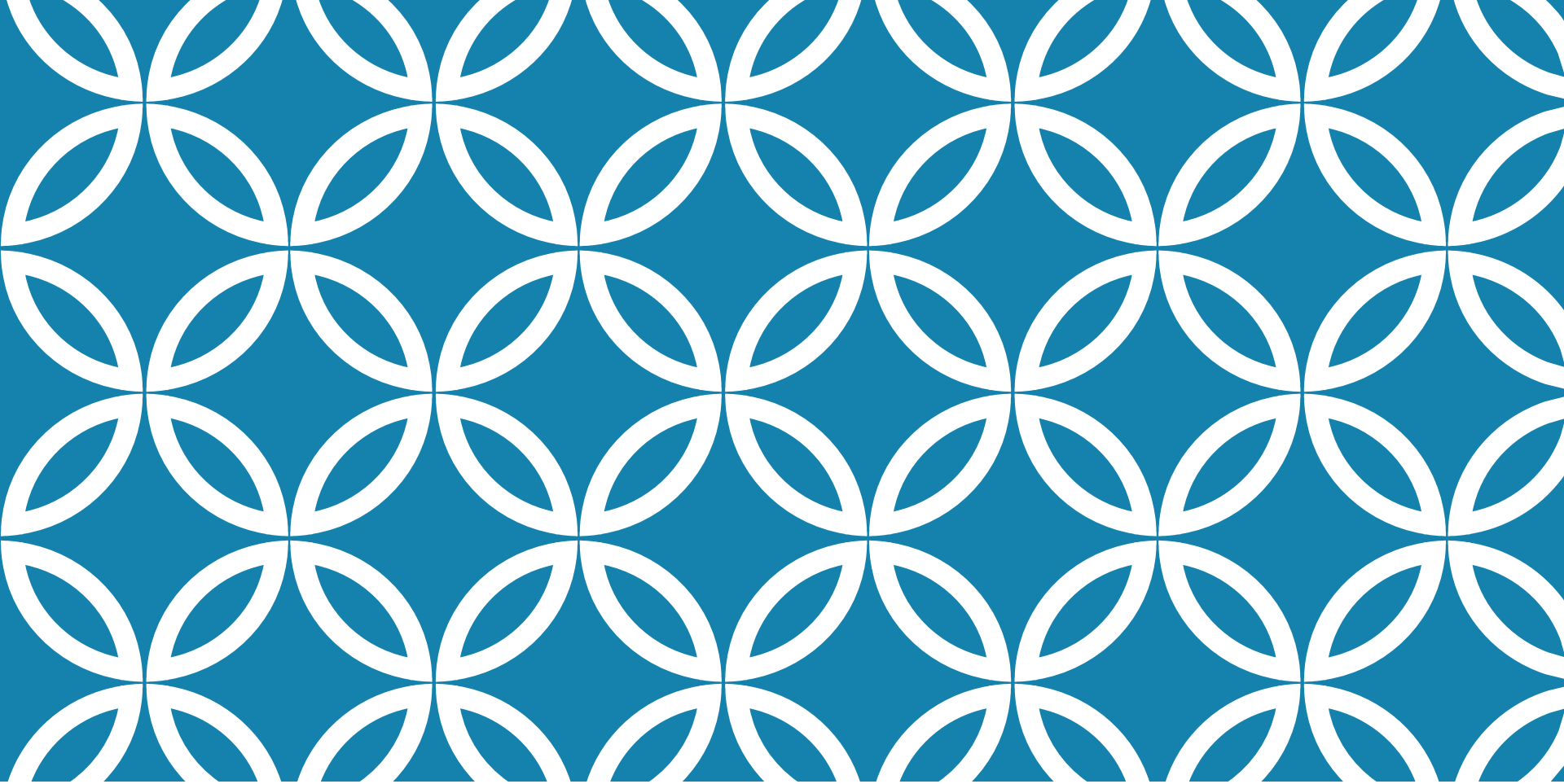


STRUCTURAL DESIGN PATTERNS

Lecture 10

CONTENT

Design Patterns

- Structural Patterns
  - Adapter
  - Composite
  - Decorator
  - Proxy
  - Bridge

REFERENCES

- Erich Gamma, et.al, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994, ISBN 0-201-63361-2.

DESIGN PATTERN SPACE

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Singleton Prototype	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Command Chain of Responsibility Strategy Visitor Iterator Mediator Memento Observer State

STRUCTURAL PATTERNS

Structural patterns are concerned with how classes and objects are composed to form larger structures.

Structural **class** patterns use **inheritance** to compose interfaces or implementations.

Rather than composing interfaces or implementations, structural **object** patterns describe ways to **compose** objects to realize new functionality.

PAYING BY PAYPAL

```
class PayPal {
     public PayPal() {
         // constructor code here //
     }
     public void sendPayment(double amount) {
         // Paying via Paypal //
     }
}
// Client code
PayPal pay = new PayPal();
pay.sendPayment(2629);
```

CHANGES

```
sendPayment(double amount)->
payAmount(double amount)
```

COMMIT TO AN INTERFACE, NOT AN IMPLEMENTATION!

```
// Simple Interface for each Adapter we create
interface PaymentAdapter {
    public pay(double amount);
}
class PaypalAdapter implements PaymentAdapter {
     private PayPal paypal;
     public PaypalAdapter(PayPal p) {
         this.paypal = p;
     }
     public void pay(double amount) {
         paypal.sendPayment(amount);
     }
}

// Client Code
PaymentAdapter pp = new PaypalAdapter(new PayPal());
pp.pay(2629);
```

CHANGING THE PAY SERVICE

```
class PayServiceAdapter implements PaymentAdapter {
    private PayService pay;
    public PayServiceAdapter(PayService p) {
        this.pay = p;
    }
    public void pay(double amount) {
        pay.payAmount(amount);
    }
}
```

//Client code

??

ADAPTER PATTERN

Intent

Convert the interface of a class into another interface clients expect.

Adapter enables classes to work together that couldn't otherwise because of incompatible interfaces.

Also known as

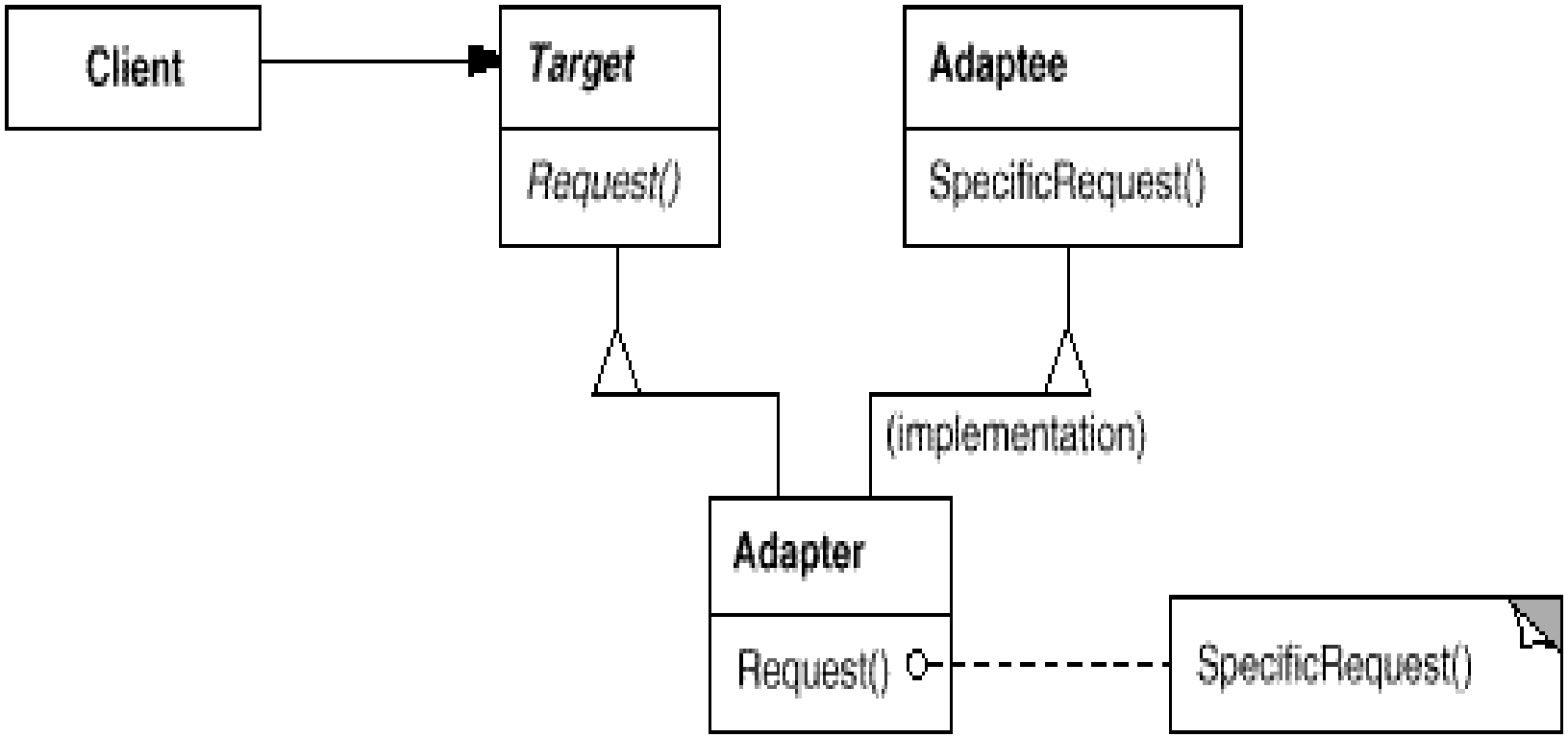
Wrapper

ADAPTER PATTERN

Applicability

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

ADAPTER PATTERN - INHERITANCE-BASED




```
class Adapter extends Adaptee implements Target
```

ADAPTER PATTERN - PARTICIPANTS

Target

- defines the domain-specific interface that Client uses.

Client

- collaborates with objects conforming to the Target interface.

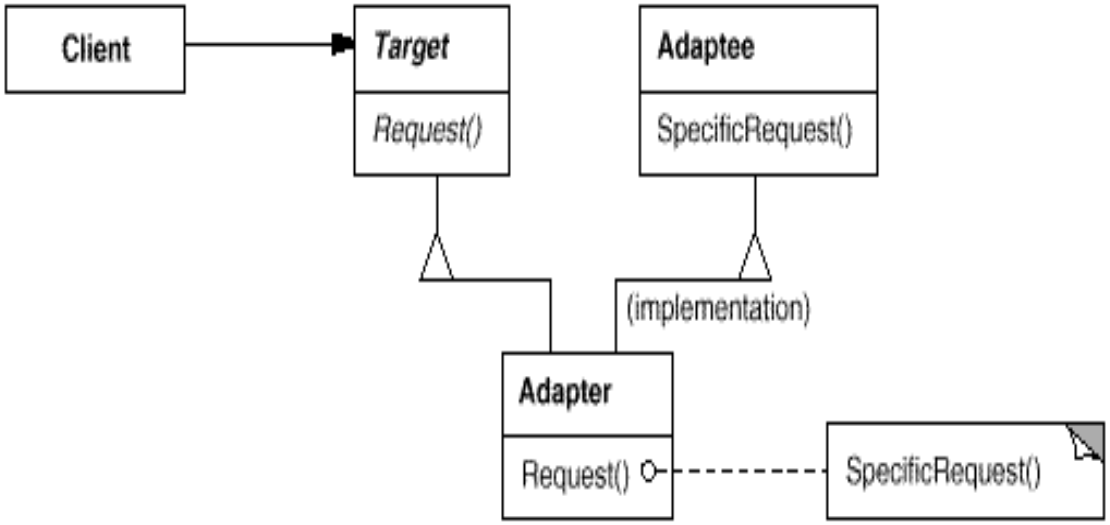
Adaptee

- defines an existing interface that needs adapting.

Adapter

- adapts the interface of Adaptee to the Target interface.

