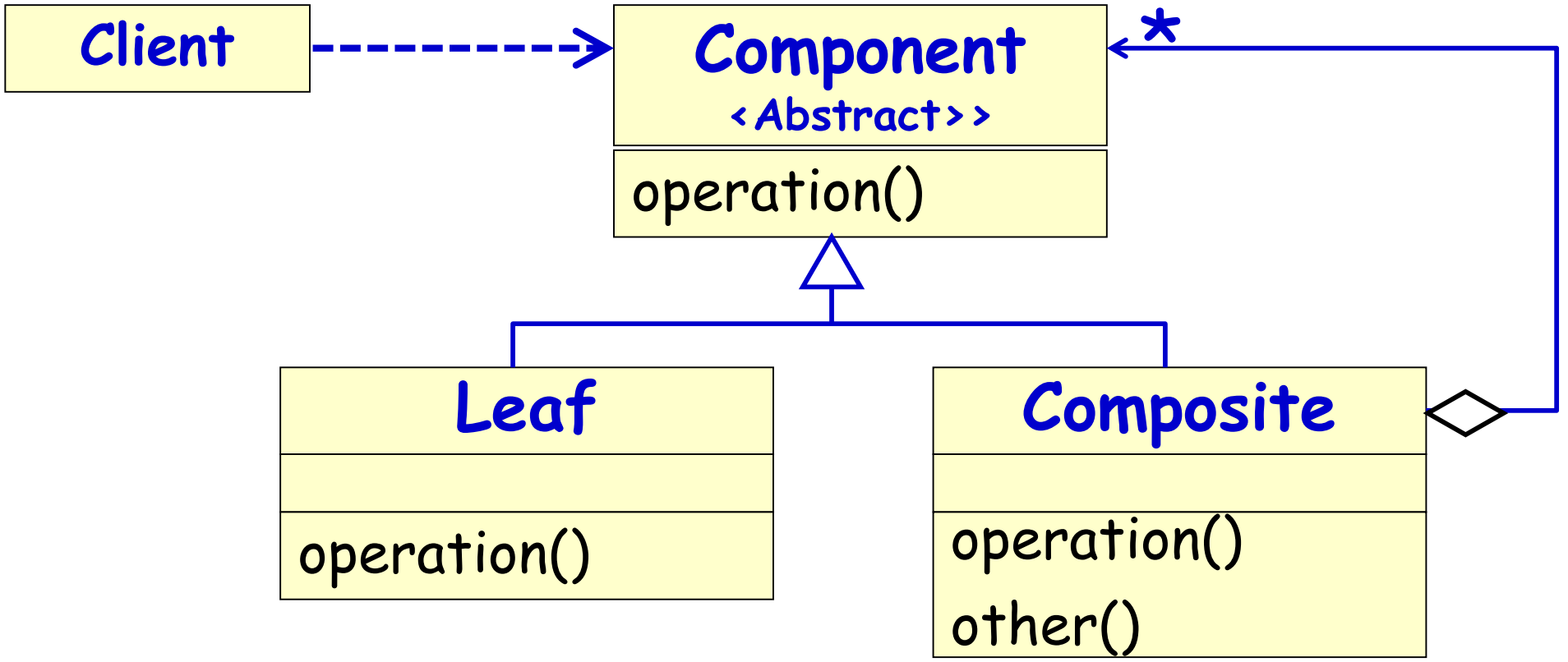
Remaining parts of Composite Pattern and then Adapter and Bridge Patterns

