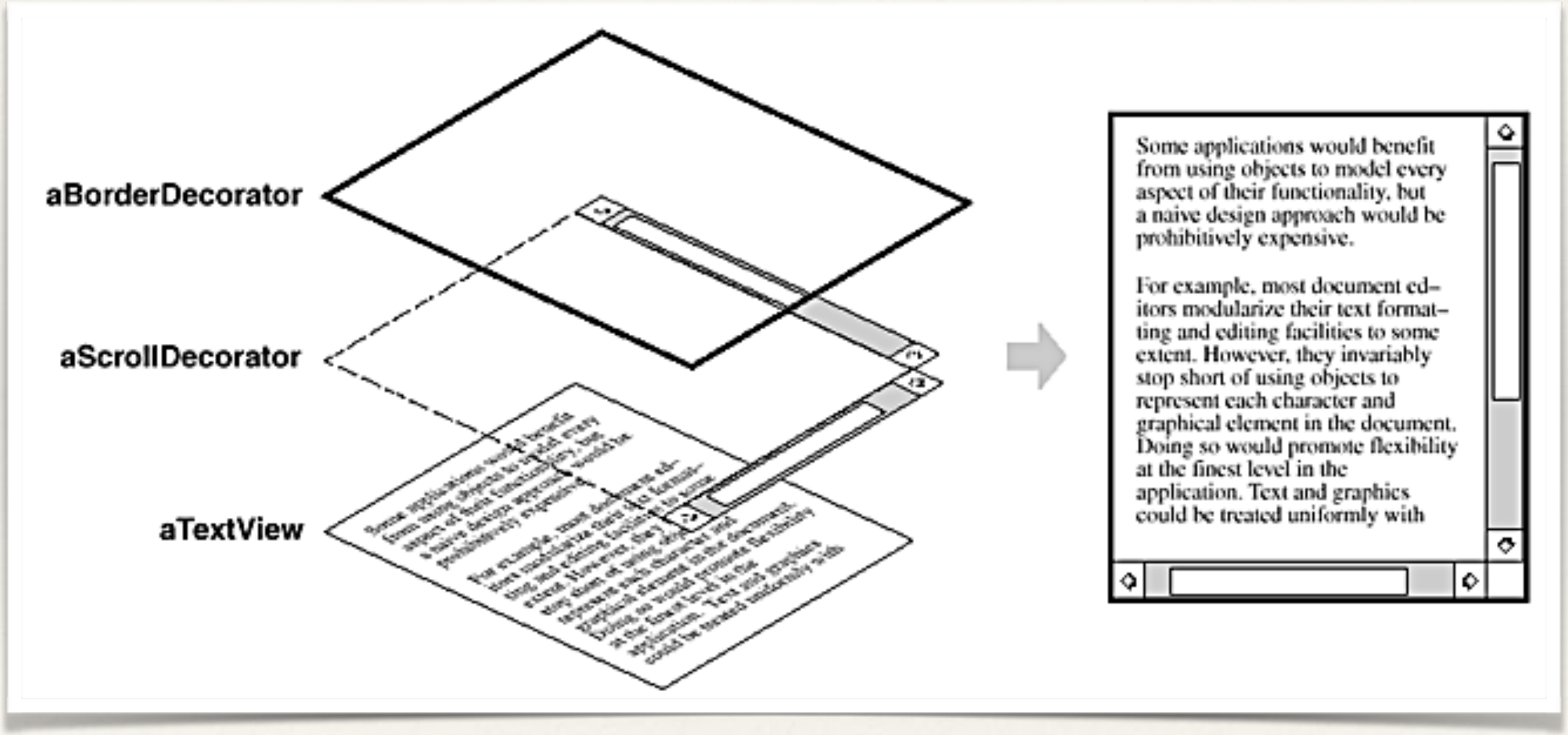
```
margin = c.getMargin();
```

Scenario

Initial design

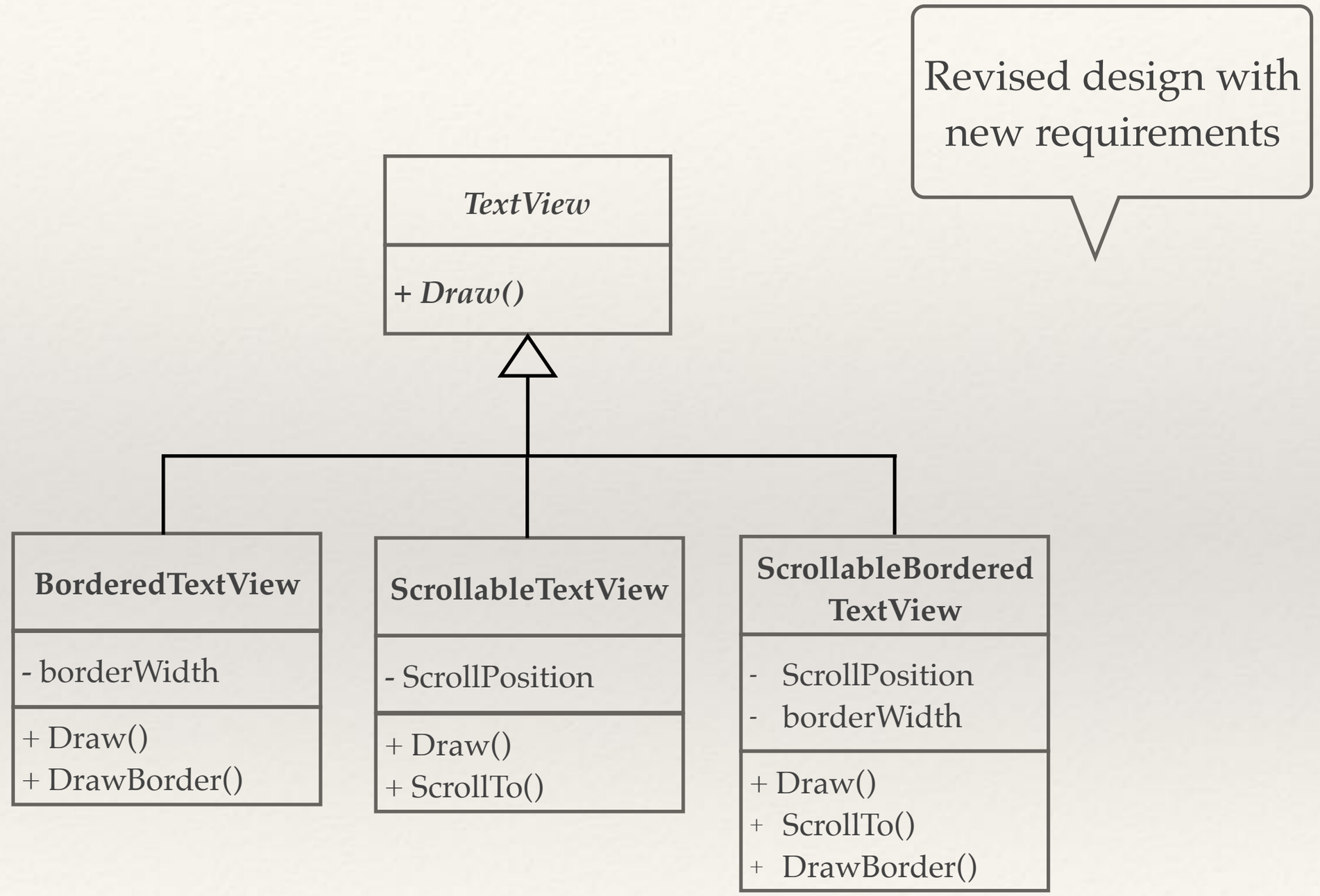


Scenario



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

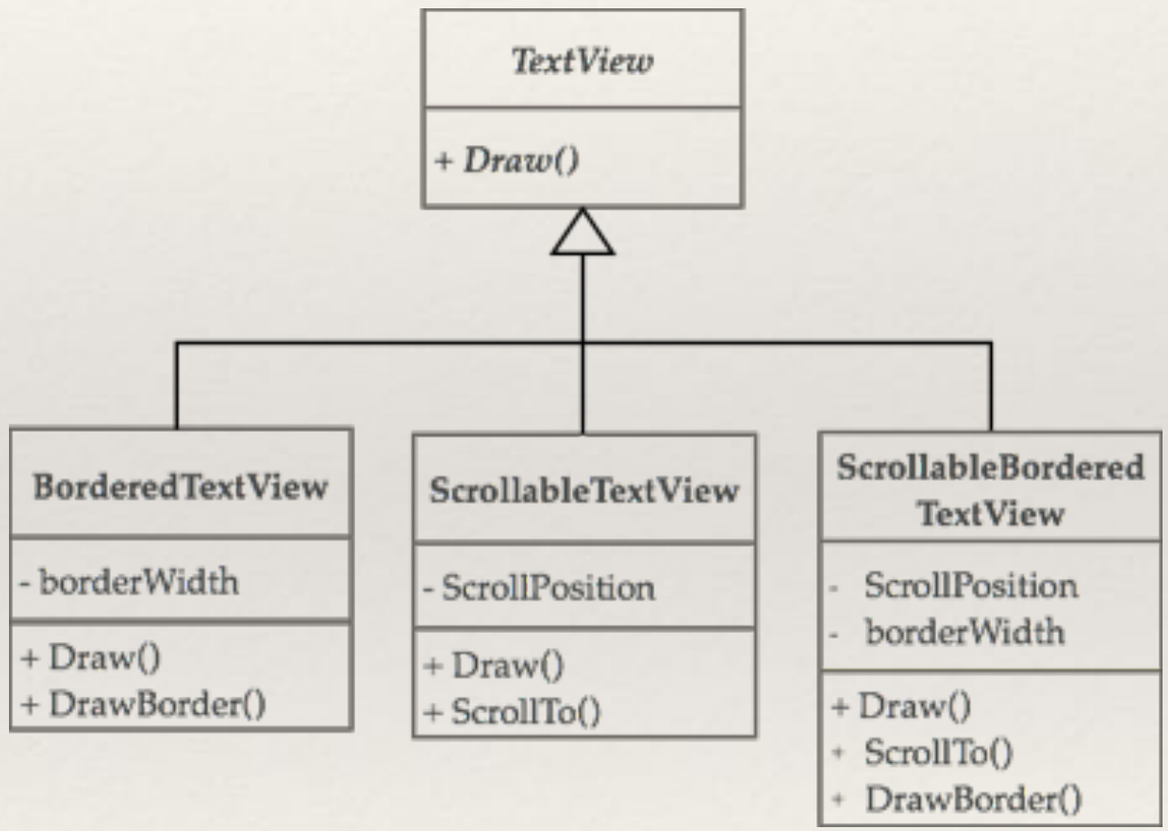
Supporting new requirements



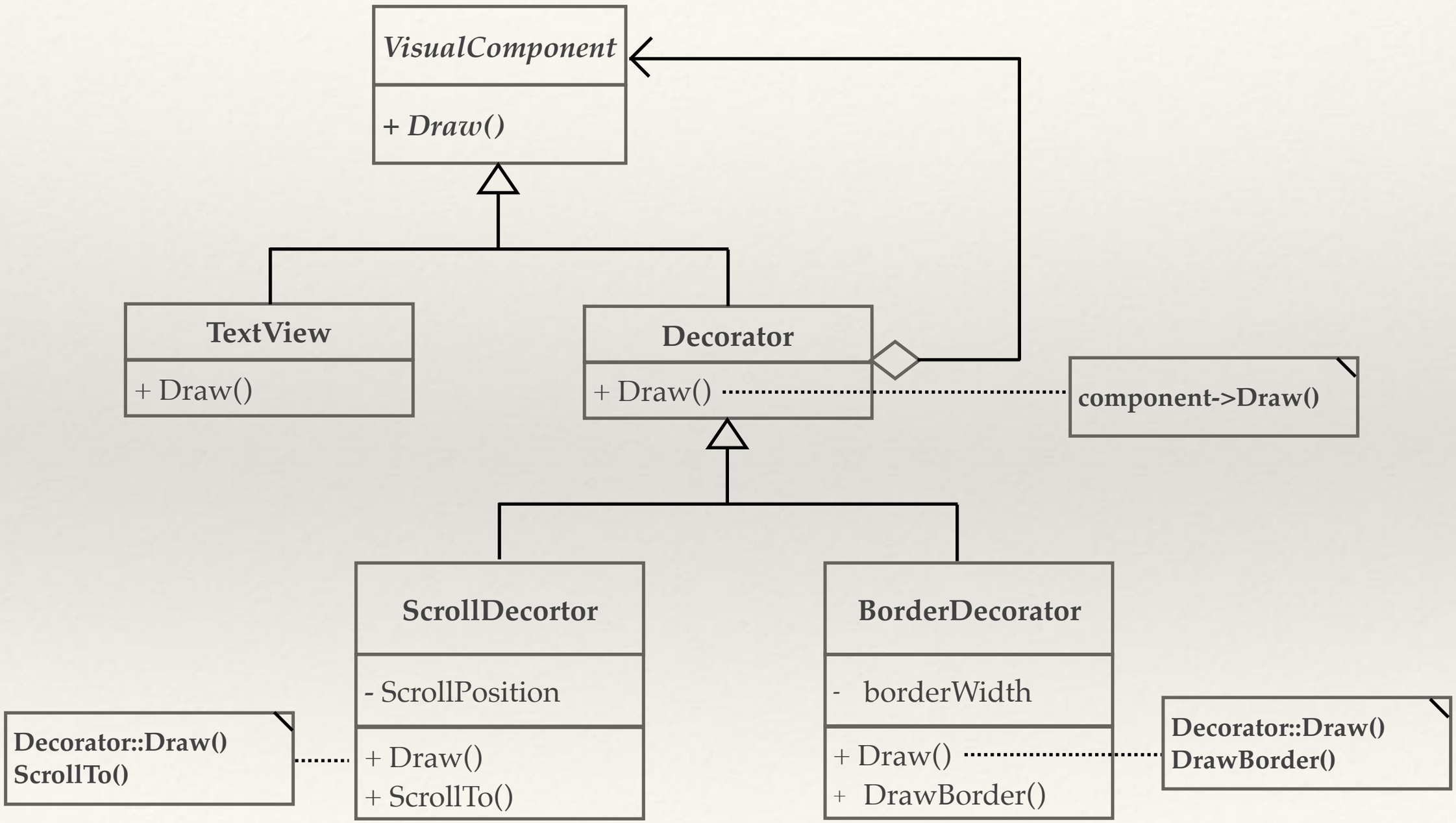
Scenario

Next change: smelly design

- 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?

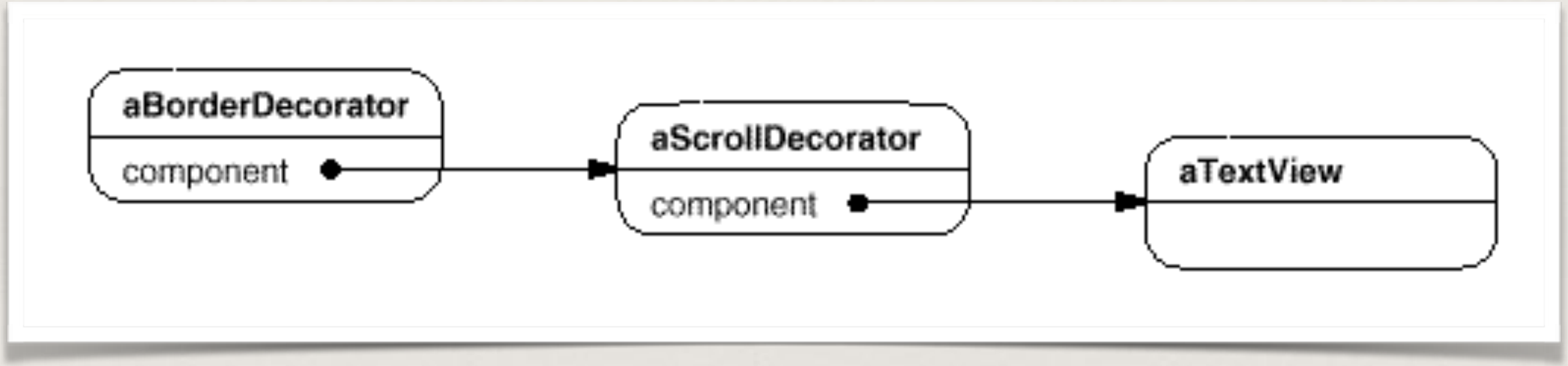


How about this solution?



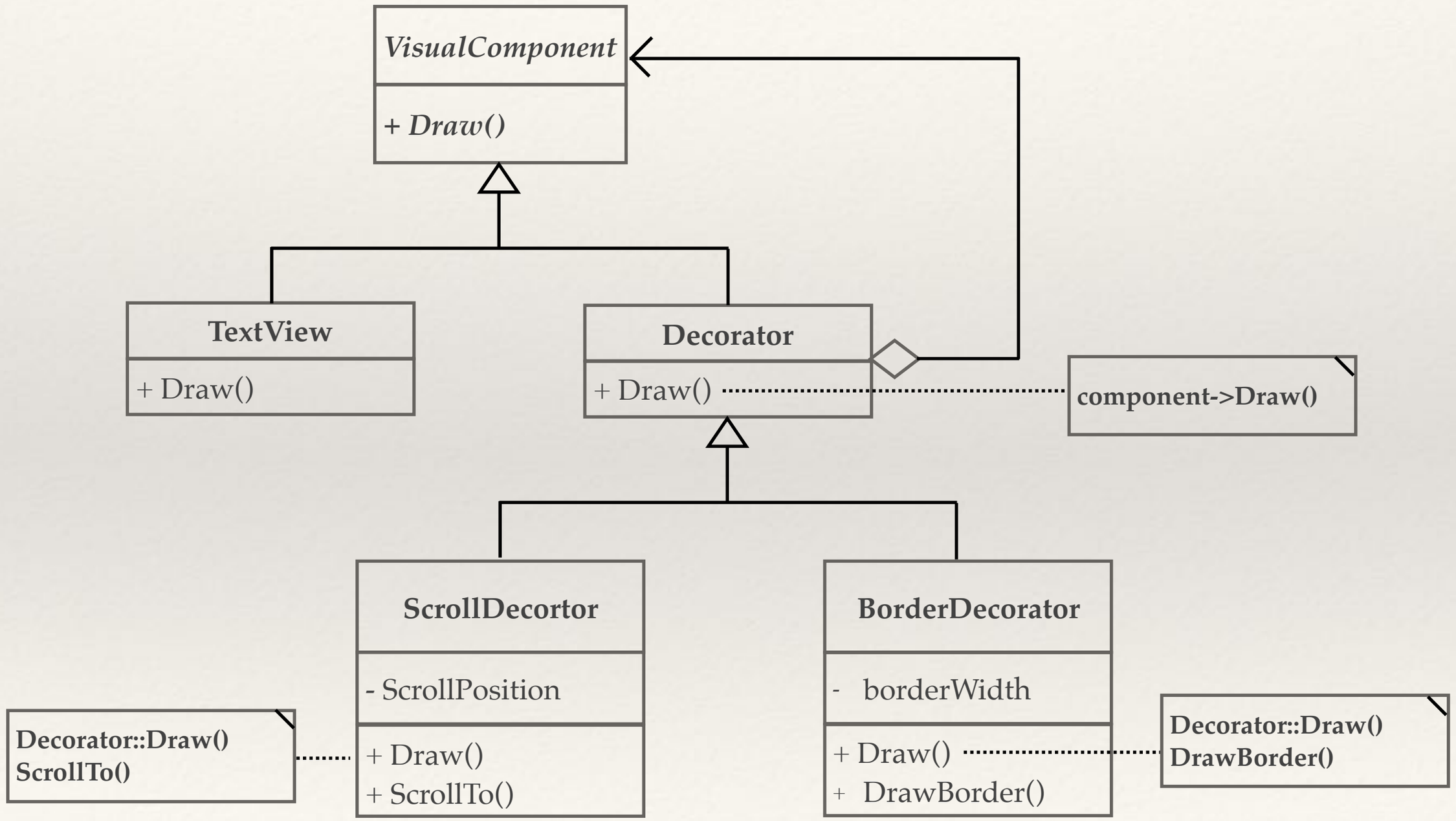
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

At runtime (object diagram)



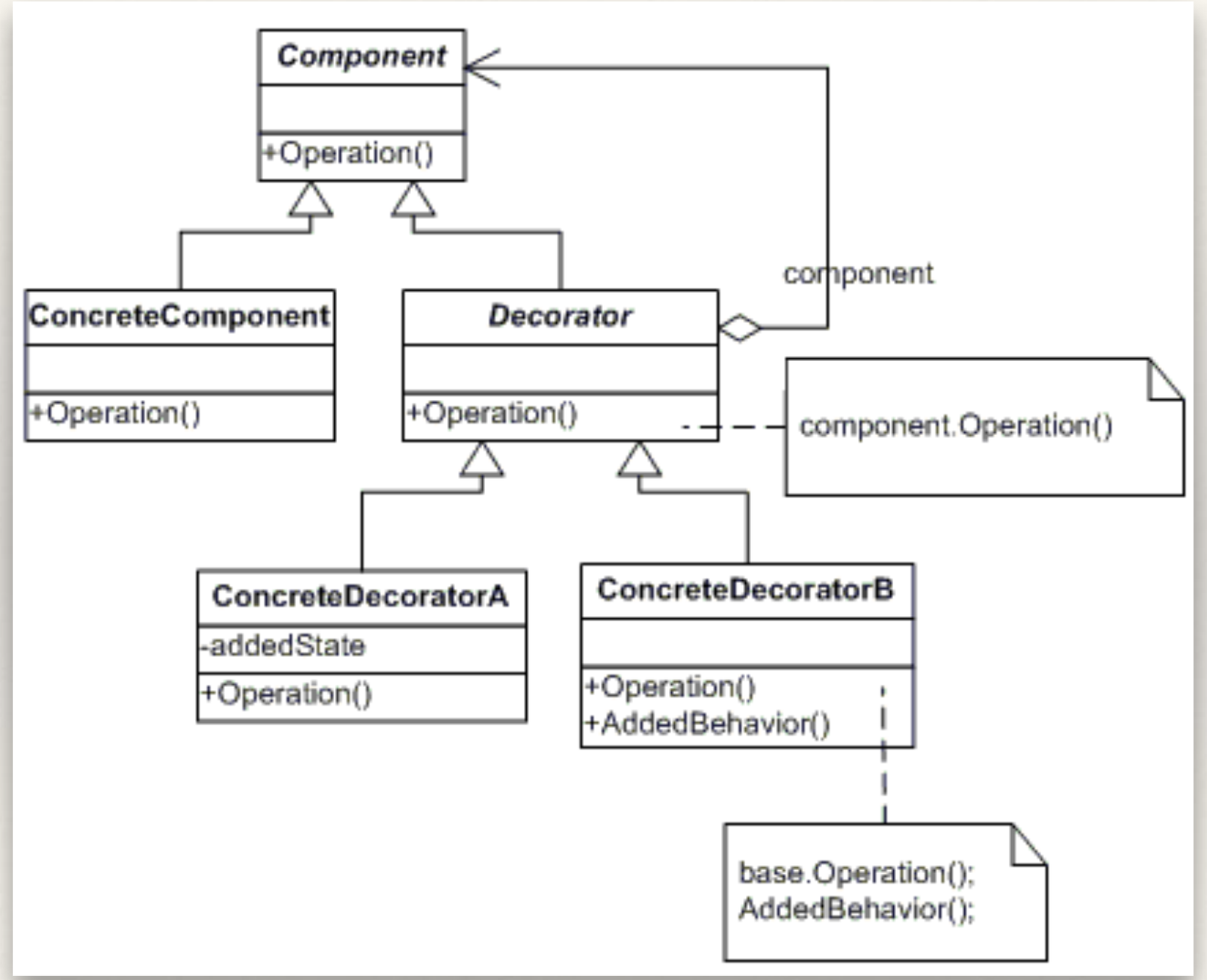
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

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

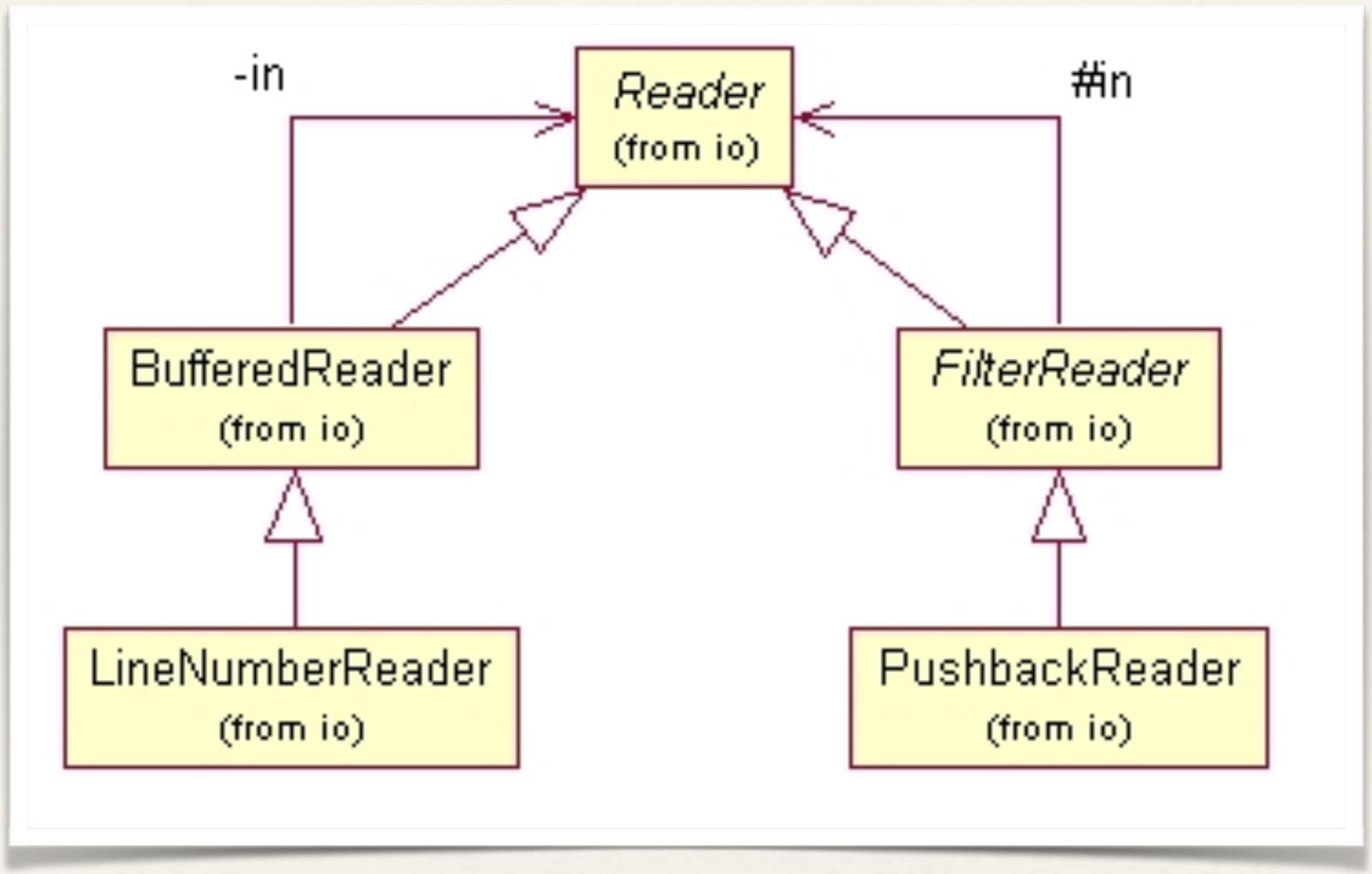
- Want to add responsibilities to individual objects (not an entire class)
- One way is to use inheritance
  - Inflexible; static choice
  - Hard to add and remove responsibilities dynamically

- 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

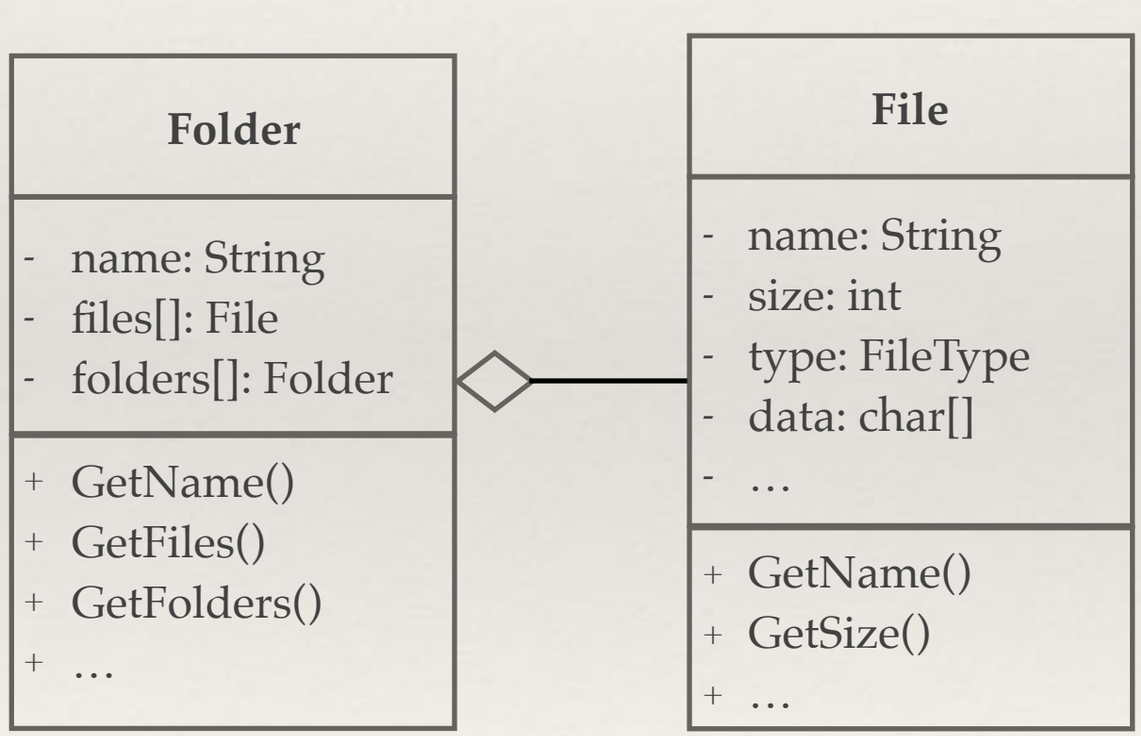
```
LineNumberReader lnr =
    new LineNumberReader(
    new BufferedReader(
    new FileReader(“./test.c")));

String str = null;

while((str = lnr.readLine()) != null)
      System.out.println(lnr.getLineNumber() + ": " + str);
```

Decorator pattern in Reader

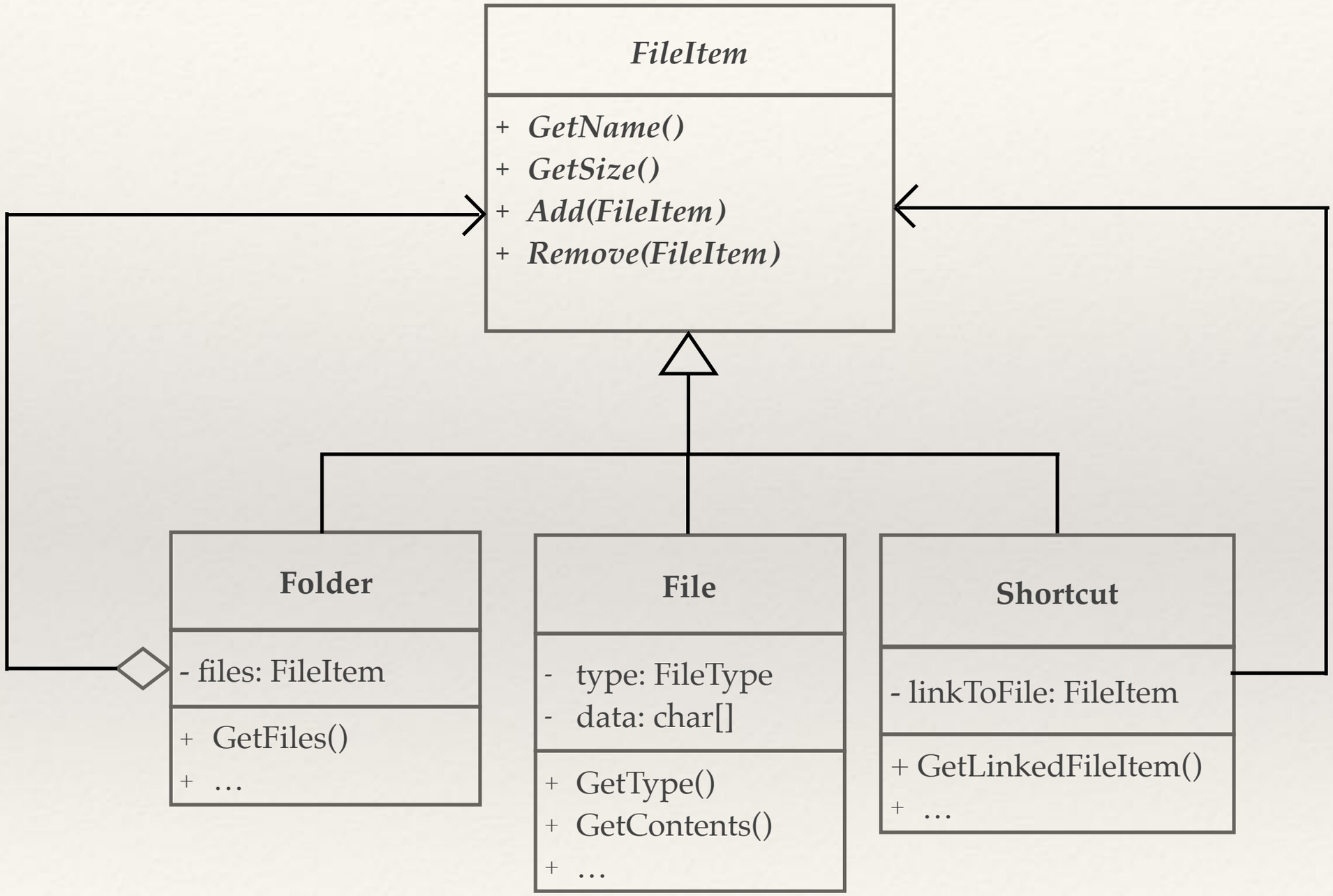


Scenario

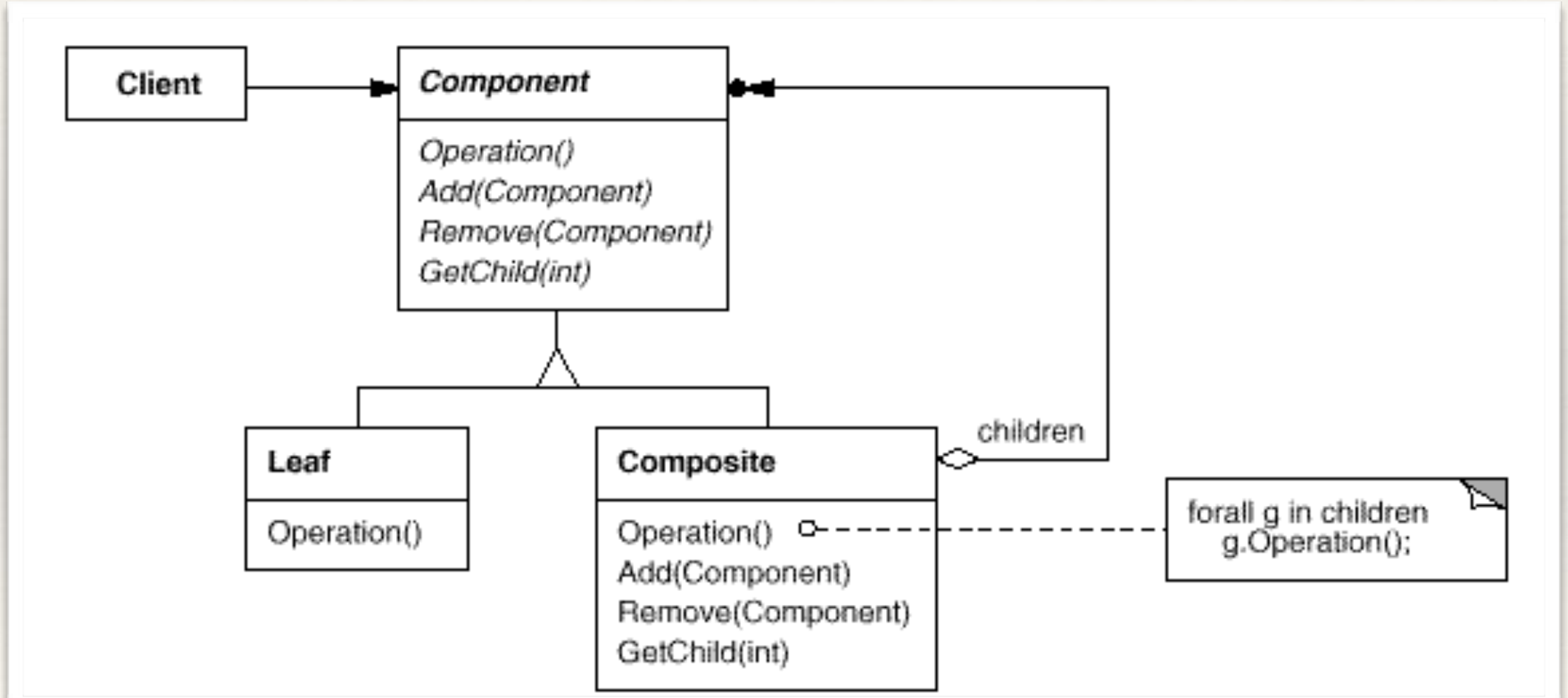
a) How do we treat files and folders alike?
b) How can we handle shortcuts?



How about this solution?