CONSEQUENCES



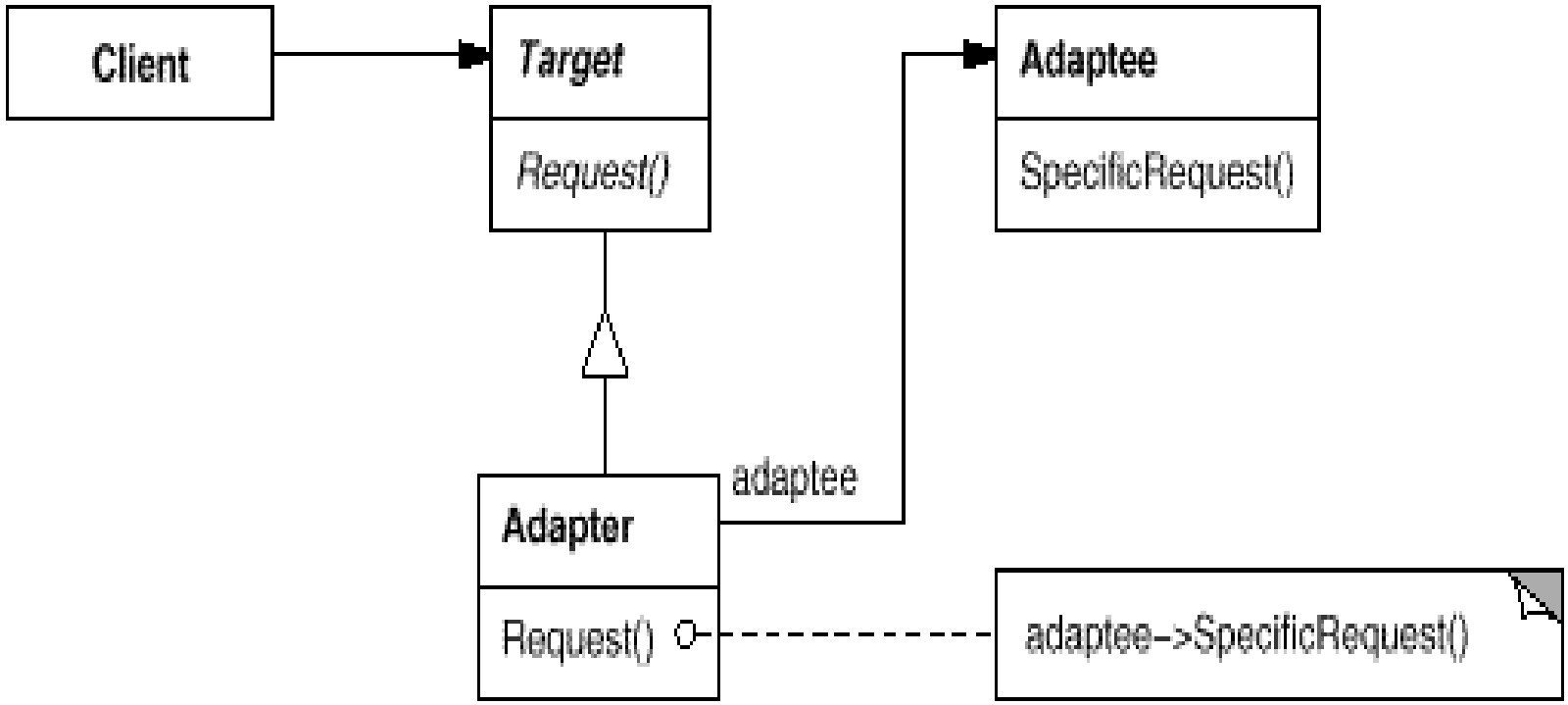
Class Adapter (Inheritance based)

- adapts Adaptee to Target by committing to a concrete Adapter class => a class adapter won't work when we want to adapt a class *and* all its subclasses.

- lets Adapter **override** some of Adaptee's behavior, since Adapter is a subclass of Adaptee.

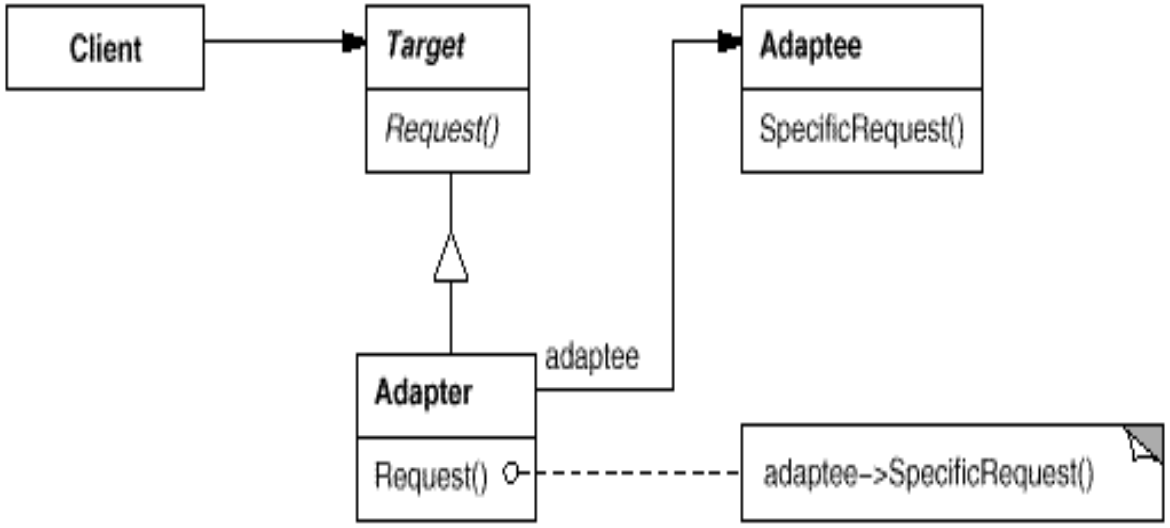
- introduces only one object, and **no additional pointer indirection** is needed to get to the Adaptee.

ADAPTER PATTERN – COMPOSITION BASED




```
class Adapter implements Target {
      Adaptee adaptee;
…}
```

CONSEQUENCES



Object Adapter (Composition based)

- lets a single Adapter work with many Adaptees (that is, the Adaptee itself and all of its subclasses (if any)). The Adapter can also add functionality to all Adaptees at once.

- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

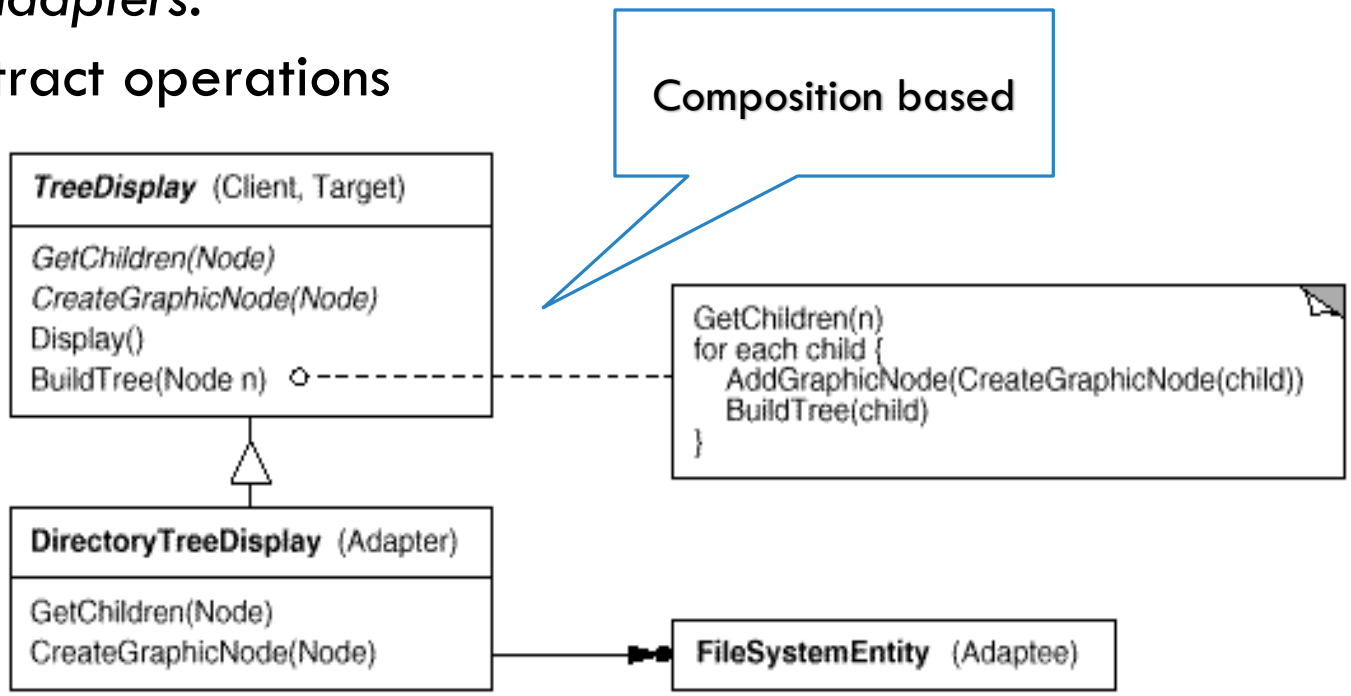
OTHER ISSUES

How much adapting does Adapter do?

- from simple interface conversion, to supporting an entirely different set of operations

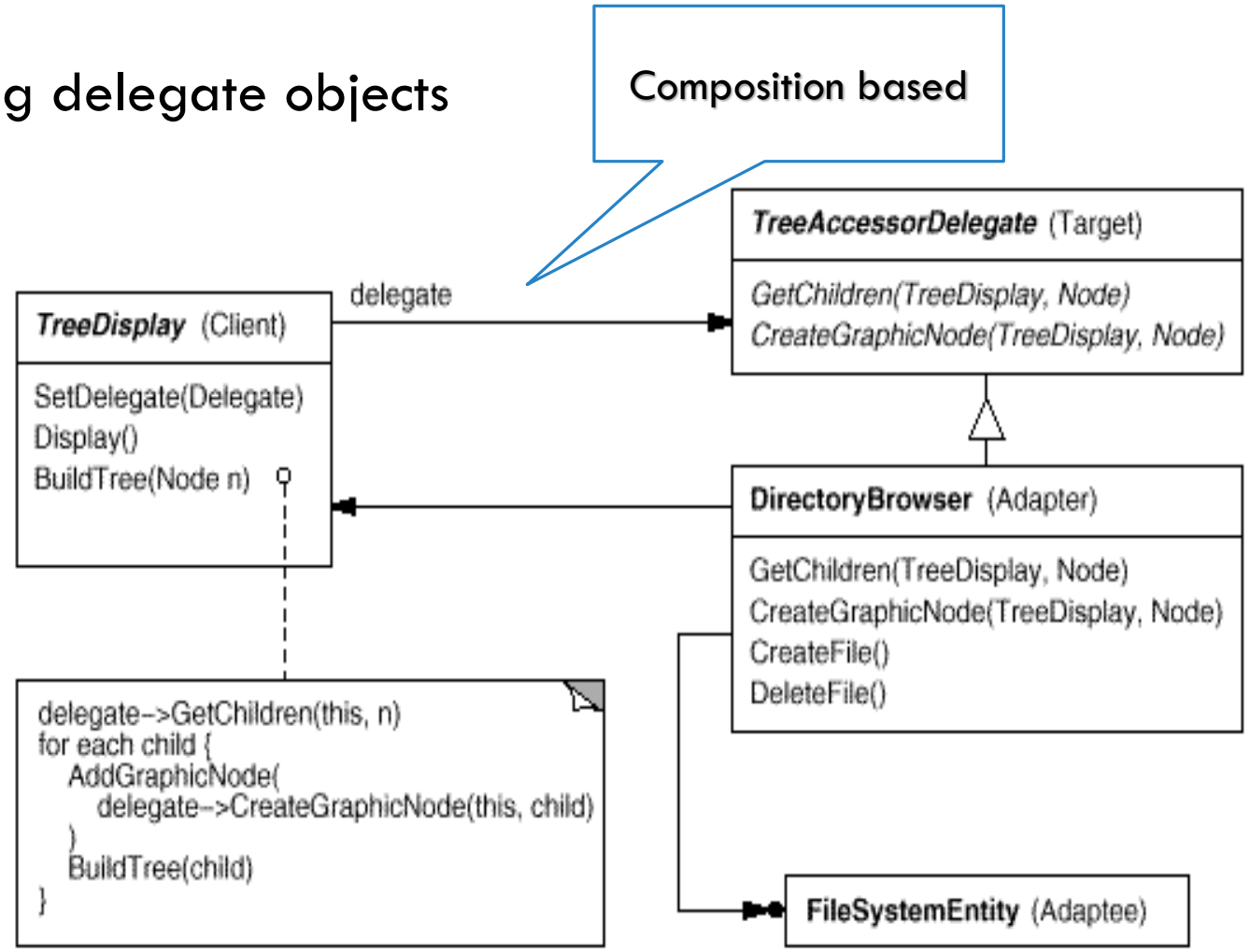
Pluggable adapters.