Lect 22--23

16-10-2023

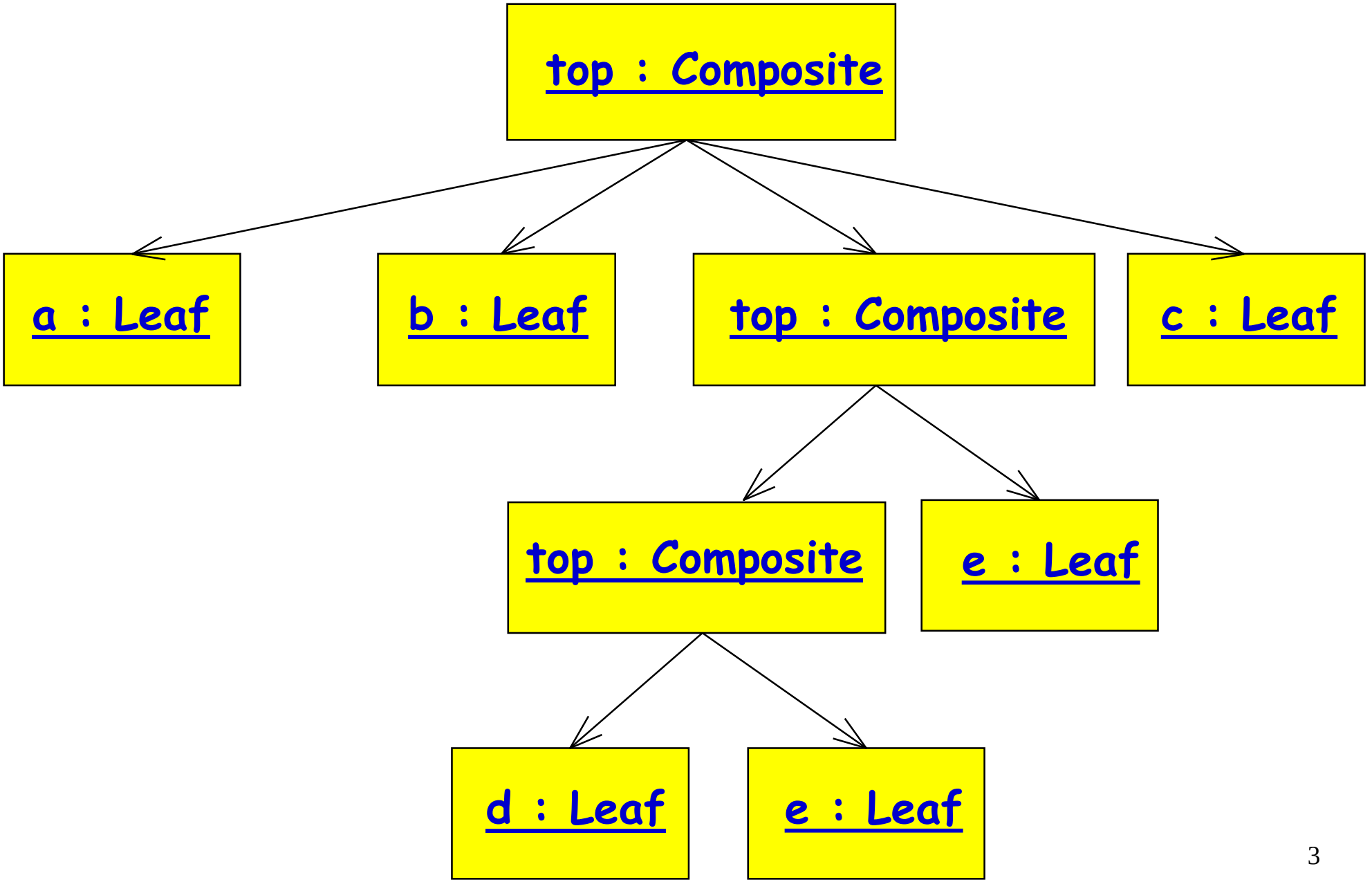
Composite Pattern: Class Diagram



·Each node of the Component structure can respond to some common operation(s).

·The client can call operation of the Component and the structure responds "appropriately".

Composite: Object Diagram



Consequences

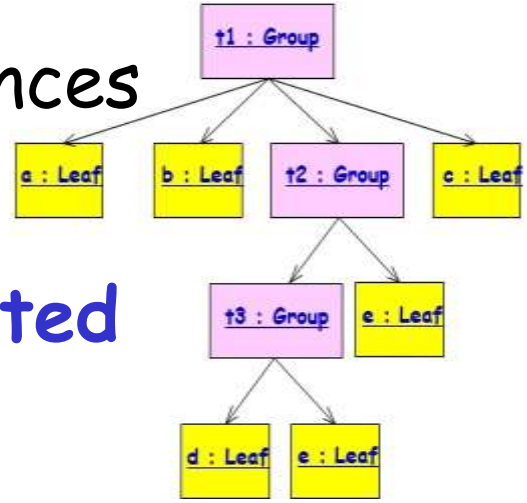
- **Makes it possible to define recursive compositions of primitive and composite objects.**

- Makes the client simple.
  - They don't need to know whether they are dealing with leaves or composites.

- Makes it easier to add new kinds of components.