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

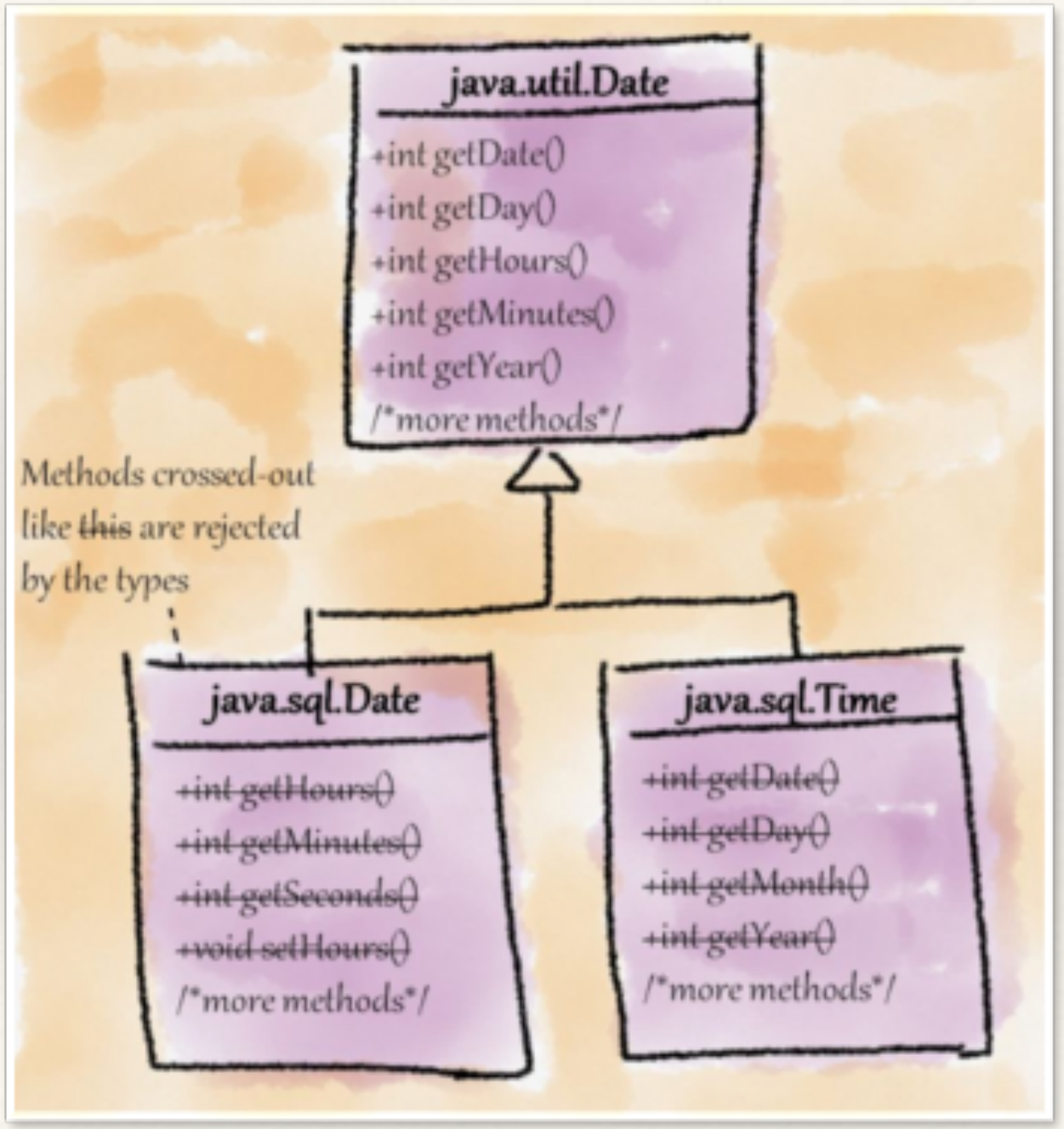
- Perform recursive composition of components
  - Clients don't have to treat container and primitive components differently

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.

What's that smell?



"Refused bequest" smell

Liskov's Substitution Principle (LSP)

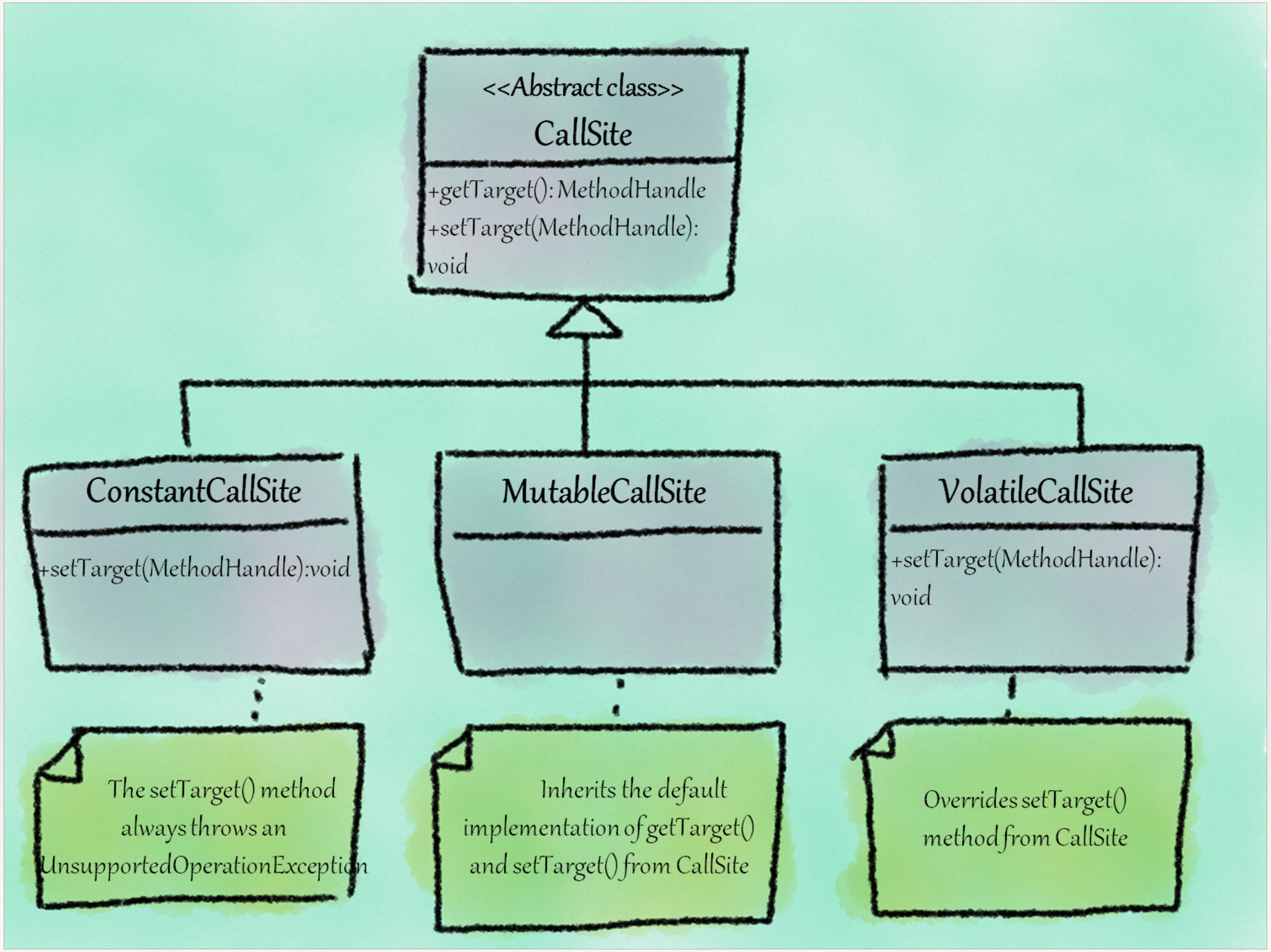
It should be possible to replace objects of supertype with objects of subtypes without altering the desired behavior of the program

Barbara Liskov

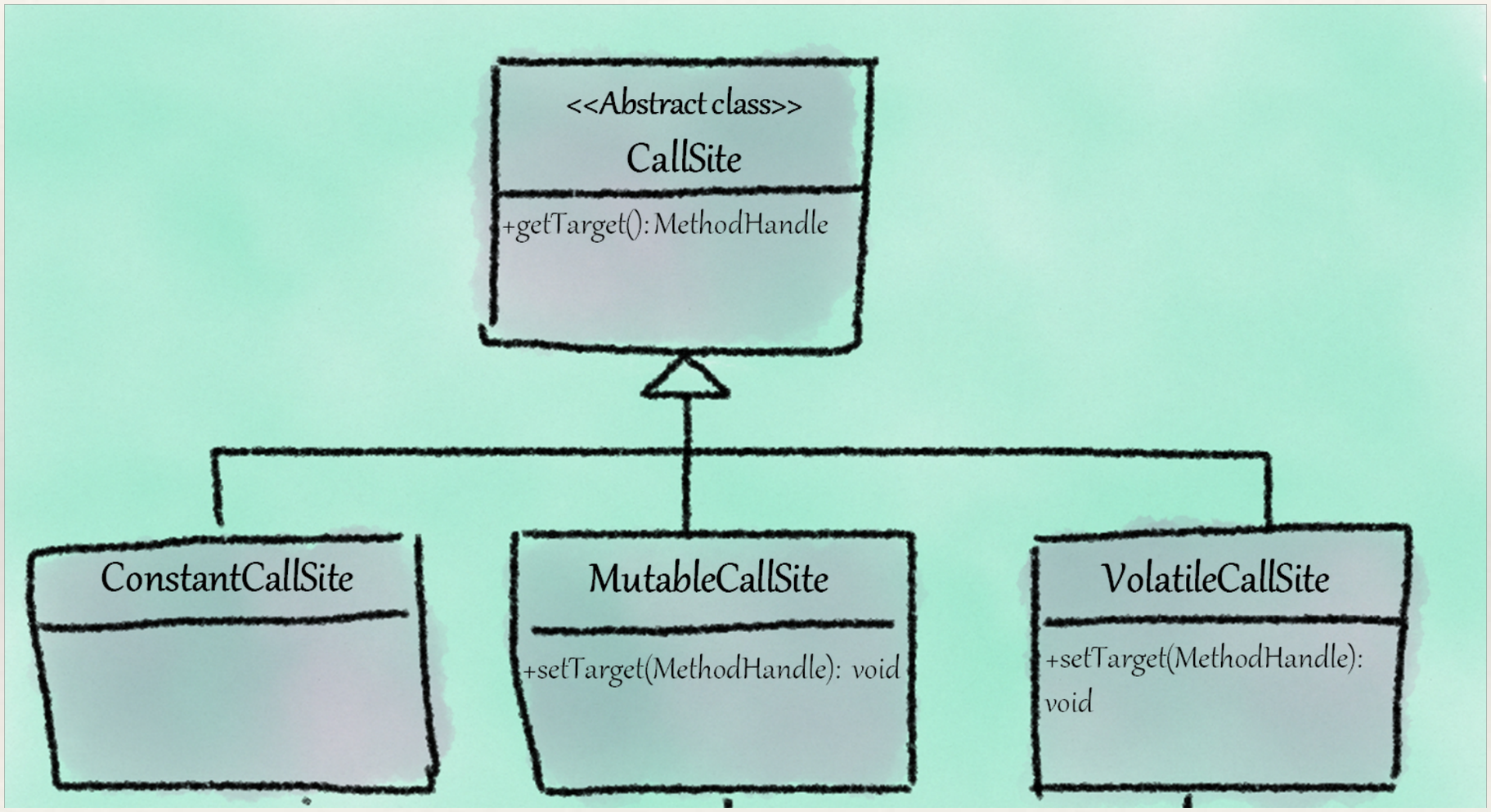
Refused bequest smell

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

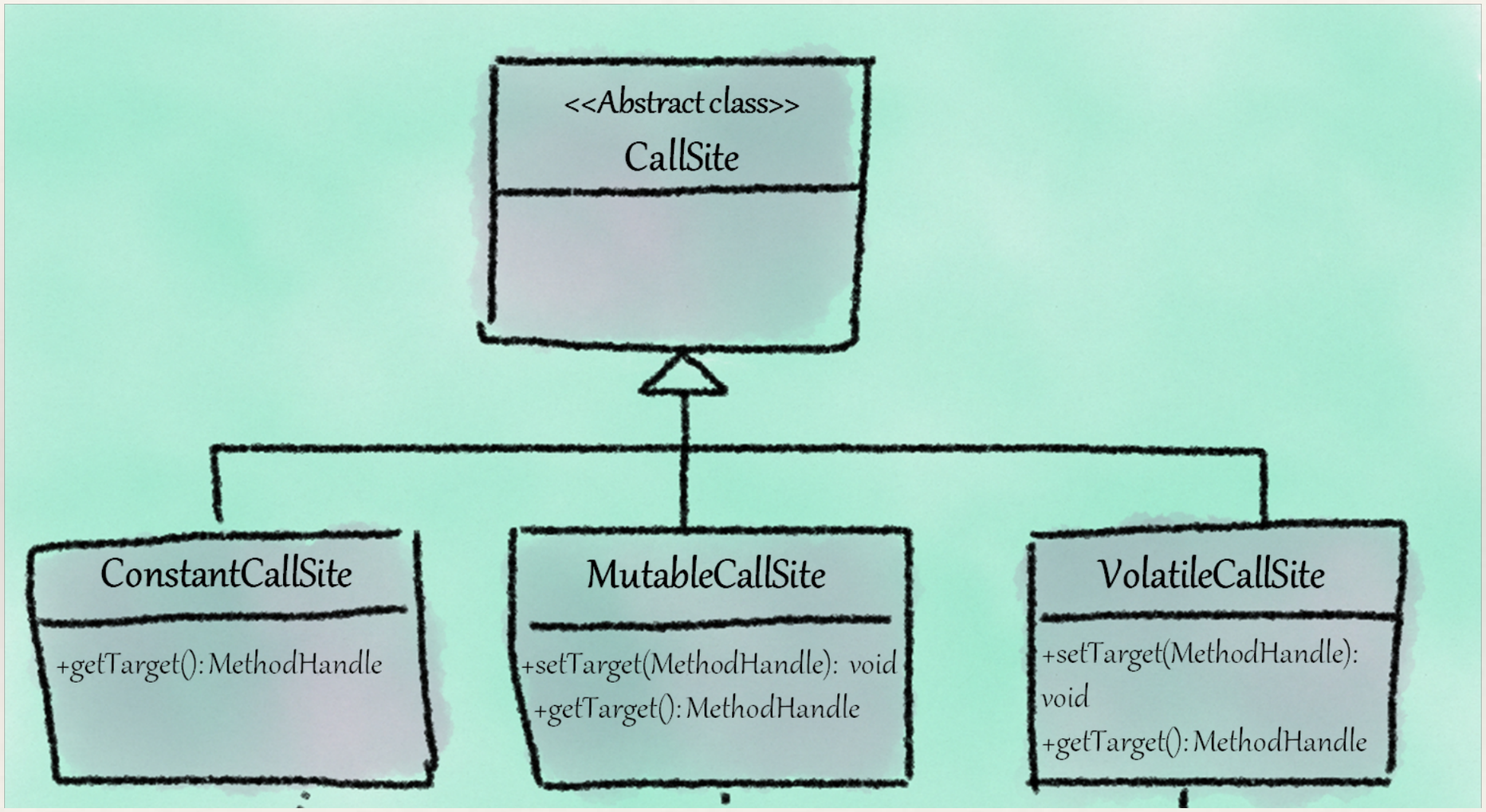
What's that smell?



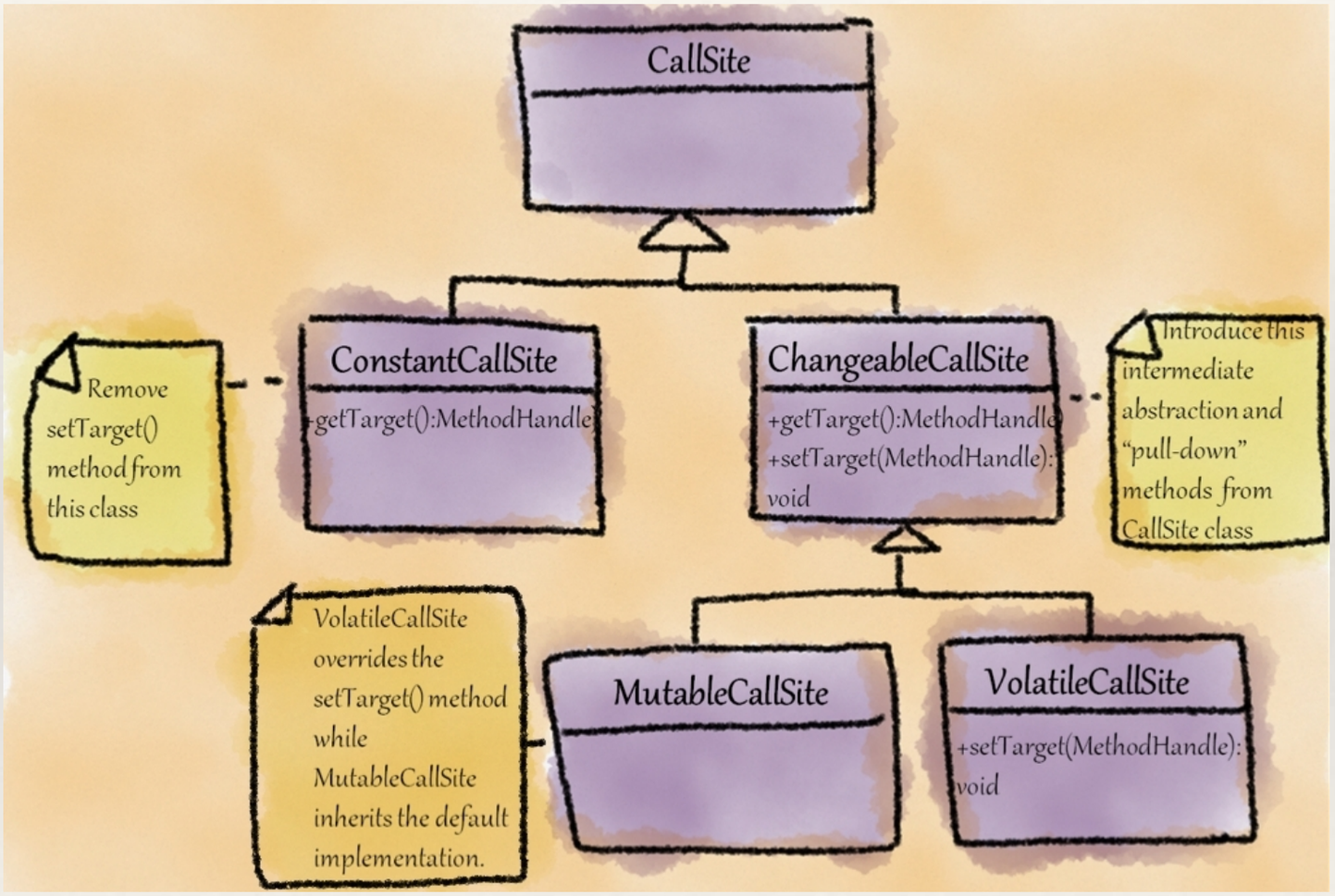
How about this refactoring?



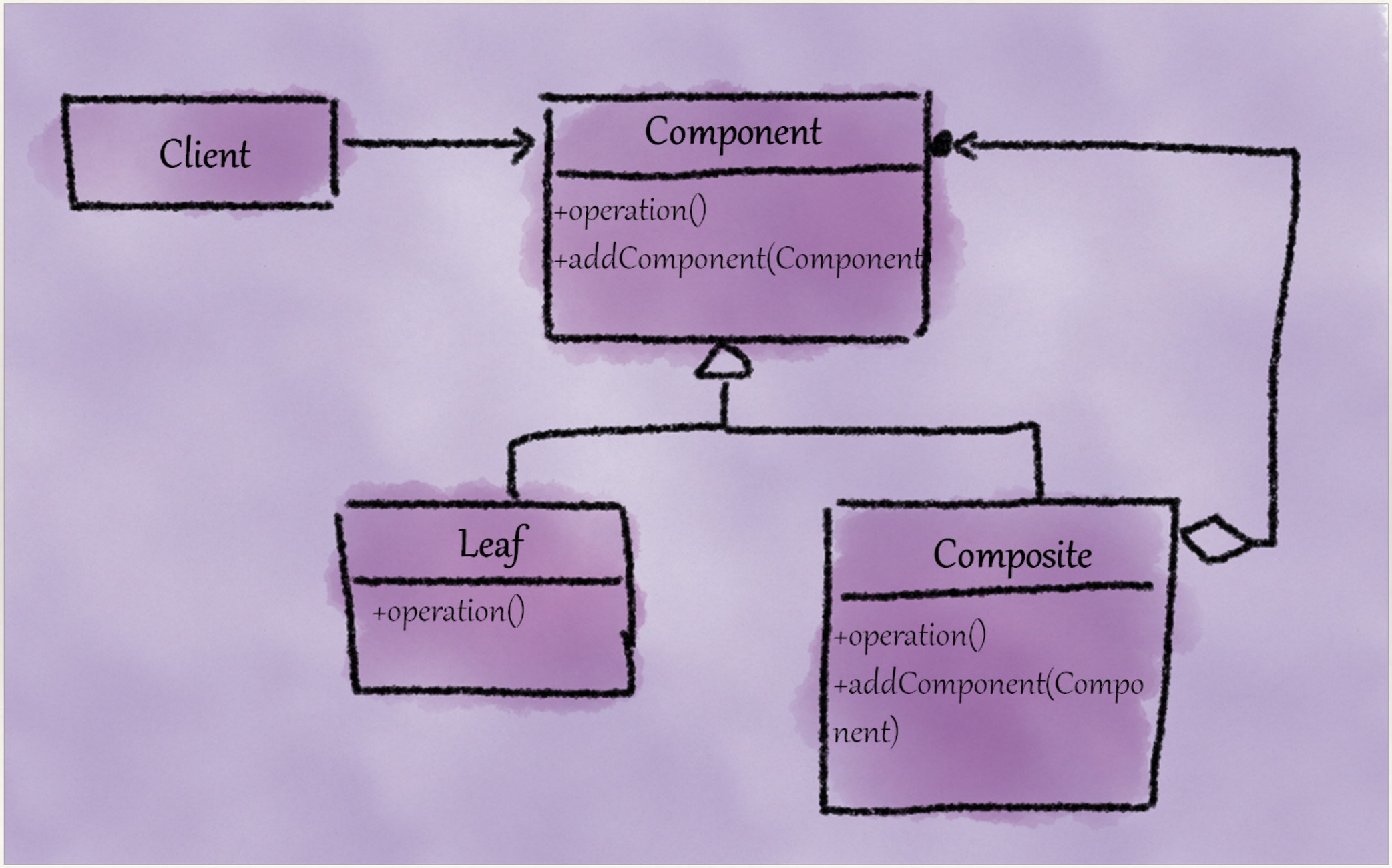
How about this refactoring?



Suggested refactoring

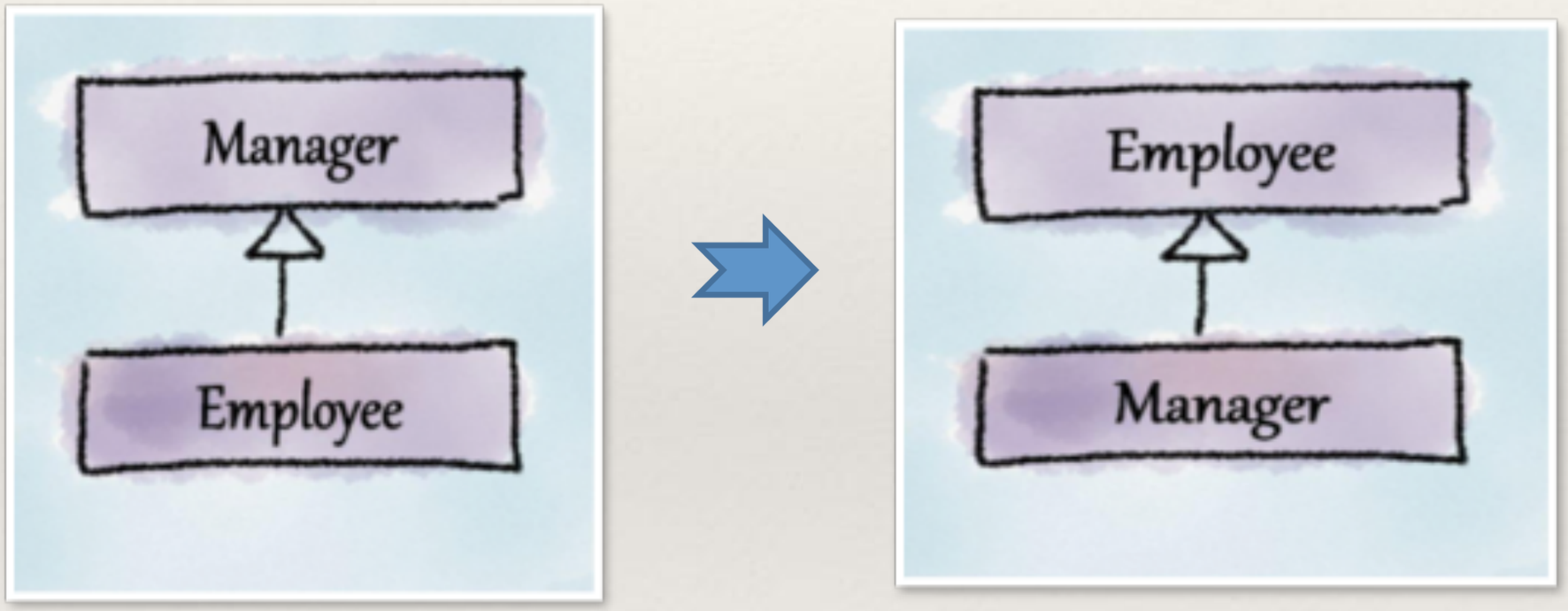


Practical considerations

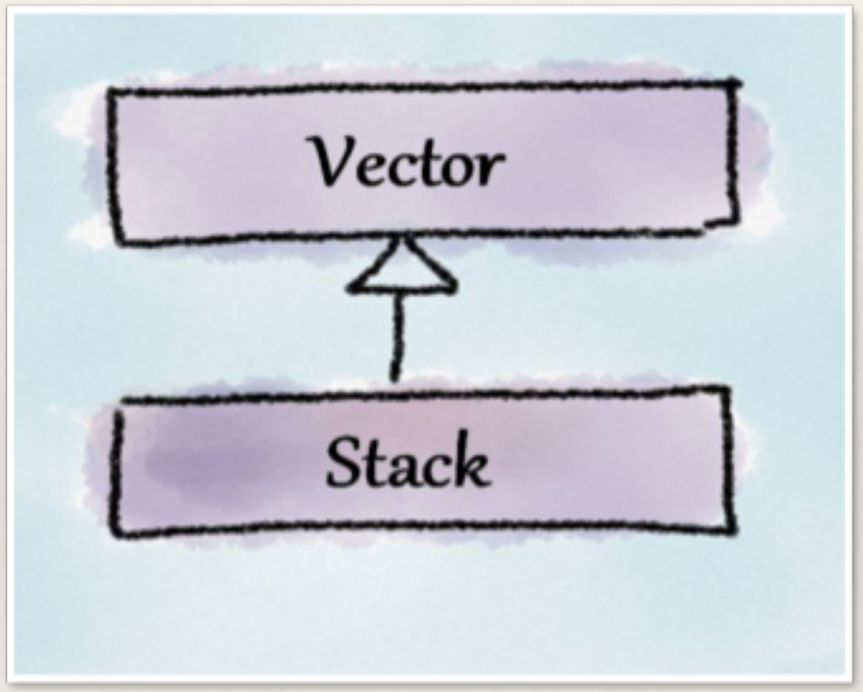


What's that smell?

Refactoring

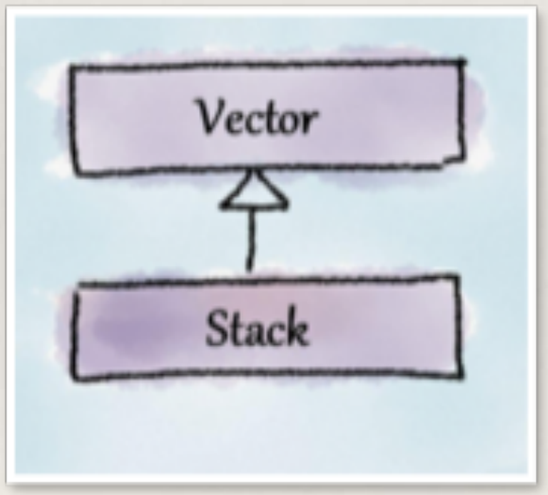


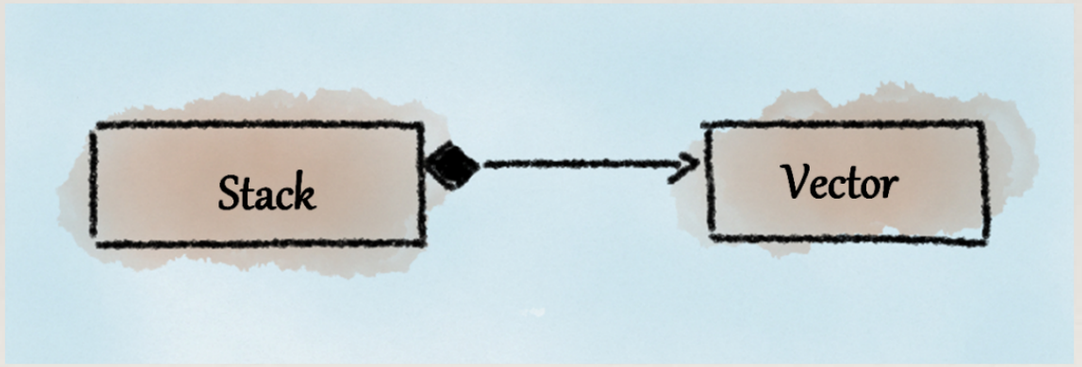
What's that smell?



Refactoring

Replace inheritance with delegation

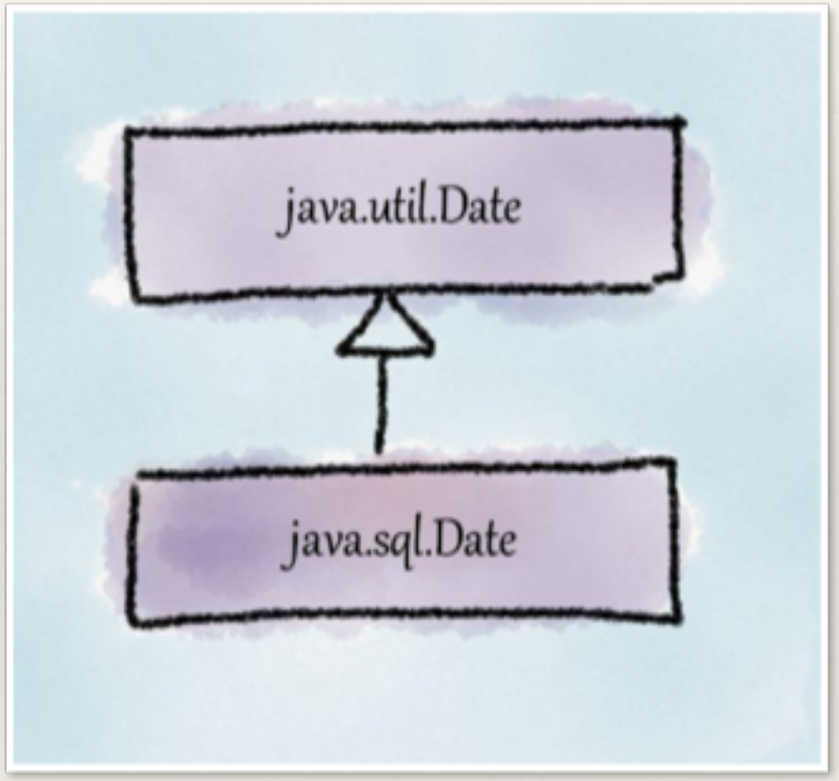




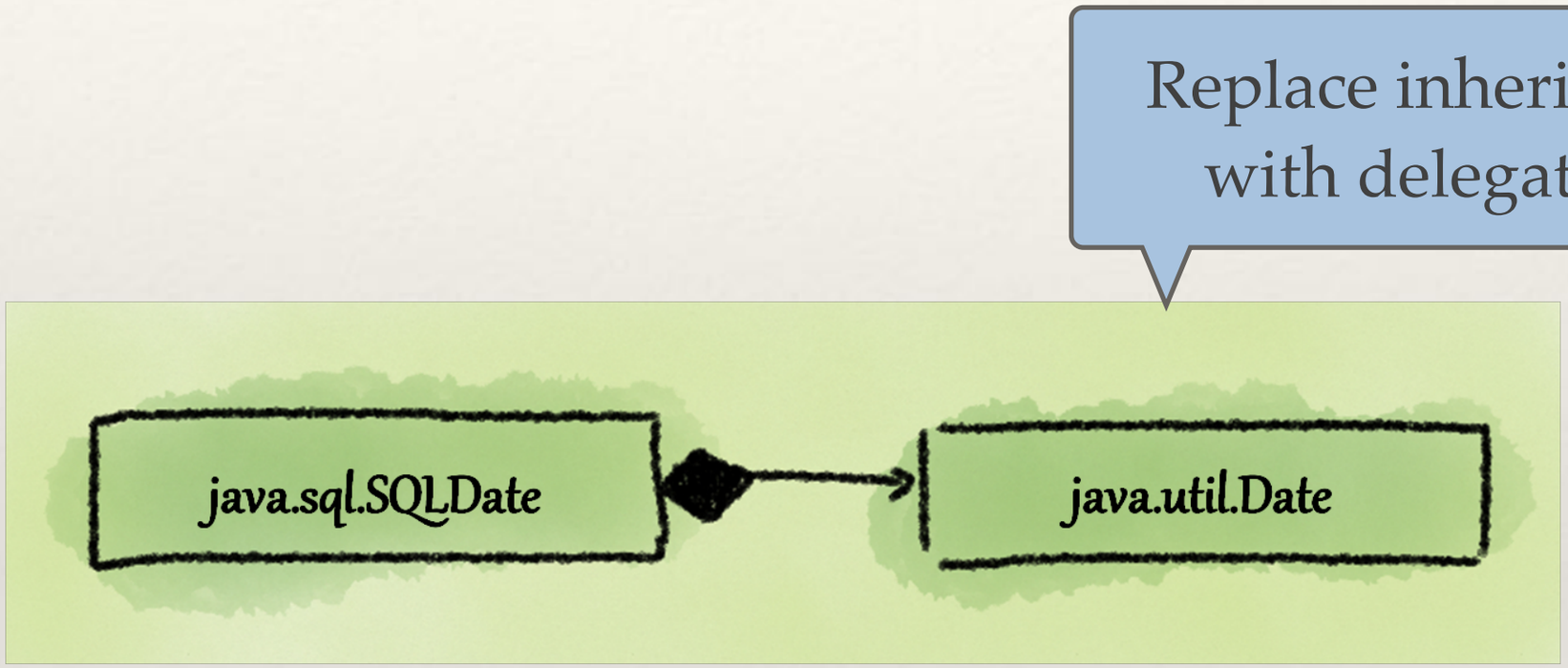
Hands-on exercise

- Refactor java.util.Stack!
- Question: Is it a “safe” refactoring?

What's that smell?



Refactoring for Date



Agenda

- SOLID Foundations
- Single Responsibility Principle
- Open-Closed Principle
- Liskov's Substitution Principle
- **Interface Segregation Principle**
- Dependency Inversion Principle
- Applying Principles in Practice

Interface Segregation Principle (ISP)

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

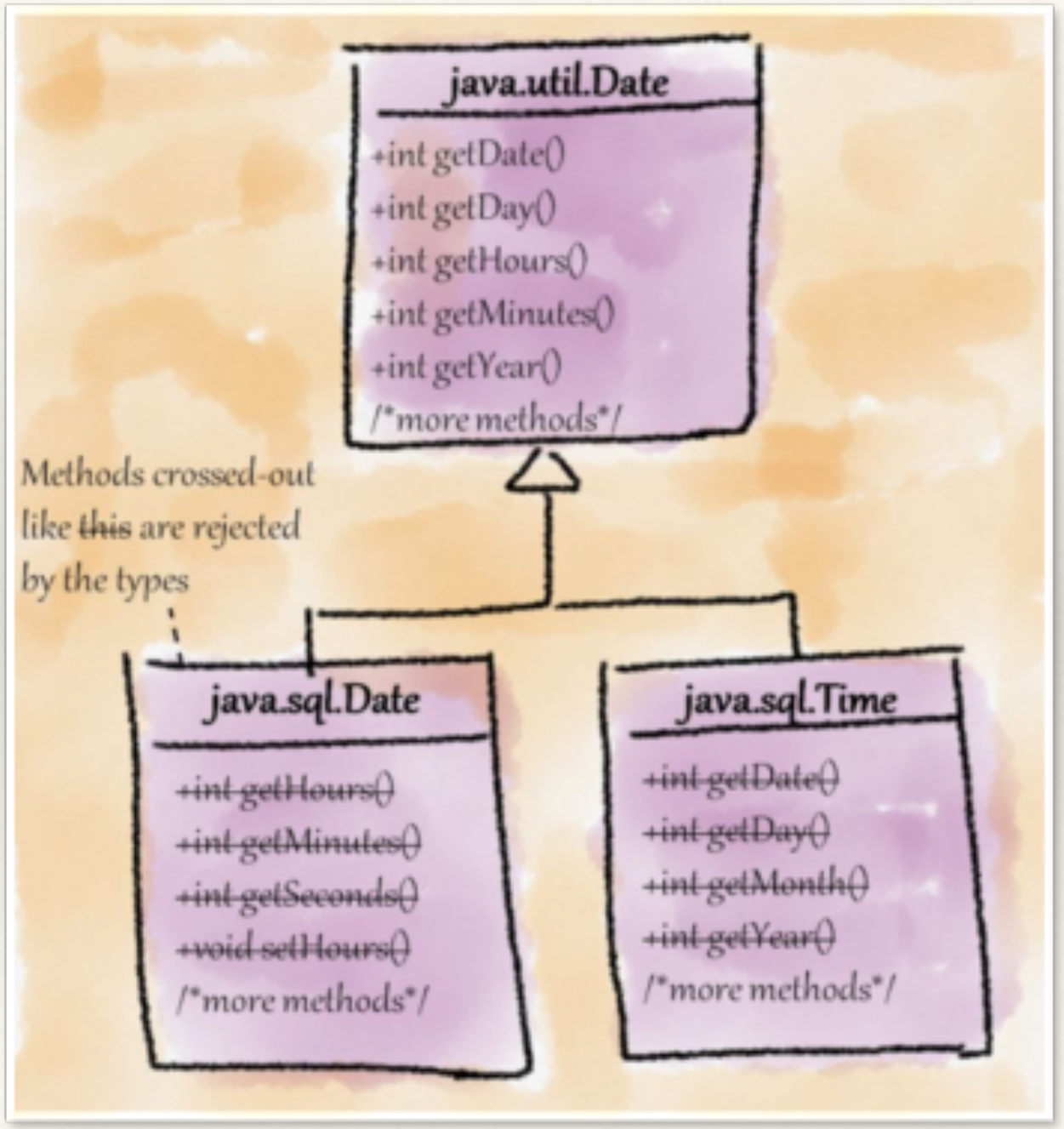
How java.util.Date violates ISP

```
// using java.util.Date
Date today = new Date();
System.out.println(today);
```

```
$ java DateUse
Wed Dec 02 17:17:08 IST 2015
```

Why should we get the time and timezone details if I only want a date? Can I get rid of these parts? No!

What's that smell?