- Using abstract operations



PLUGGABLE ADAPTERS CONTINUED

- Using delegate objects

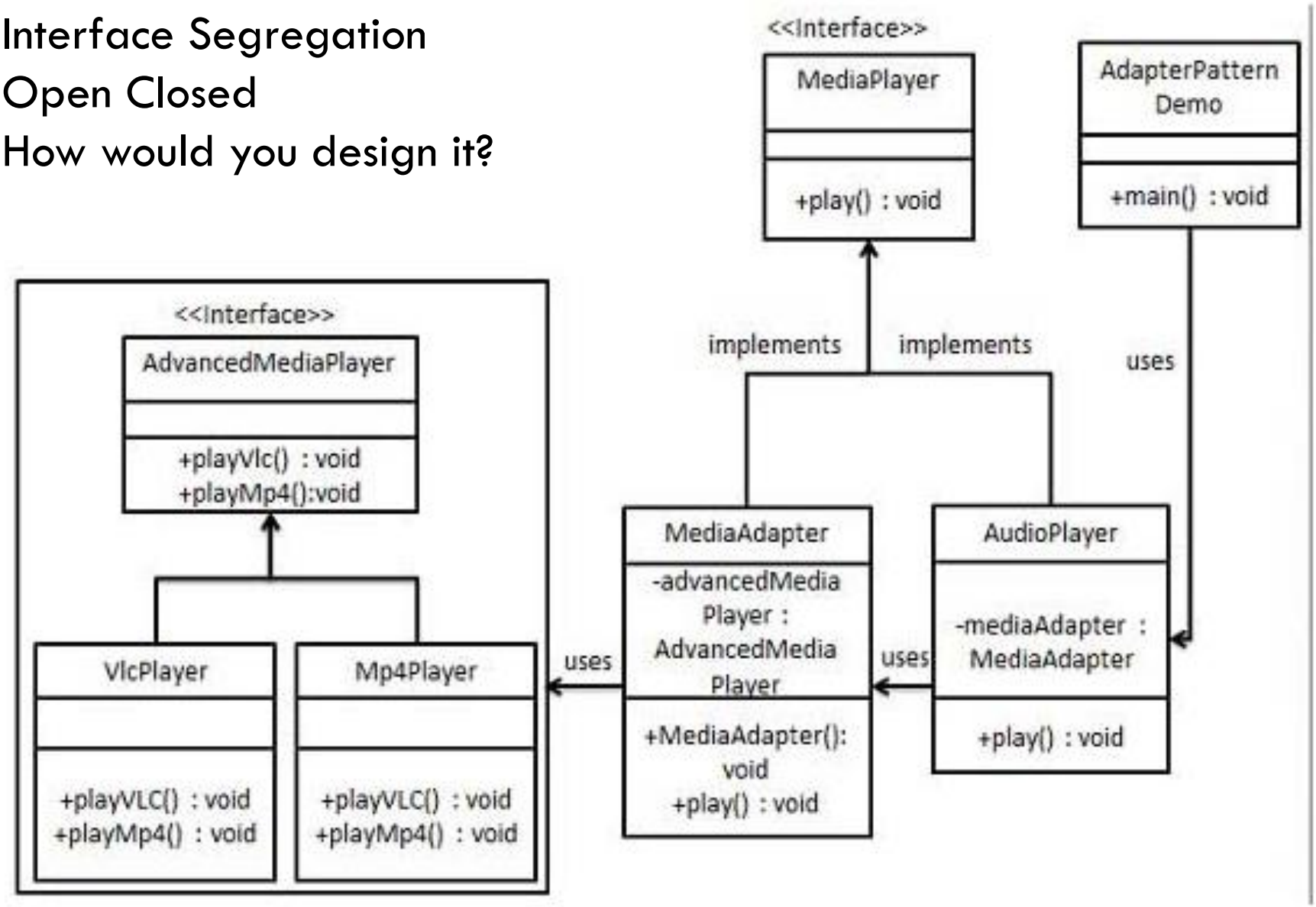


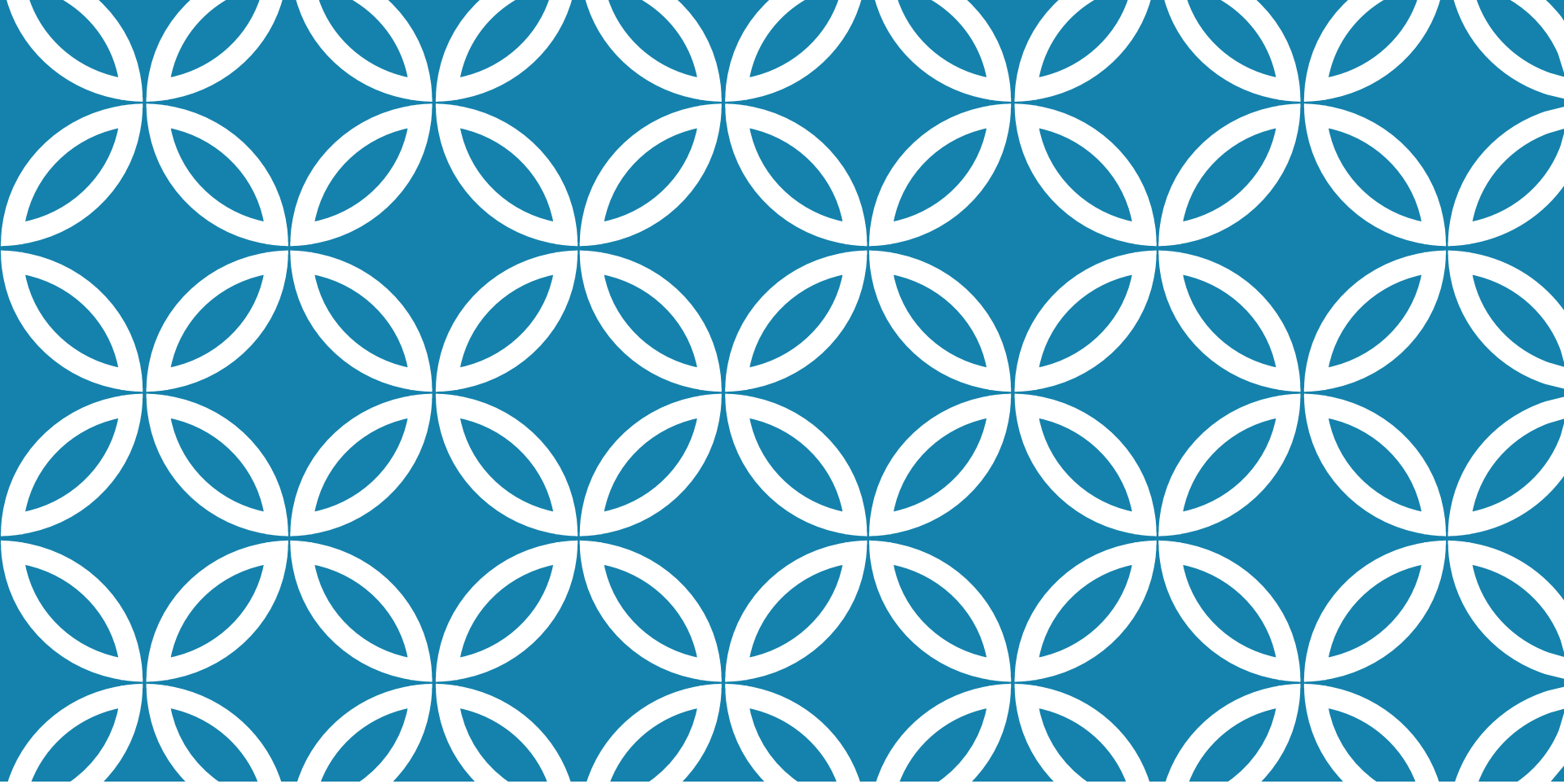
ANOTHER (BAD!) EXAMPLE

Interface Segregation

Open Closed

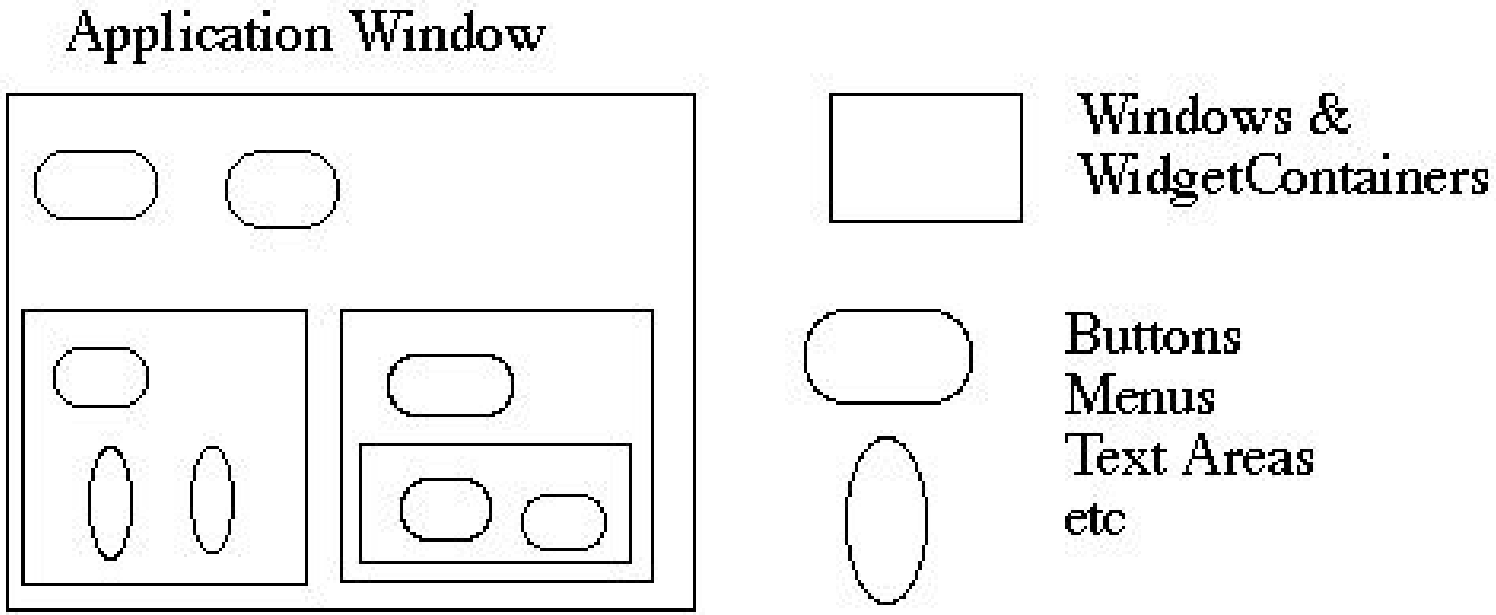
How would you design it?





COMPOSITE PATTERN

MOTIVATION



GUI Windows and GUI elements

- How does the window hold and deal with the different items it has to manage?
- Widgets are different than WidgetContainers

IMPLEMENTATION IDEAS

Nightmare Implementation

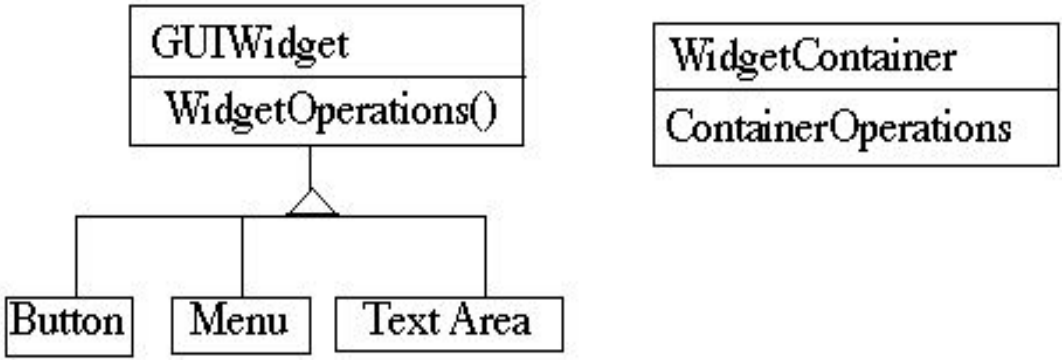
- for each operation **deal** with each category of objects **individually**
- **no uniformity** and **no hiding of complexity**
- a lot of **code duplication**

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if ( myButtons != null )
            for ( int k = 0; k < myButtons.length(); k++ )
                myButtons[k].refresh();
        if ( myMenus != null )
            for ( int k = 0; k < myMenus.length(); k++ )
                myMenus[k].display();
        if ( myTextAreas != null )
            for ( int k = 0; k < myButtons.length(); k++ )
                myTextAreas[k].refresh();
        if ( myContainers != null )
            for (int k = 0; k < myContainers.length();k++)
                myContainers[k].updateElements();
        // ...etc.
    }

    public void fooOperation()
        {
        if ( blah ) etc.
        // again and again.. . .
        }
    }
```

PROGRAM TO AN INTERFACE



- **uniform** dealing with widget operations
- but still containers are treated different