Applicability

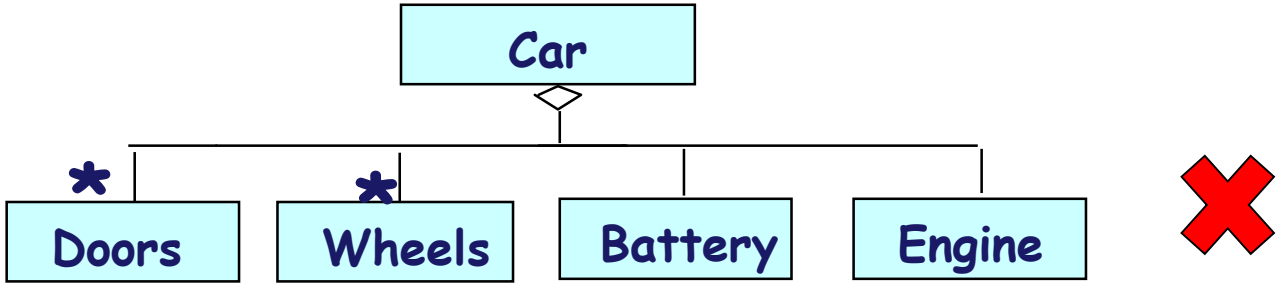
- Use the Composite pattern when:
  - You need to represent part-whole hierarchies of objects
  - You want clients to ignore the differences between parts and wholes
  - **Especially use, if the parts are created dynamically – at run time:**
    - **Example:** to build a complex system from primitive components and previously defined subsystems.



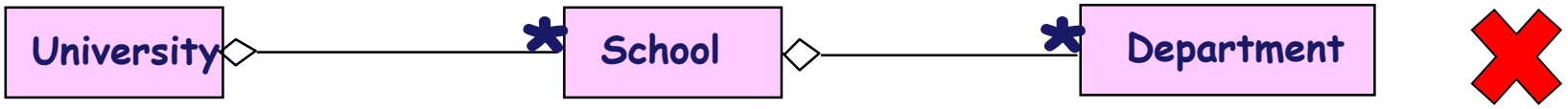
- **Composite is clearly applicable when the construction process will use primitive objects, as well reuse subsystems defined earlier.**

Use Composite Pattern to only model dynamic aggregates

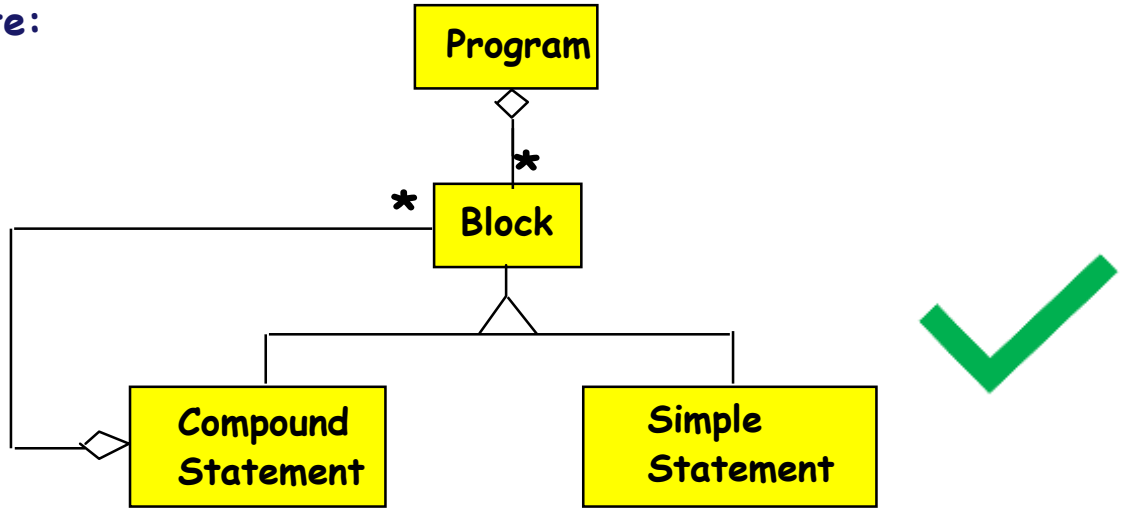
Fixed Structure:



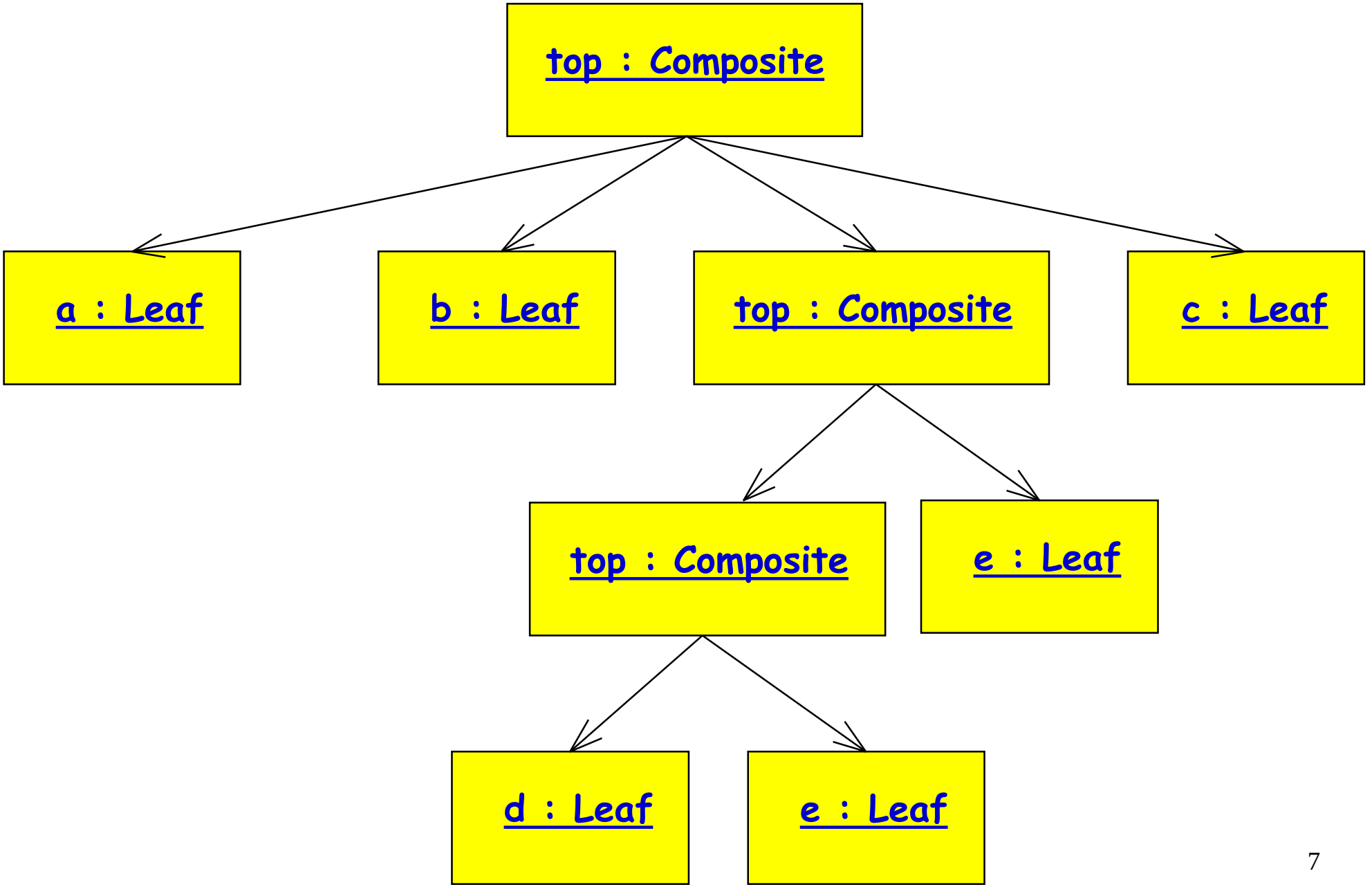
Organization Chart (variable aggregate):



Dynamic (recursive) aggregate:



Composite: Object Diagram



```
// "Component"
  abstract class Component
  {protected string name;

    // Constructor
    public Component(string name)
            {this.name = name;}

    public abstract void Add(Component c);
    public abstract void Remove(Component c);
    public abstract void Display();
  }
```