"Refused bequest" smell

How java.time API follows ISP

```
// using java.time.LocalDate
LocalDate today = LocalDate.now();
System.out.println(today);
```

```
$ java DateUse
2015-12-02
```

I can use (and hence depend upon) only date related functionality (not time, zone, etc)

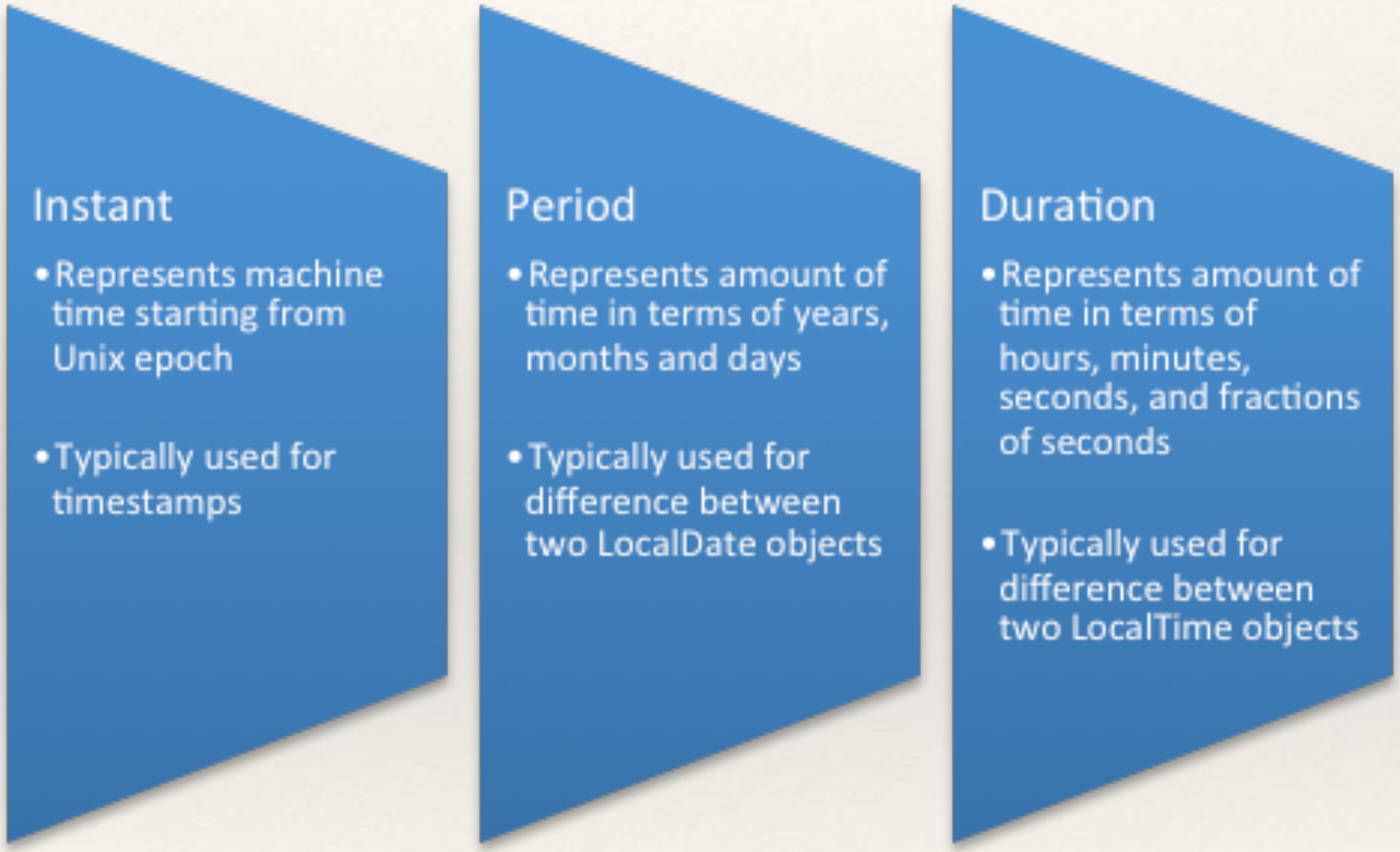
How java.time API follows ISP

```
LocalDate today = LocalDate.now();
System.out.println(today);

LocalTime now = LocalTime.now();
System.out.println(now);

LocalDateTime todayAndNow = LocalDateTime.now();
System.out.println(todayAndNow);

ZonedDateTime todayAndNowInTokyo = ZonedDateTime.now(ZoneId.of("Asia/Tokyo"));
System.out.println(todayAndNowInTokyo);
```

You can use only date, time, or even timezone, and combine them as needed!

```
2015-12-02
17:37:22.647
2015-12-02T17:37:22.648
2015-12-02T21:07:22.649+09:00[Asia/Tokyo]
```

More classes in Date/Time API



How about this one?

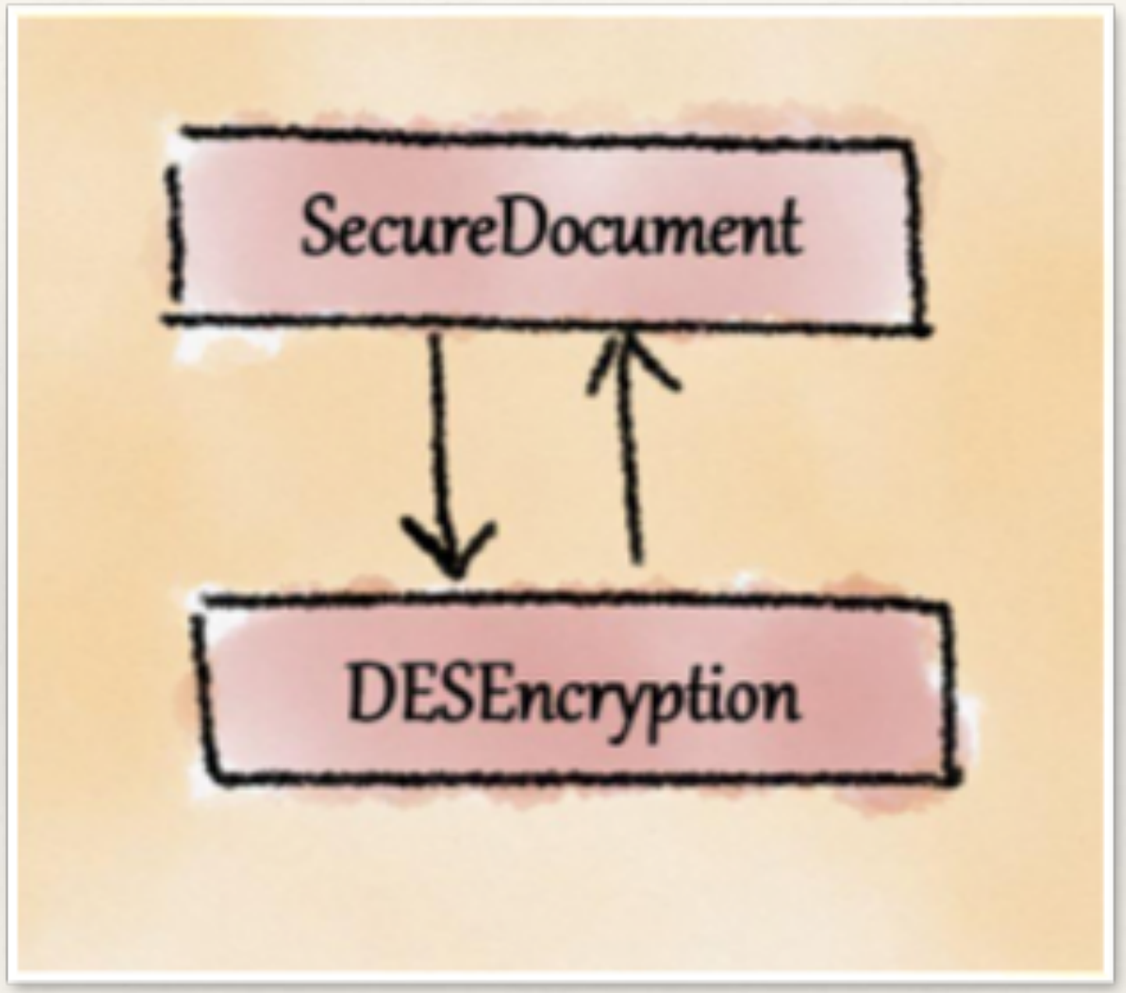
```
public interface MouseListener extends EventListener {

    /**
     * Invoked when the mouse button has been clicked (pressed
     * and released) on a component.
     */
    public void mouseClicked(MouseEvent e);

    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e);

    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e);

    /**
     * Invoked when the mouse enters a component.
     */
    public void mouseEntered(MouseEvent e);

    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e);
}
```

Dependency Inversion Principle (DIP)

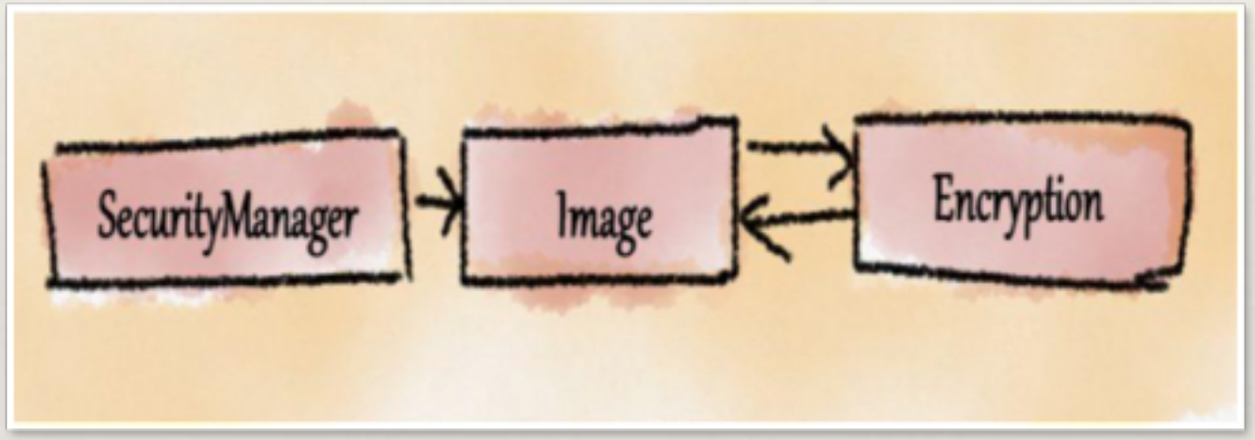
A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

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

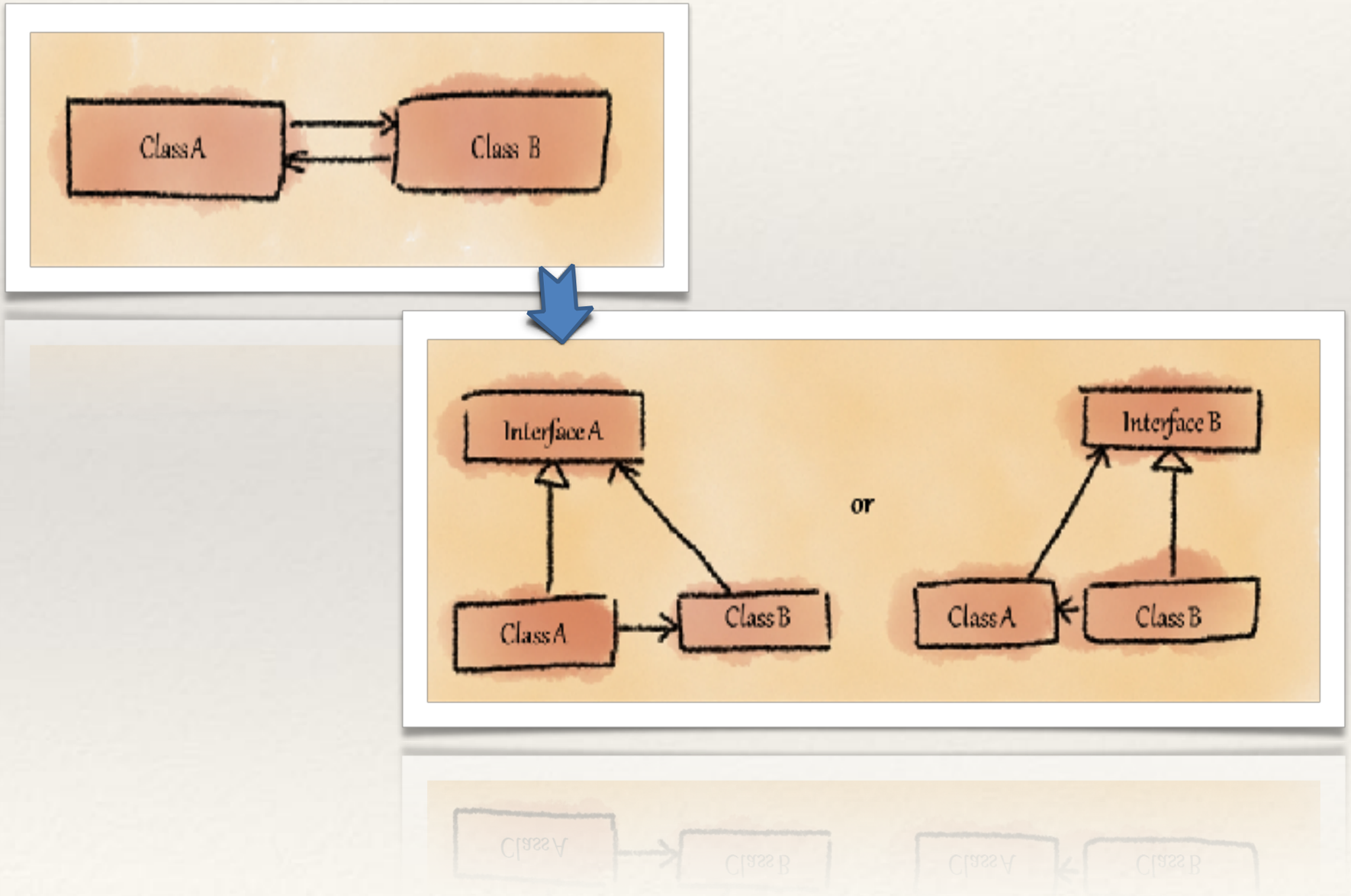
What's That Smell?



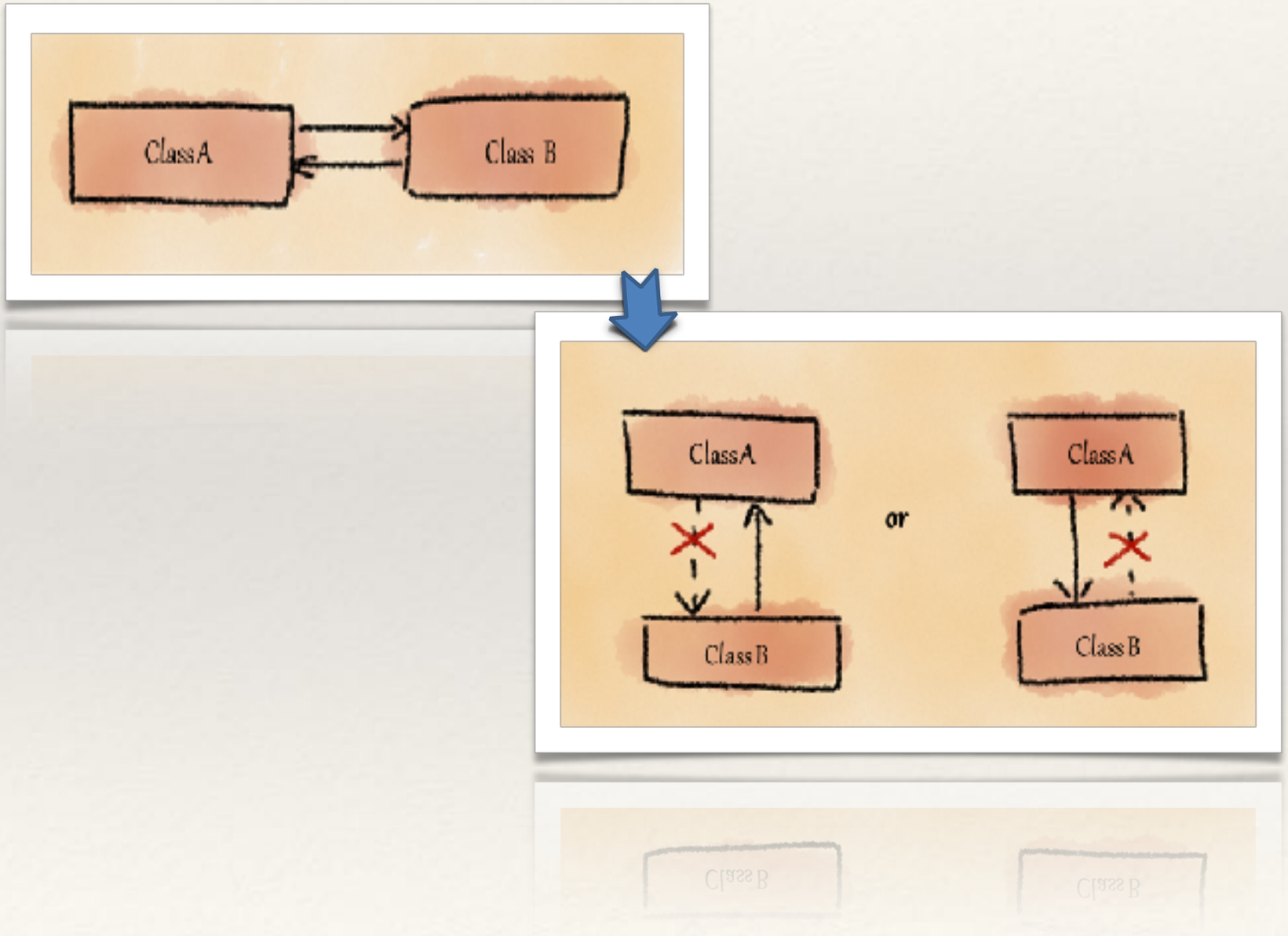
What's That Smell?



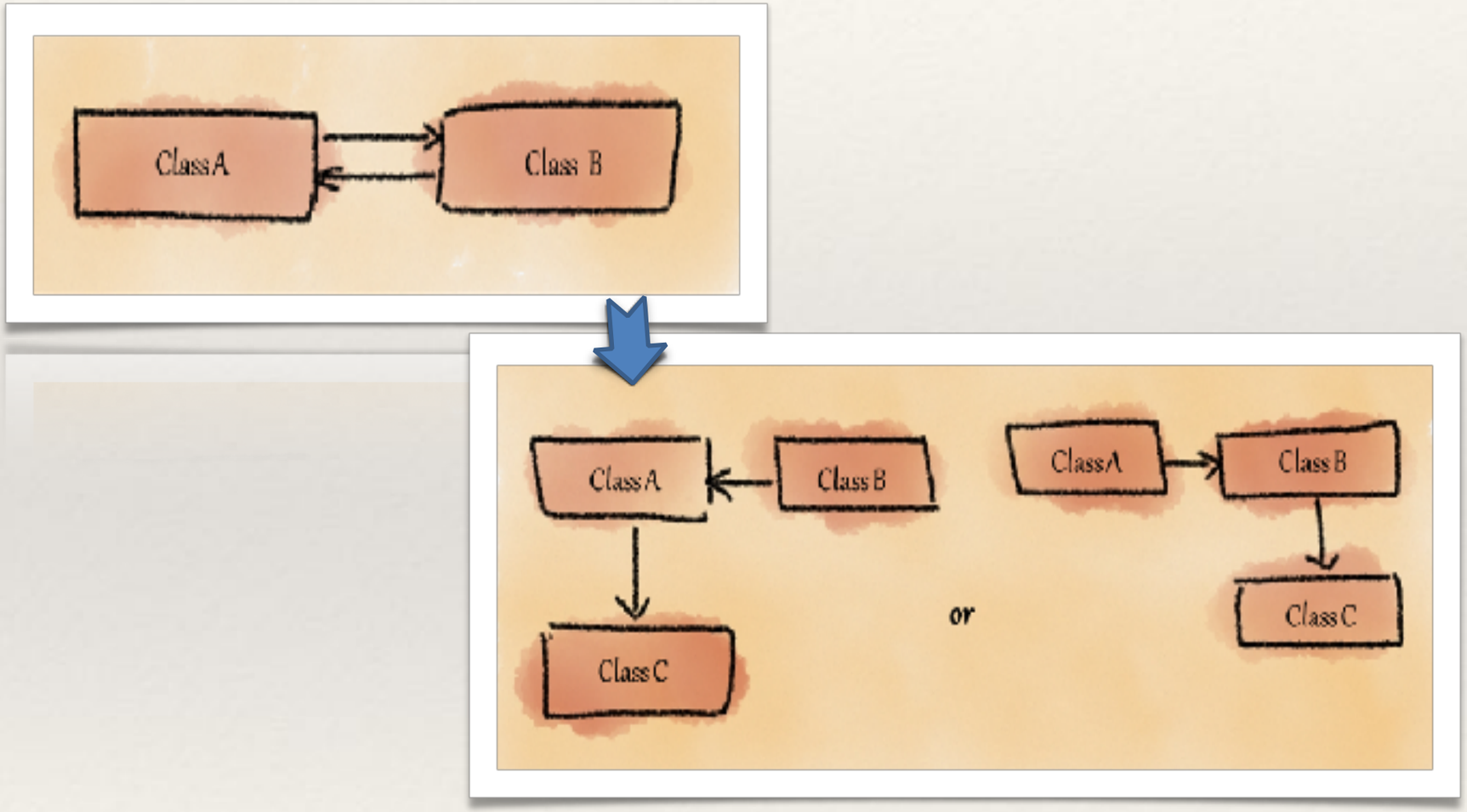
Suggested refactoring for this smell



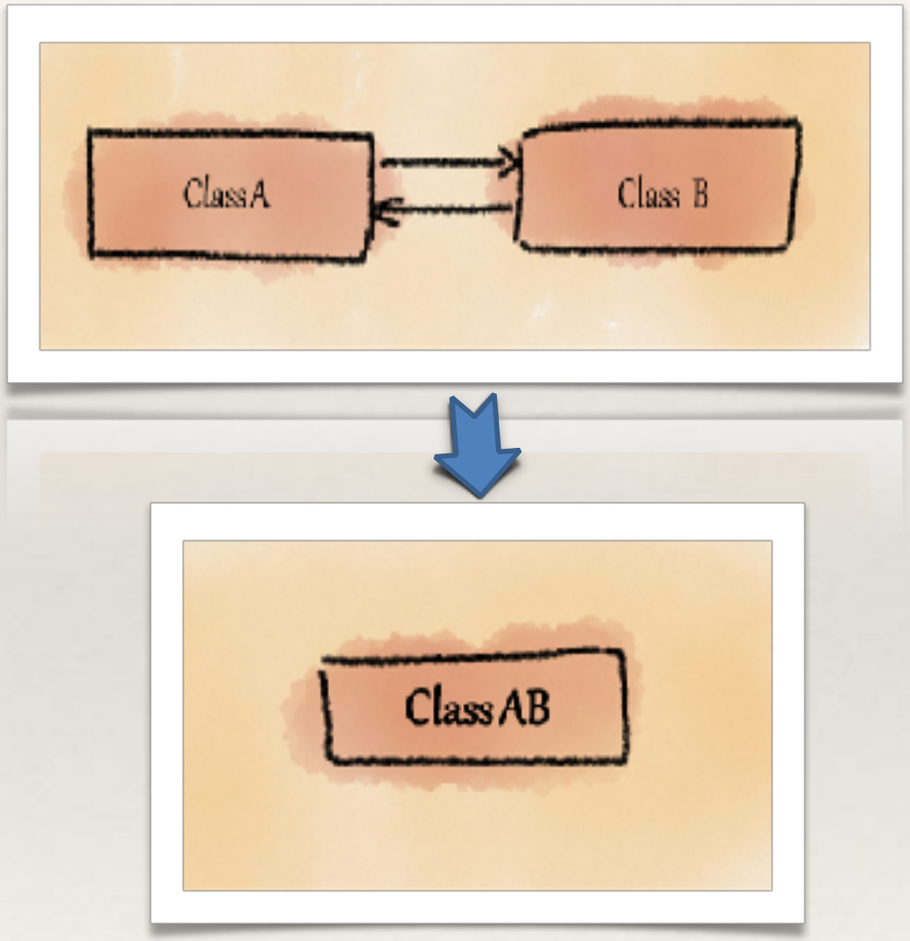
Suggested refactoring for this smell



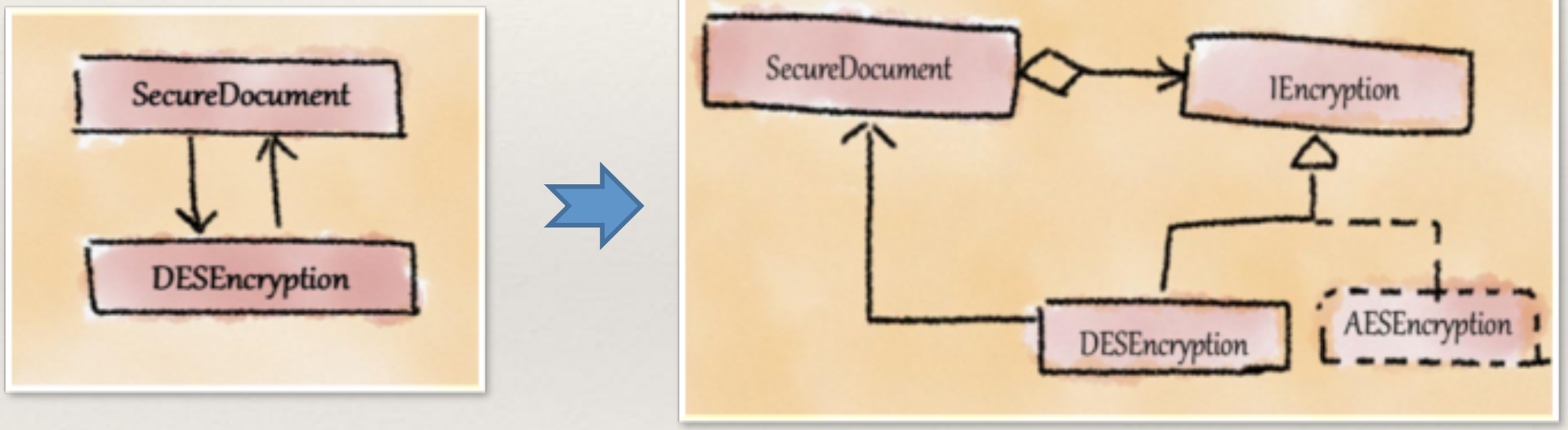
Suggested refactoring for this smell



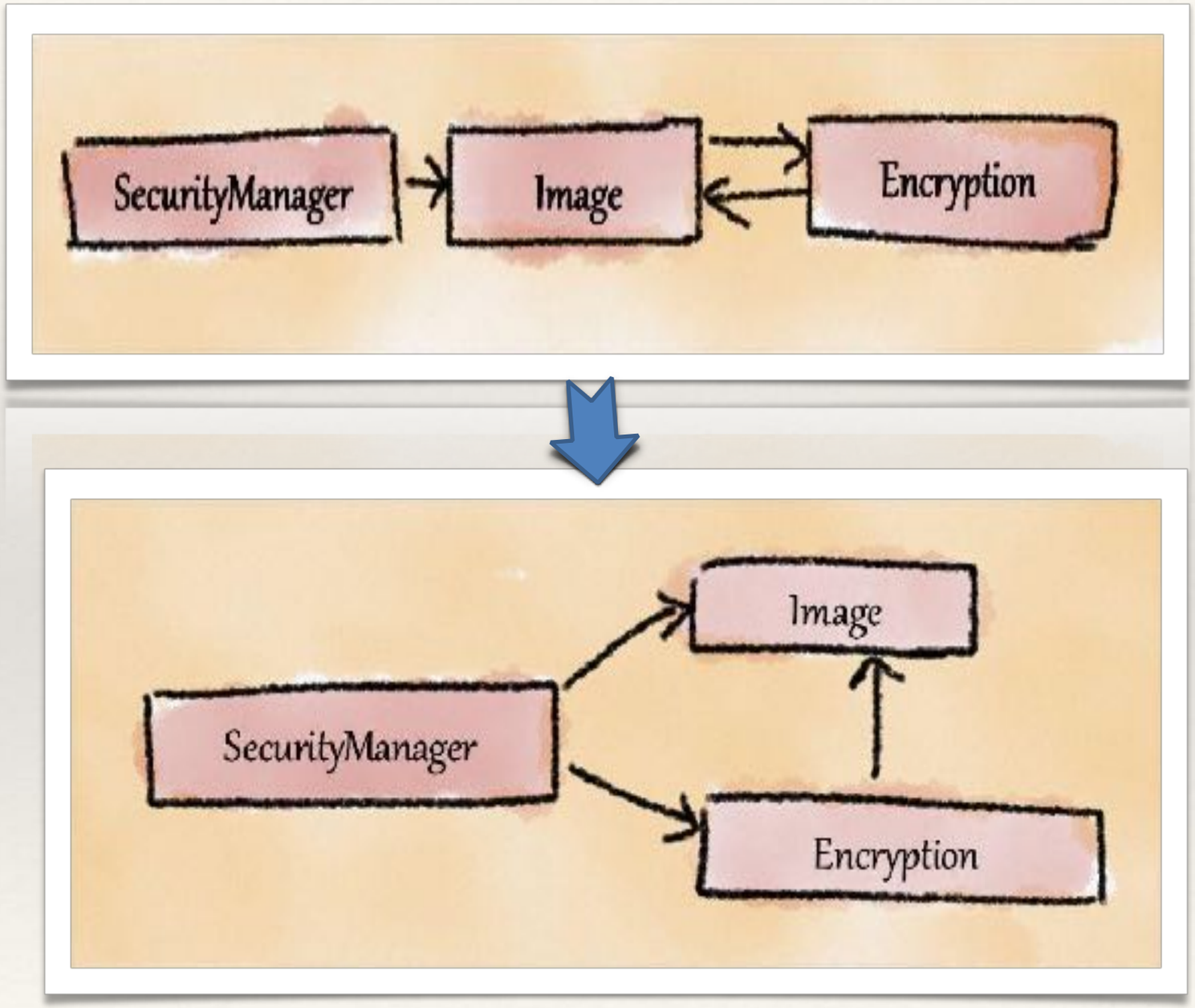
Suggested refactoring for this smell



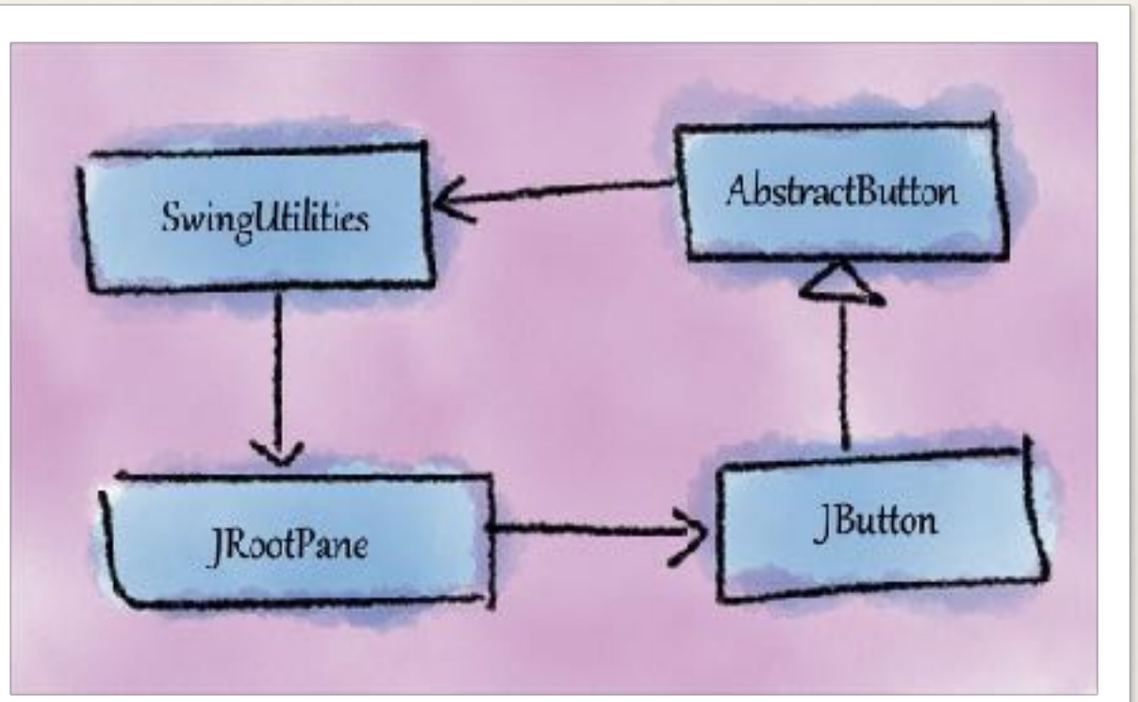
Suggested refactoring for this smell



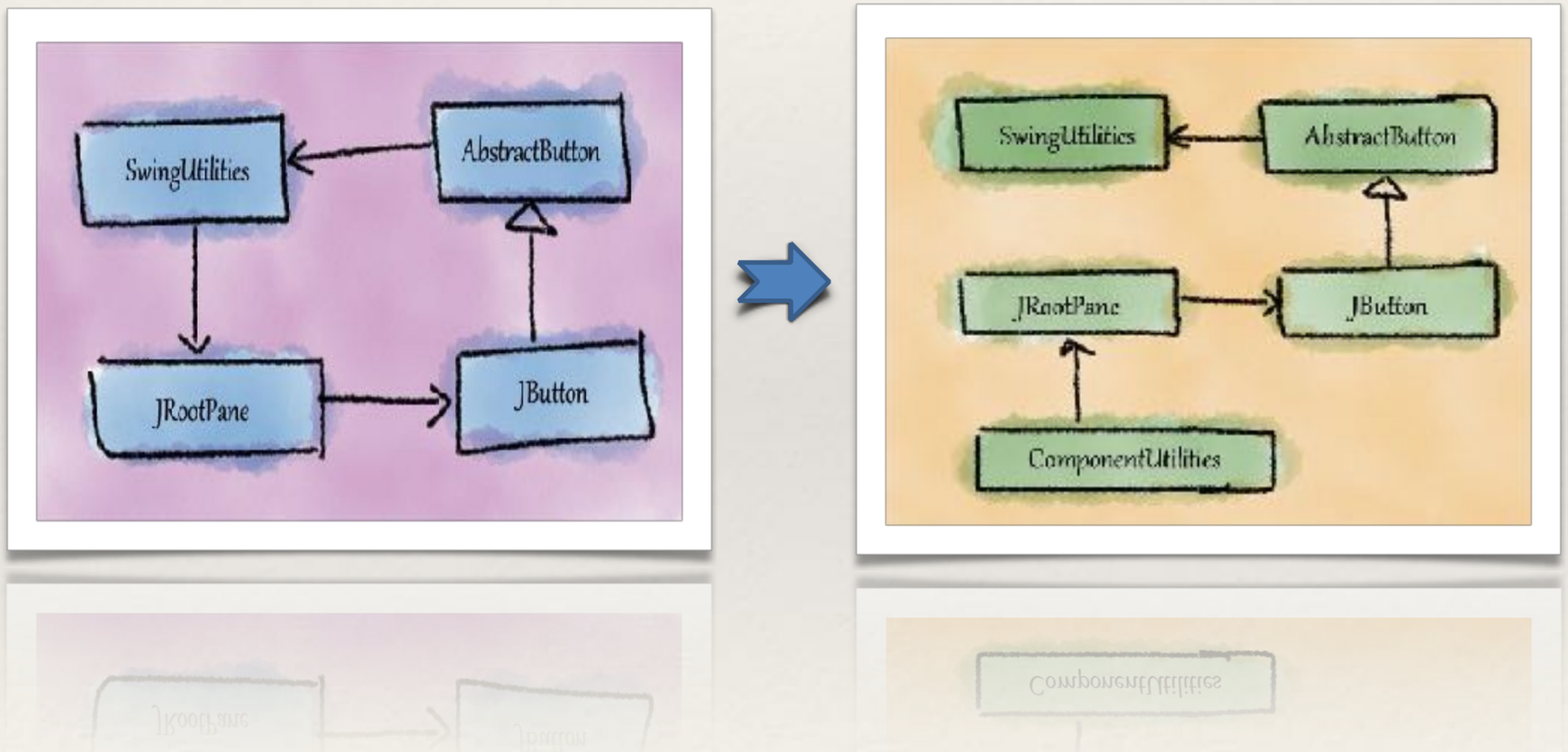
Suggested refactoring for this smell



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

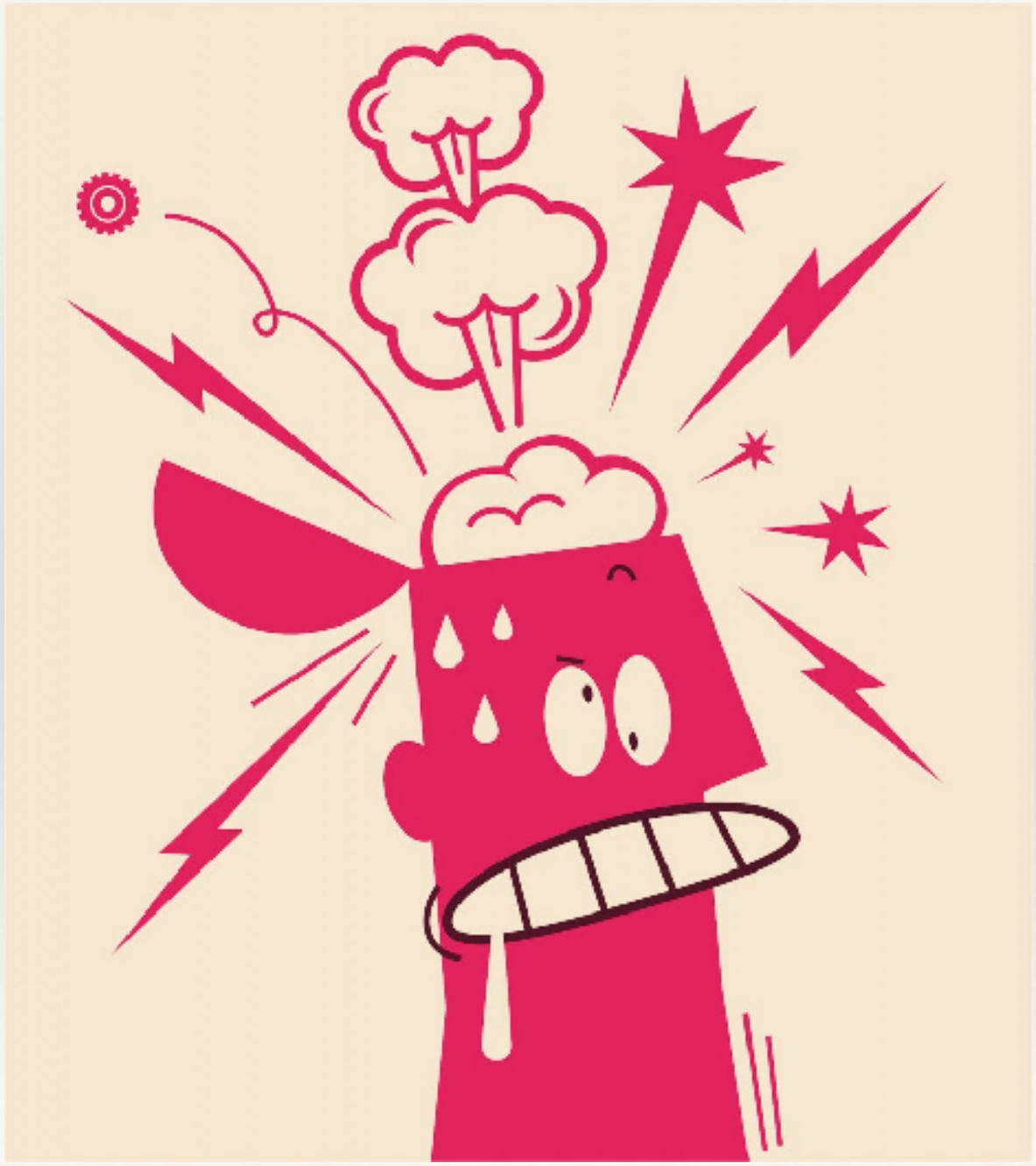
3 principles behind patterns

Program to an interface, not to an implementation

Favor object composition over inheritance

Encapsulate what varies

Cover key patterns through examples



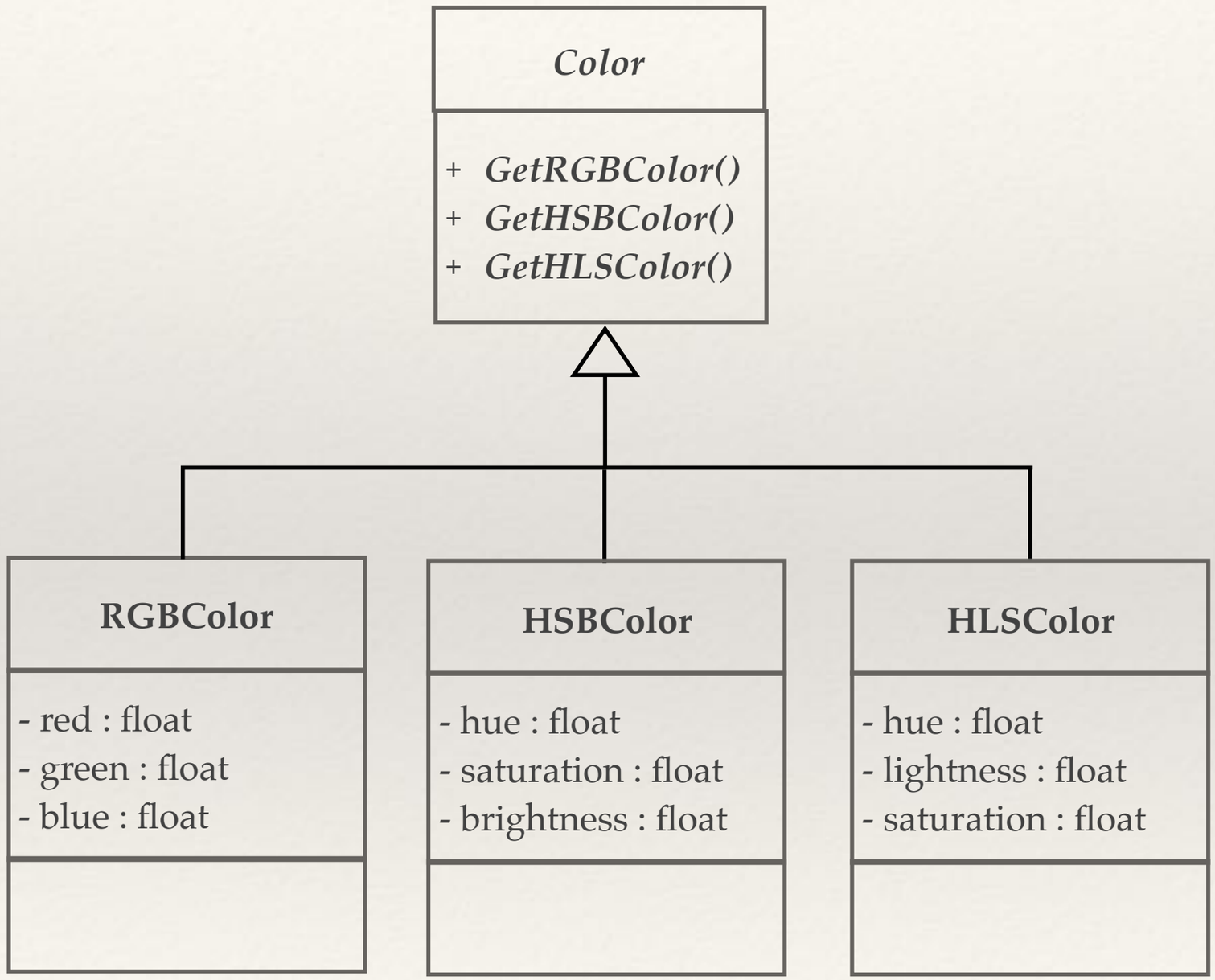
It is not about the number of patterns you know, but how well you understand “why, when, where, and how” to apply them effectively