```
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update() {
      if(myWidgets != null)
        for (int k = 0; k < myWidgets.length(); k++)
           myWidgets[k].update();
      if(myContainers != null)
        for (int k = 0; k < myContainers.length(); k++)
           myContainers[k].updateElements();
       // .. .. etc.
    }
}
```

BASIC ASPECTS OF COMPOSITE PATTERN

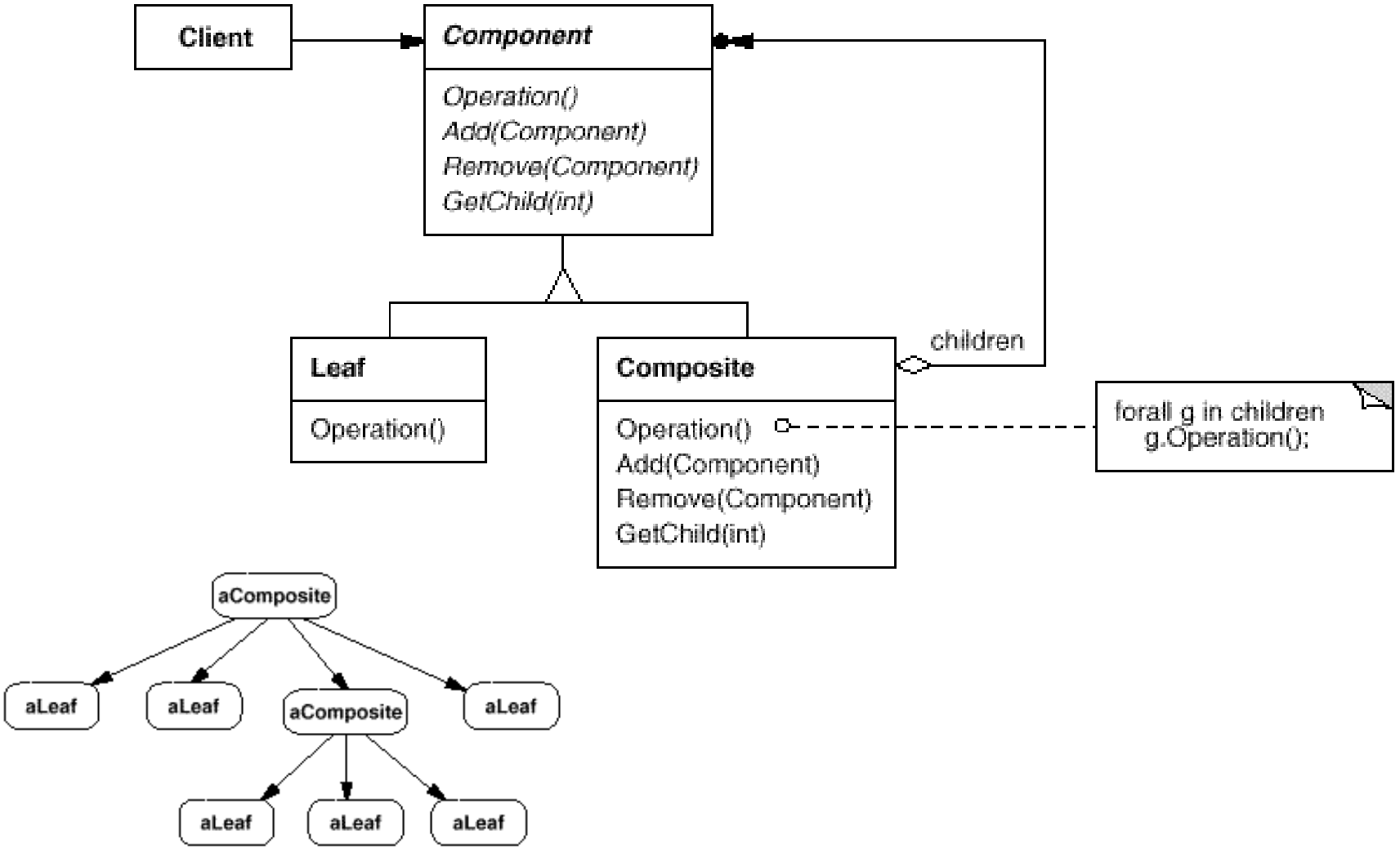
Intent

- Treat individual objects and compositions of these object uniformly
- Compose objects into tree-structures to represent recursive aggregations

Applicability

- represent part-whole hierarchies of objects
- be able to ignore the difference between compositions of objects and individual objects

STRUCTURE


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();
aComposite
aLeaf
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf

PARTICIPANTS & COLLABORATIONS

Component

- declares interface for objects in the composition
- implements default behavior for components when possible

Composite

- defines behavior for components having children
- stores child components
  - implement child-specific operations

Leaf

- defines behavior for primitive objects in the composition

Client

- manipulates objects in the composition through the Component interface

CONSEQUENCES

Defines uniform class hierarchies

- recursive composition of objects

Make clients simple

- don't know whether dealing with a leaf or a composite
- simplifies code because it avoids to deal in a different manner with each class

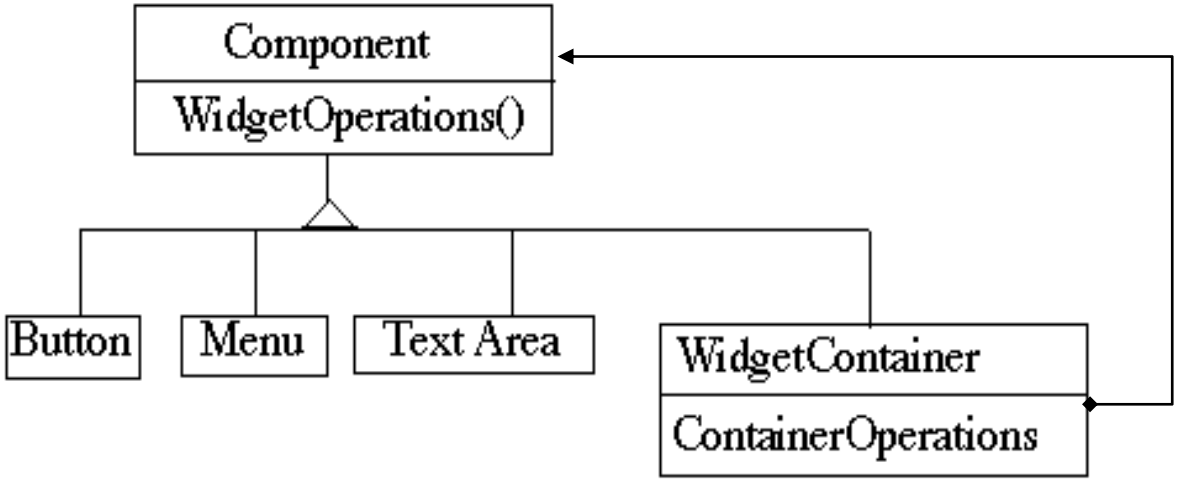
Easier to extend

- easy to add new Composite or Leaf classes

Design excessively general

- **type checks needed to restrict the types admitted in a particular composite structure**

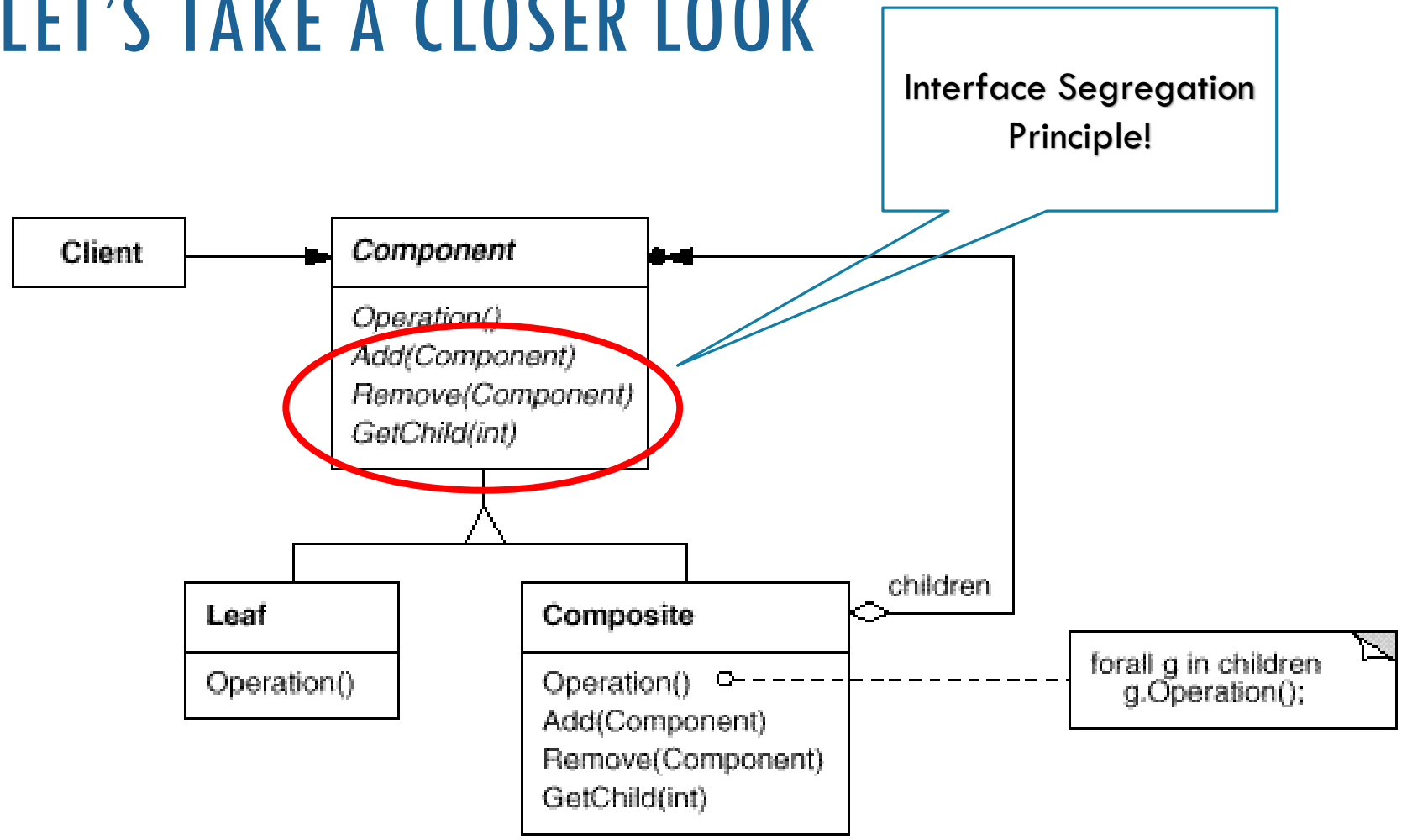
APPLYING COMPOSITE TO WIDGET PROBLEM




```
class WidgetContainer {
   Component[] myComponents;