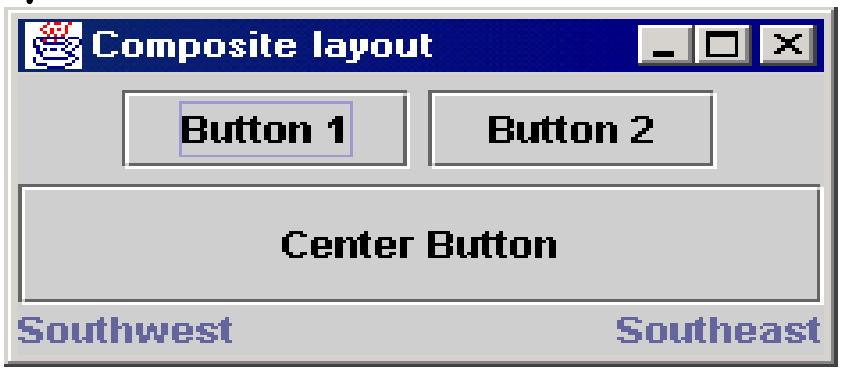
```
// "Composite"
class Composite extends Component
{private ArrayList children = new ArrayList();

  // Constructor
  public Composite(string name)  {super(name); }

  public void Add(Component component)
  {children.Add(component);}

  public void Remove(Component component)
  {children.Remove(component);}

  public void Display(int depth)   //override
  {System.out.println(new String('-', depth) + name);

    // Recursively display child nodes
   for(Component component : children)
        component.Display()
  }
}
```

```
// "Leaf"
 class Leaf extends Component
 {// Constructor
   public Leaf(string name) {super(name);}

   public void Add(Component c) // override
   {System.out.println("Cannot add to a leaf");}

   public void Remove(Component c) //override
   {System.out.println("Cannot remove from a leaf");}

   public void Display(int depth) //override
   {System.out.println(new String('-', depth) + name);}
 }
}
```

```
class MainApp  {
  static void Main(){
    // Create a tree structure
    Composite root = new Composite("root");
    root.Add(new Leaf("Leaf A"));
    root.Add(new Leaf("Leaf B"));

    Composite comp = new Composite("Composite X");
    comp.Add(new Leaf("Leaf XA"));
    comp.Add(new Leaf("Leaf XB"));

    root.Add(comp);
    root.Add(new Leaf("Leaf C"));

    // Add and remove a leaf
    Leaf leaf = new Leaf("Leaf D");
    root.Add(leaf);
    root.Remove(leaf);

    // Recursively display tree
    root.Display(); }
}
```

```
-root
   ---Leaf A
   ---Leaf B
   ---Composite X
       -----Leaf XA
       -----Leaf XB
   ---Leaf C
```

```
Container north = new JPanel(new FlowLayout());
north.add(new JButton("Button 1"));
north.add(new JButton("Button 2"));
```

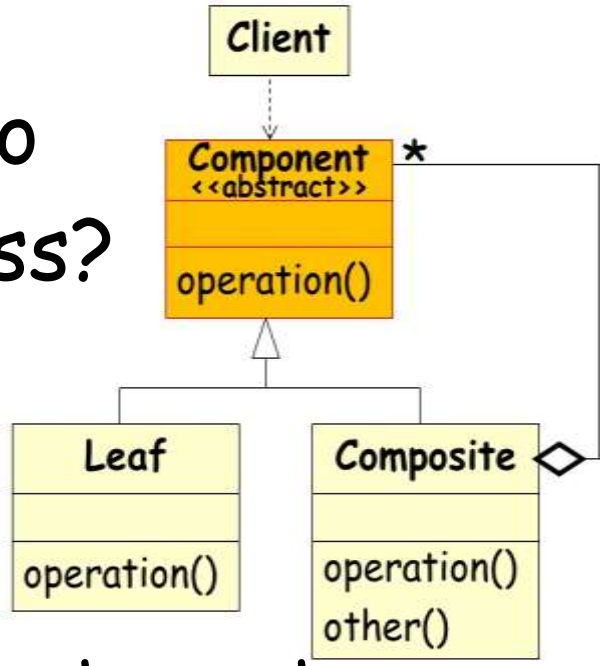


Composite example: Jpanel

```
Container south = new JPanel(new BorderLayout());
south.add(new JLabel("Southwest"), BorderLayout.WEST);
south.add(new JLabel("Southeast"), BorderLayout.EAST);

// overall panel contains the smaller panels (composite)
JPanel overall = new JPanel(new BorderLayout());
overall.add(north, BorderLayout.NORTH);
overall.add(new JButton("Center Button"),
   BorderLayout.CENTER);
overall.add(south, BorderLayout.SOUTH);
frame.add(overall);
```