Scenario

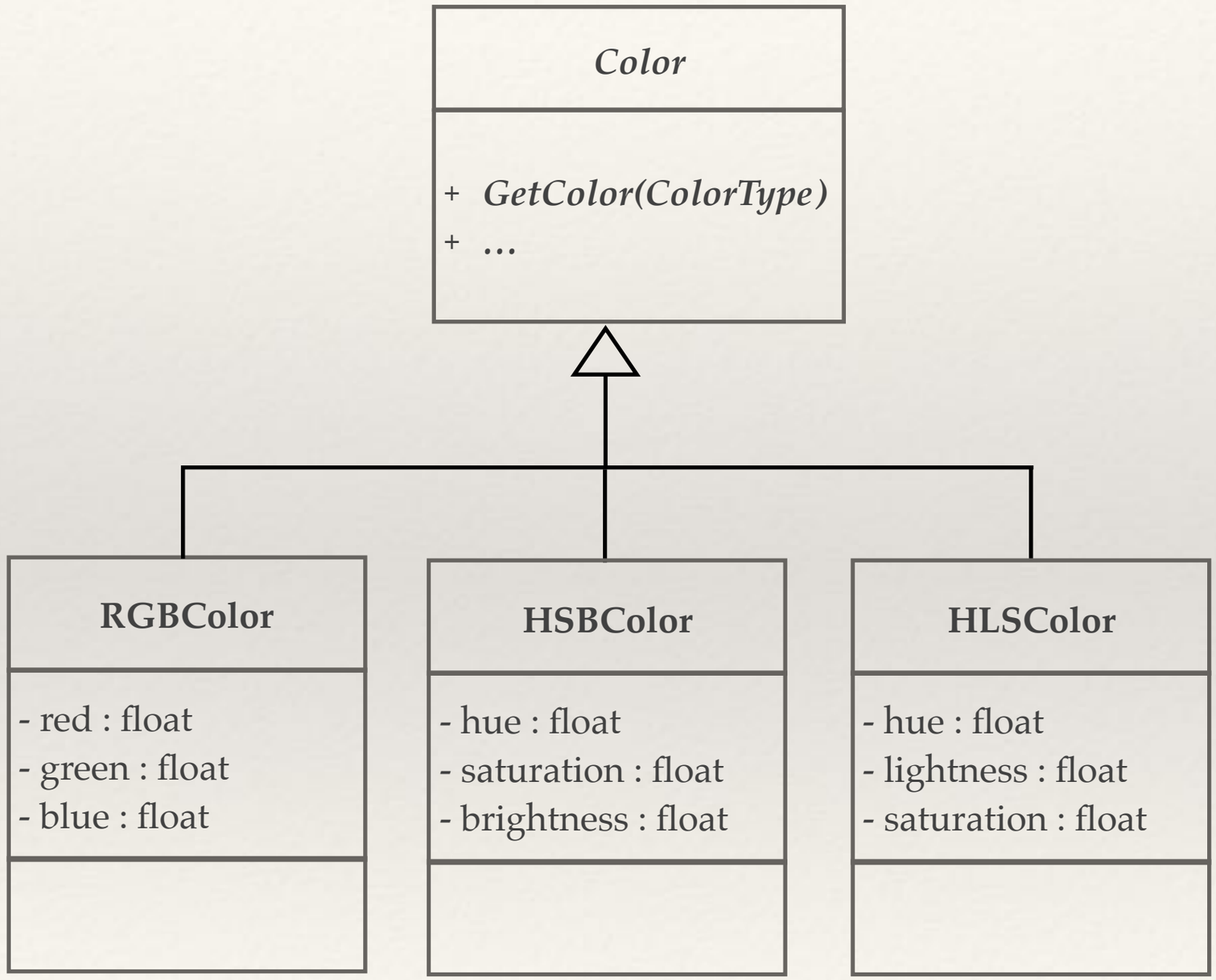
- Assume that you need to support different Color schemes in your software
  - RGB (Red, Green, Blue), HSB (Hue, Saturation, Brightness), and HLS (Hue, Lightness, and Saturation) schemes
- Overloading constructors and differentiating them using enums can become confusing
- What could be a better design?

```
enum ColorScheme { RGB, HSB, HLS, CMYK }
class Color {
        private float red, green, blue;                         // for supporting RGB scheme
        private float hue1, saturation1, brightness1;       // for supporting HSB scheme
        private float hue2, lightness2, saturation2;        // for supporting HLS scheme
        public Color(float arg1, float arg2, float arg3, ColorScheme  cs) {
                switch (cs) {
                        // initialize arg1, arg2, and arg3 based on ColorScheme value
                }
        }
}
```

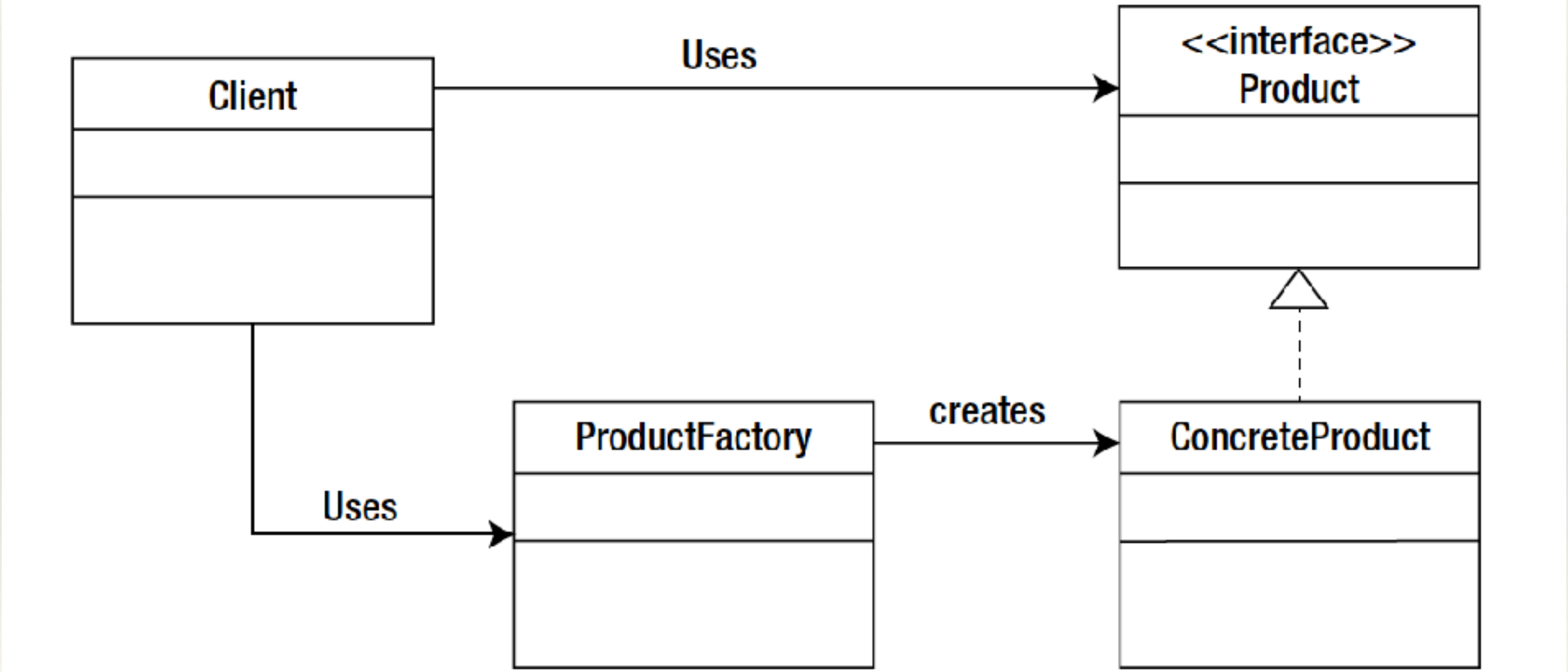
A solution using Factory Method pattern



A solution using Factory Method pattern



Factory Method pattern: Structure



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.

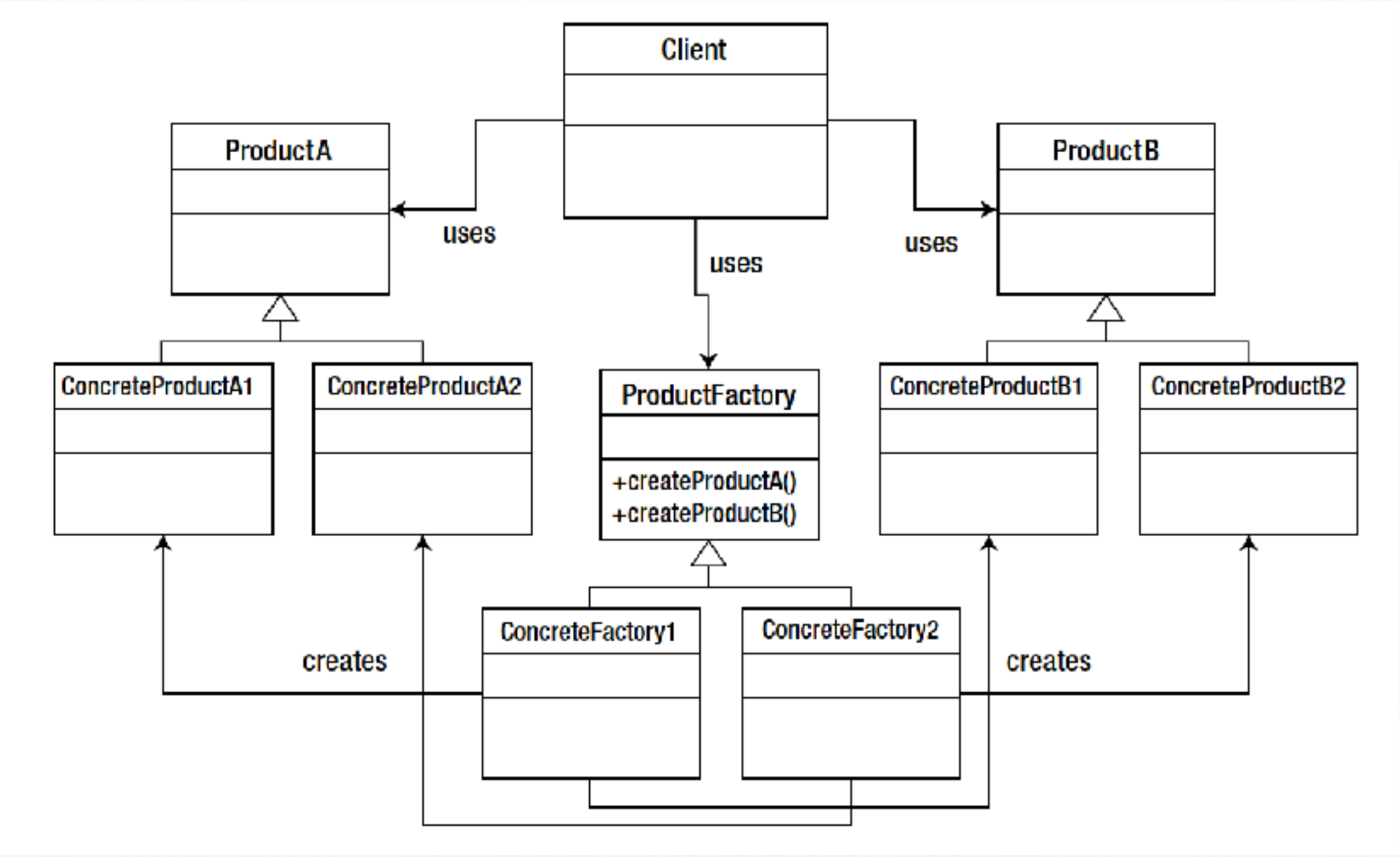
- A class cannot anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates

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

Factory method: Java library example

```
class SimpleThreadFactory implements ThreadFactory {
   public Thread newThread(Runnable r) {
     return new Thread(r);
   }
}
```

Abstract Factory pattern

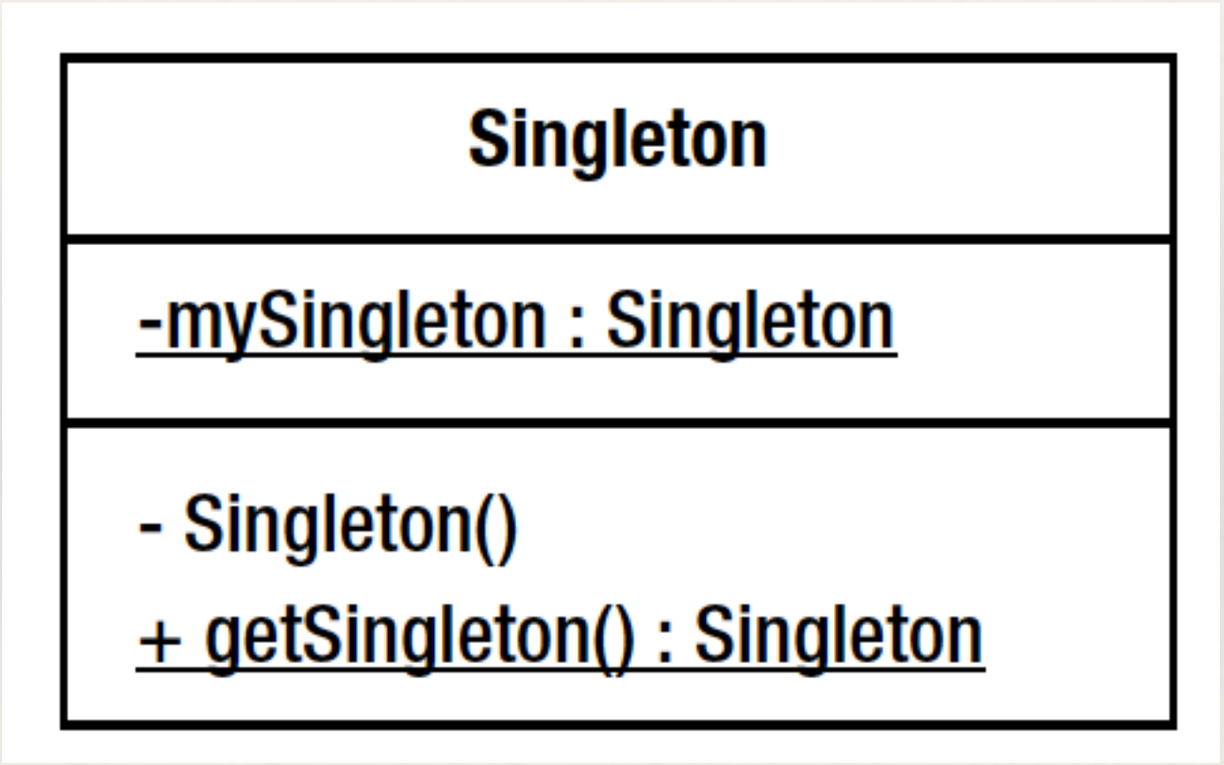


Abstract Factory in JDK

- XMLFactory

(returns XPathFactory, SchemaFactory, SAXParserFactory, etc)

Singleton pattern



Singleton pattern

```
// Logger class must be intantiated only once in the application; it is to ensure that the
// whole of the application makes use of that same logger instance
public class Logger {
   // declare the constructor private to prevent clients
   // from instantiating an object of this class directly
   private Logger() { }
   private static Logger myInstance;   // by default, this field is initialized to null
   // the static method to be used by clients to get the instance of the Logger class
   public static Logger getInstance() {
      if(myInstance == null) {
         // this is the first time this method is called, and that's why myInstance is null
         myInstance = new Logger();
      }
      // return the same object reference any time and every time getInstance is called
      return myInstance;
   }
   public void log(String s) {
      // a trivial implementation of log where we pass the string to be logged to console
      System.err.println(s);
   }
}
```

Singleton pattern

```
public class ThreadsafeLogger {
   private ThreadsafeLogger() {
      // private constructor
   }
   public static ThreadsafeLogger myInstance;
   public static class LoggerHolder {
      public static ThreadsafeLogger logger = new ThreadsafeLogger();
   }
   public static ThreadsafeLogger getInstance() {
      return LoggerHolder.logger;
   }
   public void log(String s) {
      // log implementation
      System.err.println(s);
   }
}
```

Singleton pattern in JDK

- java.lang.Runtime
- java.lang.System

Scenario

- Assume that you have a Locale class constructor that takes many “optional arguments”
- Constraint: Only certain variants are allowed - you need to “disallow” inappropriate combinations(e.g., invalid combination of country and variant) by throwing IllformedLocaleException.
  - Overloading constructors will result in “too many constructors”
- How will you design a solution for this?

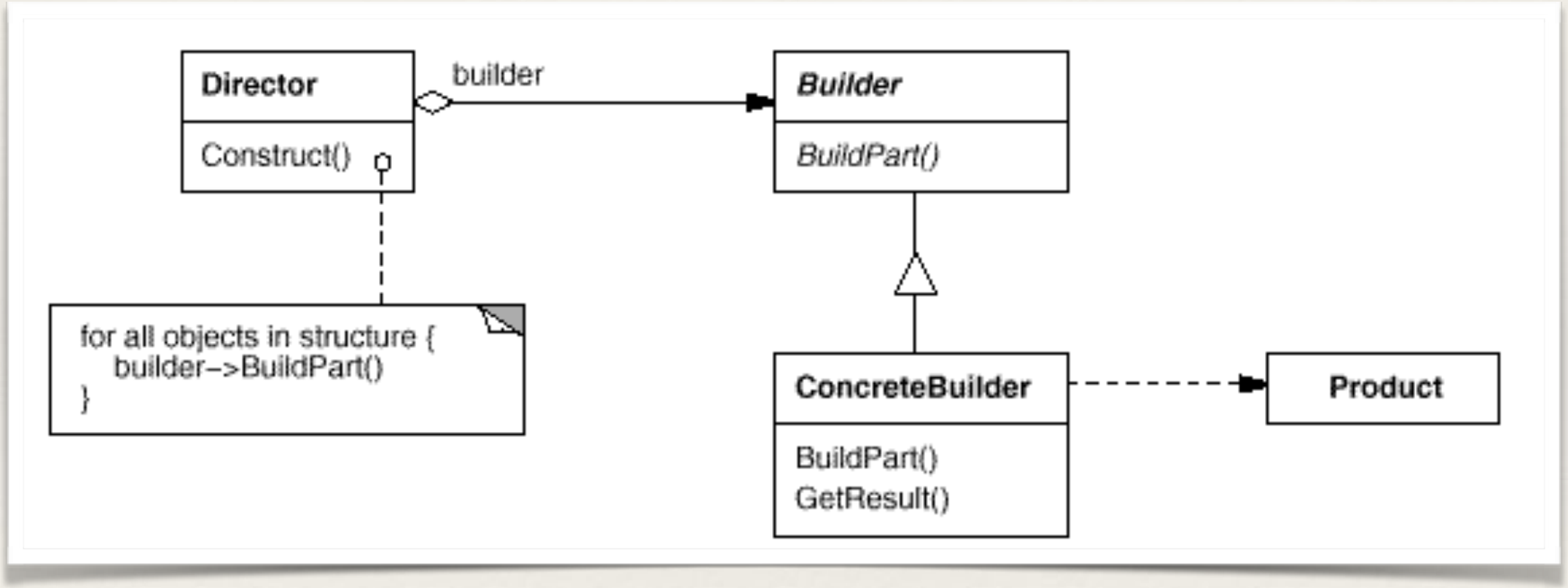
```
public Locale (String language,                  // e.g. “en" for English
               String script,                    // e.g., “Arab” for Arabic
               String country,                   // e.g., “us” for United States
               String variant,                   // e.g., “TH” for Thai
               LocaleExtensions extensions)      // e.g., “ca-buddhist” for Thai Buddhist Calendar
```

Recommended solution

- Create a Locale Builder that “builds” and returns an object step-by-step
- Validation will be performed by the individual set methods
- The build() method will return the “built” object

```
Locale aLocale =
                    new Locale.Builder()
                    .setLanguage(“sr")
                    .setScript(“Latn")
                    .setRegion("RS")
                    .build();
```

Builder pattern: structure



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

Builder 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

Most common example: StringBuilder

```
StringBuilder builder = new StringBuilder(64);
if (scheme != null) {
    builder.append(scheme);
    builder.append(':');
}
if (ssp != null) {
    builder.append(ssp);
}
return builder.toString();
```

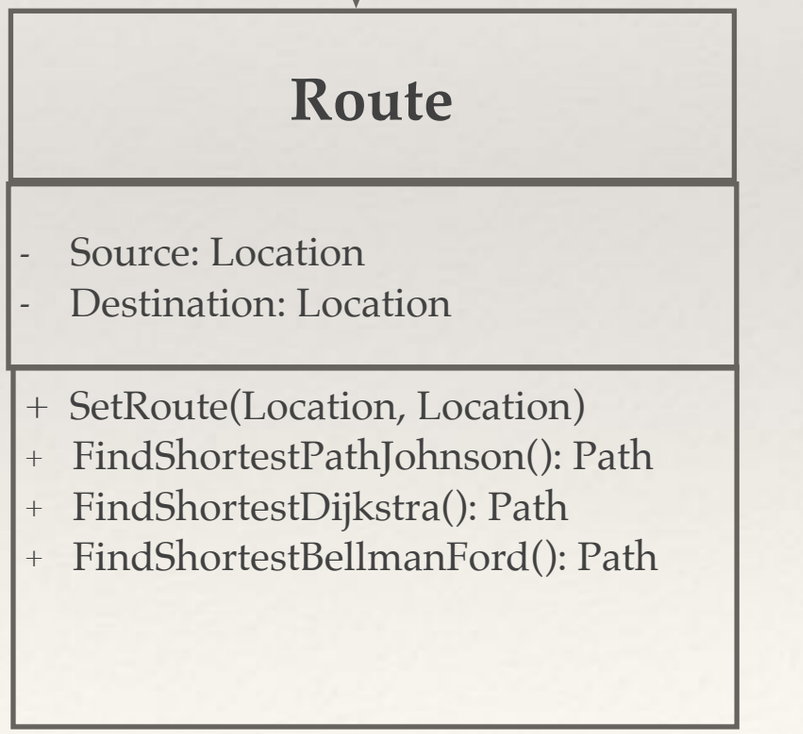
Builders common for complex classes

```
Calendar.Builder b = new Calendar.Builder();
Calendar calendar = b
                        .set(YEAR, 2003)
                        .set(MONTH, APRIL)
                        .set(DATE, 6)
                        .set(HOUR, 15)
                        .set(MINUTE, 45)
                        .set(SECOND, 22)
                        .setTimeZone(TimeZone.getDefault())
                        .build();
System.out.println(calendar);
```

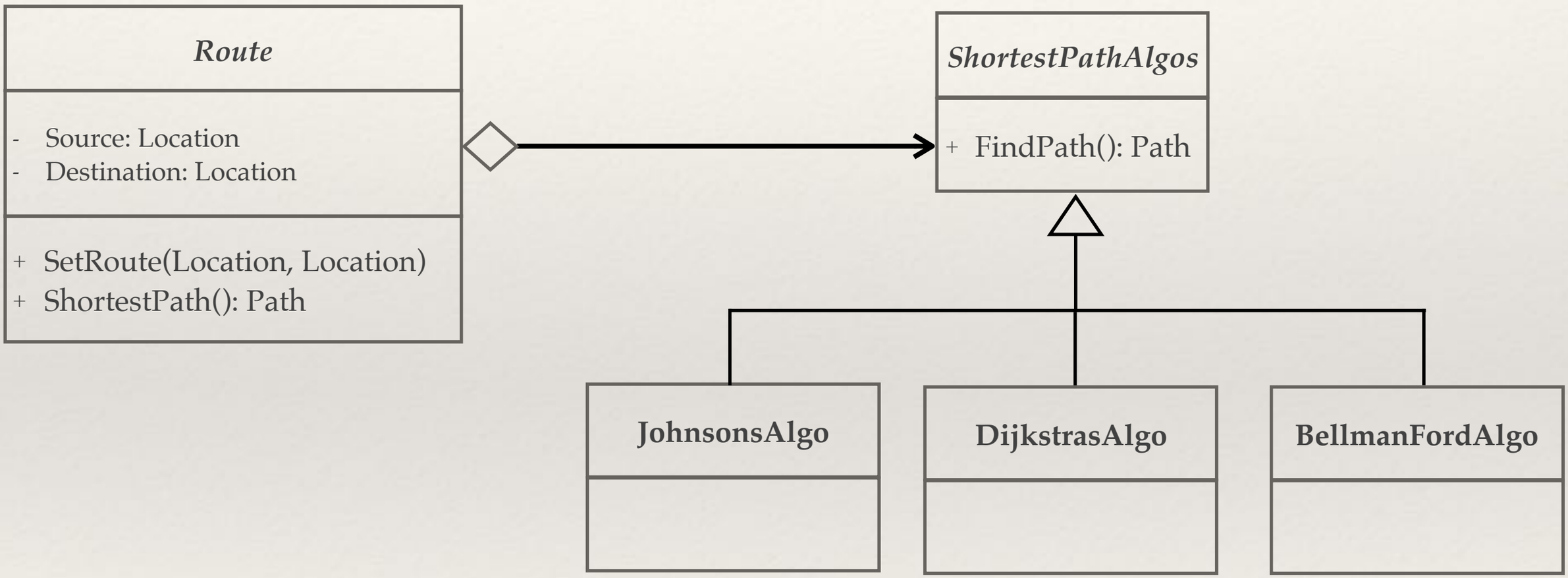
Scenario

- Consider a Route class in an application like Google Maps
- For finding shortest path from source to destination, many algorithms can be used
- The problem is that these algorithms get embedded into Route class and cannot be reused easily (smell!)

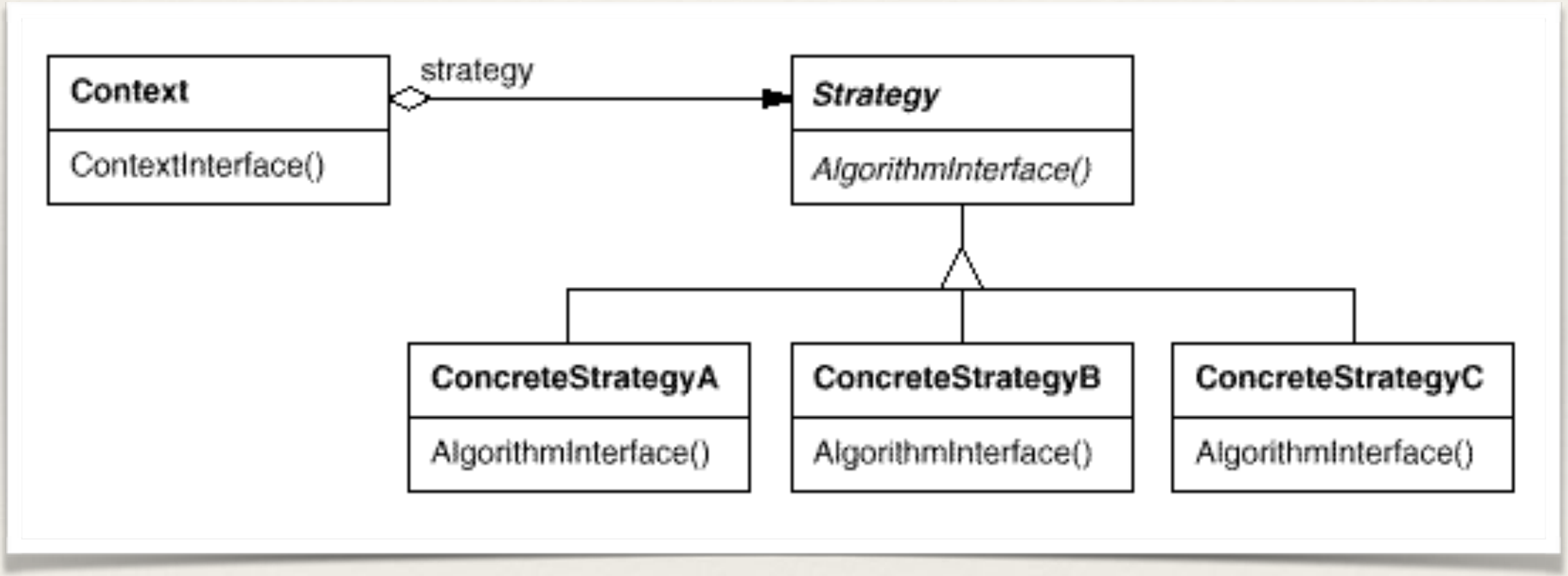
How will you refactor such that
a) Support for shortest path algorithm can be added easily?
b) Separate path finding logic from dealing with location information.



How about this solution?



You're right: Its Strategy pattern!



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

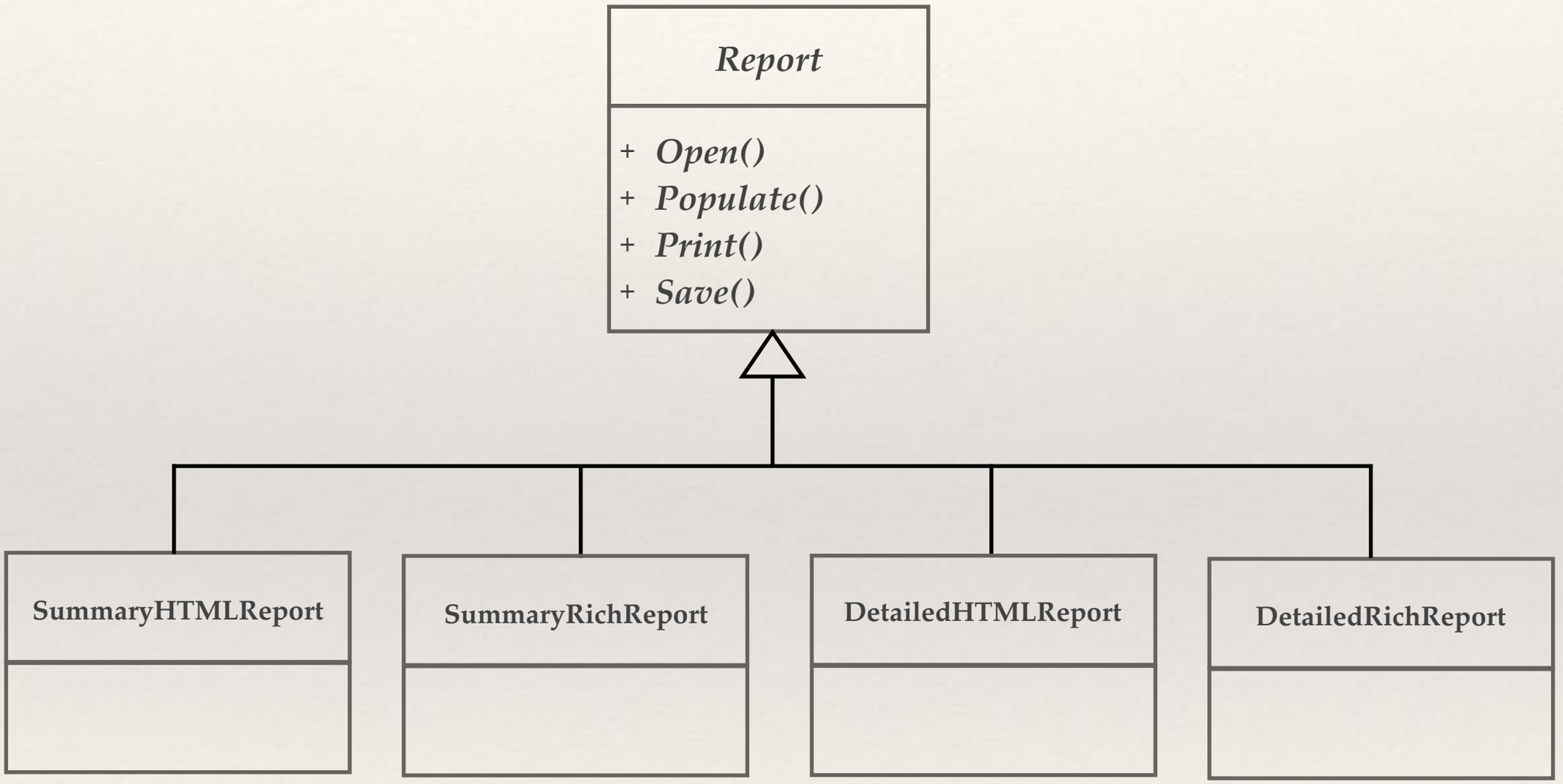
Strategy pattern: Discussion

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

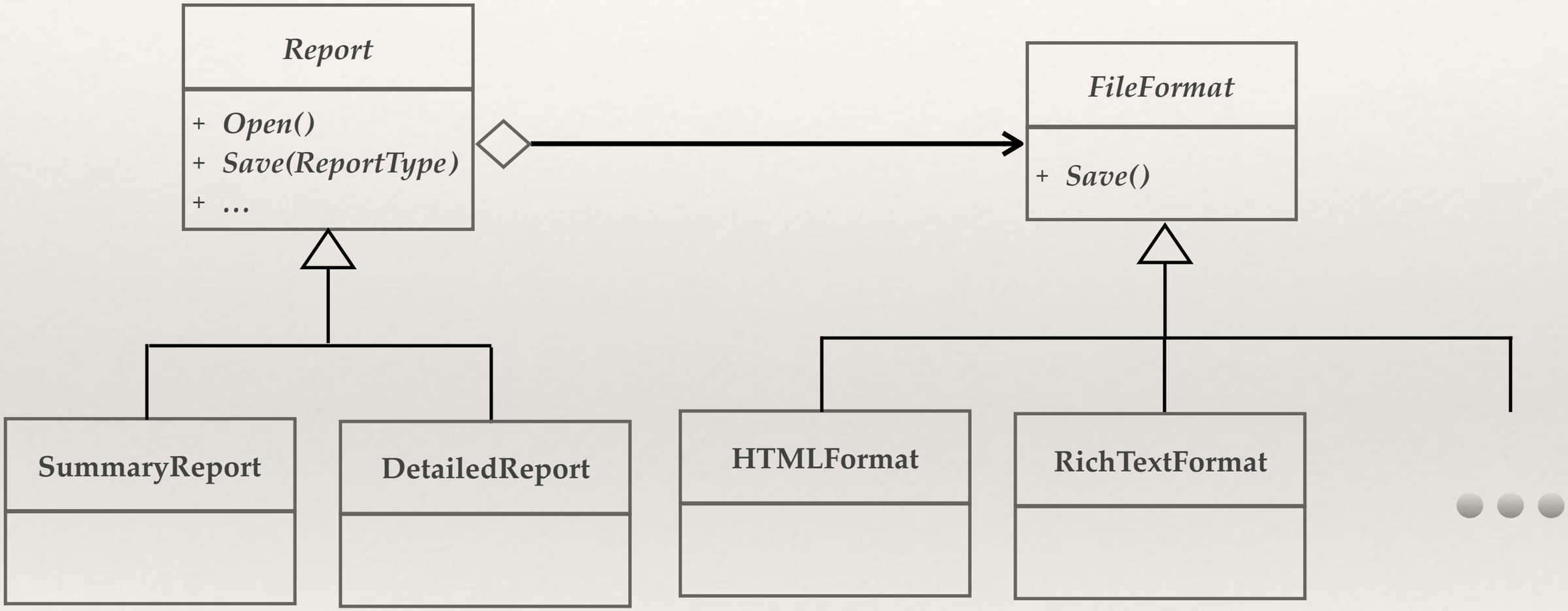
- Useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm that suits its current need

- 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

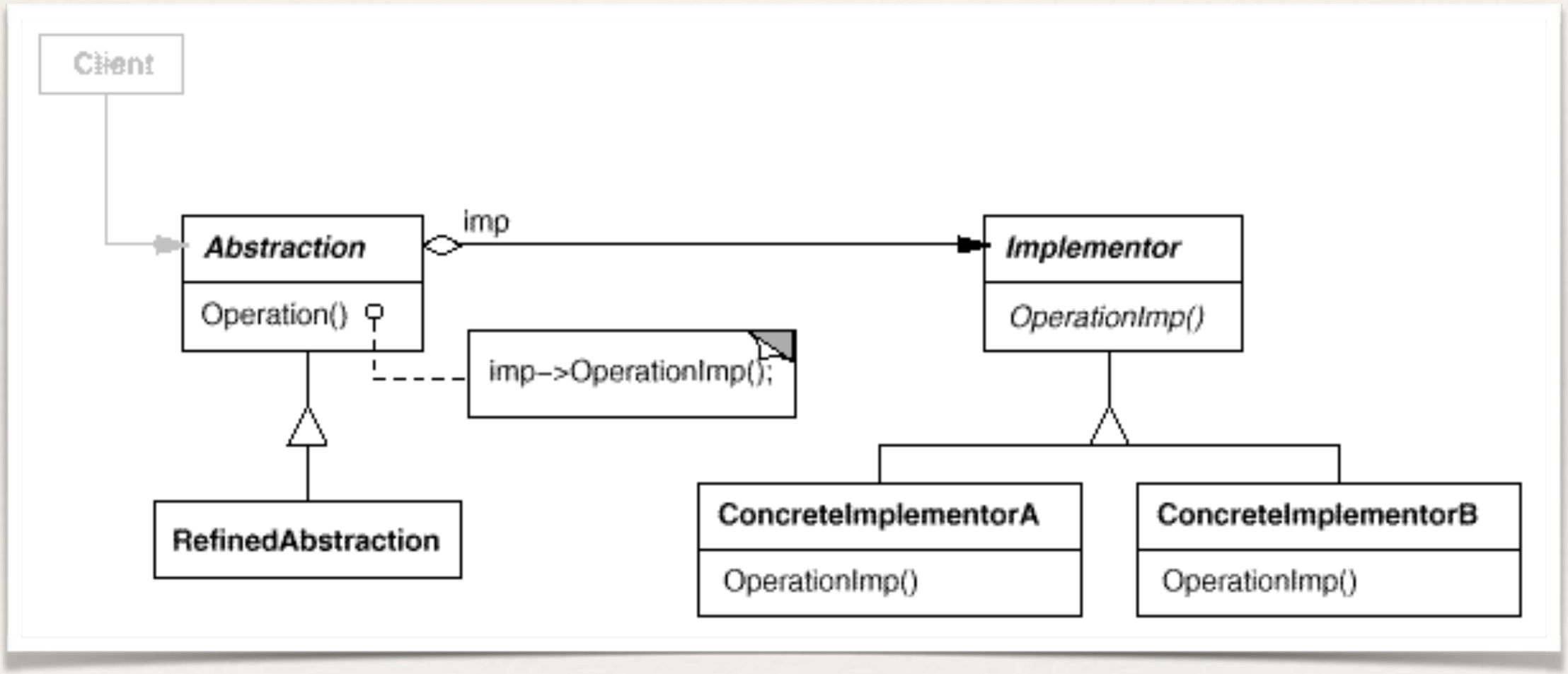
Scenario


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

How about this solution?