   public void update() {
     if ( myComponents != null )
       for( int k = 0; k < myComponents.length(); k++ )
          myComponents[k].update();
     }
}
```

LET'S TAKE A CLOSER LOOK



WHERE TO PLACE CONTAINER OPERATIONS ?

Adding, deleting, managing components in composite

- should they be placed in Component or in Composite?

Pro-Transparency Approach

- Declaring them in the Component gives all subclasses the same interface
  - All subclasses can be treated alike.
- Safety costs!
  - clients may do stupid things like adding objects to leaves
  - getComposite() to improve safety.

Pro-Safety Approach

- Declaring them in Composite is safer
  - Adding or removing widgets to non-Widget Containers is an error

OTHER IMPLEMENTATION ISSUES

Explicit references to Composite

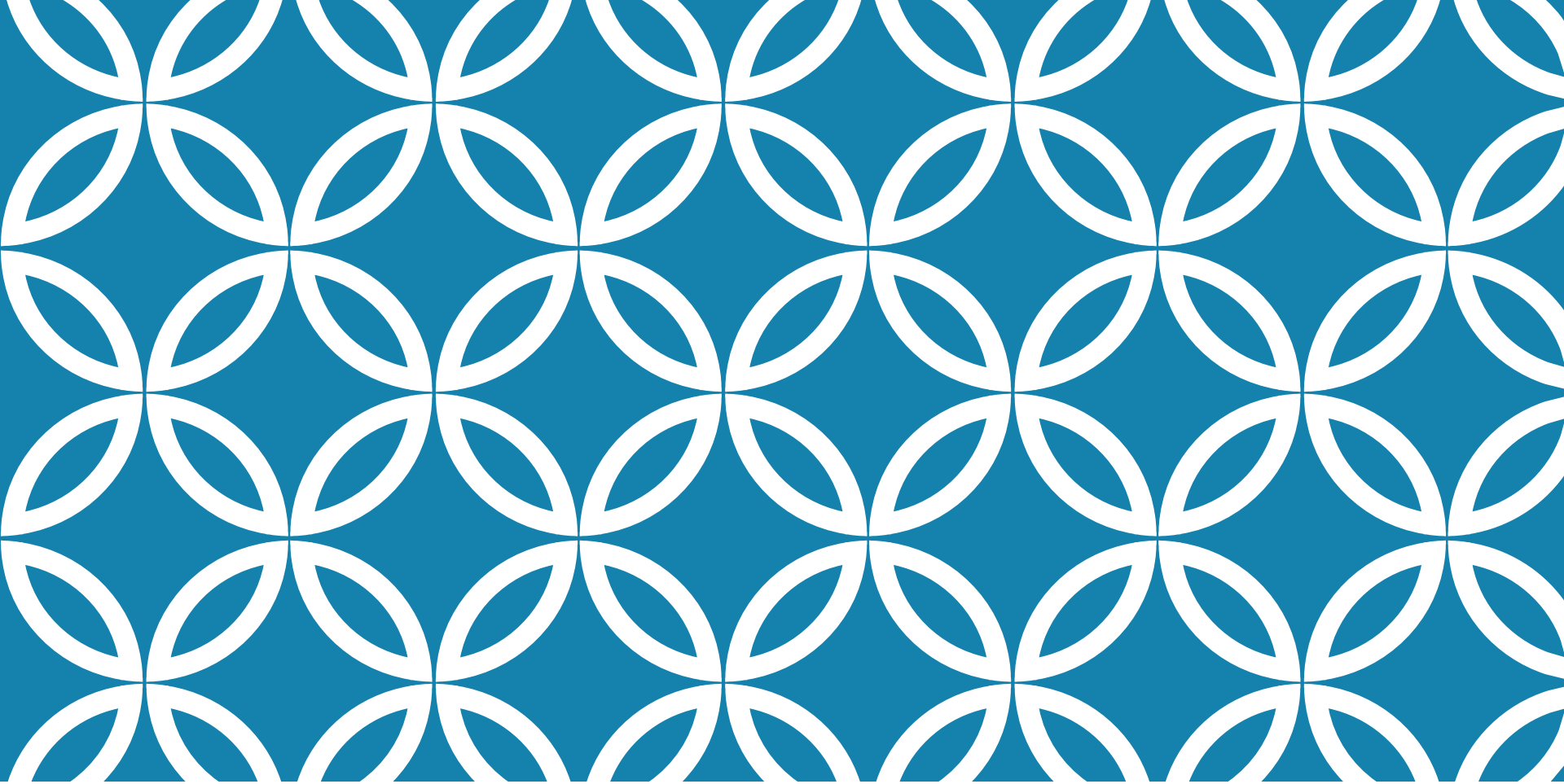
- simplifies traversal
- place it in Component
- the consistency issue
  - change reference to Composite **only** when add or remove component

Component Ordering in Composites

- consider using Iterator

Who should **delete** components?

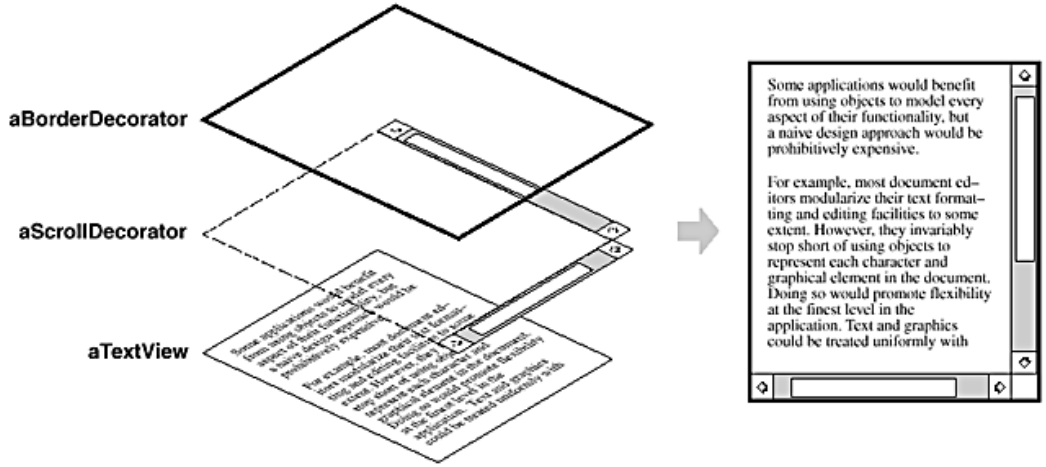
- Composite should delete its components



DECORATOR PATTERN

Changing the skin of an object

MOTIVATION



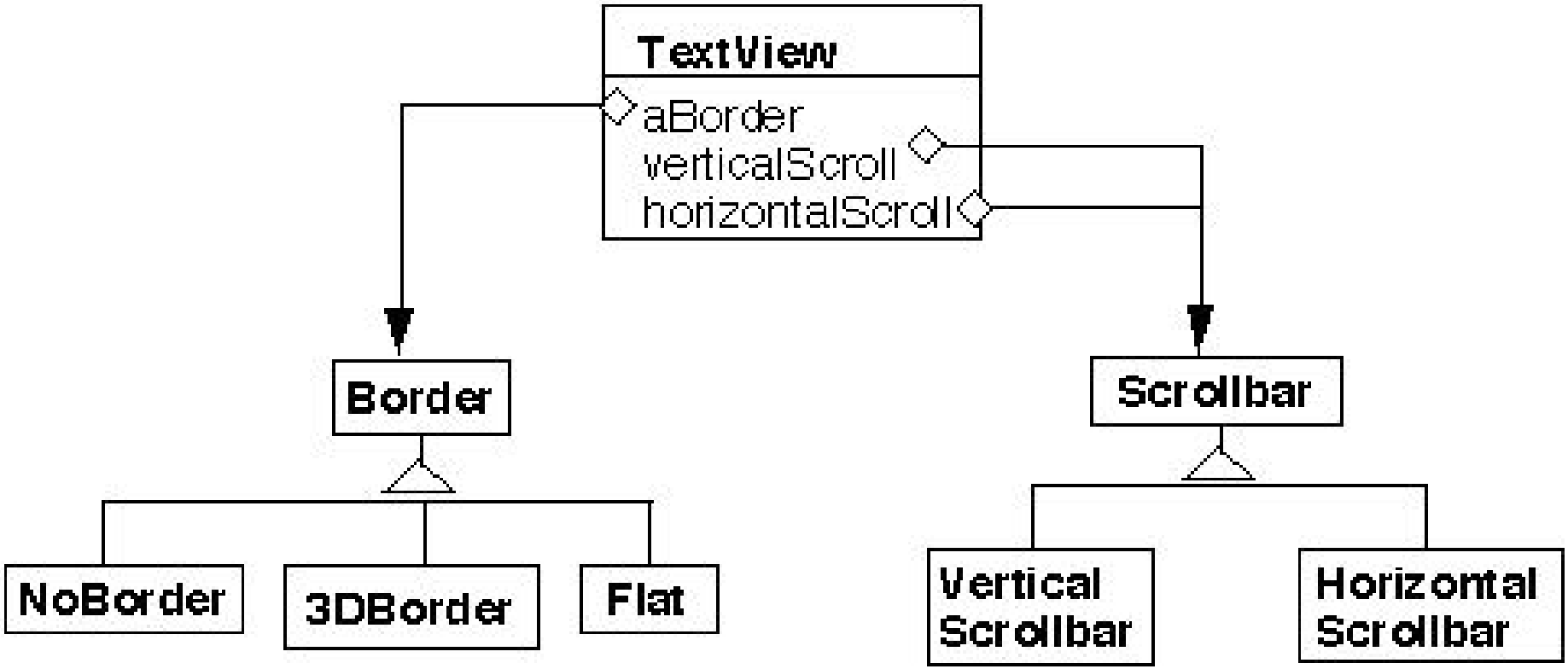
A TextView has 2 features:

- borders:
  - 3 options: none, flat, 3D
- scroll-bars:
  - 4 options: none, side, bottom, both

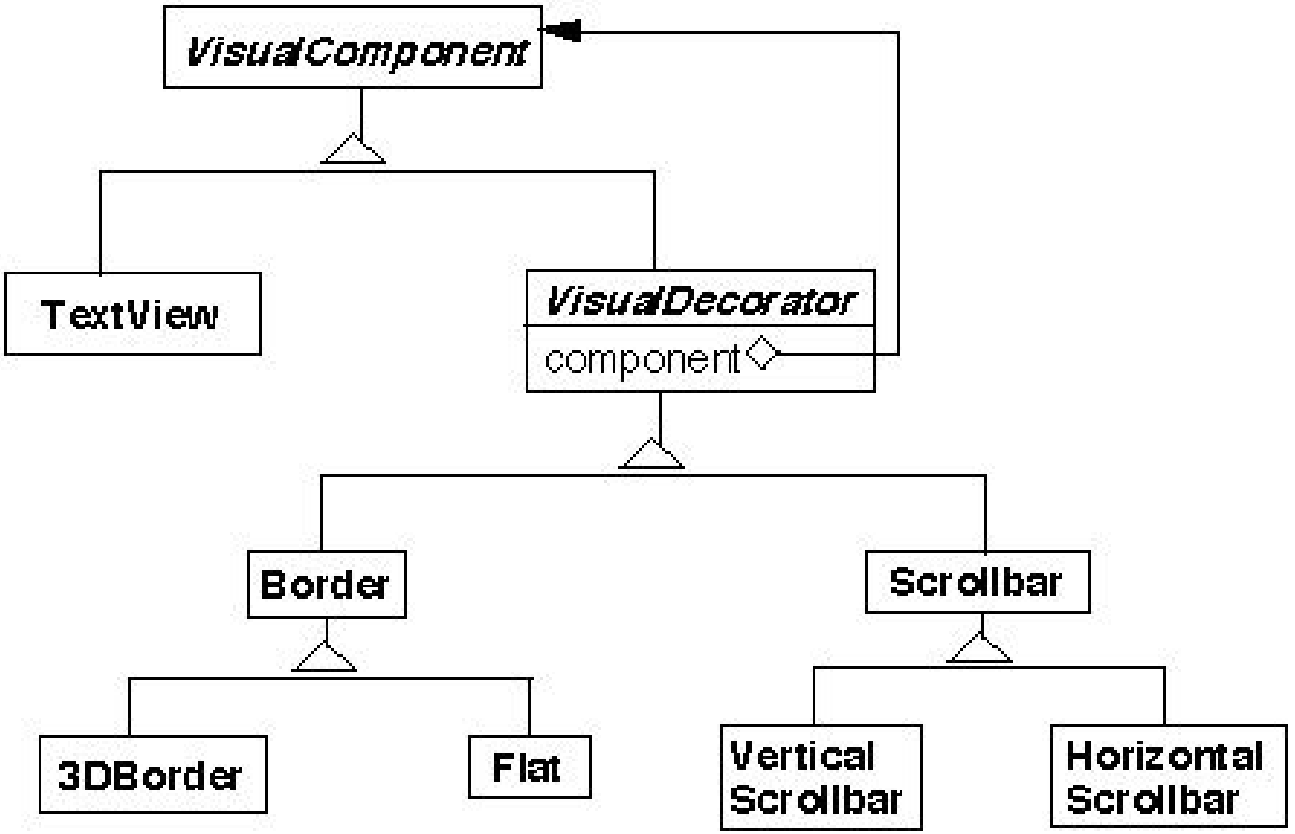
Inheritance => How many Classes?

- 3 x 4 = **12** !!!
  - e.g. TextView, TextViewWithNoBorder&SideScrollbar, TextViewWithNoBorder&BottomScrollbar, TextViewWithNoBorder&Bottom&SideScrollbar, TextViewWith3DBorder, TextViewWith3DBorder&SideScrollbar, TextViewWith3DBorder&BottomScrollbar, ...

SOLUTION 1: USE OBJECT COMPOSITION



SOLUTION 2: CHANGE THE SKIN, NOT THE GUTS!



TextView has **no** borders or scrollbars!

Add borders and scrollbars on top of a TextView

BASIC ASPECTS

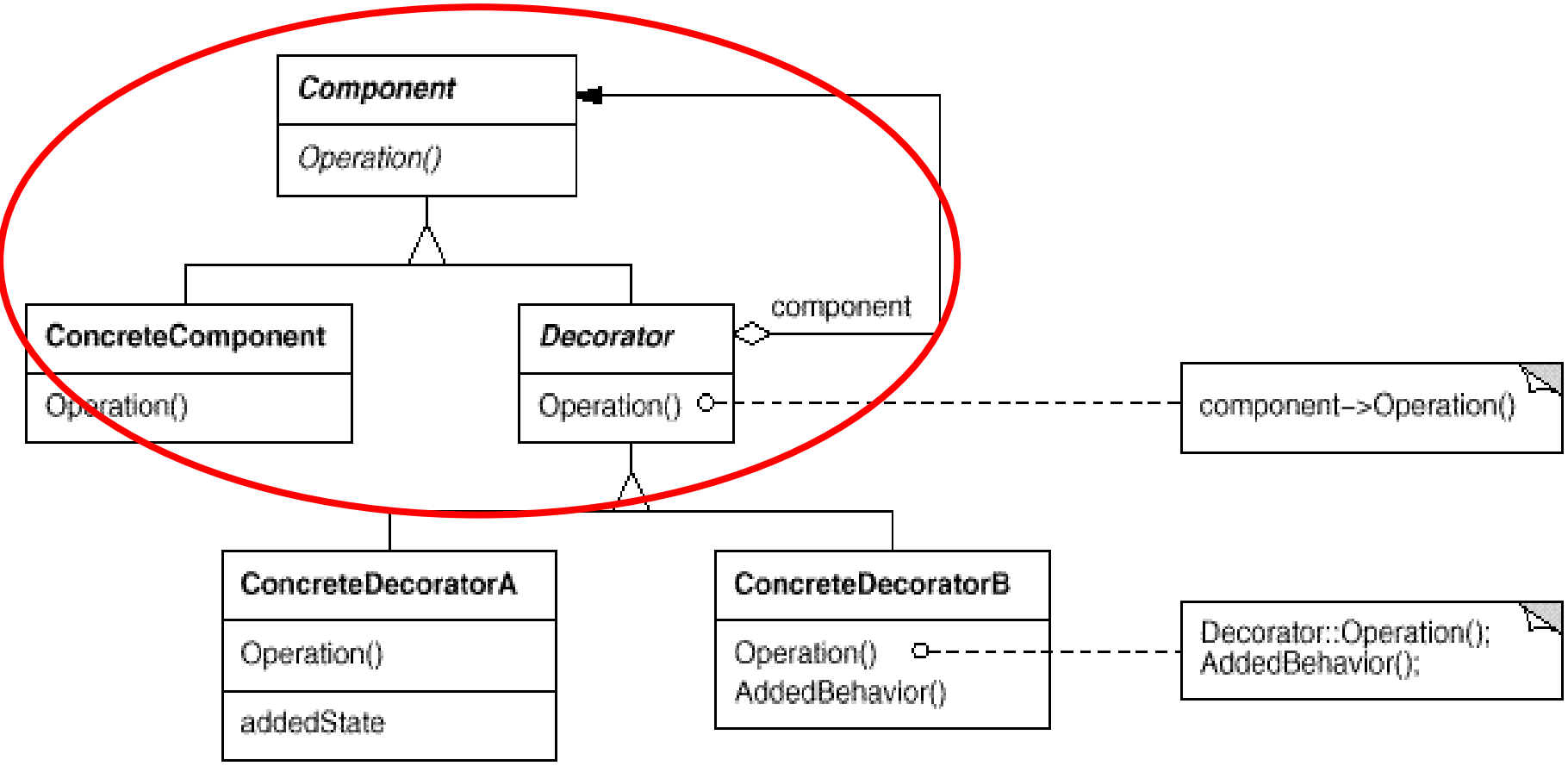
Intent

- *Add responsibilities to a particular object rather than its class*
  - Attach additional responsibilities to an object dynamically.
- Provide a flexible alternative to subclassing

Applicability

- Add responsibilities to objects transparently and dynamically
  - i.e. without affecting other objects
- Extension by subclassing is impractical
  - may lead to too many subclasses

STRUCTURE