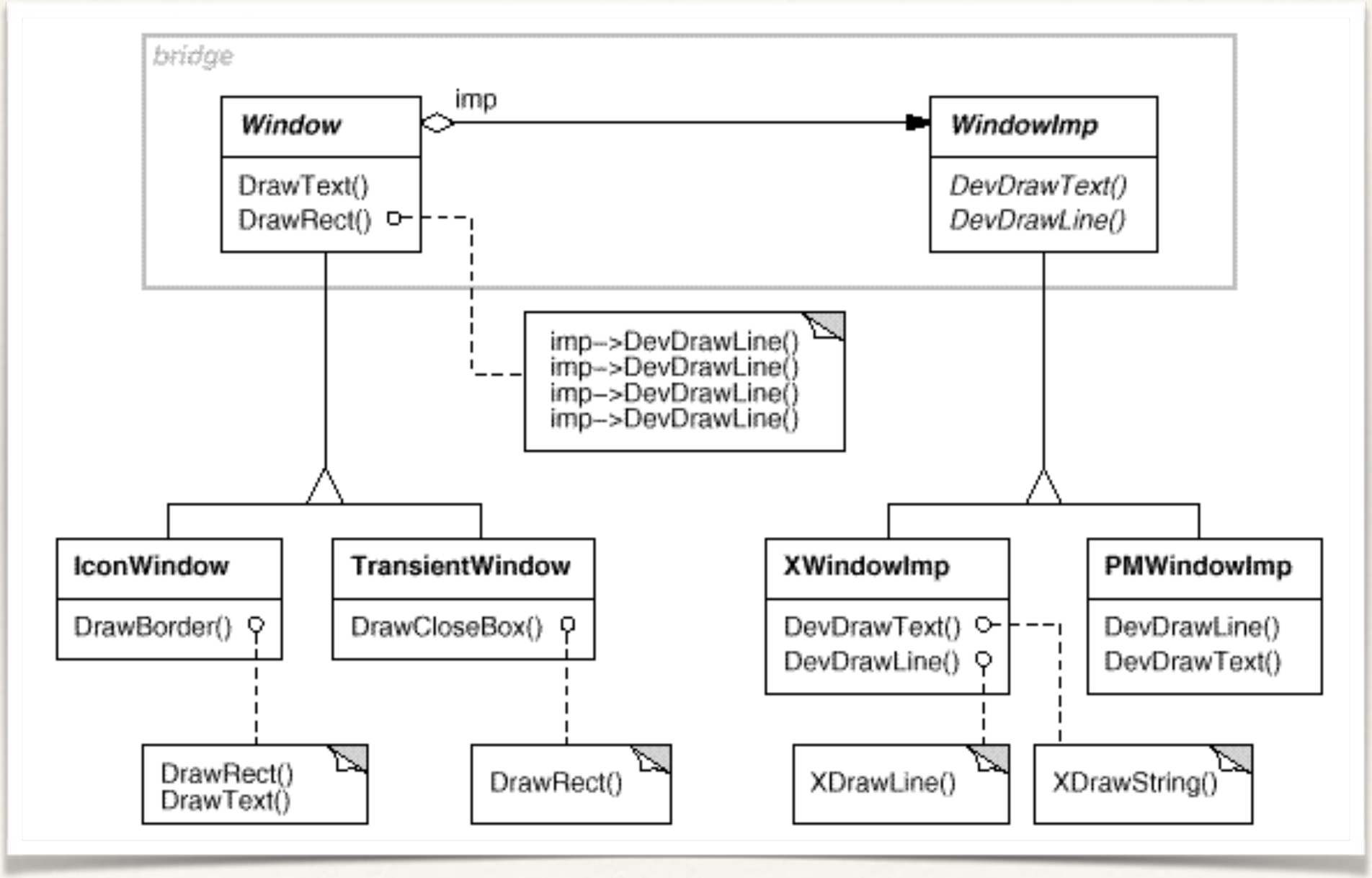


You're right: Its Bridge pattern!



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

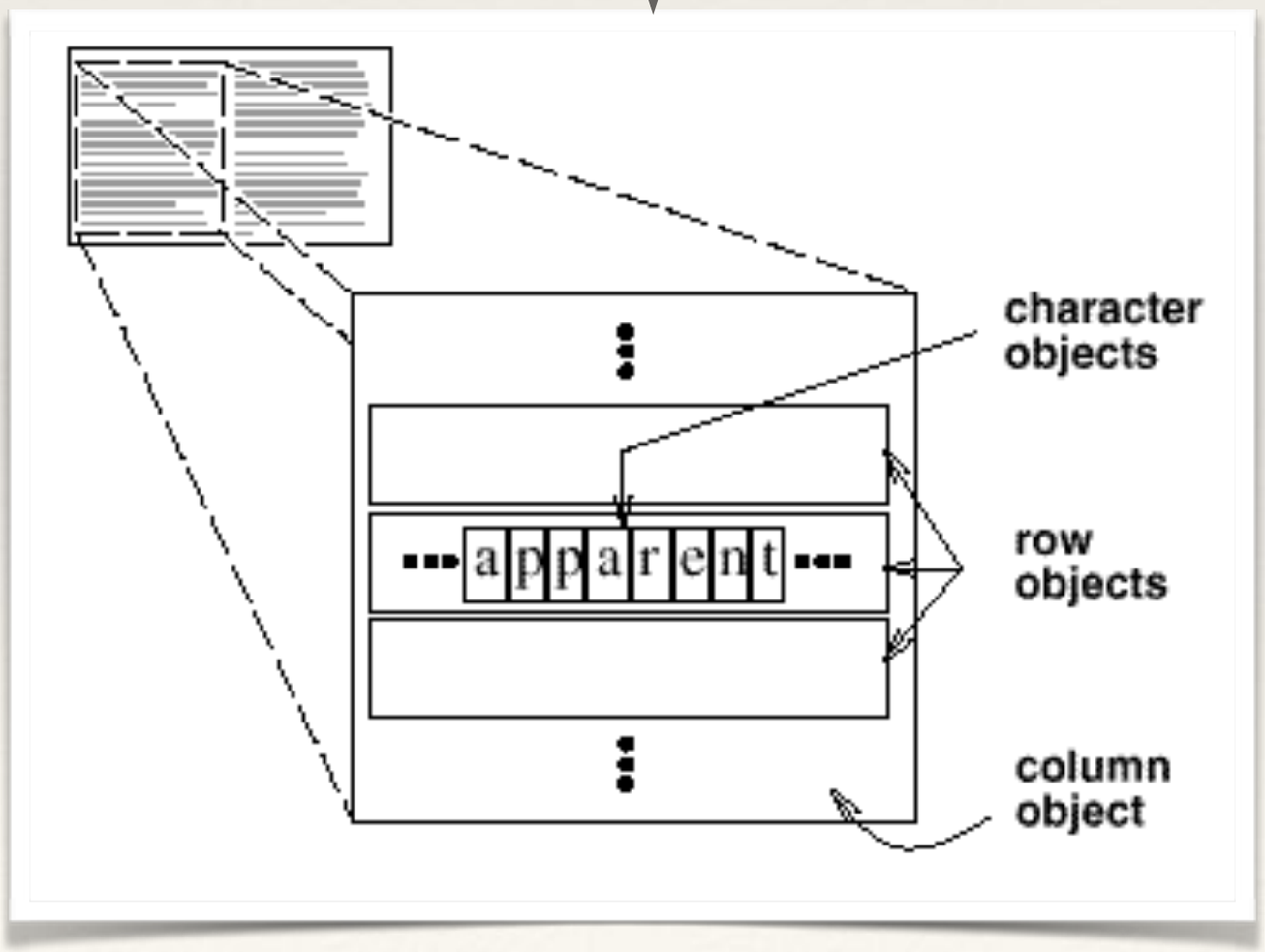
Decouples an abstraction from its implementation so that the two can vary independently

- 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

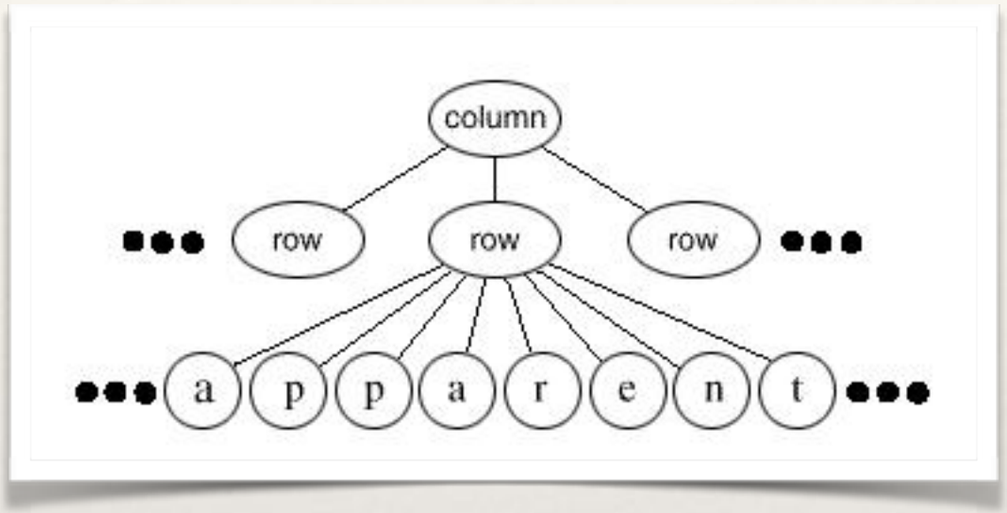
Scenario

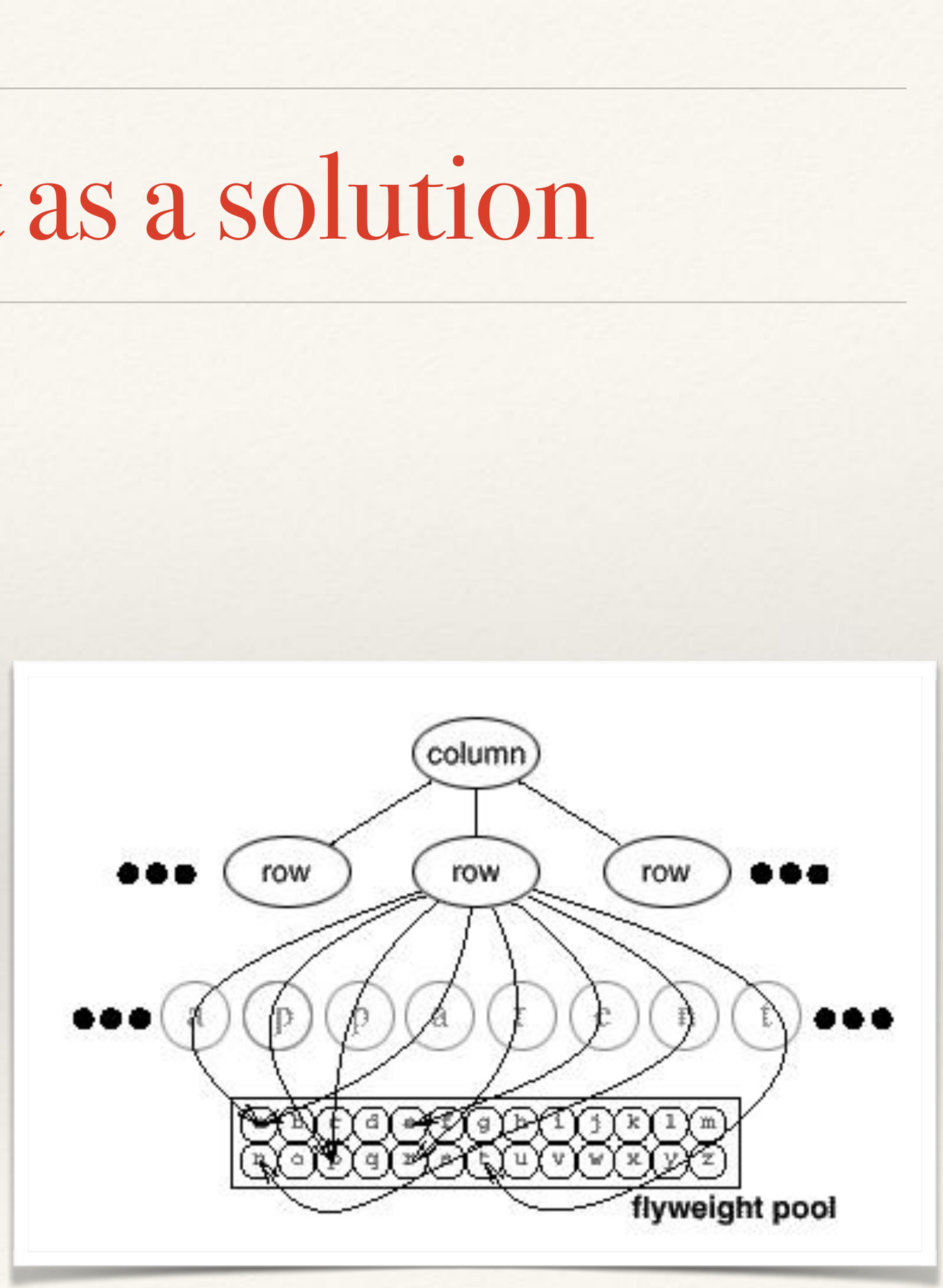
How will you design to share the characters to save space?



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

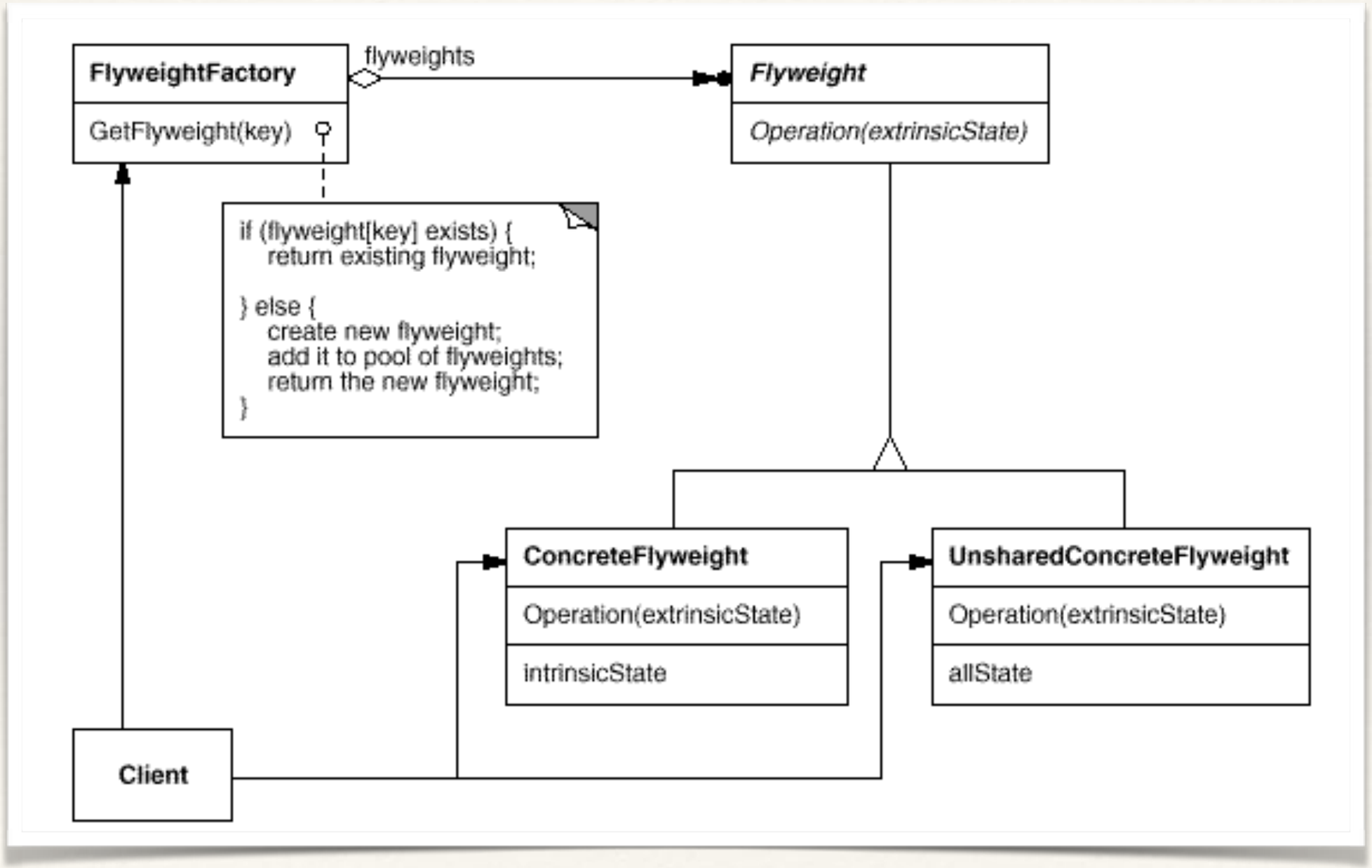
Flyweight as a solution





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

How can you “cache” common objects?

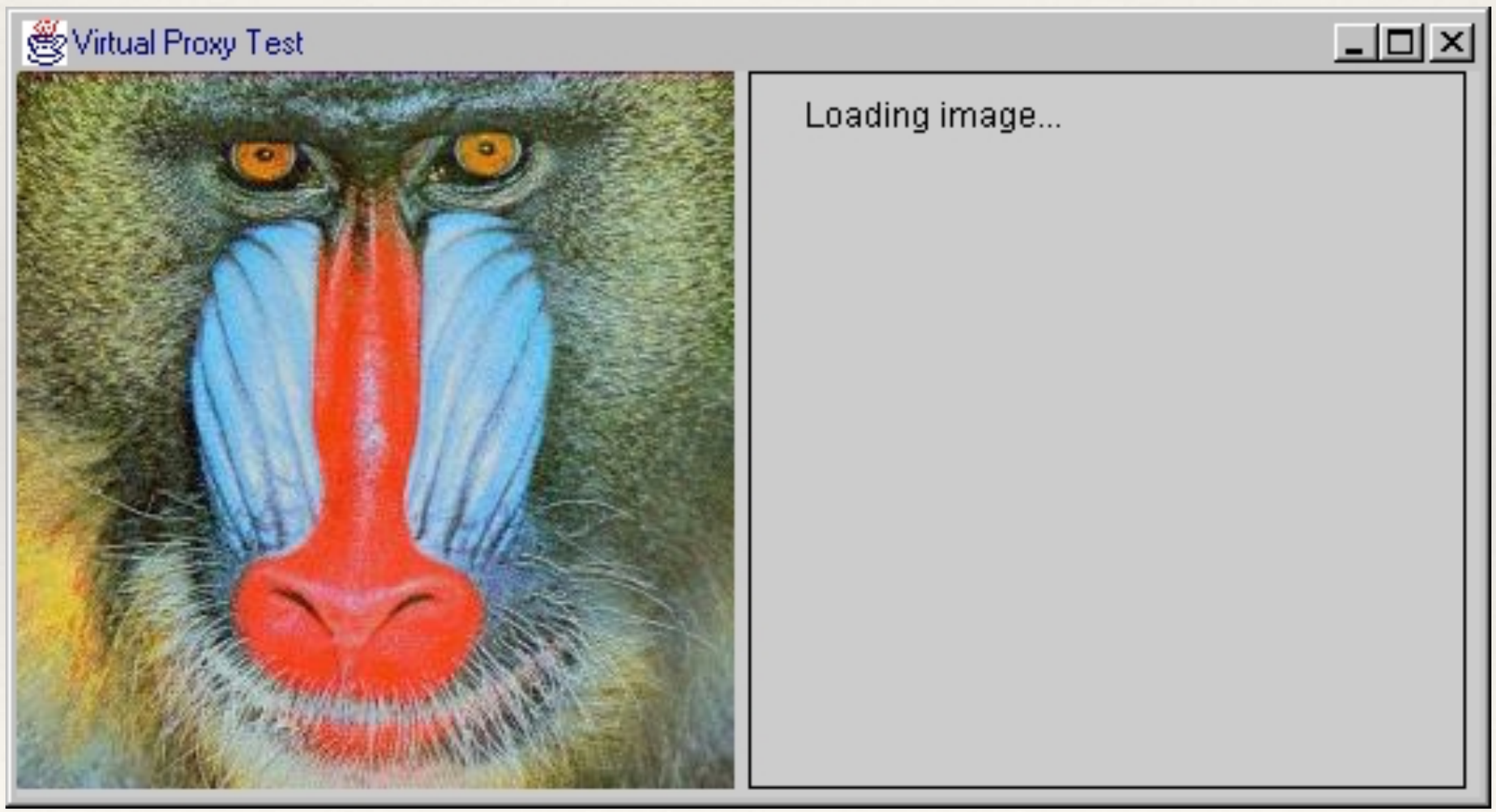
```
public static Integer valueOf(int i) {
        // insert your code here to “reuse” the common objects
        return new Integer(i);
}
```

Simplified flyweight in Java library

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

Scenario

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



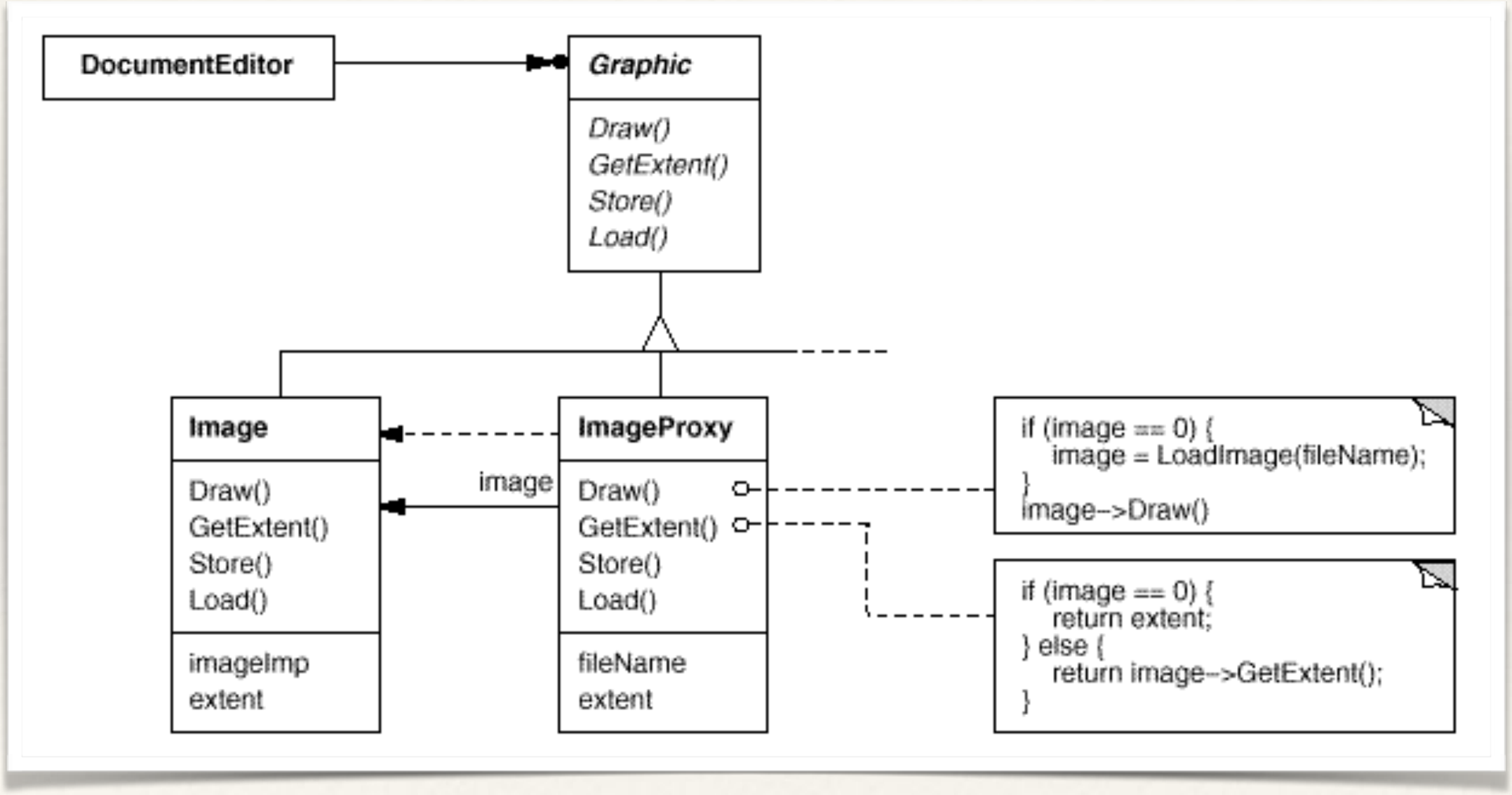
Source: http://www.javaworld.com/article/2074068/learn-java/take-control-with-the-proxy-design-pattern.html

A solution using Proxy pattern

```
// The proxy's paint() method is overloaded to draw a border
// and a message ("Loading image...") while the image
// loads. After the image has loaded, it is drawn. Notice
// that the proxy does not load the image until it is
// actually needed.
public void paintIcon(final Component c,
                      Graphics g, int x, int y) {
   if(isIconCreated) {
      realIcon.paintIcon(c, g, x, y);
   }
   else {
      g.drawRect(x, y, width-1, height-1);
      g.drawString("Loading image...", x+20, y+20);
      // The icon is created (meaning the image is loaded)
      // on another thread.
      synchronized(this) {
         SwingUtilities.invokeLater(new Runnable() {
              public void run() {
               try {
                  // Slow down the image-loading process.
                  Thread.currentThread().sleep(2000);
                  // ImageIcon constructor creates the image.
                  realIcon = new ImageIcon(imageName);
                  isIconCreated = true;
               }
               catch(InterruptedException ex) {
                    ex.printStackTrace();
               }
               // Repaint the icon's component after the
               // icon has been created.
               c.repaint();
            }
         });
      }
   }
}
```

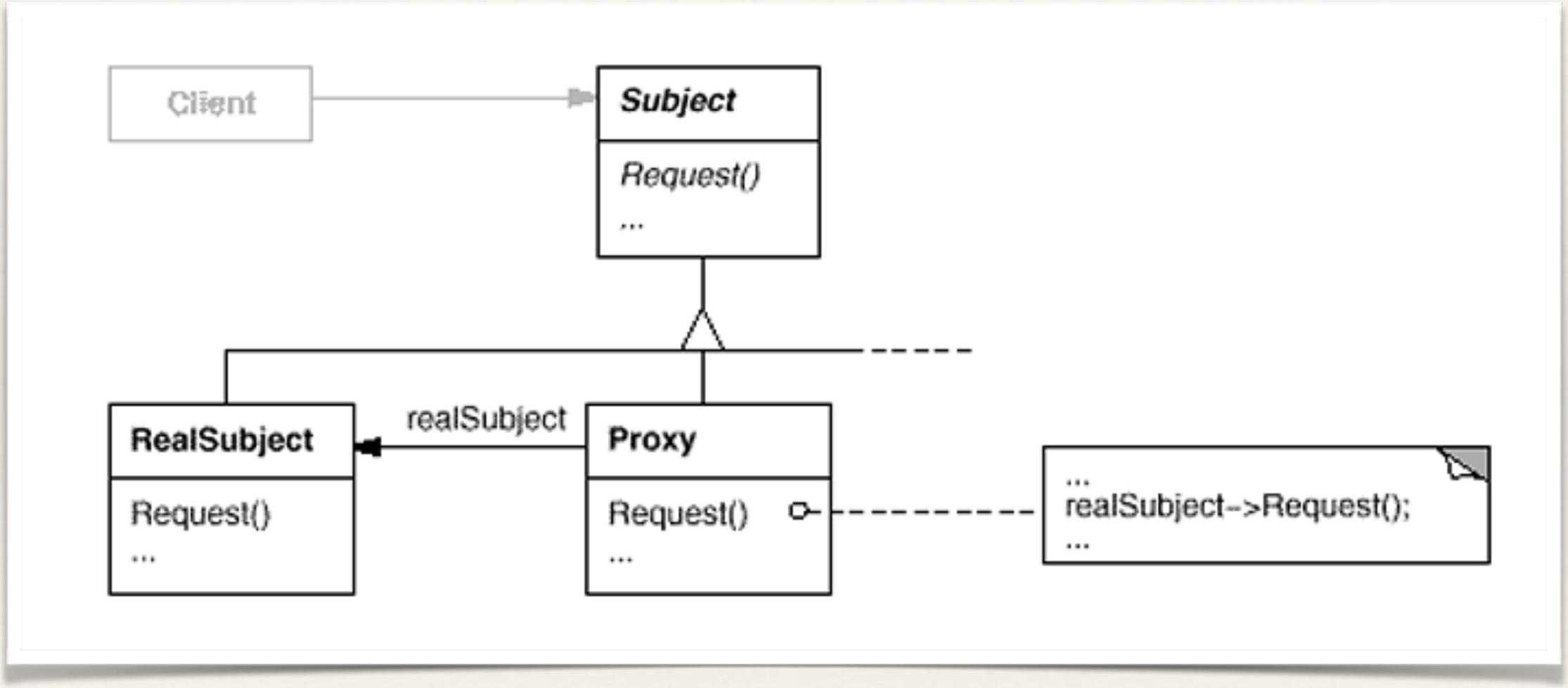
Source: http://www.javaworld.com/article/2074068/learn-java/take-control-with-the-proxy-design-pattern.html

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?

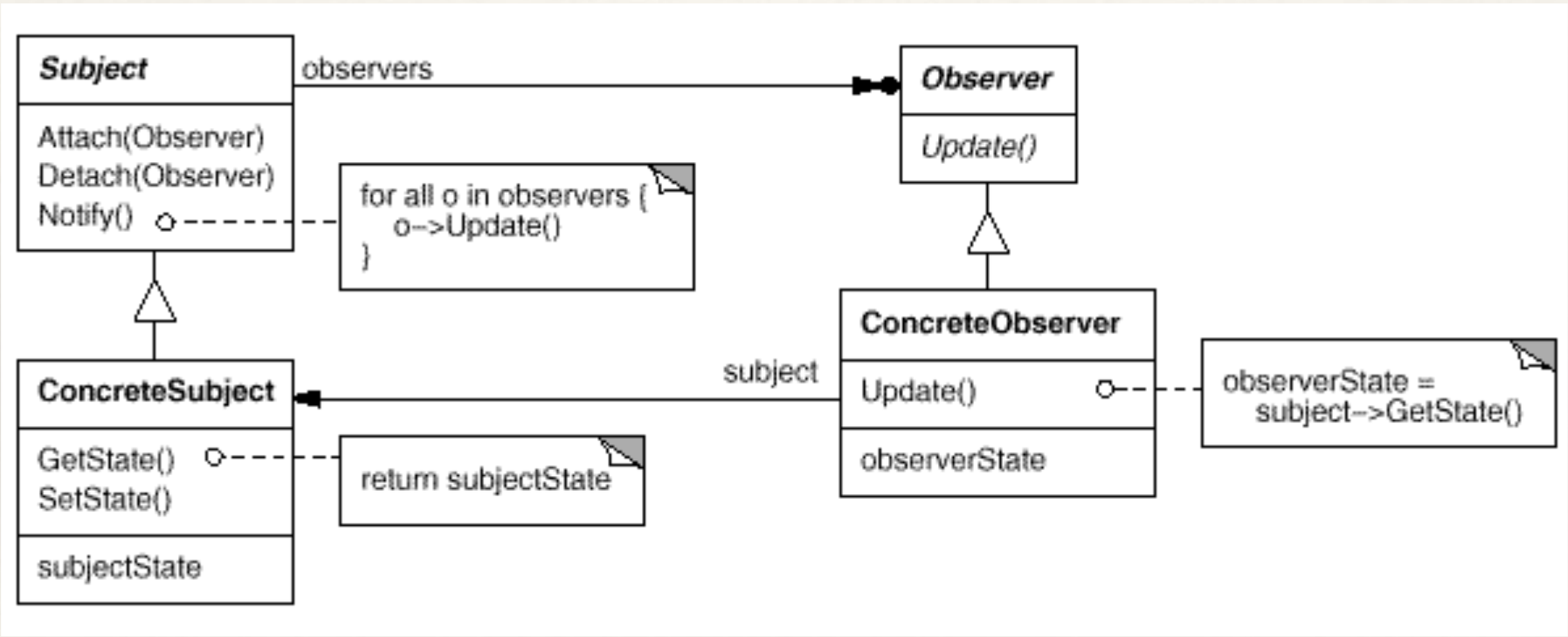
- When it is possible to share objects
  - When object's value remain the same irrespective of the contexts - they can be shared
- Share the commonly used objects in a pool

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



Scenario

How to exploit:
a) common code segments?
b) provide extensibility for supporting other kinds of compressed files (e.g., rar)?

- You have code segments and steps for creating different kinds of compressed files, such as zip file and gzip files. How will you unify the common steps for compressing files and support new compression formats easily?

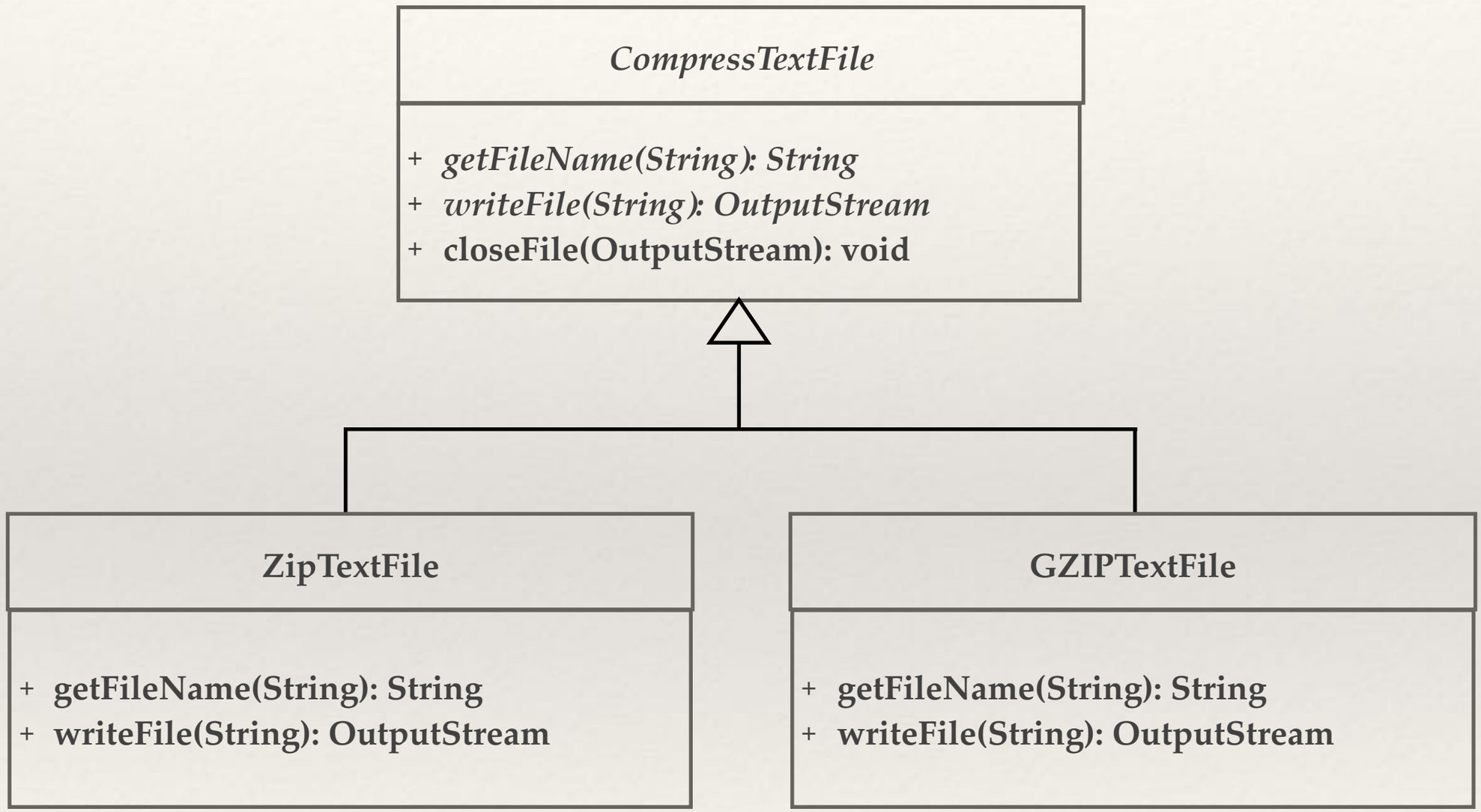
zip file creation example

```
import java.util.zip.*;
import java.io.*;

// class ZipTextFile takes the name of a text file as input and creates a zip file
// after compressing that text file.
class ZipTextFile {
     private static final int CHUNK = 1024; // to help copy chunks of 1KB
     public ZipTextFile(String fileName) throws Exception {
          String zipFileName = getZipFileName(fileName);
          OutputStream zipFile = writeZipFile(zipFileName);
          closeZipFile(zipFile);
     }

     private String getZipFileName(String fileName) {
          // name of the zip file is the input file name with the suffix ".zip"
          return fileName + ".zip";
     }

     private OutputStream writeZipFile(String zipFileName) throws Exception {
             byte [] buffer = new byte[CHUNK];
          // these constructors can throw FileNotFoundException
          ZipOutputStream zipFile = new ZipOutputStream(new FileOutputStream(zipFileName));
          FileInputStream fileIn = new FileInputStream(zipFileName);
               // putNextEntry can throw IOException
               zipFile.putNextEntry(new ZipEntry(zipFileName));
               int lenRead = 0; // variable to keep track of number of bytes successfully read
               // copy the contents of the input file into the zip file
               while((lenRead = fileIn.read(buffer)) > 0) {
                    // both read and write methods can throw IOException
                    zipFile.write (buffer, 0, lenRead);
               }
          return zipFile;
         }

     private void closeZipFile(OutputStream zipFile) throws Exception {
          zipFile.flush();
          zipFile.close();
     }

     public static void main(String []args) throws Exception {
          if(args.length == 0) {
                    System.out.println("Pass the name of the file in the current directory to be zipped as an argument");
                    System.exit(-1);
          }
          new ZipTextFile(args[0]);
     }
}
```

gzip file creation example

```
import java.util.zip.*;
import java.io.*;

// class GZIPTextFile takes the name of a text file as input and creates a gzip file
// after compressing that text file.
class GZIPTextFile {
     private static final int CHUNK = 1024; // to help copy chunks of 1KB
     public GZIPTextFile(String fileName) throws Exception {
          String gzipFileName = getGZIPFileName(fileName);
          OutputStream gzipFile = writeGZIPFile(gzipFileName);
          closeGZIPFile(gzipFile);
     }

     private String getGZIPFileName(String fileName) {
          // name of the gzip file is the input file name with the suffix ".gzip"
          return fileName + ".gzip";
     }

     private OutputStream writeGZIPFile(String gzipFileName) throws Exception {
             byte [] buffer = new byte[CHUNK];
          // these constructors can throw FileNotFoundException
          GZIPOutputStream gzipFile = new GZIPOutputStream(new FileOutputStream(gzipFileName));
          FileInputStream fileIn = new FileInputStream(gzipFileName);
               int lenRead = 0; // variable to keep track of number of bytes successfully read
               // copy the contents of the input file into the zip file
               while((lenRead = fileIn.read(buffer)) > 0) {
                    // both read and write methods can throw IOException
                    gzipFile.write (buffer, 0, lenRead);
               }
          return gzipFile;
        }

     private void closeGZIPFile(OutputStream gzipFile) throws Exception {
          gzipFile.flush();
          gzipFile.close();
     }

     public static void main(String []args) throws Exception {
          if(args.length == 0) {
                   System.out.println("Pass the name of the file in the current directory to be gzipped as an argument");
                   System.exit(-1);
          }
          new GZIPTextFile(args[0]);
     }
}
```

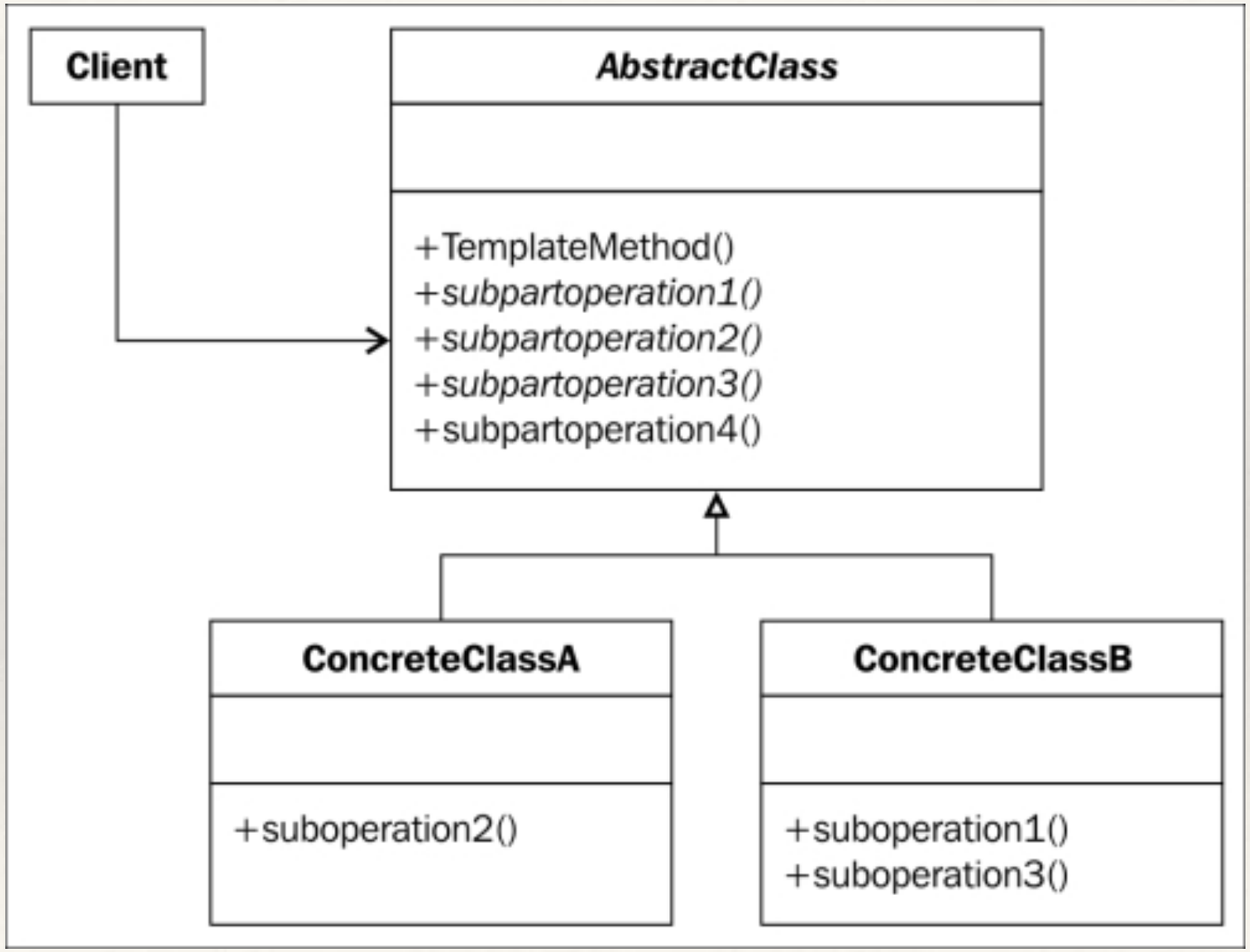
Refactoring to Template Method



Hands-on exercise

- Refactor “ZipTextFile.java” and “GZIPTextFile.java” to use template method design pattern

Template Method: Structure

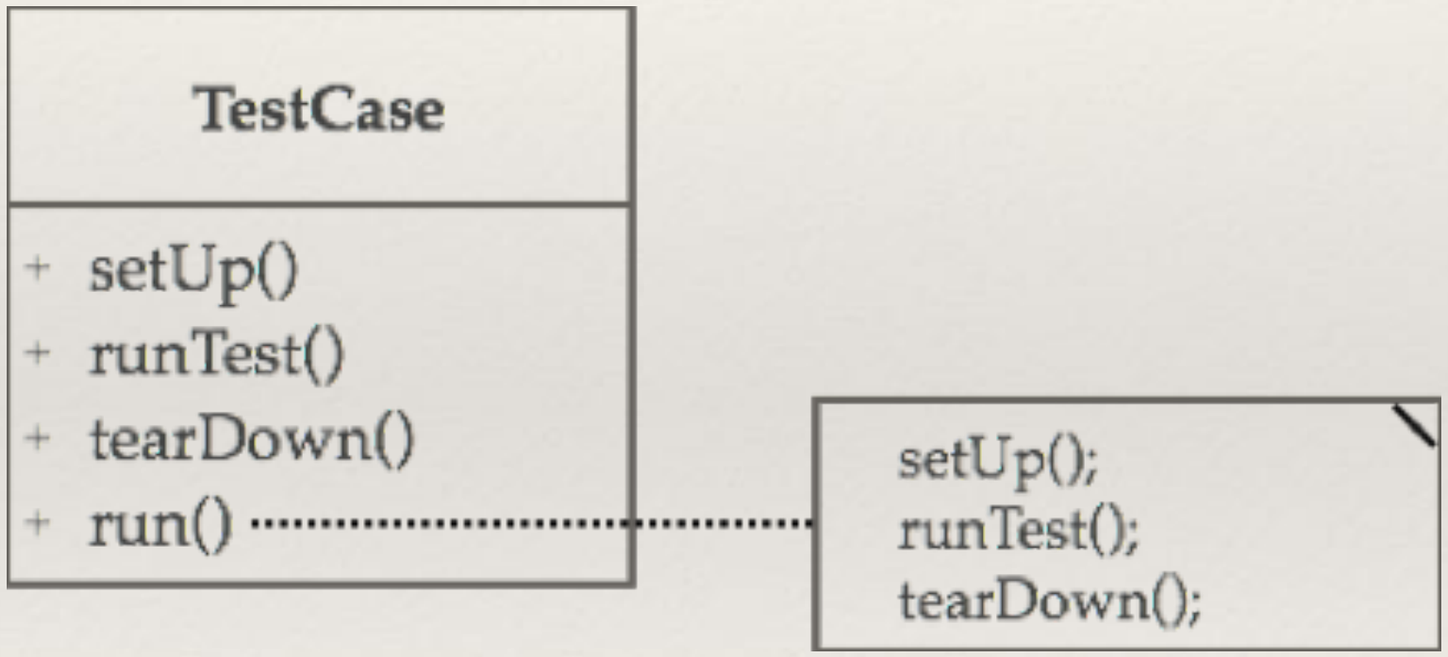


Source: https://www.safaribooksonline.com/library/view/swift-2-design/9781785887611/graphics/4582_05_07.jpg

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.

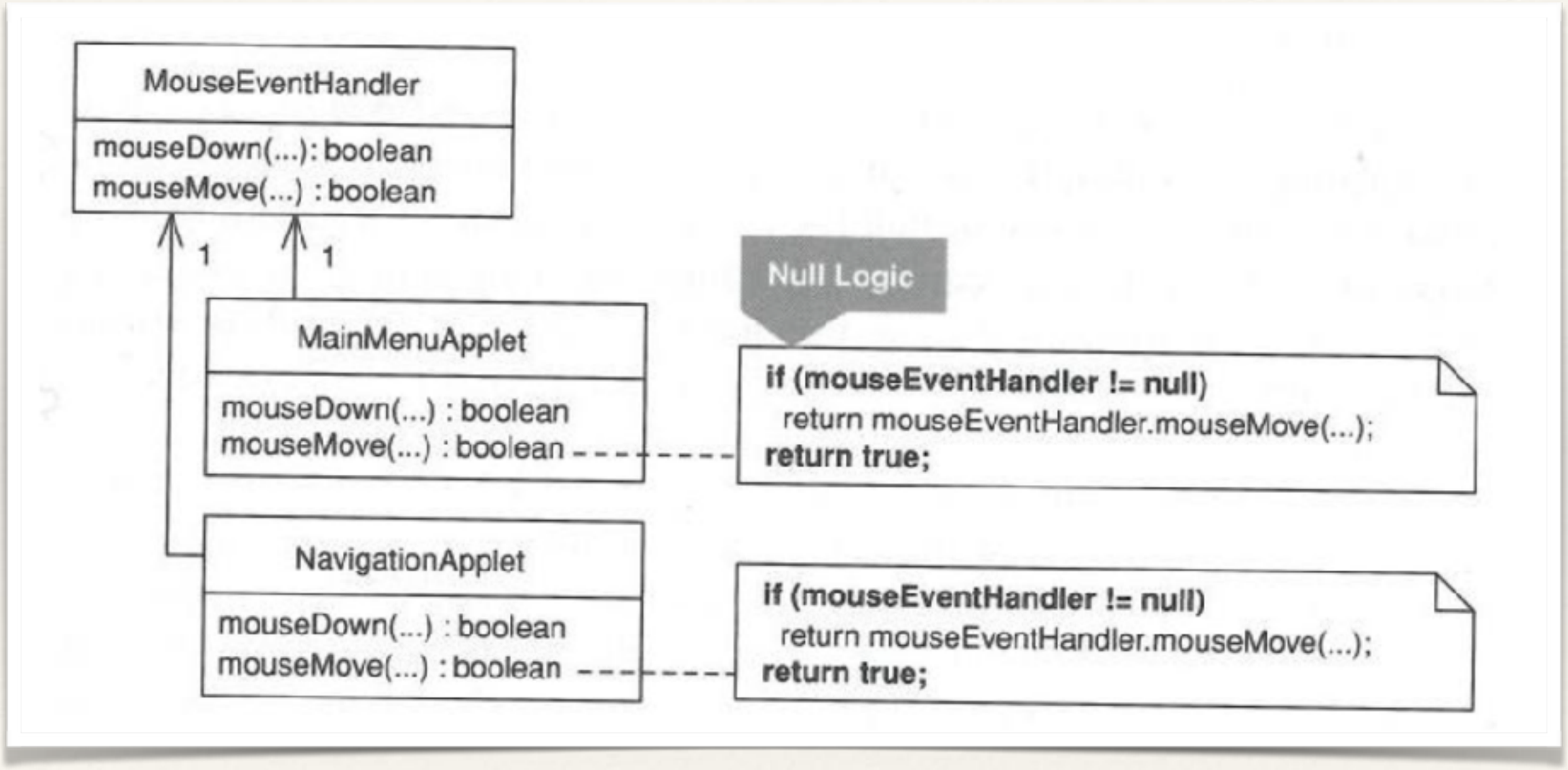
Template Method in JUnit



Patterns discussed so far

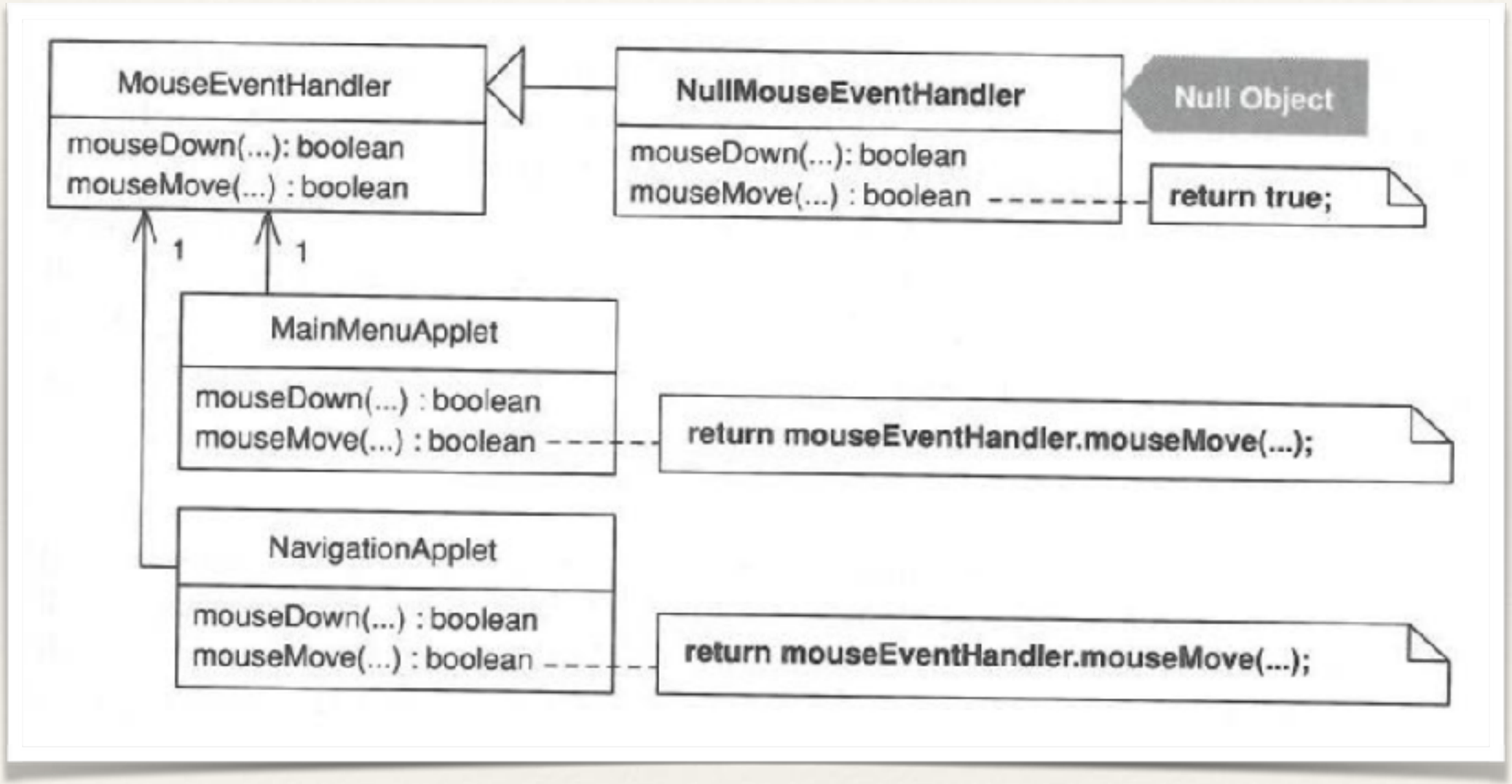
- ✓ Builder pattern
- ✓ Factory method pattern
- ✓ Strategy pattern
- ✓ Bridge pattern
- ✓ Decorator pattern
- ✓ Observer pattern
- ✓ Composite pattern
- ✓ Flyweight pattern
- ✓ Proxy pattern
- ✓ Visitor pattern
- ✓ Template method pattern
- ✓ Interpreter pattern

Many other patterns other than GoF



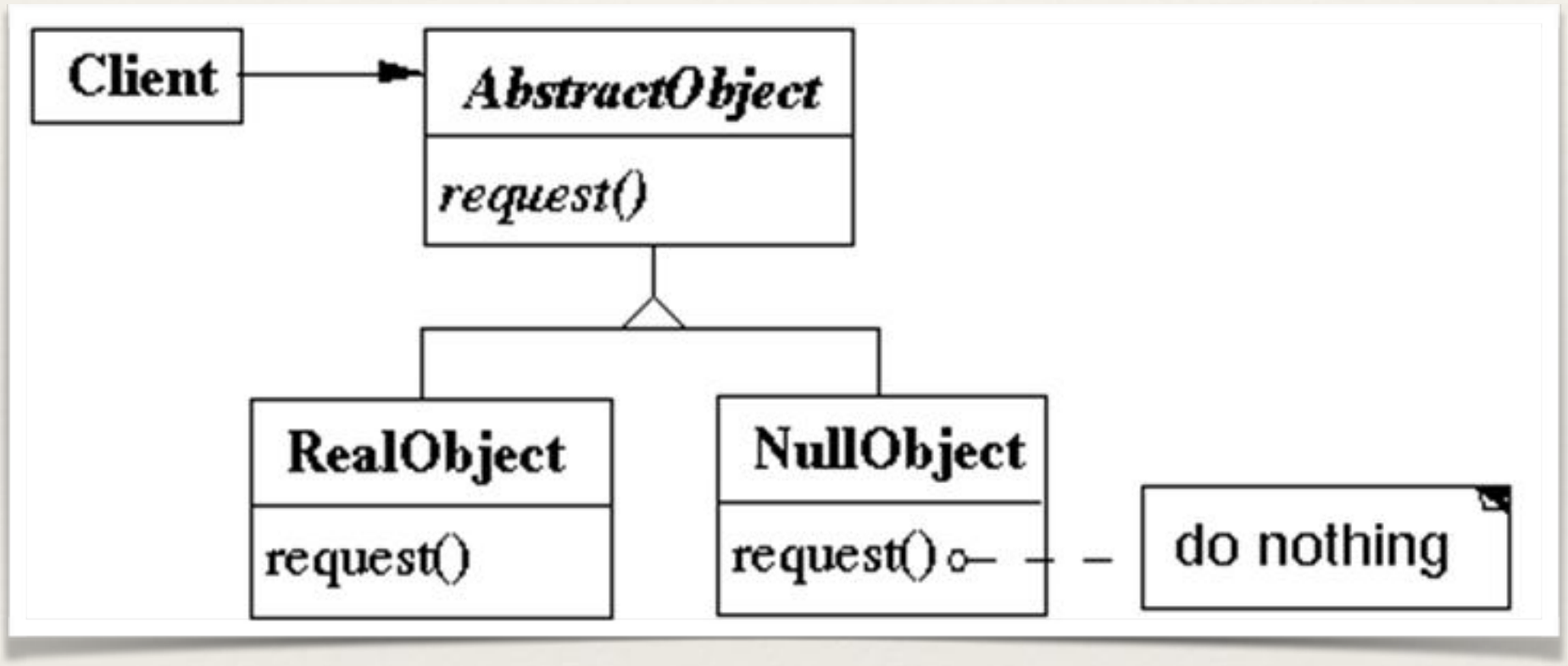
Source: “Refactoring to Patterns”, Joshua Kerievsky, Addison Wesley, 2004

Null Object pattern



Source: “Refactoring to Patterns”, Joshua Kerievsky, Addison Wesley, 2004

Null Object pattern: structure



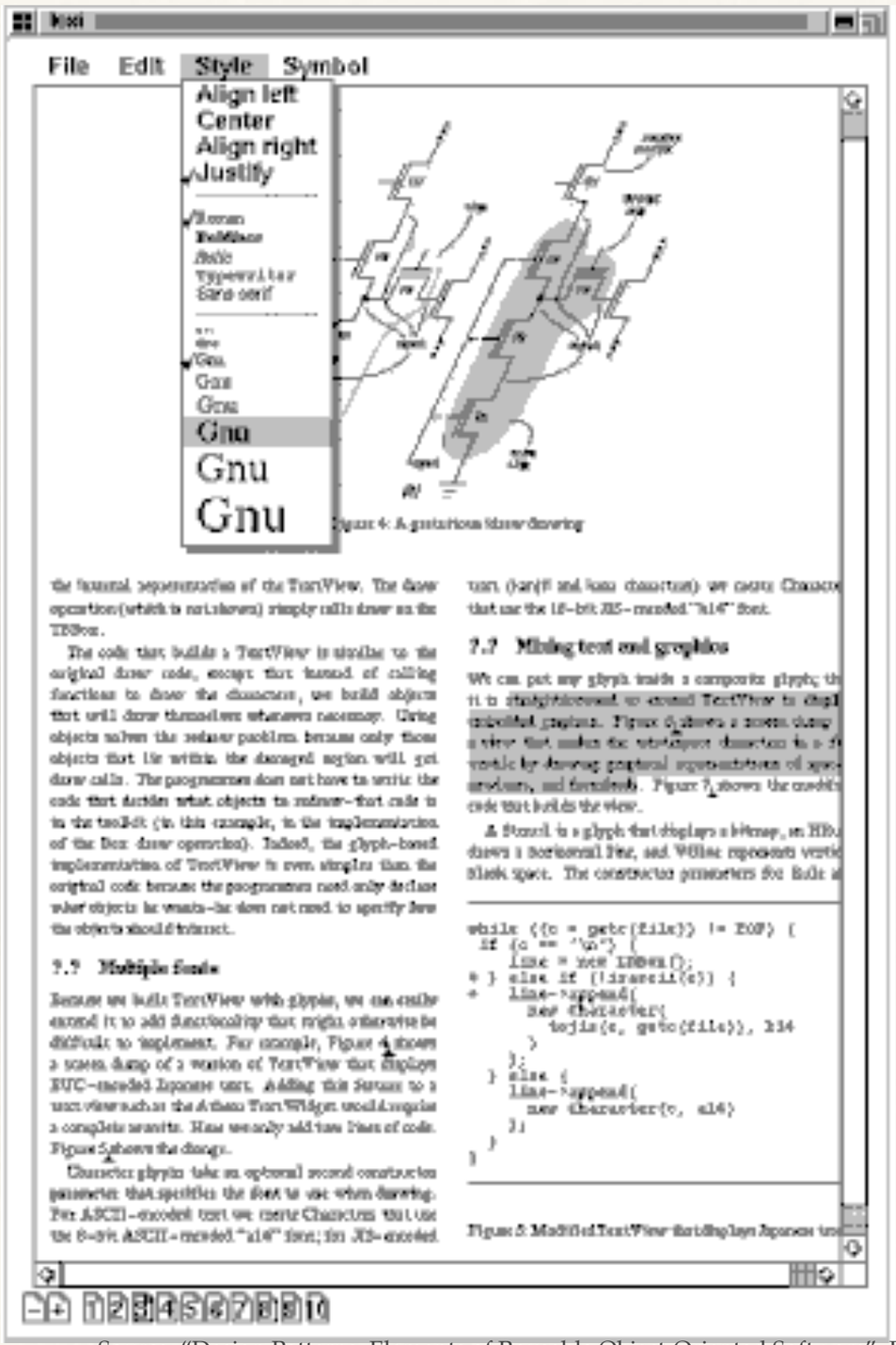
EmptySet from Java library

```
private static class EmptySet<E>
        extends AbstractSet<E>
        implements Serializable
    {
        private static final long serialVersionUID = 1582296315990362920L;

        public Iterator<E> iterator() { return emptyIterator(); }

        public int size() {return 0;}
        public boolean isEmpty() {return true;}

        public boolean contains(Object obj) {return false;}
        public boolean containsAll(Collection<?> c) { return c.isEmpty(); }

        public Object[] toArray() { return new Object[0]; }

        public <T> T[] toArray(T[] a) {
            if (a.length > 0)
                a[0] = null;
            return a;
        }

        // Preserves singleton property
        private Object readResolve() {
            return EMPTY_SET;
        }
    }
```

Agenda

- Introduction
- Design patterns though exercises
- **Patterns through a case-study**
- Tools for refactoring
- Wrap-up & key-takeaways

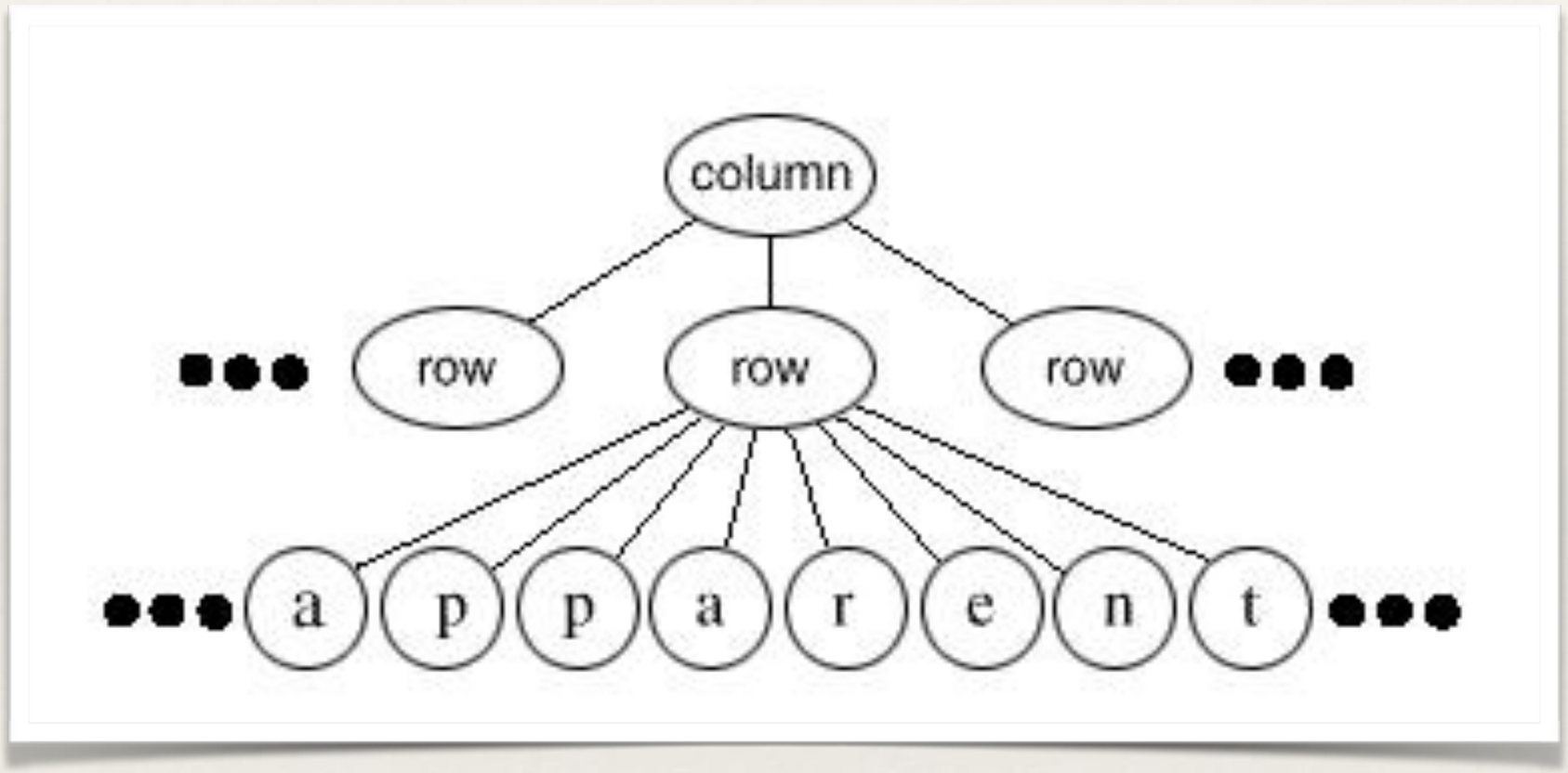
Designing a document editor



- How to maintain a document structure?
- How to support formatting?
- How to support embellishments (highlighting, marking, etc) with ease?
- Support multiple look-and-feel standards
- Support multiple windowing systems (mac, windows, motif, etc)
- How to support user operations and handle user events?
- How to support spellchecking and hyphenation?
- ….

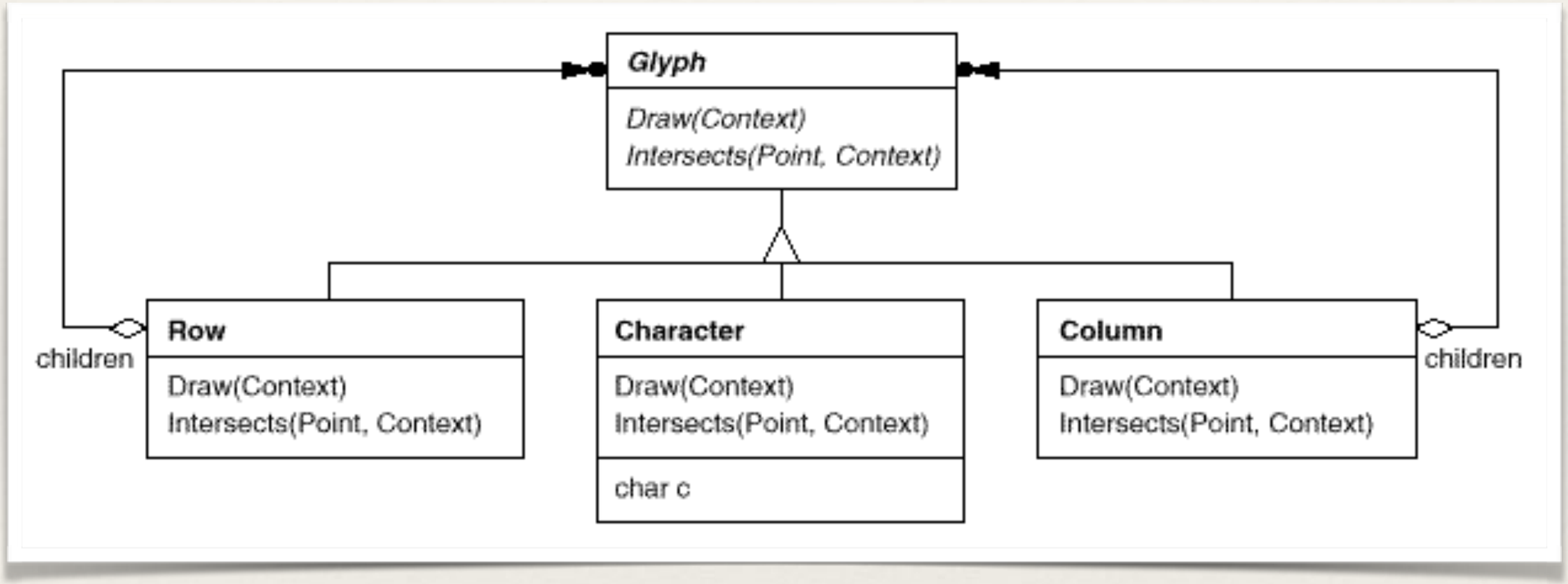
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

“Recursive composition”



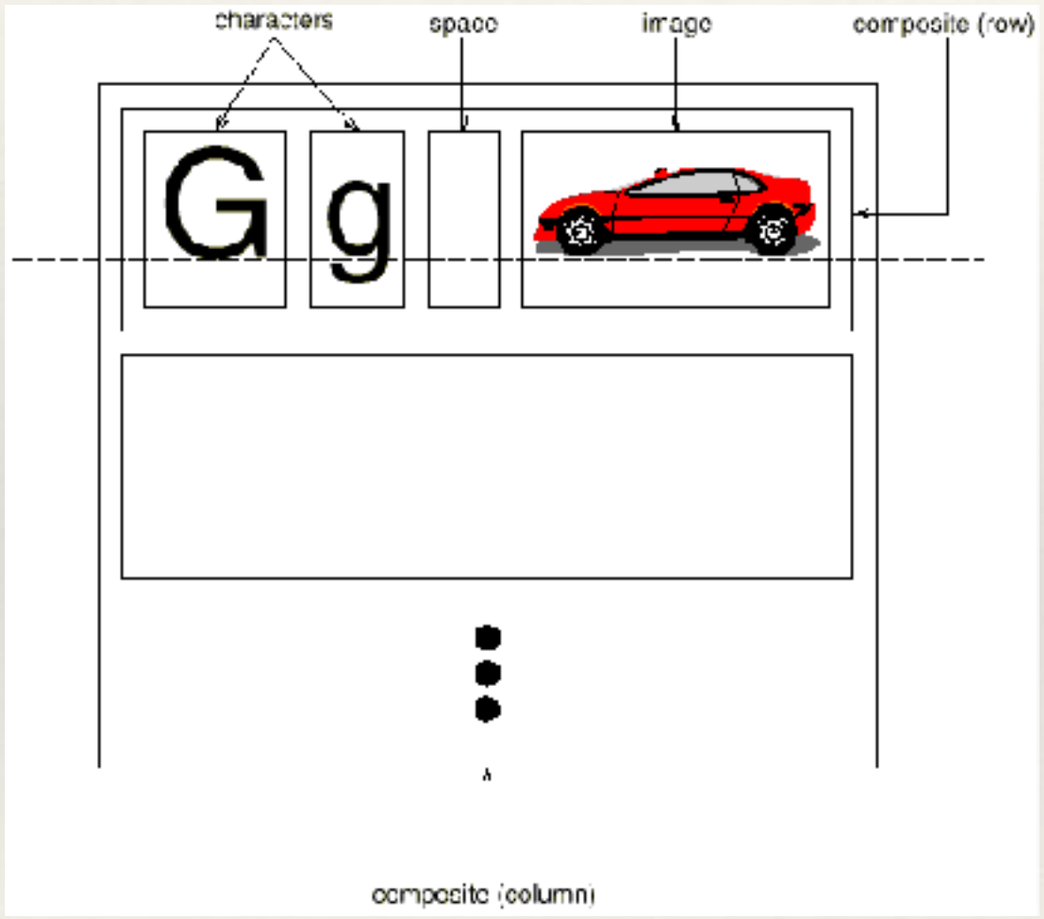
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