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

PARTICIPANTS & COLLABORATIONS

Component

- defines the interface for objects that can have responsibilities added dynamically

ConcreteComponent

- the "bases" object to which additional responsibilities can be added

Decorator

- defines an interface conformant to Component's interface
  - for transparency
- maintains a reference to a Component object

ConcreteDecorator

- adds responsibilities to the component

CONSEQUENCES

More **flexibility** than static inheritance

- allows to mix and match responsibilities
- allows to apply a property twice

Avoid feature-laden classes high-up in the hierarchy

- "*pay-as-you-go*" approach
- easy to define new types of decorations

Lots of little objects

- easy to customize, but hard to learn and debug

A decorator and its component aren't identical

- checking object identification can cause problems
  - e.g. `if (aComponent instanceof TextView )`

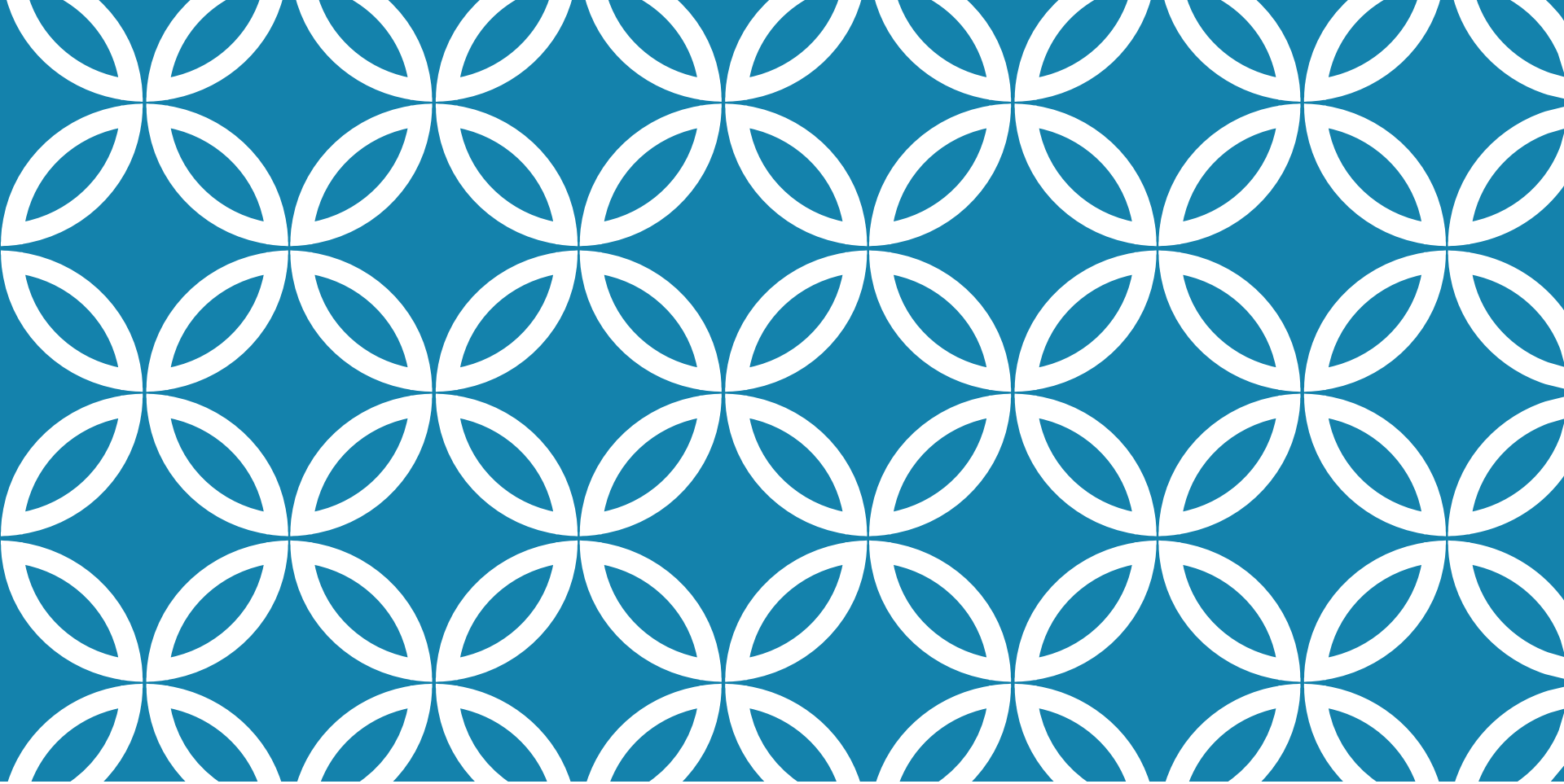
IMPLEMENTATION ISSUES

Keep Decorators lightweight

- Don't put data members in Component
- Use it for shaping the interface

Omitting the abstract Decorator class

- If only one decoration is needed
- Subclasses may pay for what they don't need



CHANGING THE GUTS

CHANGING THE "GUTS" OF AN OBJECT ...

Control

- "shield" the implementation from direct access (**Proxy**)

Decouple

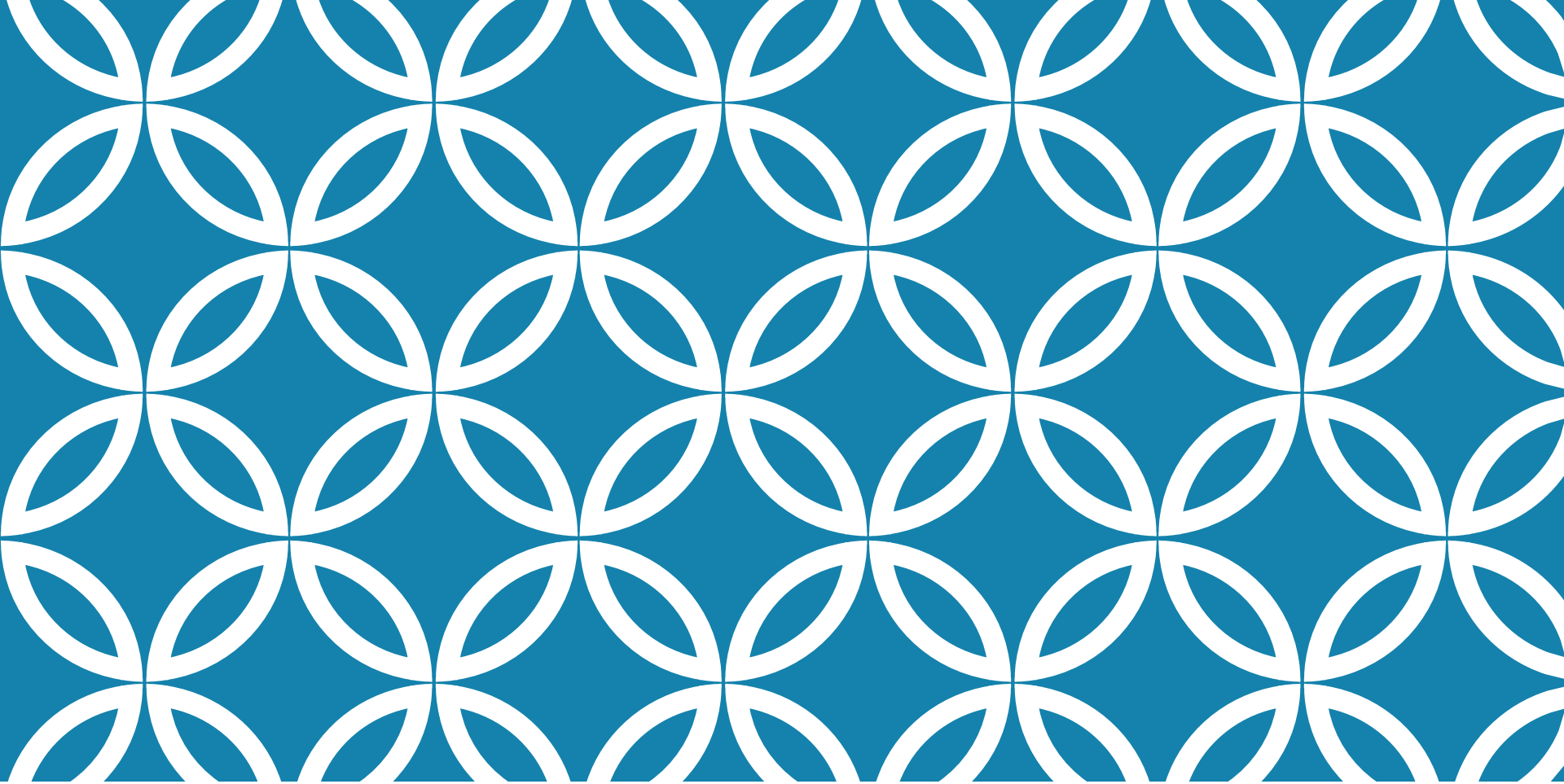
- let abstraction and implementation vary independently (**Bridge**)

Optimize

- use an alternative algorithm to implement behavior (Strategy)

Alter

- change behavior when object's state changes (State)



PROXY PATTERN

LOADING "HEAVY" OBJECTS

Document Editor that can embed multimedia objects

- MM objects are expensive to create $\Rightarrow$ opening of document slow
- avoid creating expensive objects
  - they are not all necessary as they are not all visible at the same time

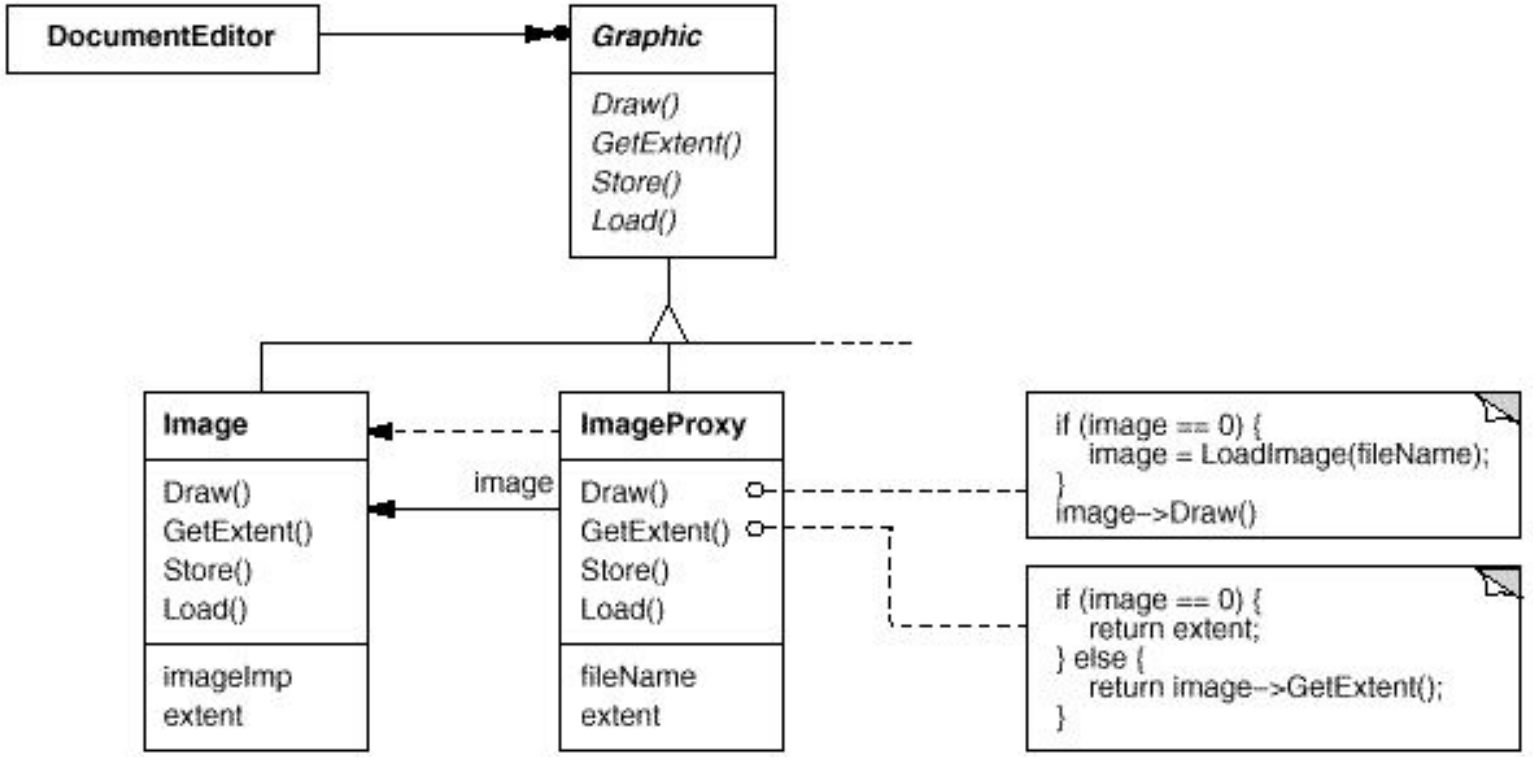
Creating each expensive object on demand !

- i.e. when image has to be displayed

What should we put instead?

- hide the fact that we are "lazy"!
- don't complicate the document editor!

IDEA: USE A PLACEHOLDER!



- create only when needed for drawing
- keeps information about the dimensions (extent)

BASIC ASPECTS

Definition: **proxy (n. pl prox-ies)** The agency for a person who acts as a substitute for another person, authority to act for another