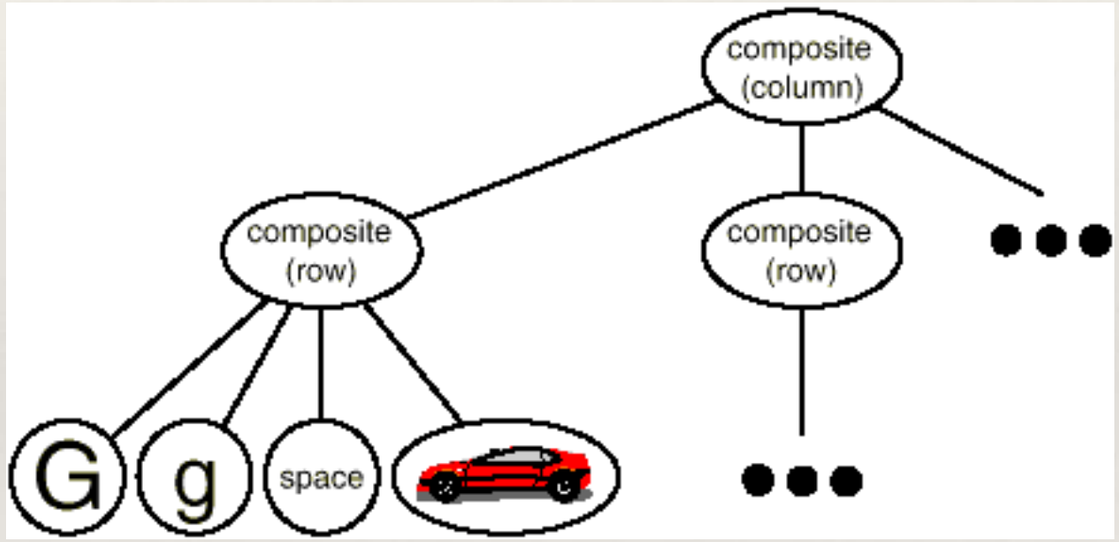
Solution using Composite pattern



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

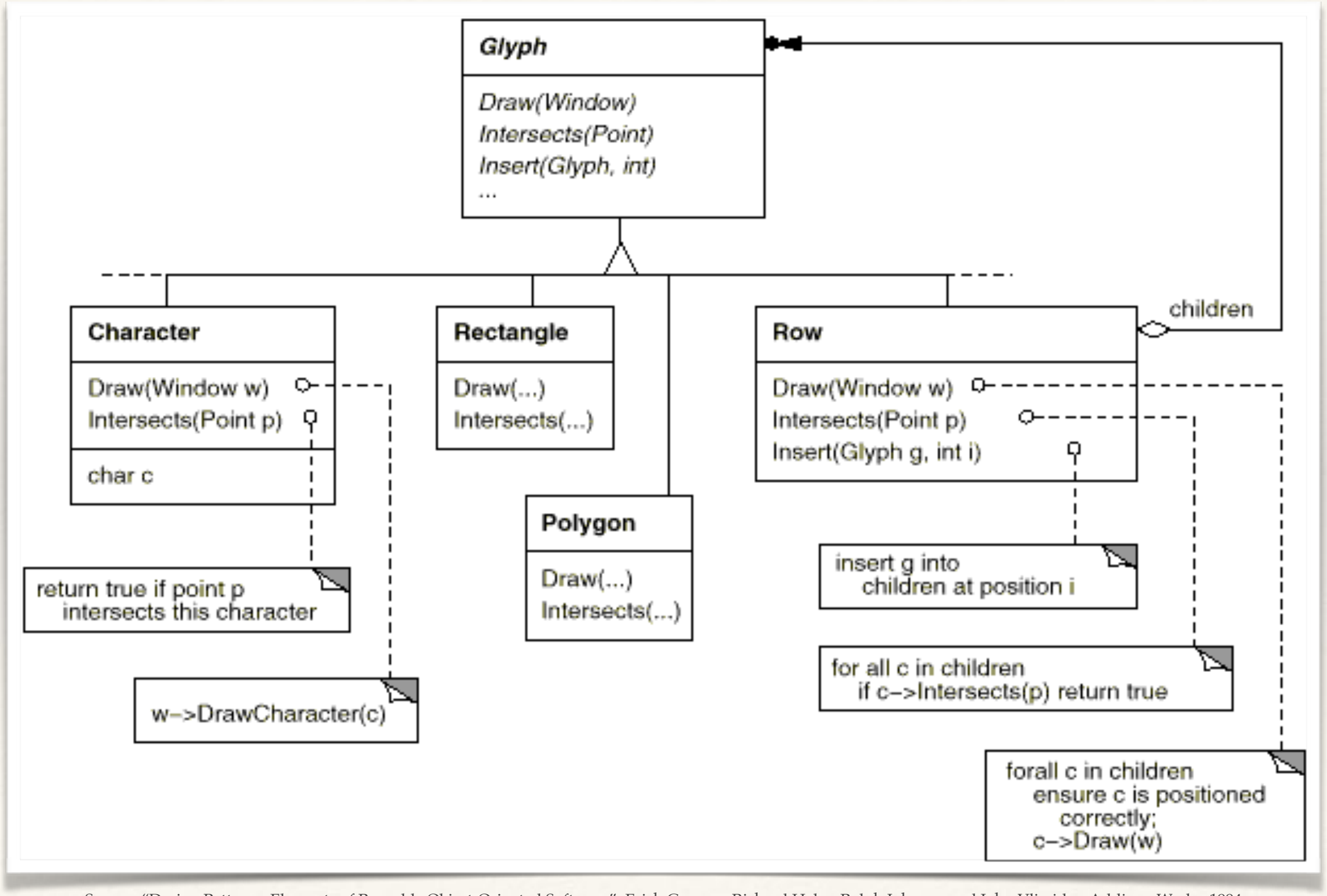
More “recursive composition”





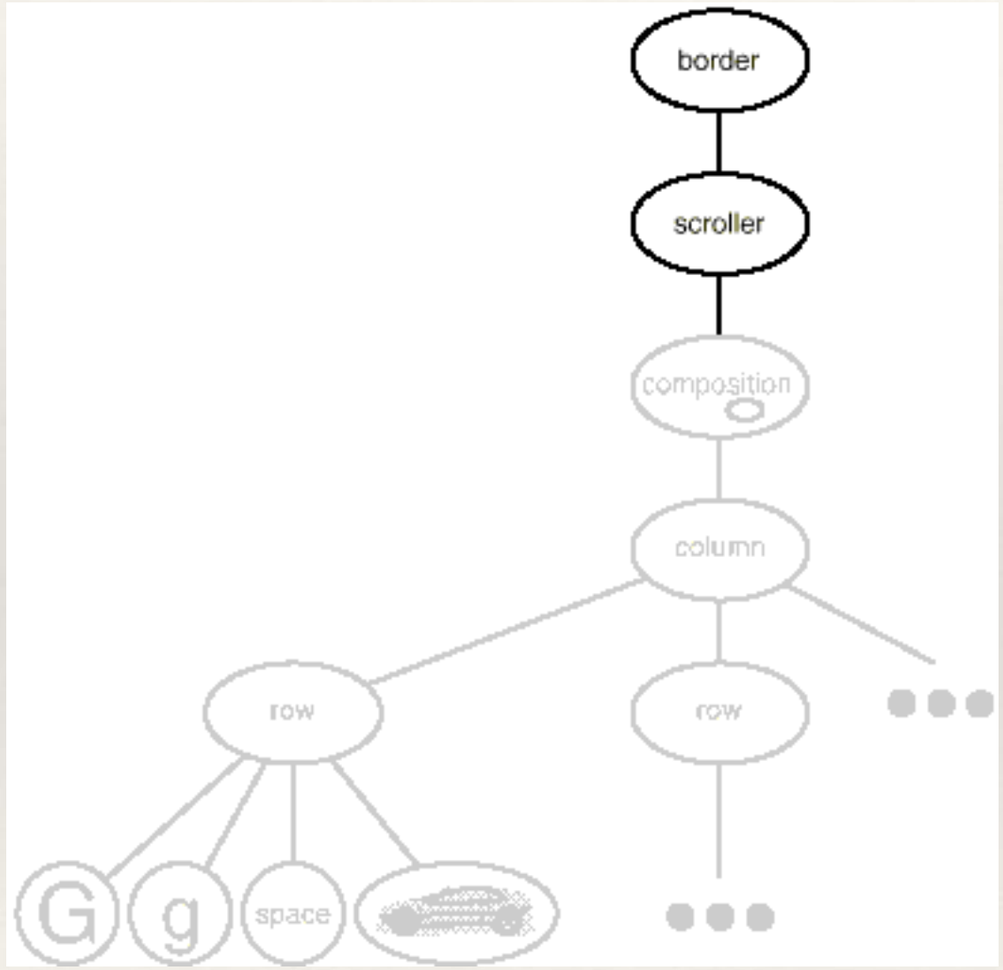
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

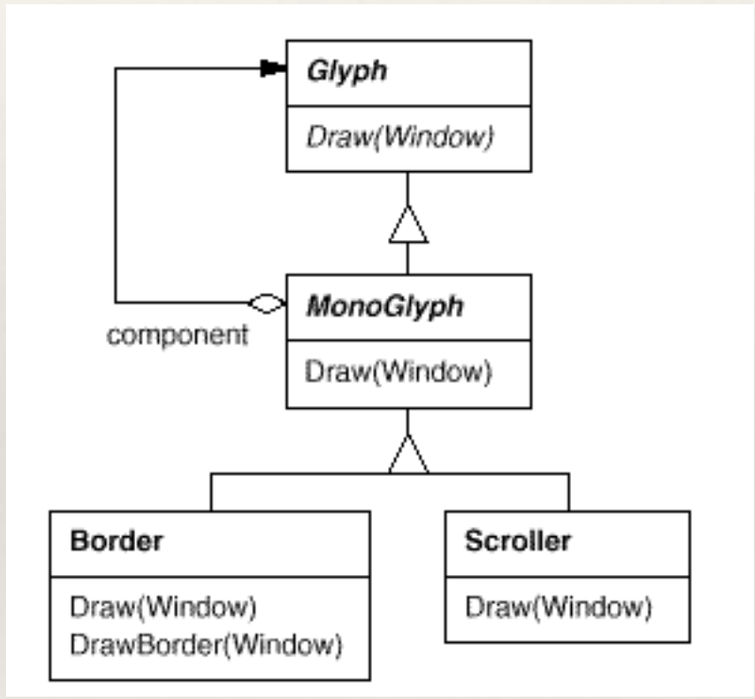
“Recursive composition” - class design



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

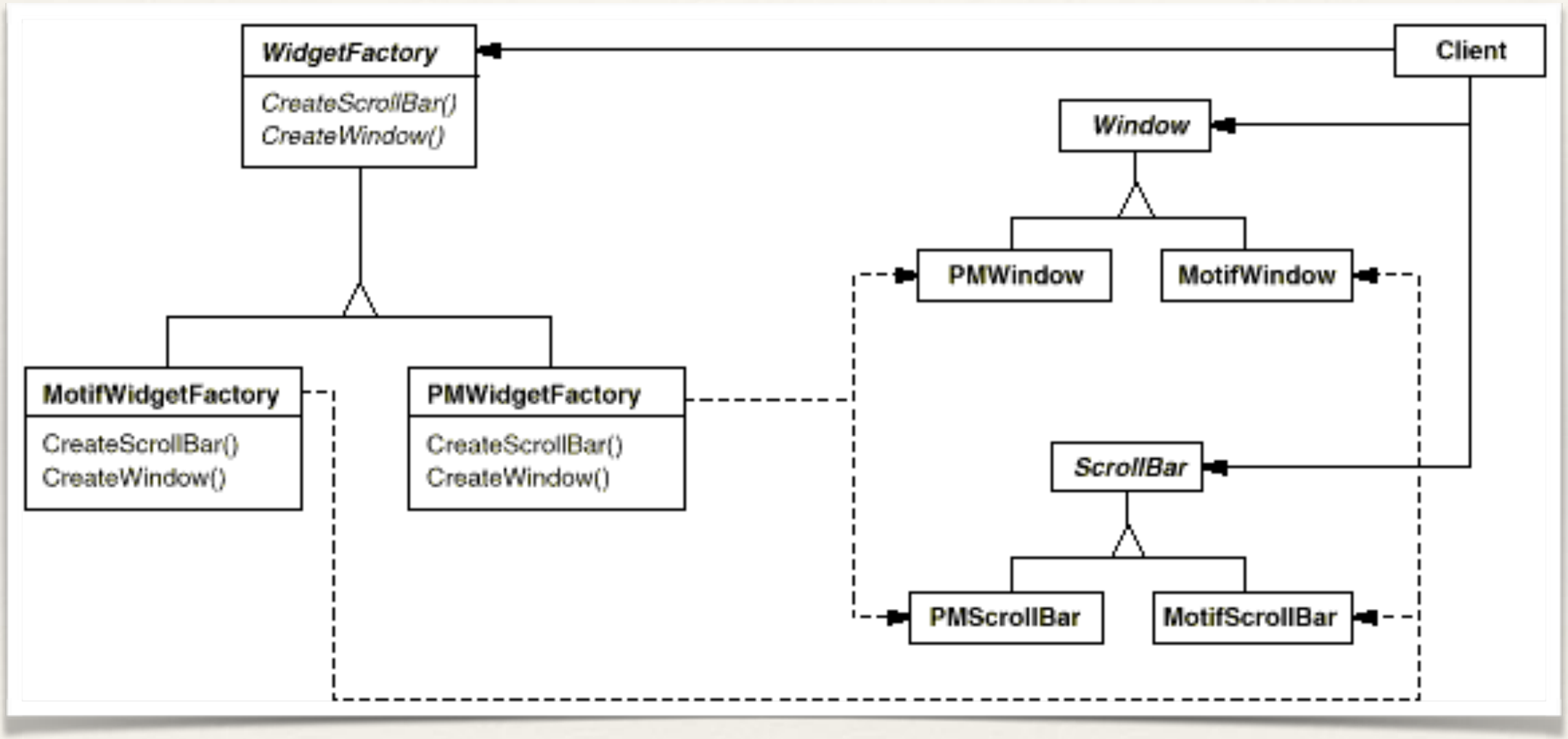
Supporting bordering, scrolling, …





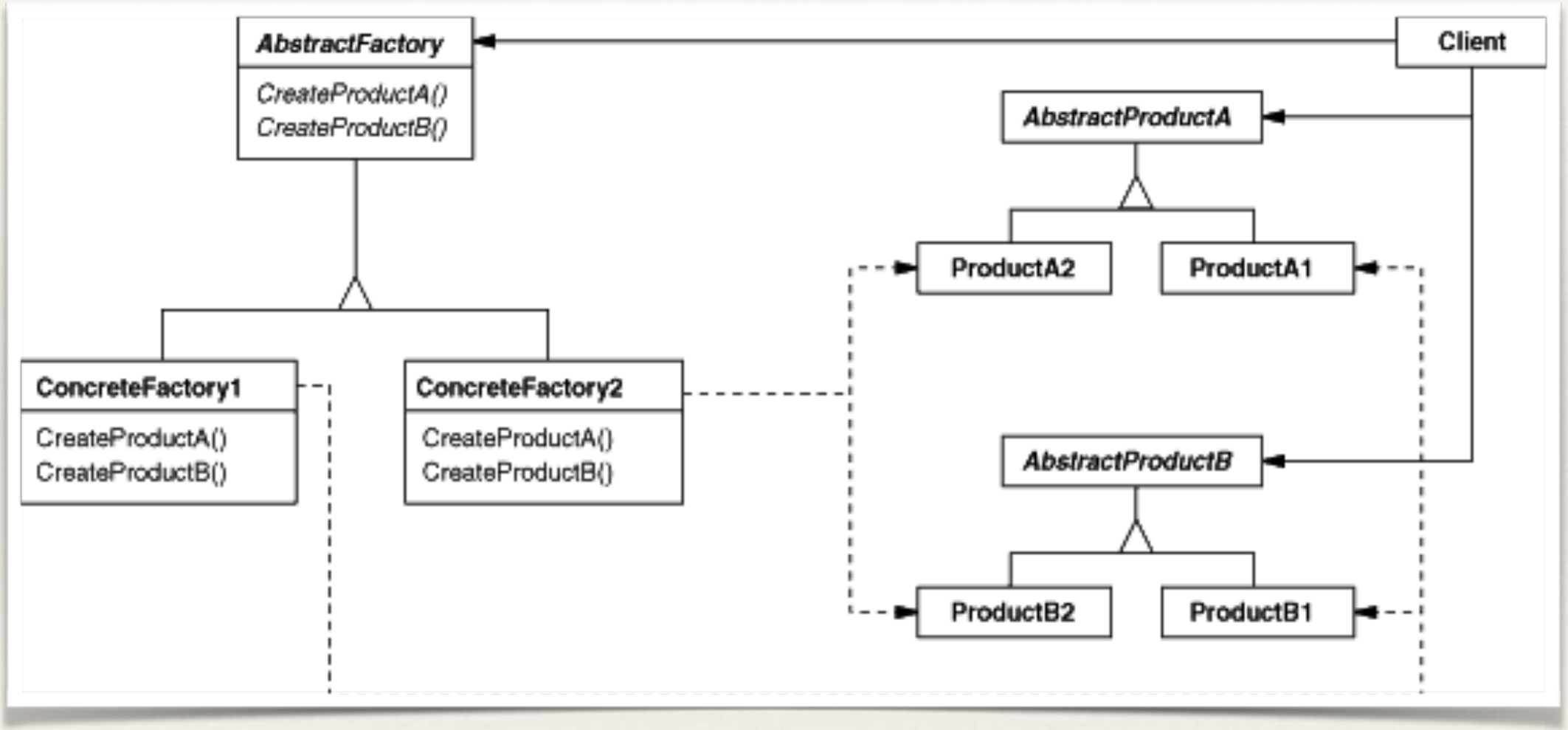
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Supporting multiple look-and-feel



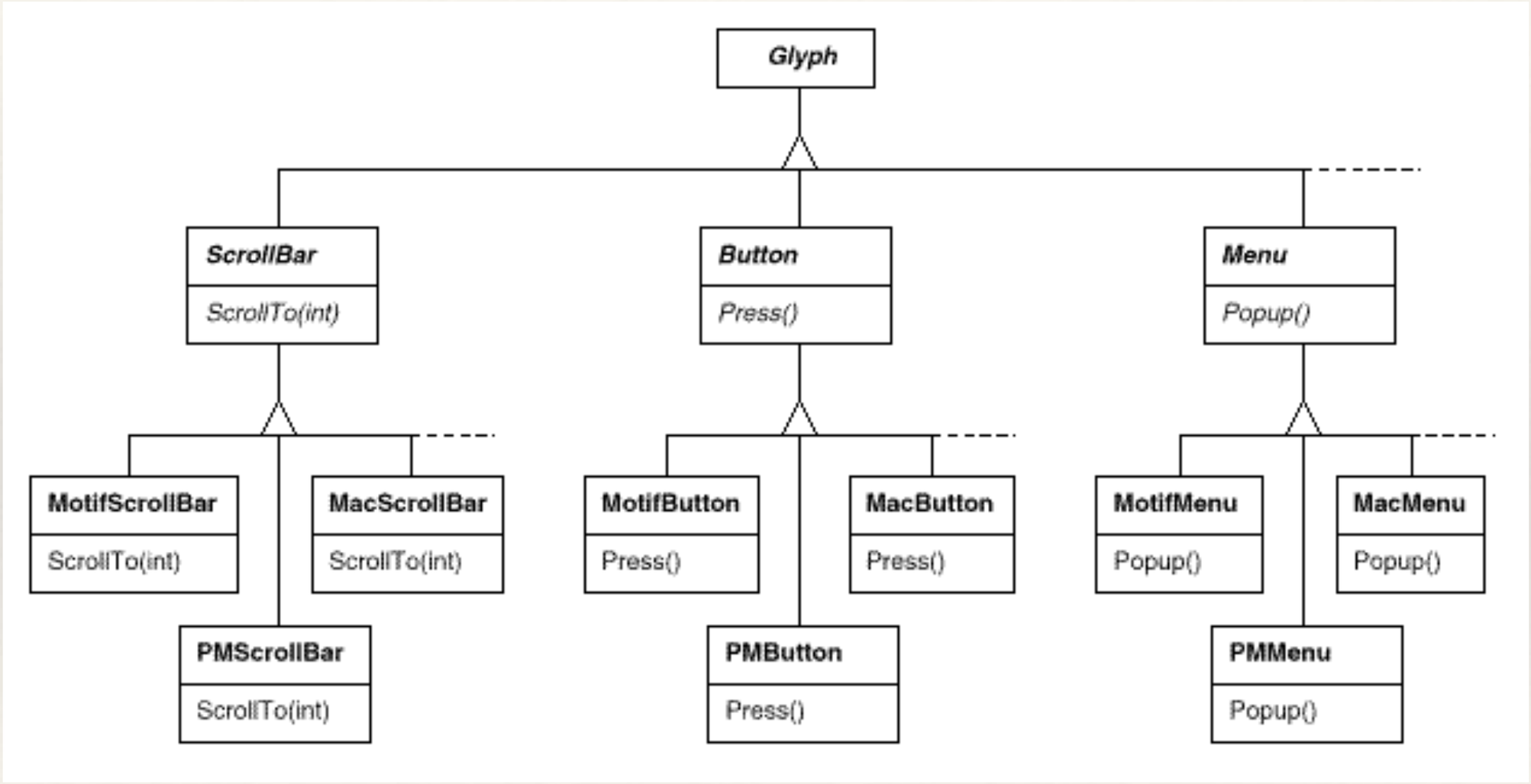
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Abstract factory: Structure



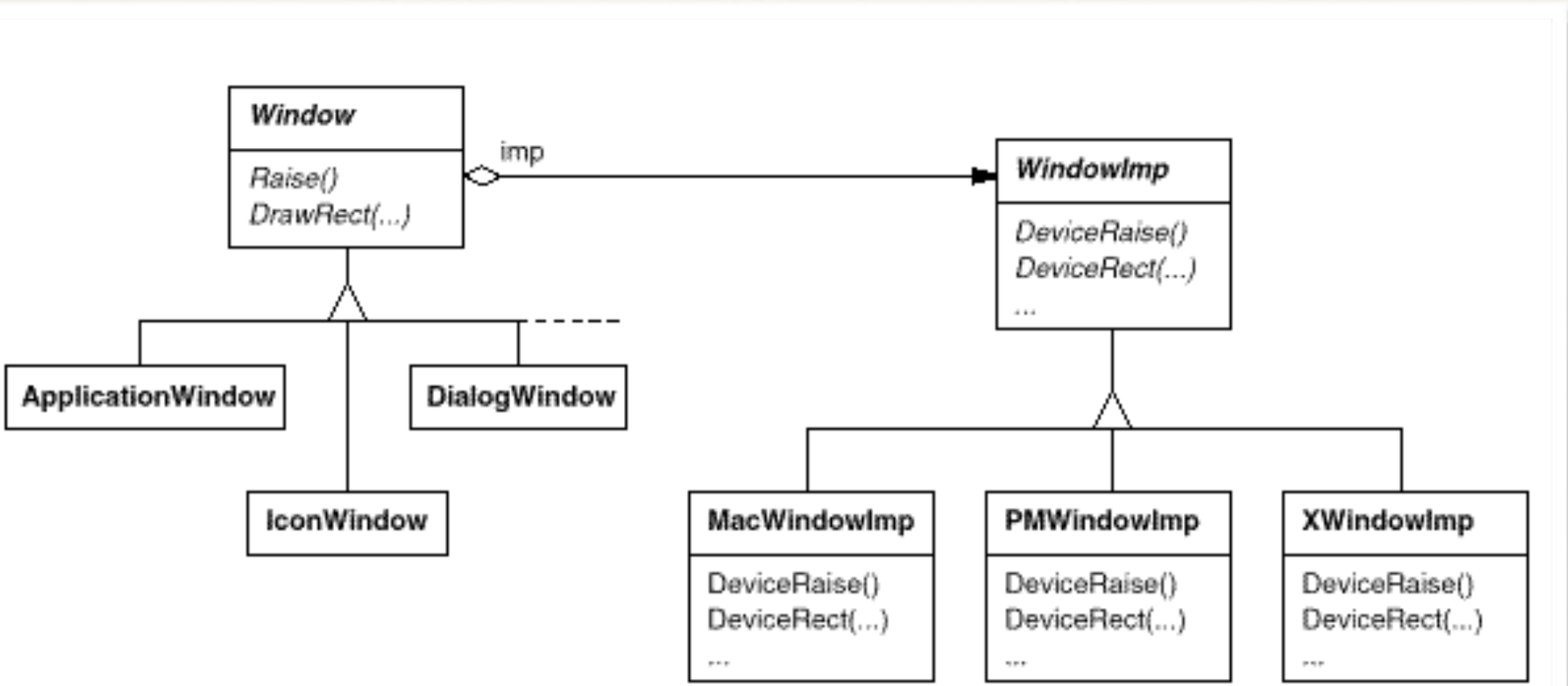
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Supporting multiple look-and-feel



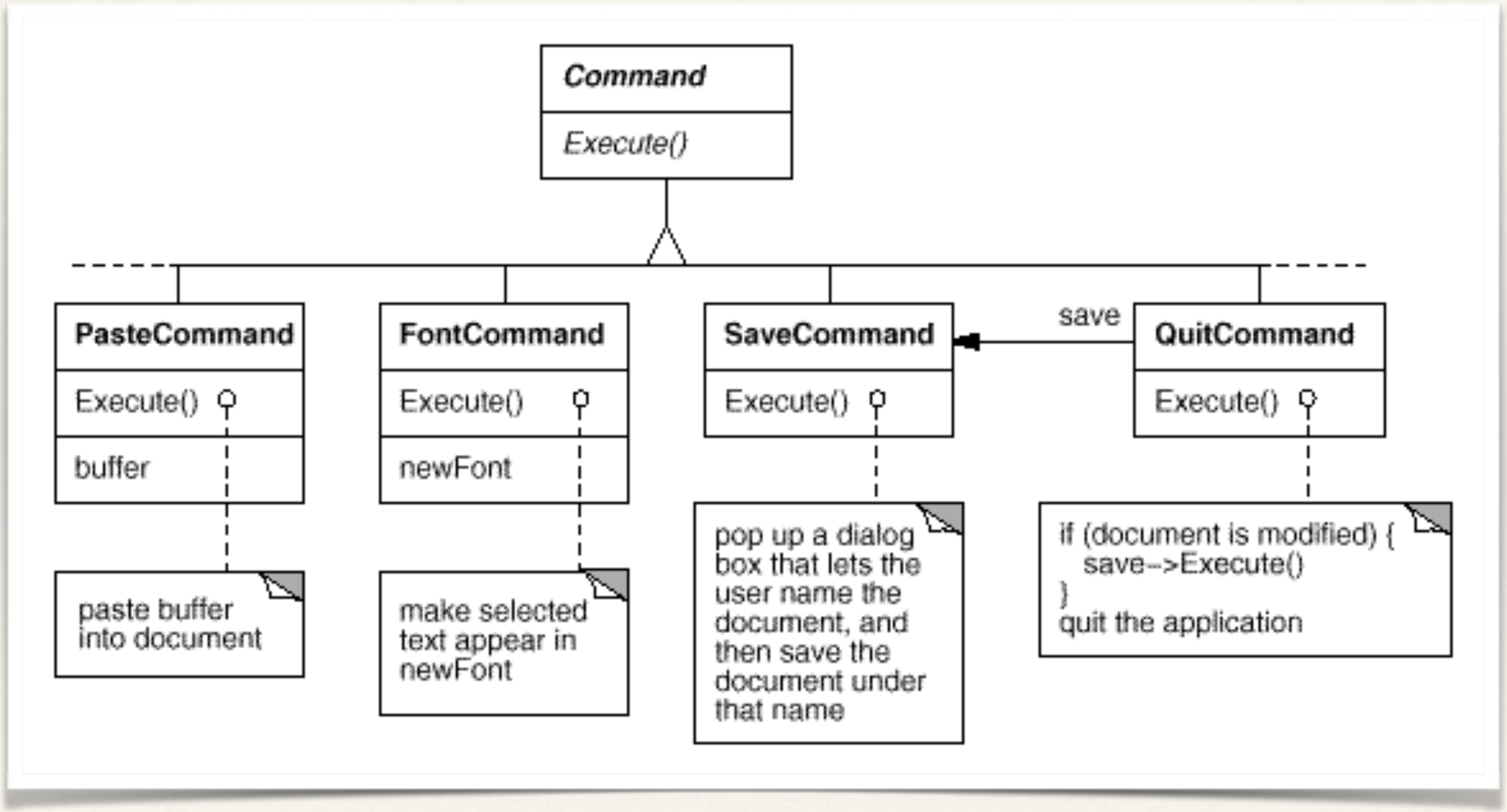
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Separating implementation dependencies



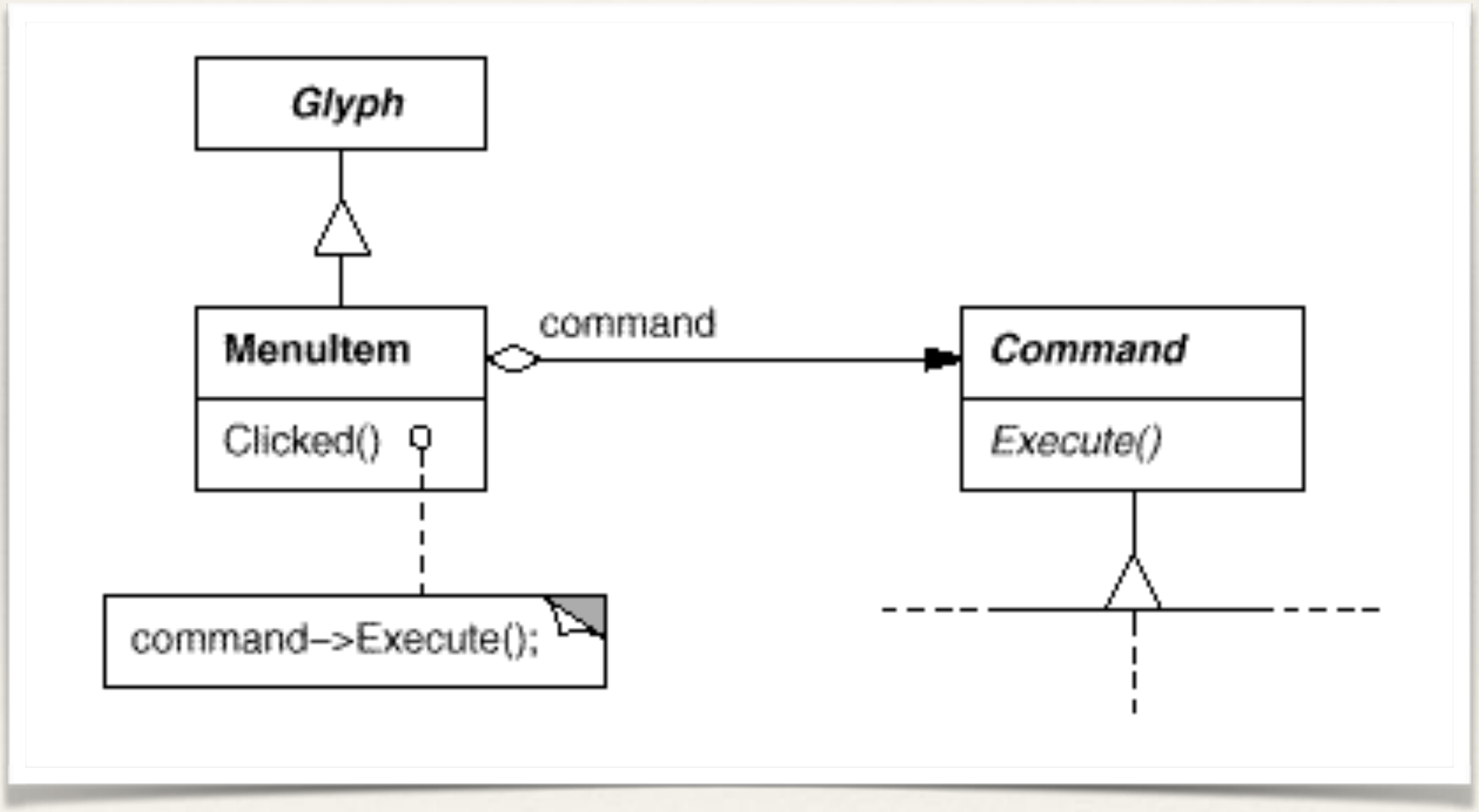
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Supporting user operations



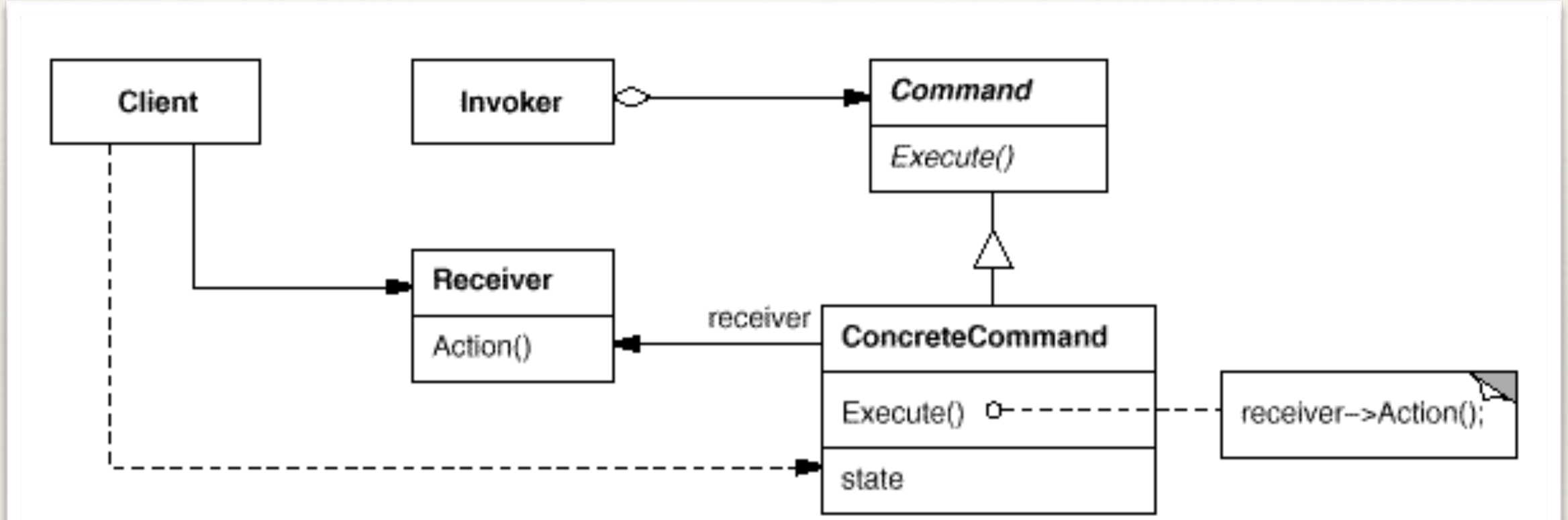
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Supporting user operations



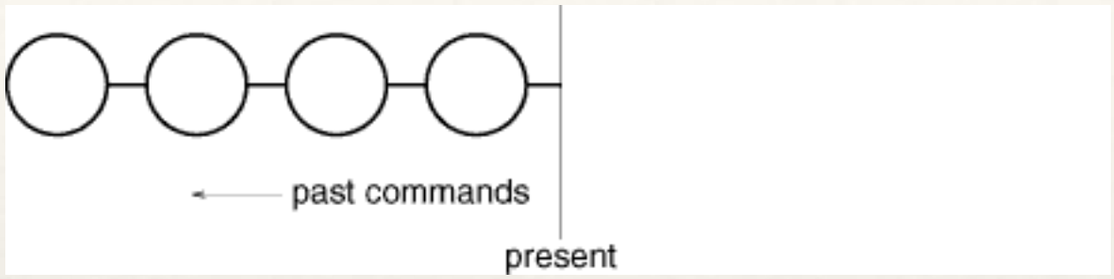
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

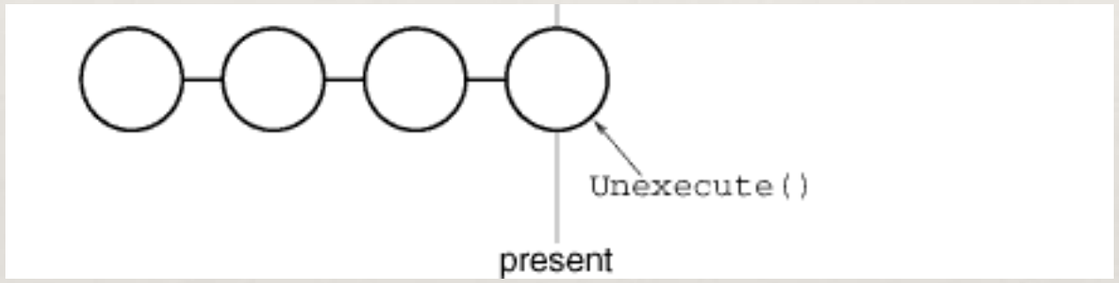
Command pattern: Structure

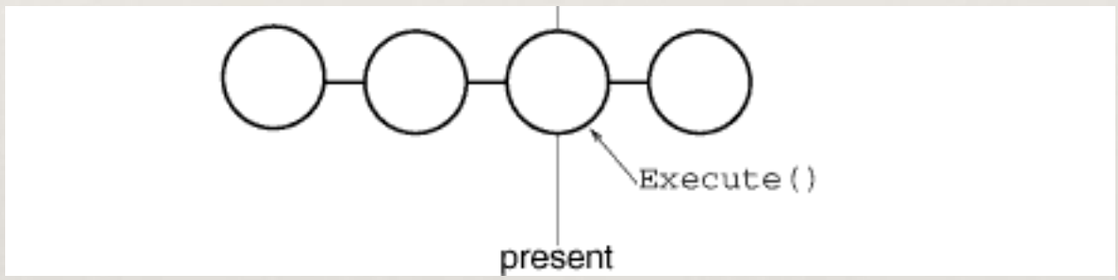


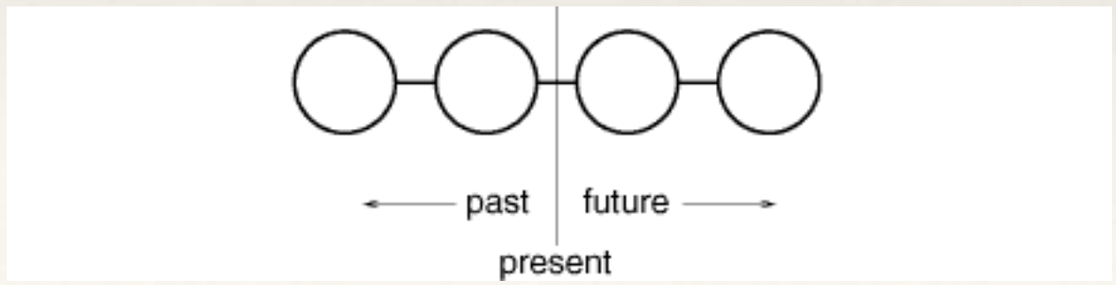
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

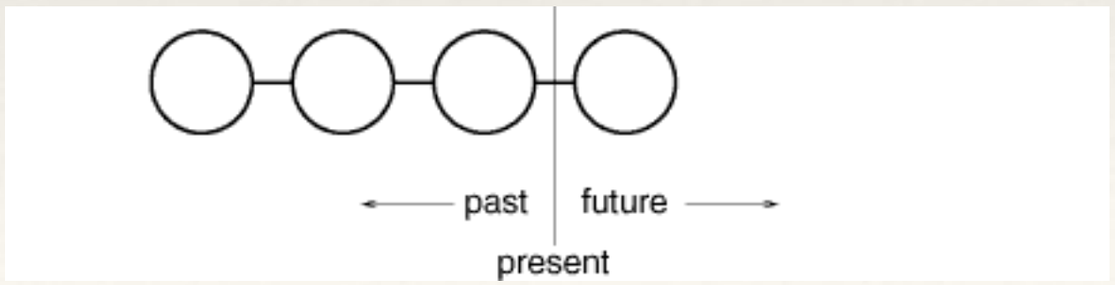
Supporting undo/redo: Command history





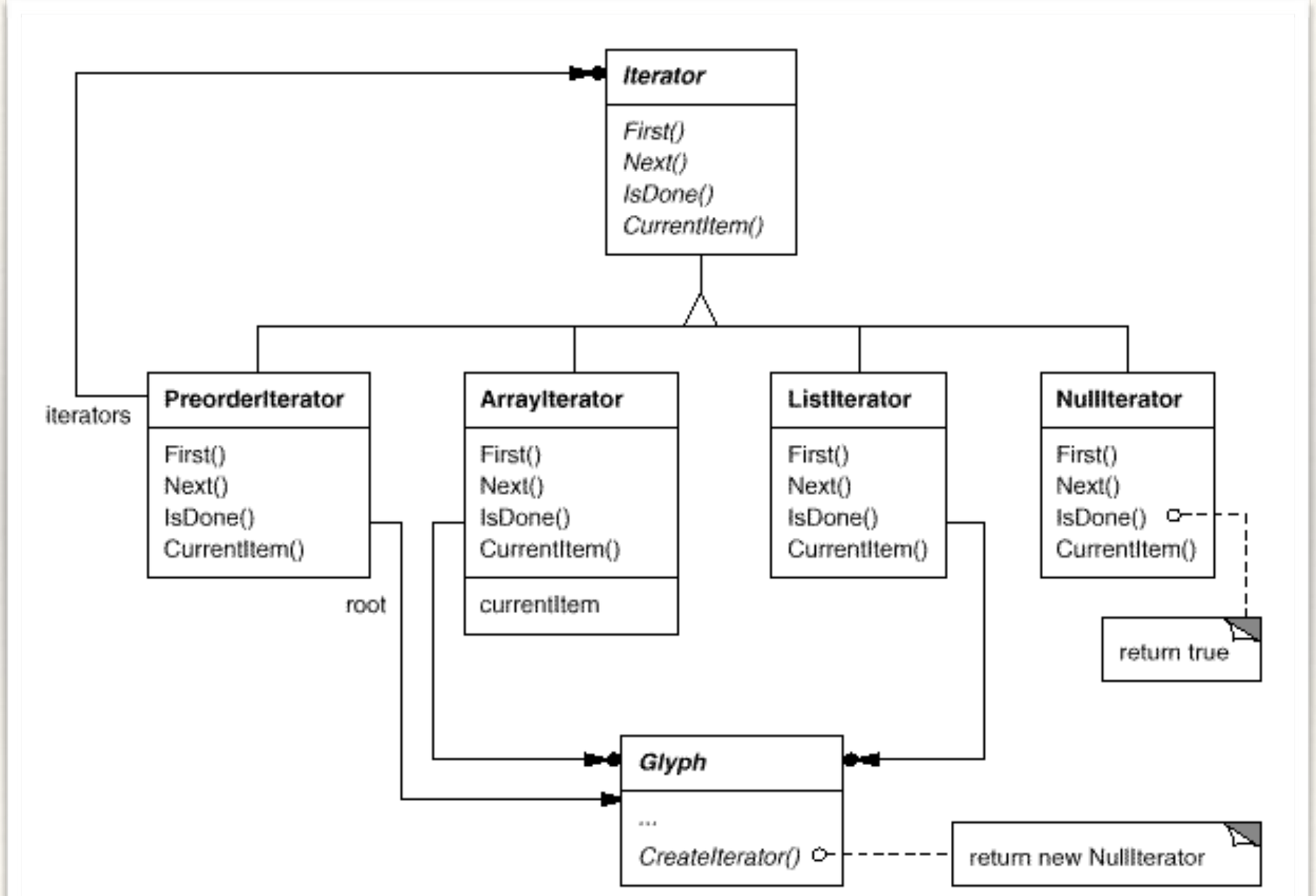






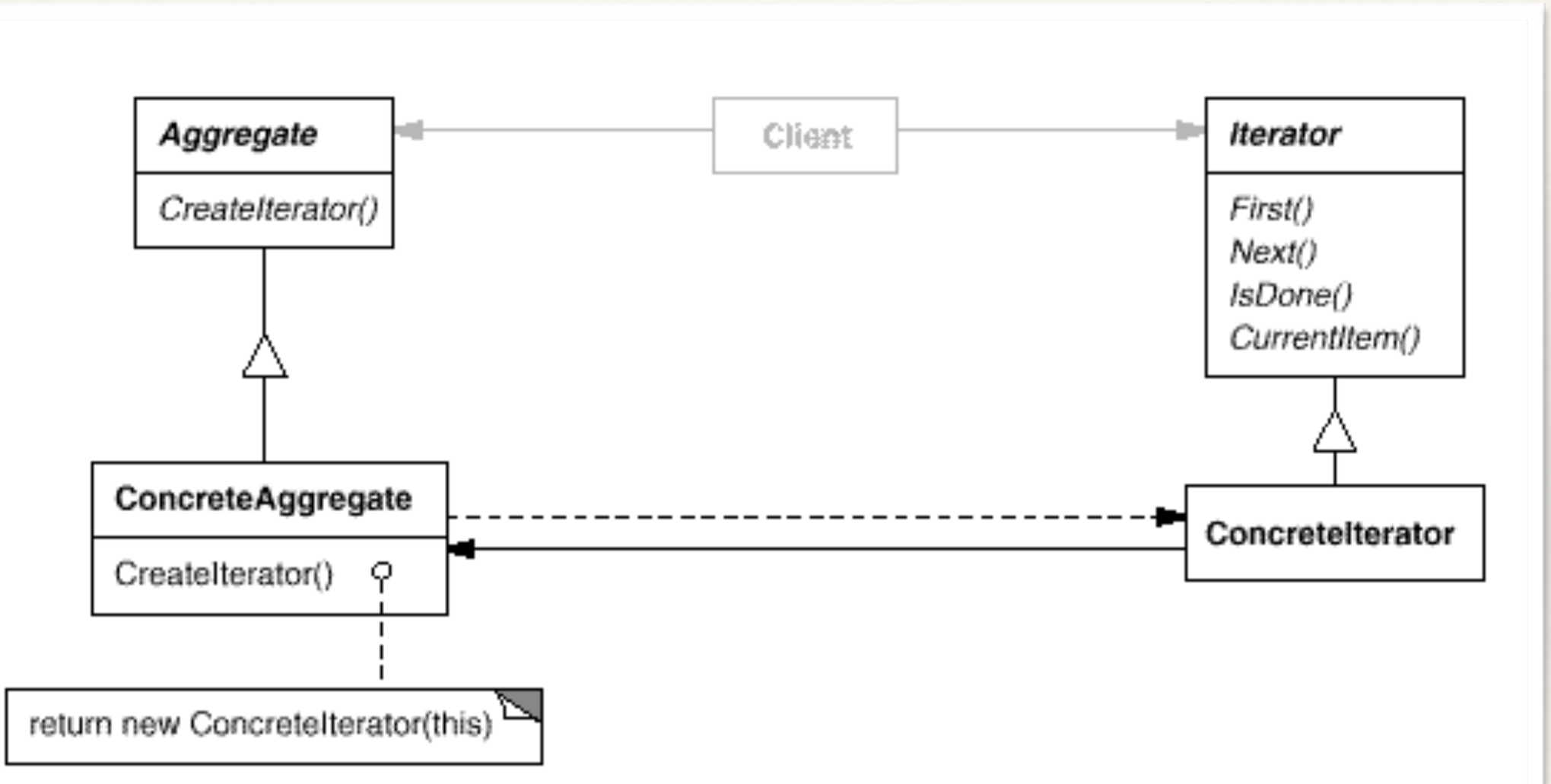
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Traversing document (e.g., spell check)



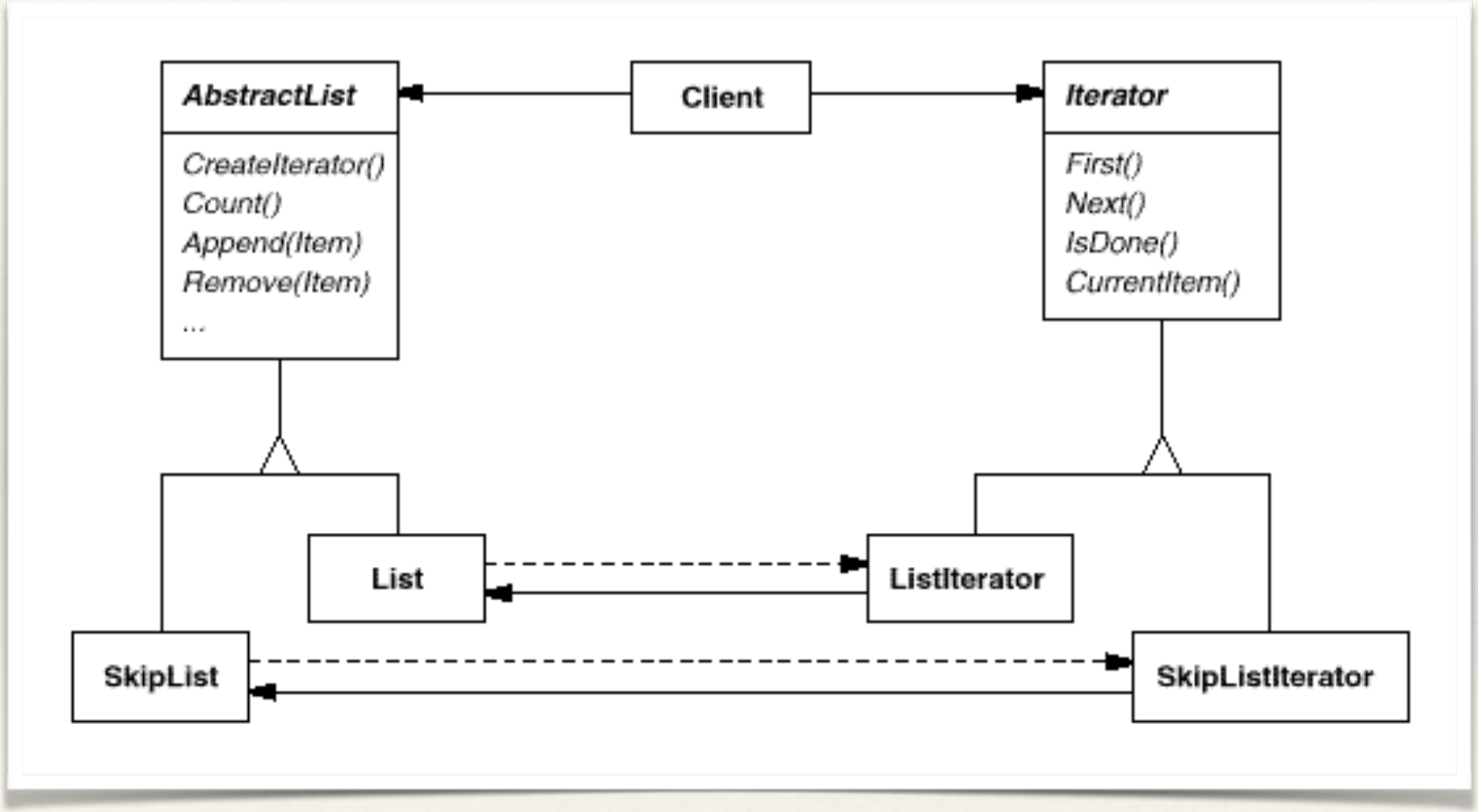
Source: “Design Patterns: Elements of Reusable Object-Oriented Software”, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison-Wesley,1994

Iterator pattern: Structure



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

Iterator pattern: Another example



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

Patterns discussed in this case-study

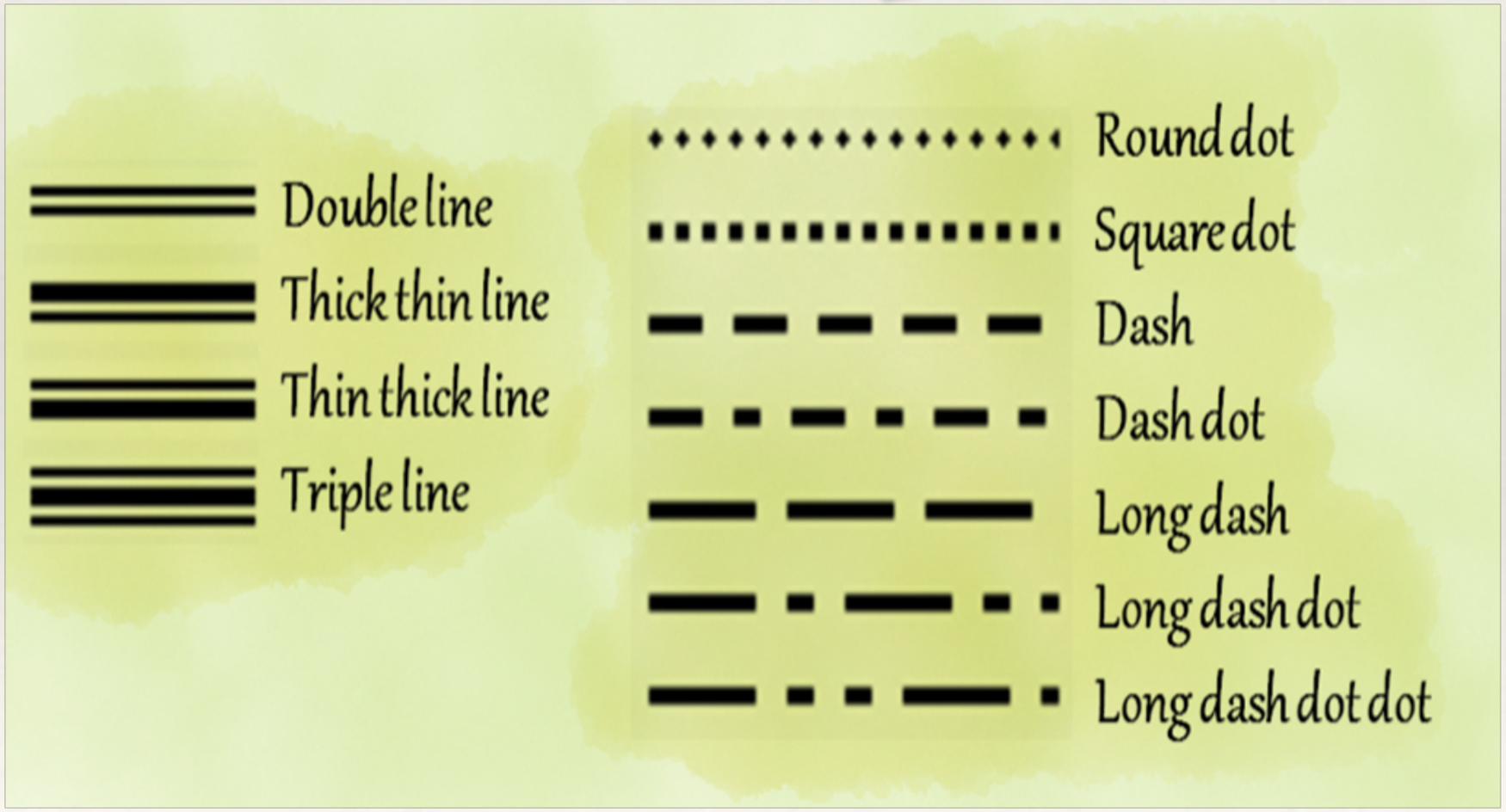
- ✓ Composite pattern
- ✓ Decorator pattern
- ✓ Abstract factory pattern
- ✓ Bridge pattern
- ✓ Command pattern
- ✓ Iterator pattern

Agenda

- Introduction
- Design patterns though exercises
- Patterns through a case-study
- **Wrap-up & key-takeaways**

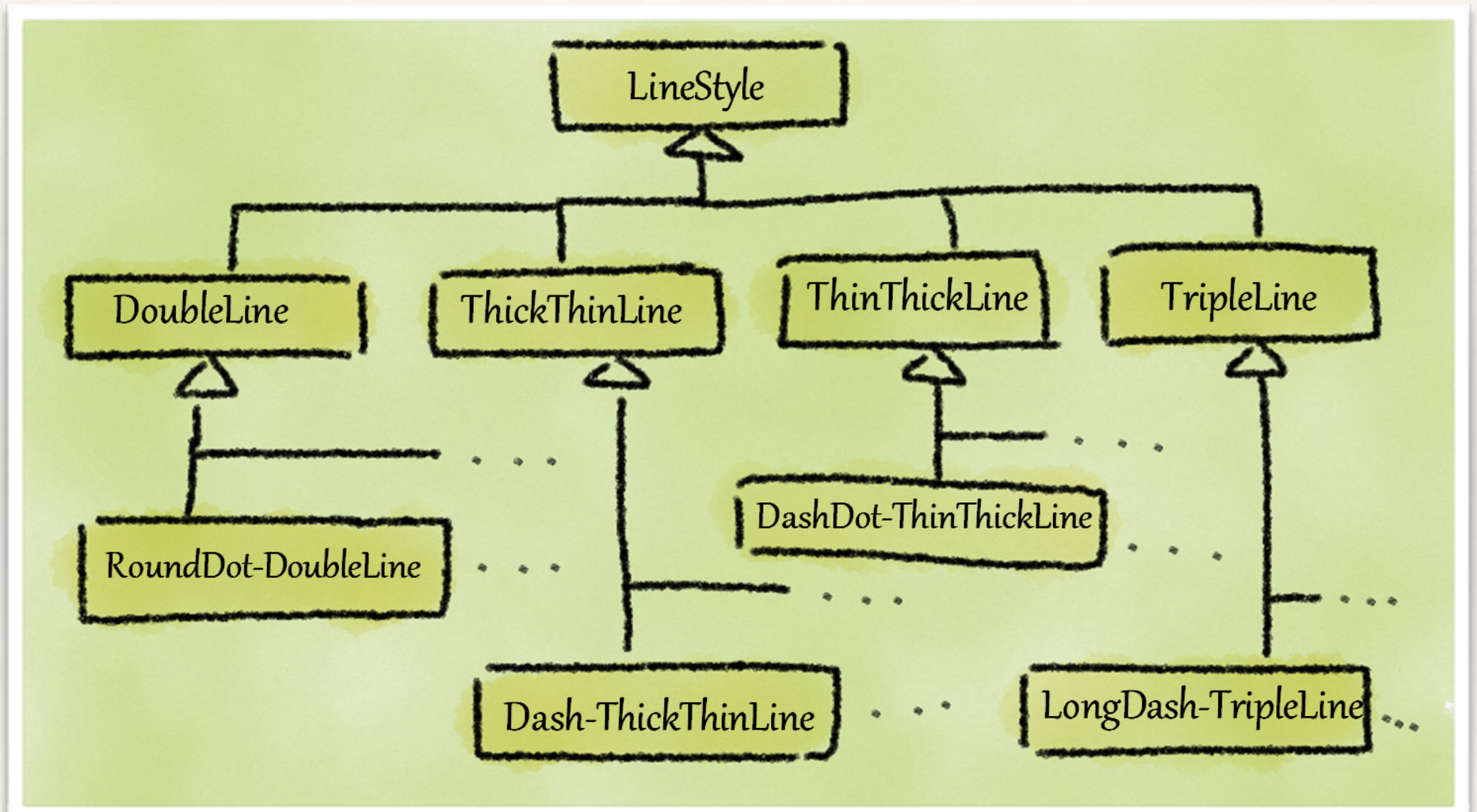
Exercise: Create a suitable design

Note: Responsibility of rendering the line in chosen style is with underlying OS / platform



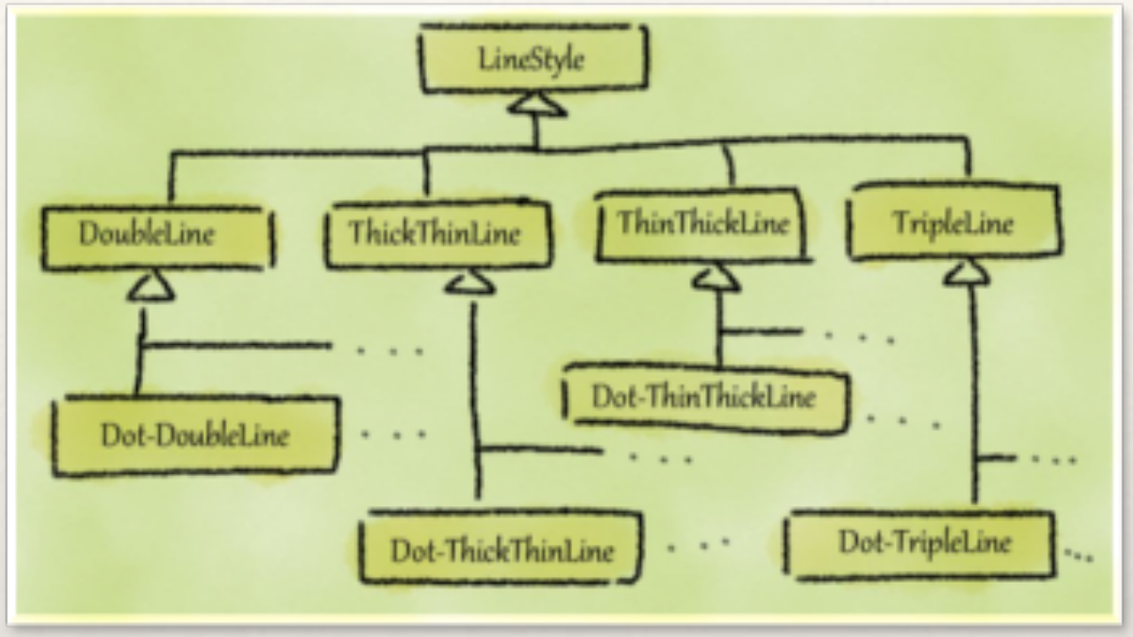
Source: "Refactoring for Software Design Smells: Managing Technical Debt", Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Morgan Kaufmann/Elsevier, 2014

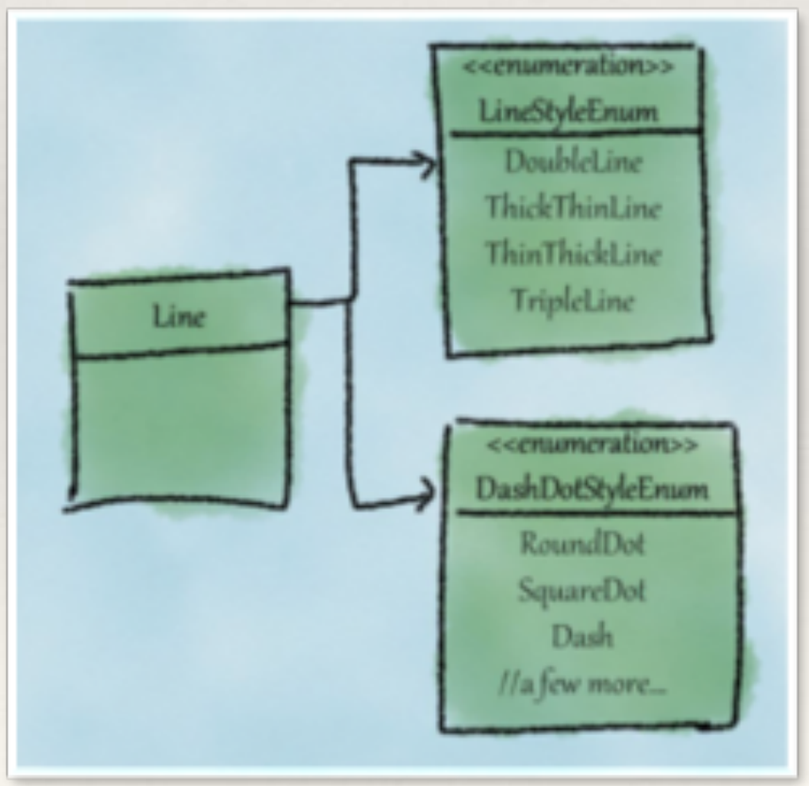
How about this solution?



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

Suggested refactoring for this smell



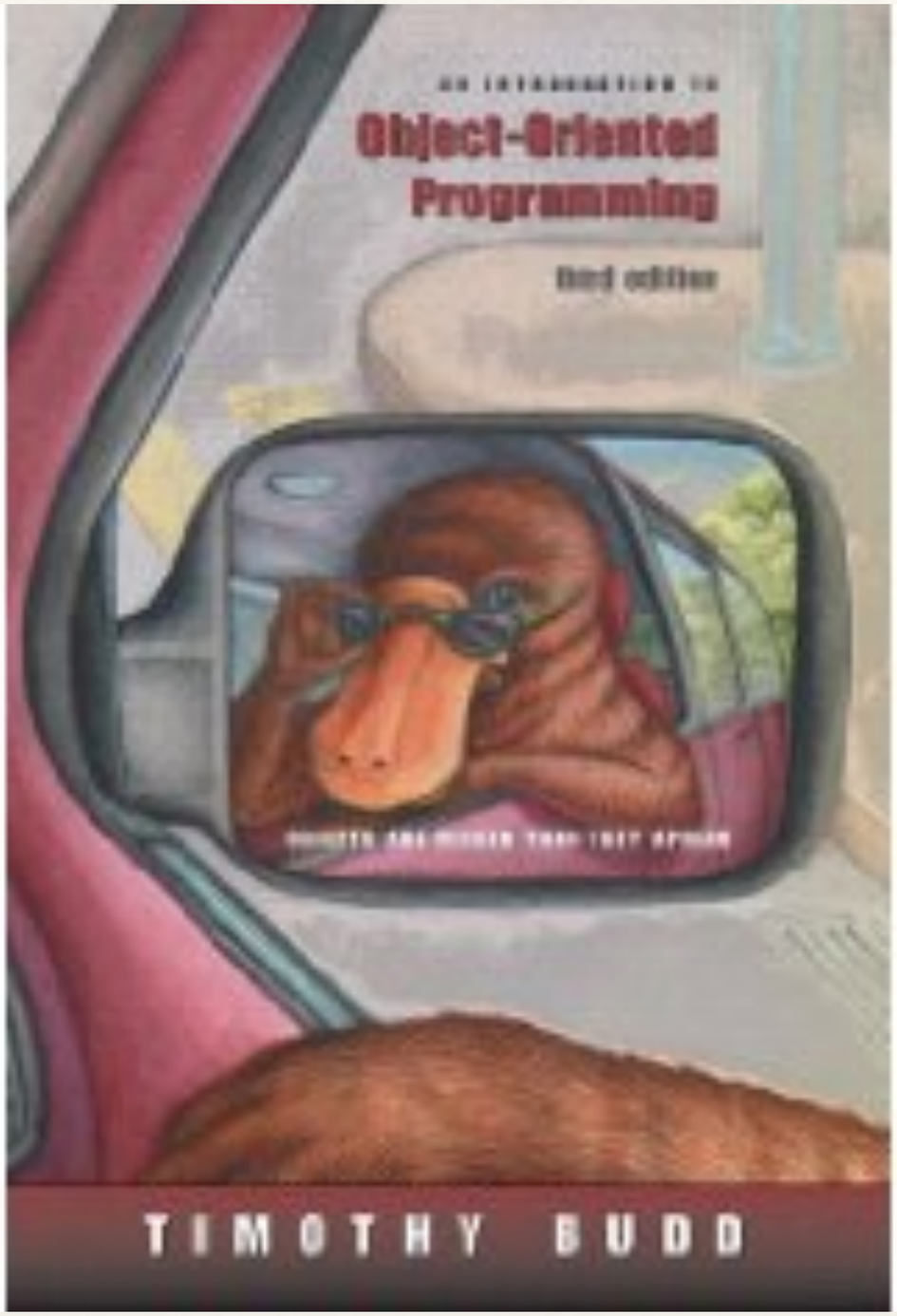


Source: "Refactoring for Software Design Smells: Managing Technical Debt", Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma, Morgan Kaufmann/Elsevier, 2014

Beware of “patterns mania”!

What are your takeaways?

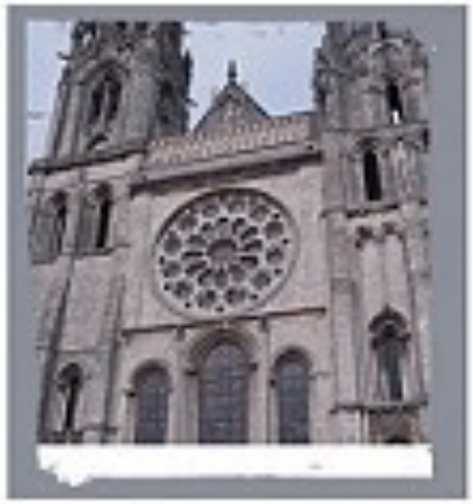
Books to read


Object-Oriented
Programming
TIMOTHY BUDD

WILEY SERIES IN
SOFTWARE DESIGN PATTERNS

PATTERN-ORIENTED SOFTWARE ARCHITECTURE

A System of Patterns



Volume 1

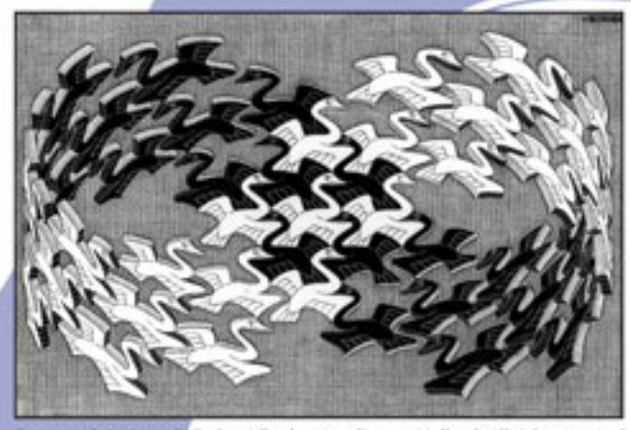
Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

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.

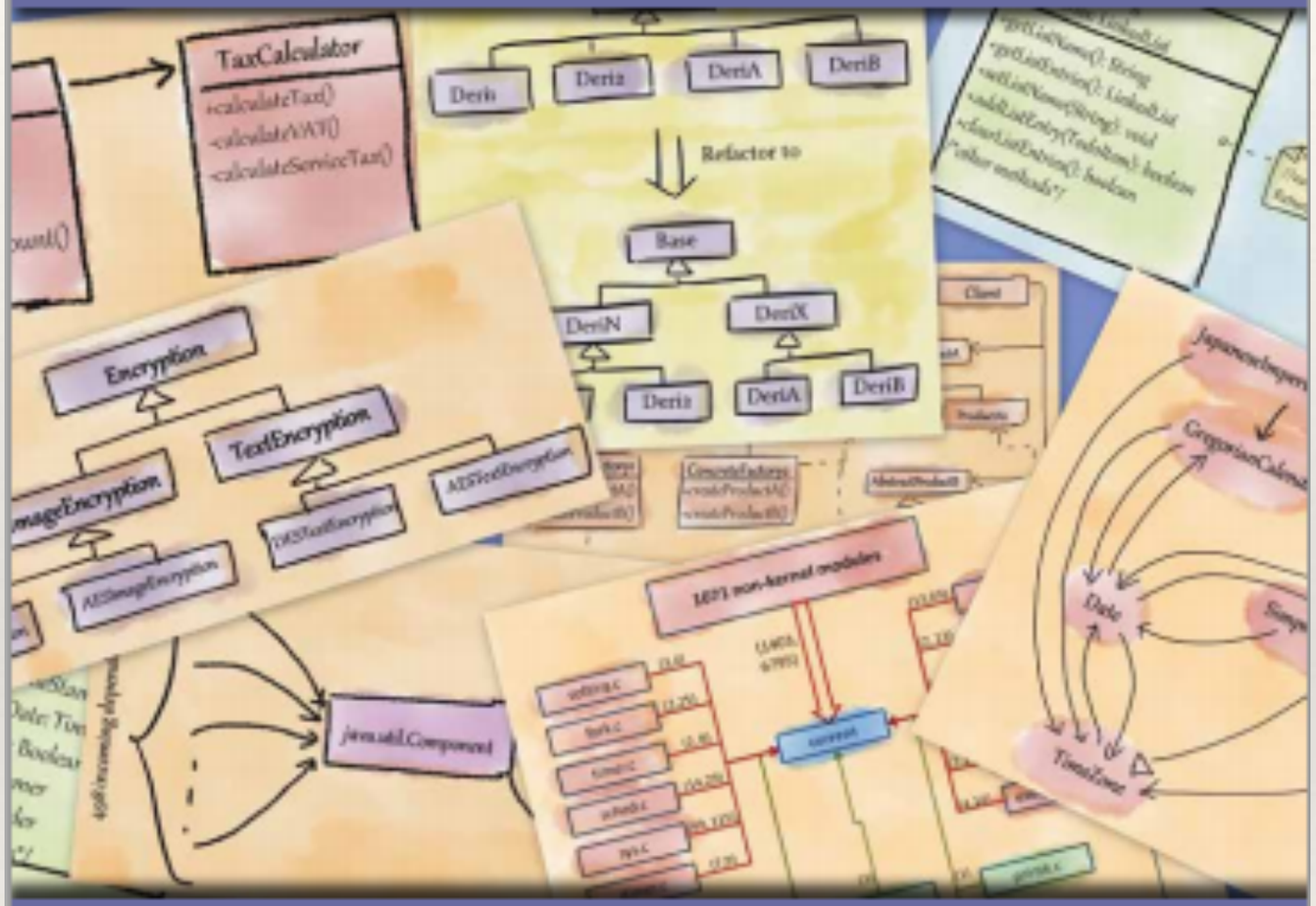
ADDISON-WESLEY OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

SERIES EDITORS

Refactoring for Software Design Smells

Managing Technical Debt



Girish Suryanarayana,
Ganesh Samarthyam, Tushar Sharma

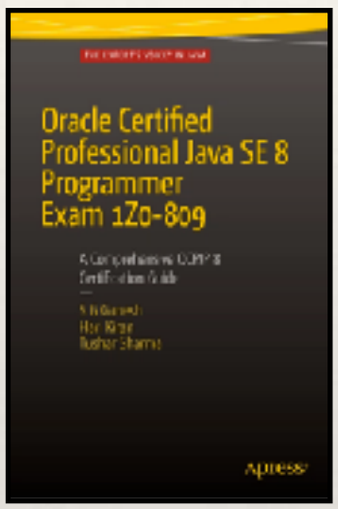
MK
MORGAN KAUFMANN

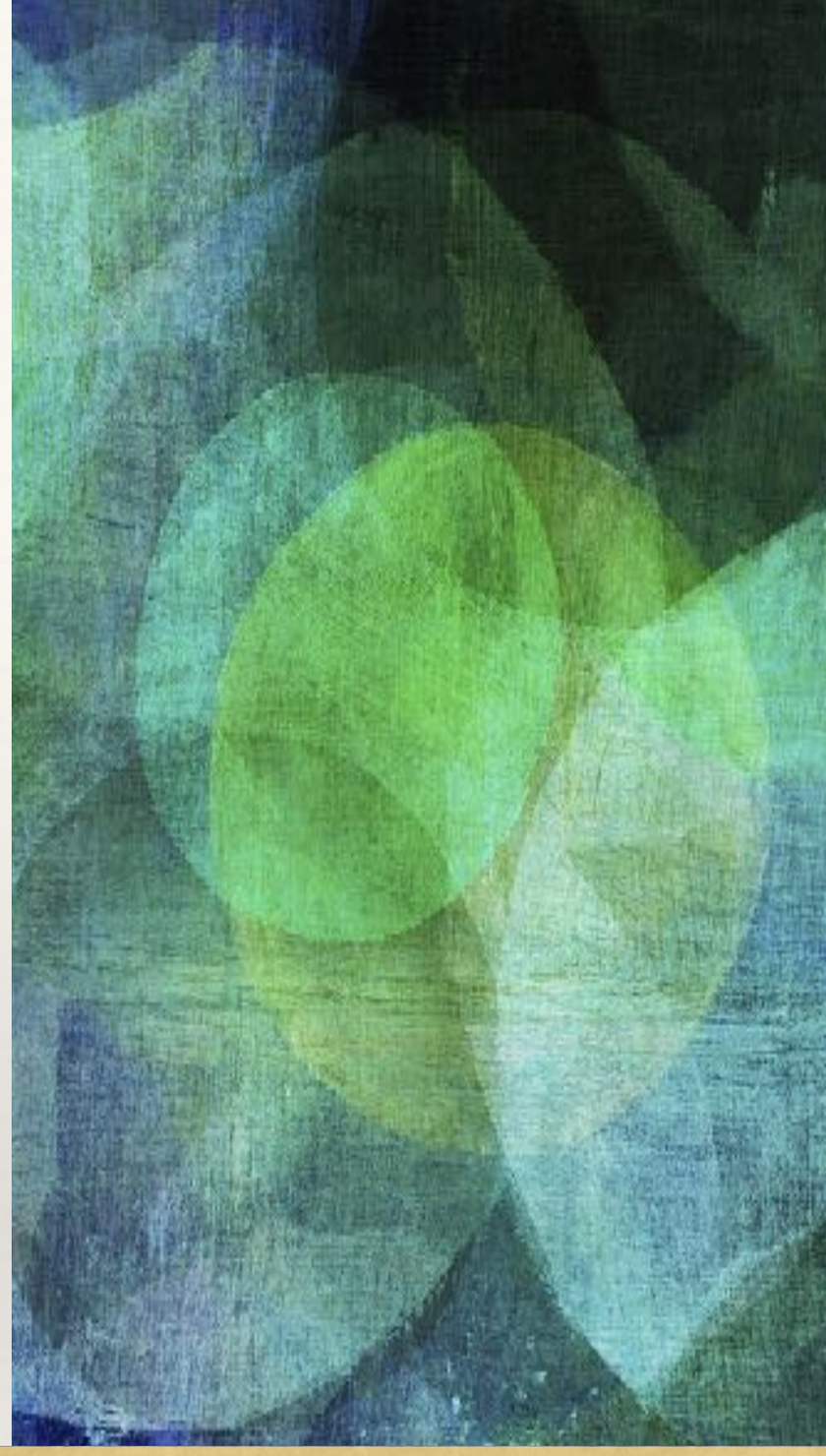
"Applying design principles is the key to creating high-quality software!"

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

Image/video credits

- http://en.wikipedia.org/wiki/Fear_of_missing_out
- http://lesliejanemoran.blogspot.in/2010_05_01_archive.html
- http://javra.eu/wp-content/uploads/2013/07/angry_laptop2.jpg
- https://www.youtube.com/watch?v=5R8XHrfJkeg
- http://womenworld.org/image/052013/31/113745161.jpg
- http://www.fantom-xp.com/wallpapers/33/I'm_not_sure.jpg
- https://www.flickr.com/photos/31457017@N00/453784086
- https://www.gradtouch.com/uploads/images/question3.jpg
- http://gurujohn.files.wordpress.com/2008/06/bookcover0001.jpg
- http://upload.wikimedia.org/wikipedia/commons/d/d5/Martin_Fowler_-_Swipe_Conference_2012.jpg
- http://www.codeproject.com/KB/architecture/csdespat_2/dpcs_br.gif
- http://upload.wikimedia.org/wikipedia/commons/thumb/2/28/Bertrand_Meyer_IMG_2481.jpg/440px-Bertrand_Meyer_IMG_2481.jpg
- http://takeji-soft.up.n.seesaa.net/takeji-soft/image/GOF-OOPLSA-94-Color-75.jpg?d=a0
- https://developer.apple.com/library/ios/documentation/cocoa/Conceptual/OOP_ObjC/Art/watchcalls_35.gif
- http://www.pluspack.com/files/billeder/Newsletter/25/takeaway_bag.png
- http://cdn1.tnwcdn.com/wp-content/blogs.dir/1/files/2013/03/design.jpg





ganesh@codeops.tech

www.codeops.tech

+91 98801 64463

@GSamarthyam

slideshare.net/sgganesh

bit.ly/sgganesh

CodeOps