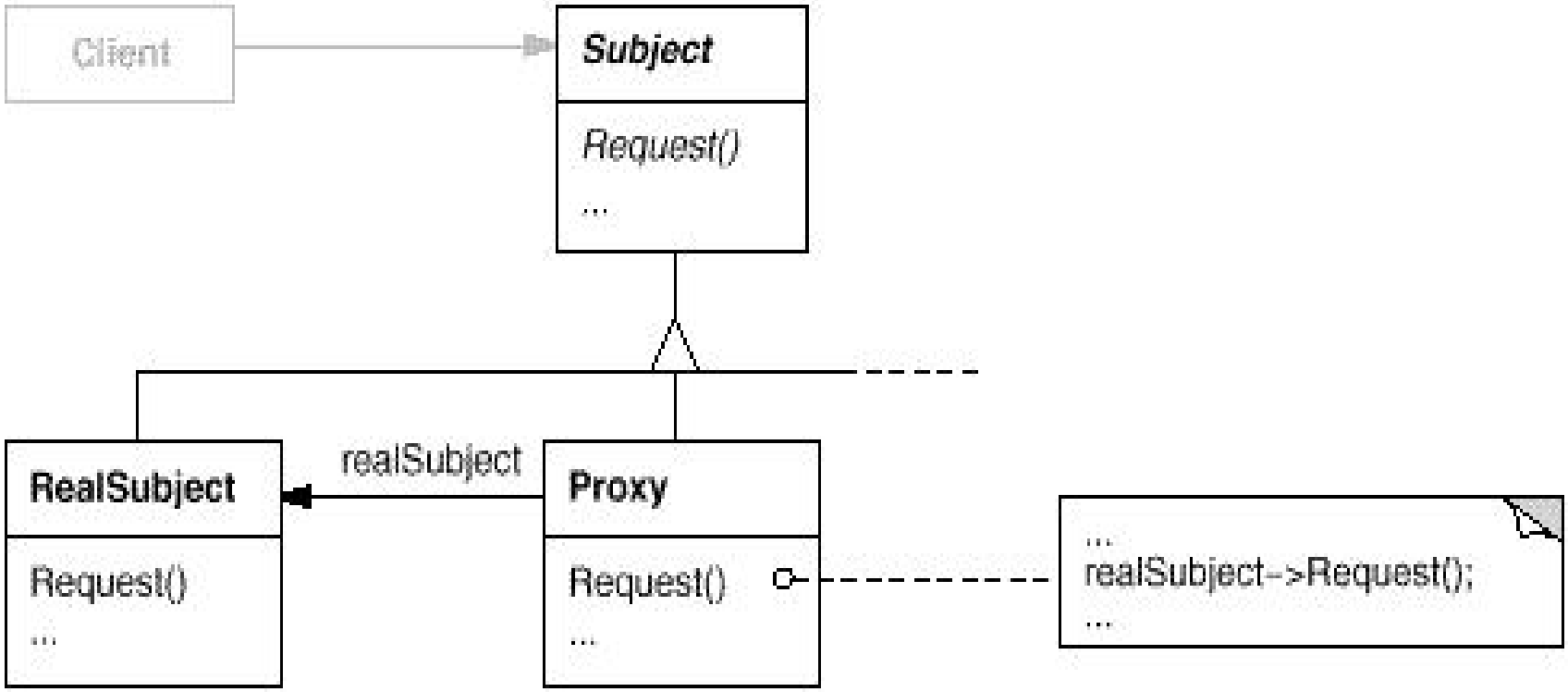
Intent

- provide a surrogate or placeholder for another object to control access to it

Applicability: whenever there is a need for a more *flexible* or *sophisticated* reference to an object than a simple pointer

- **remote** proxy … if real object is "far away"
- **virtual** proxy … if real object is "expensive"
- **protection** proxy … if real object is "vulnerable"
- **enhancement** proxies (*smart pointers*)
  - prevent accidental delete of objects (counts references)

STRUCTURE


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

PARTICIPANTS

Proxy

- maintains a reference that lets the proxy access the real subject.
- provides an interface identical to Subject's
  - so that proxy can be substituted for the real subject
- controls access to the real subject
  - may be responsible for creating or deleting it

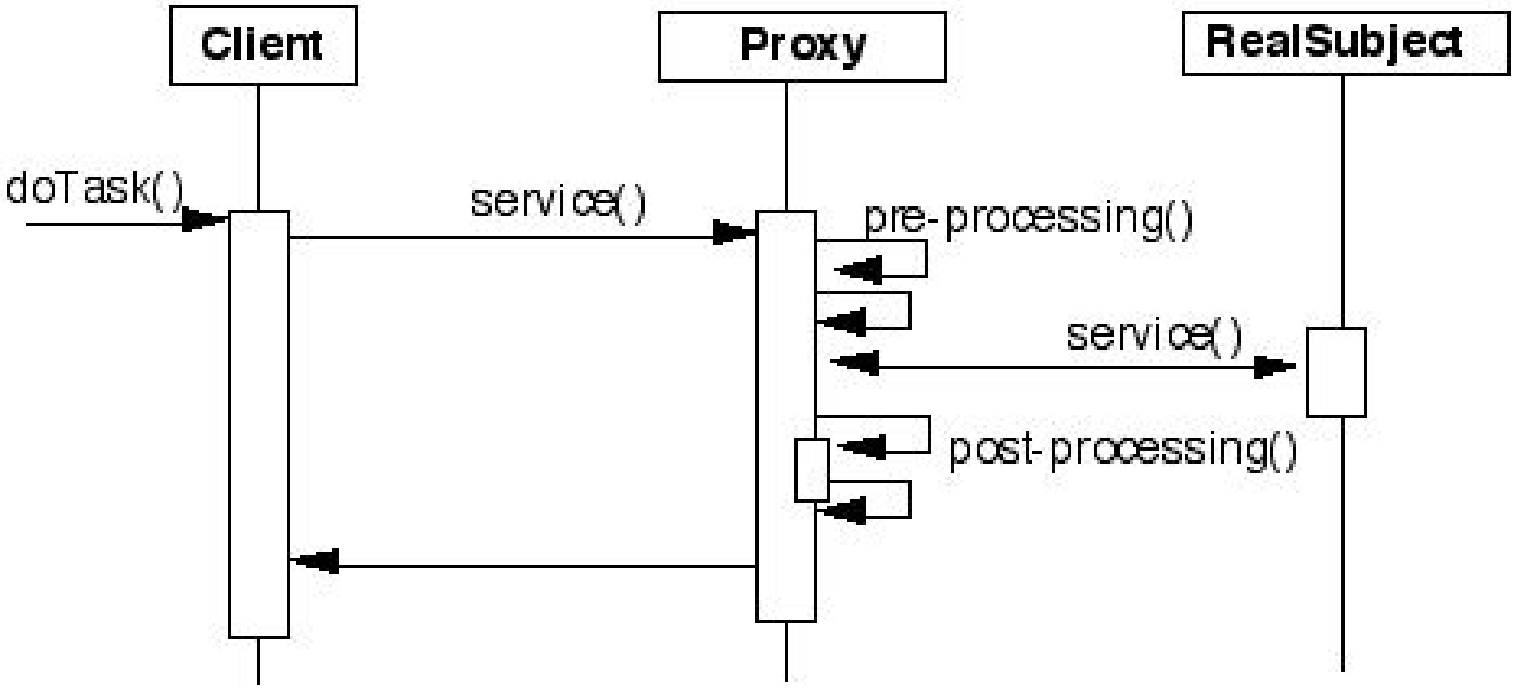
Subject

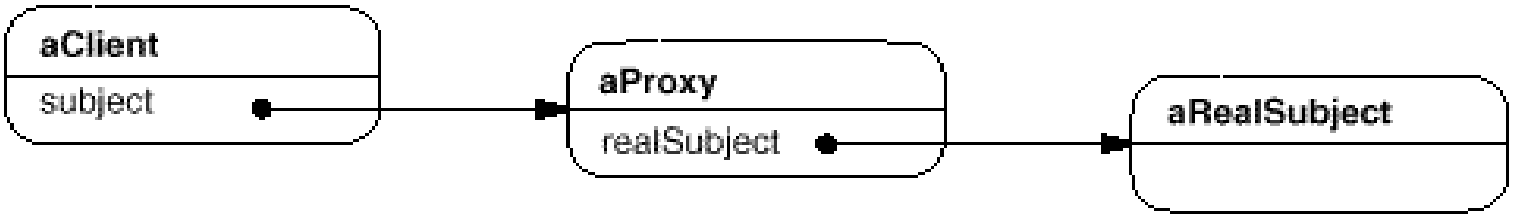
- defines the common interface for RealSubject and Proxy

RealSubject

- defines the real object that the proxy holds place for

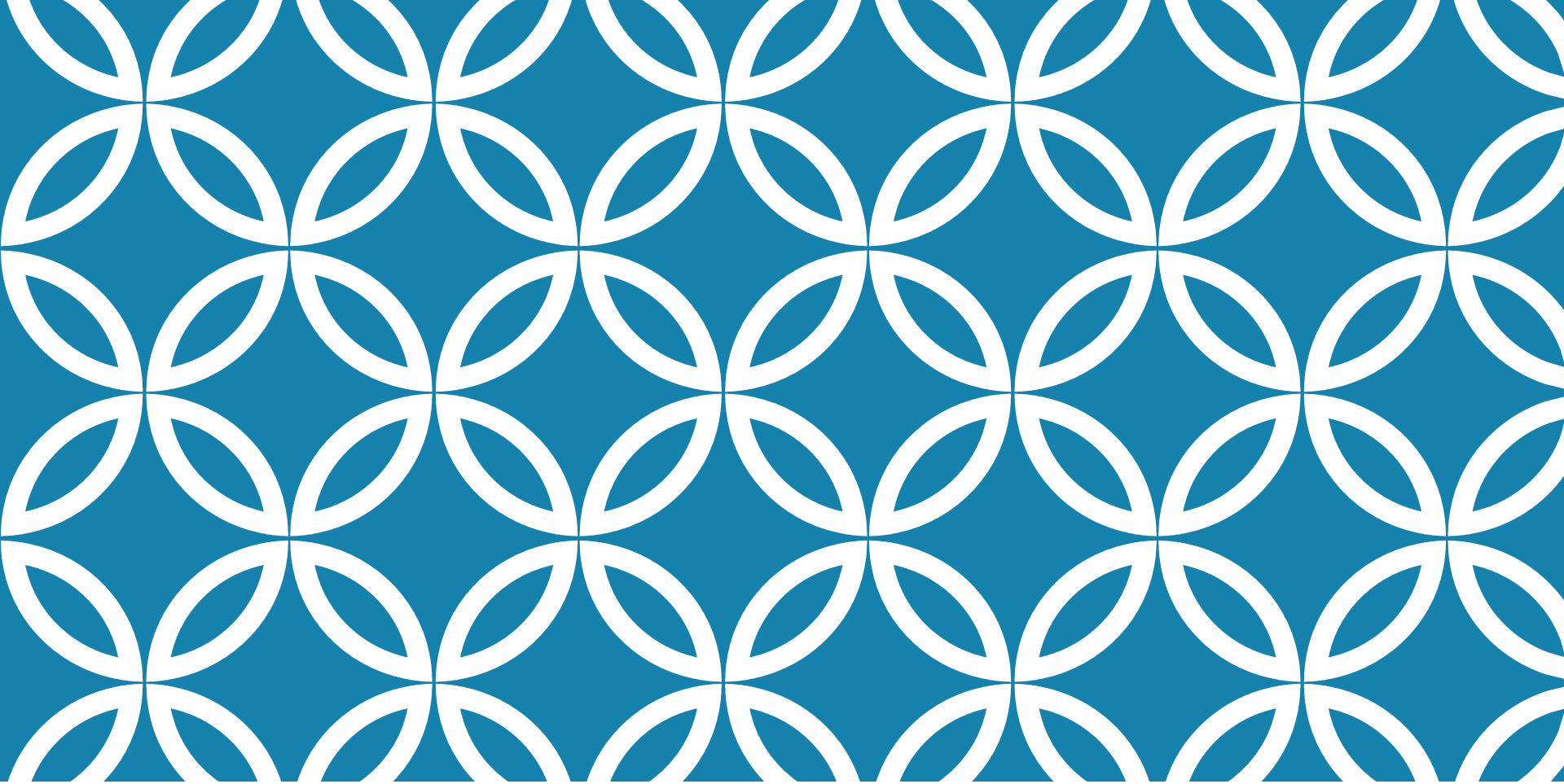
COLLABORATIONS





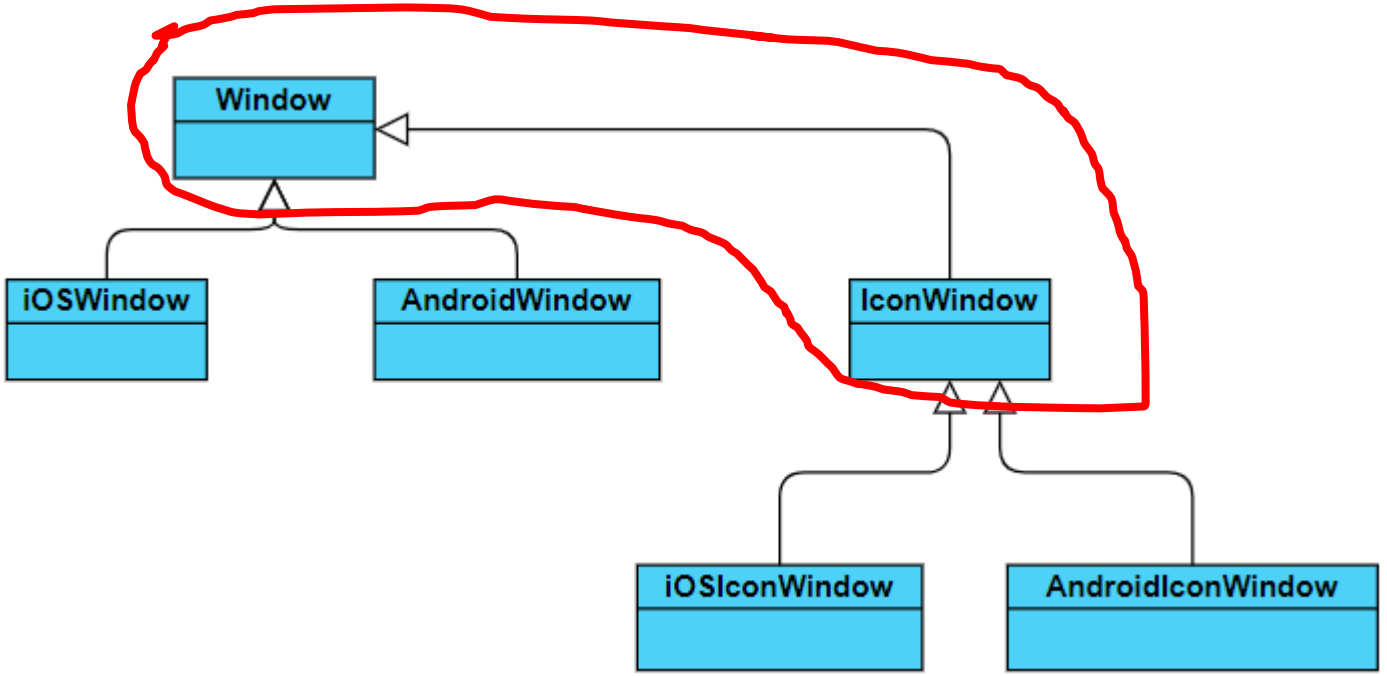
CONSEQUENCES

- introduces a level of indirection
- used differently depending on the kind of proxy:
  - hide different address space (remote p.)
  - creation on demand (virtual p.)
  - allow additional housekeeping activities (protection, smart pointers)



BRIDGE PATTERN

INHERITANCE THAT LEADS TO EXPLOSION!



BASIC ASPECTS OF BRIDGE PATTERN

Intent

- **decouple** an **abstraction** from its **implementation**
- allow implementation to vary independently from its abstraction
- abstraction defines and implements the interface
  - all operations in abstraction call methods from its implementation obj.

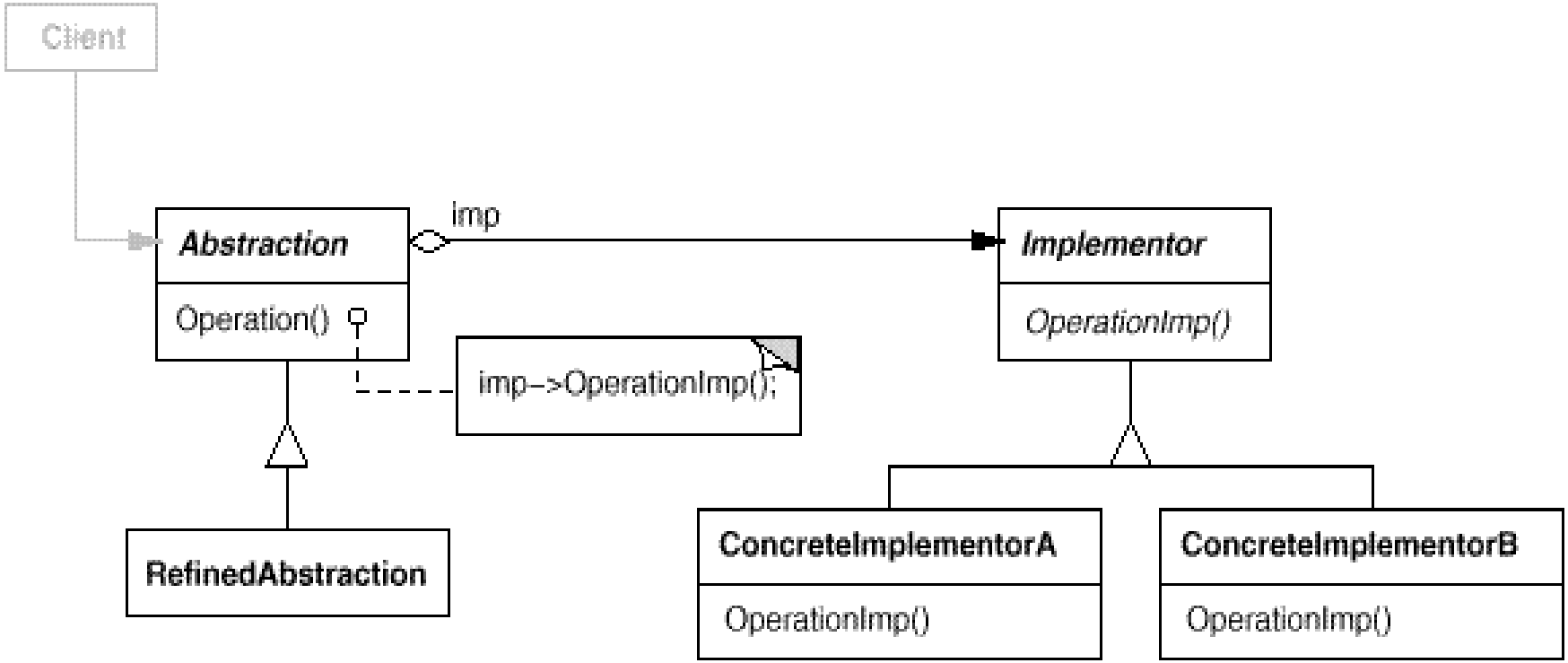
In the Bridge pattern ...

- ... an abstraction can use different implementations
- ... an implementation can be used in different abstractions

APPLICABILITY

- Avoid permanent binding btw. an abstraction and its implementation
- Abstractions and their implementations should be *independently extensible* by subclassing
- Hide the implementation of an abstraction completely from clients
  - their code should not have to be recompiled when the implementation changes
- Share an implementation among multiple objects
  - and this fact should be hidden from the client

STRUCTURE


Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp->OperationImp();
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

PARTICIPANTS

Abstraction

- defines the abstraction's interface
- maintains a reference to an object of type Implementor

Implementor

- defines the interface for implementation classes
- does not necessarily correspond to the Abstraction's interface
- Implementor contains primitive operations,
- Abstraction defines the higher-level operations based on these primitives

RefinedAbstraction

- extends the interface defines by Abstraction

ConcreteImplementer

- implements the Implementor interface, defining a concrete impl.

CONSEQUENCES

Decoupling interface and implementation

- Implementation is **configurable** and **changeable** at run-time
- reduce compile-time dependencies
  - implementation changes do not require Abstraction to recompile

Improved extensibility

- extend by subclassing independently Abstractions and Implementations

Hiding implementation details from clients

- shield clients from implementations details
  - e.g. sharing implementor objects together with reference counting

IMPLEMENTATION

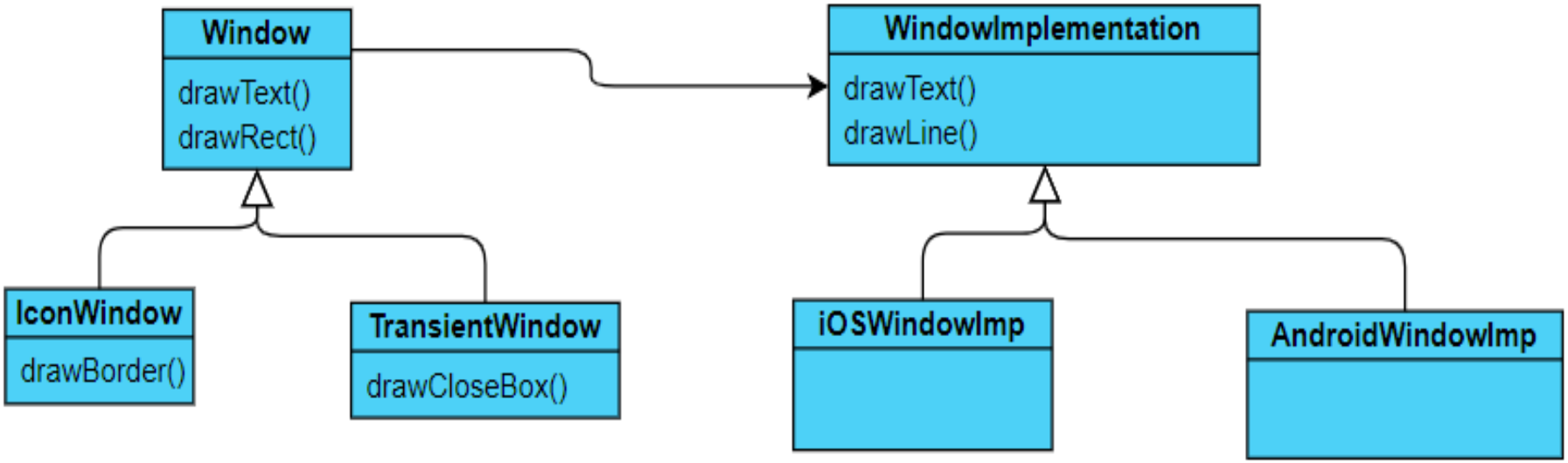
Only one Implementor

- not necessary to create an abstract implementor class
- degenerate, but useful due to decoupling

Which Implementor should I use ?

- Variant 1: let Abstraction know all concrete implem. and choose
- Variant 2: choose initially default implem. and change later
- Variant 3: use an Abstract Factory
  - no coupling btw. Abstraction and concrete implem. classes

WINDOWS EXAMPLE REVISITED



ADAPTER VS. BRIDGE

Common features

- promote flexibility by providing a level of indirection to another object.
- involve forwarding requests to this object from an interface other than its own.

Differences

- Adapter **resolves incompatibilities** between two existing interfaces. No focus on how those interfaces are implemented, nor how they might evolve independently
- Bridge **links an abstraction and its implementations**. It provides a stable interface to clients as it lets you vary the classes that implement it.
- **The Adapter pattern makes things work *after* they're designed; Bridge makes them work *before* they are.**

NEXT TIME

Behavioral DP