JPanel, a part of Java Swing package. It is a container that can store a group of components. The main task of JPanel is to organize components

Some Insights



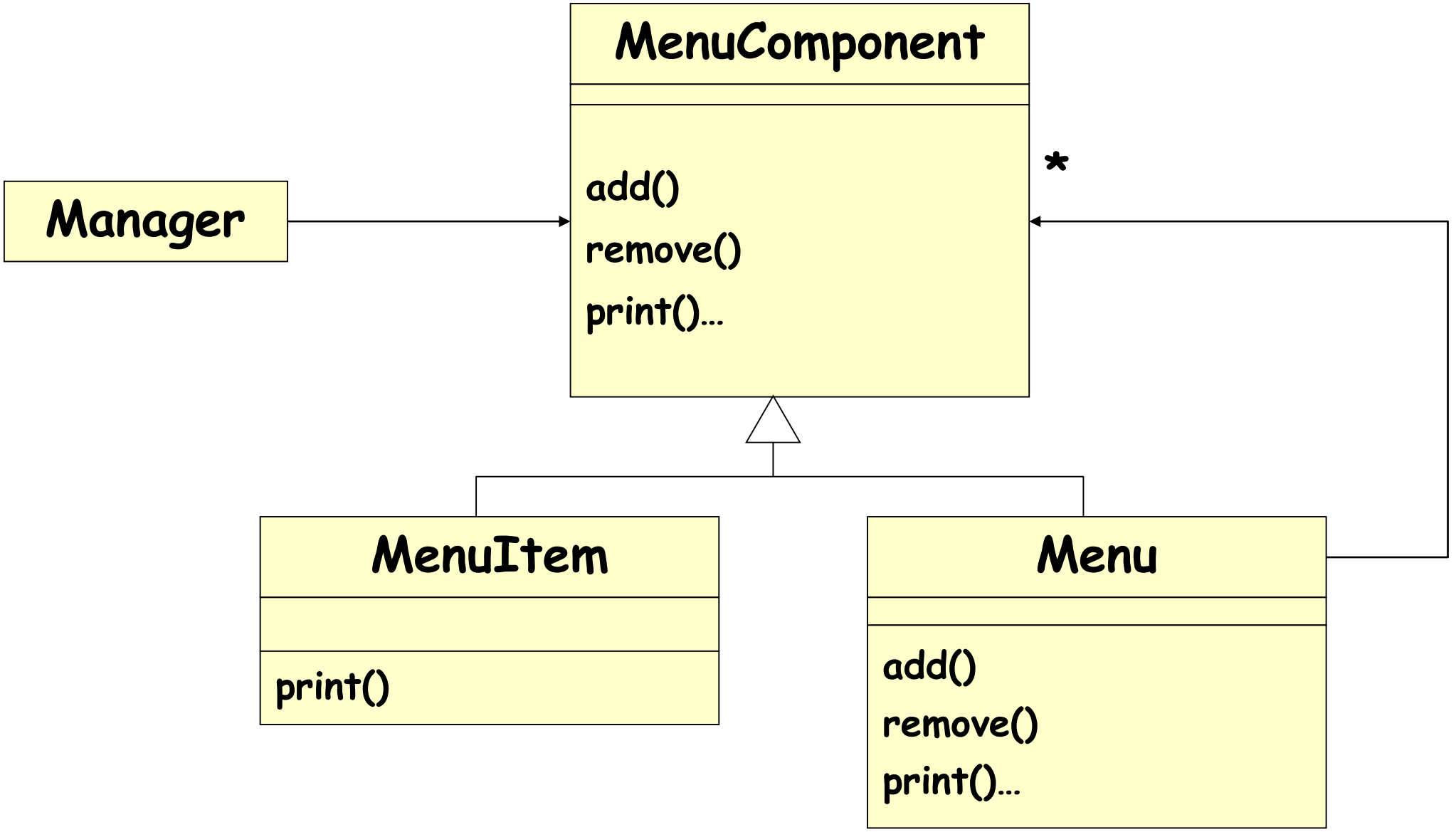
- Why do you declare the methods to handle children in the abstract class?
  - Only the composite class has any use for them?
  - Is it not poor programming practice to have these methods inherited by primitive classes, which have no use for them?
- **There is a tradeoff here between safety and elegance…**

Elegance Issue

- If the child management methods are moved from the abstract class to the composite one,
  - The client can never call these methods on primitive objects, improving elegance.

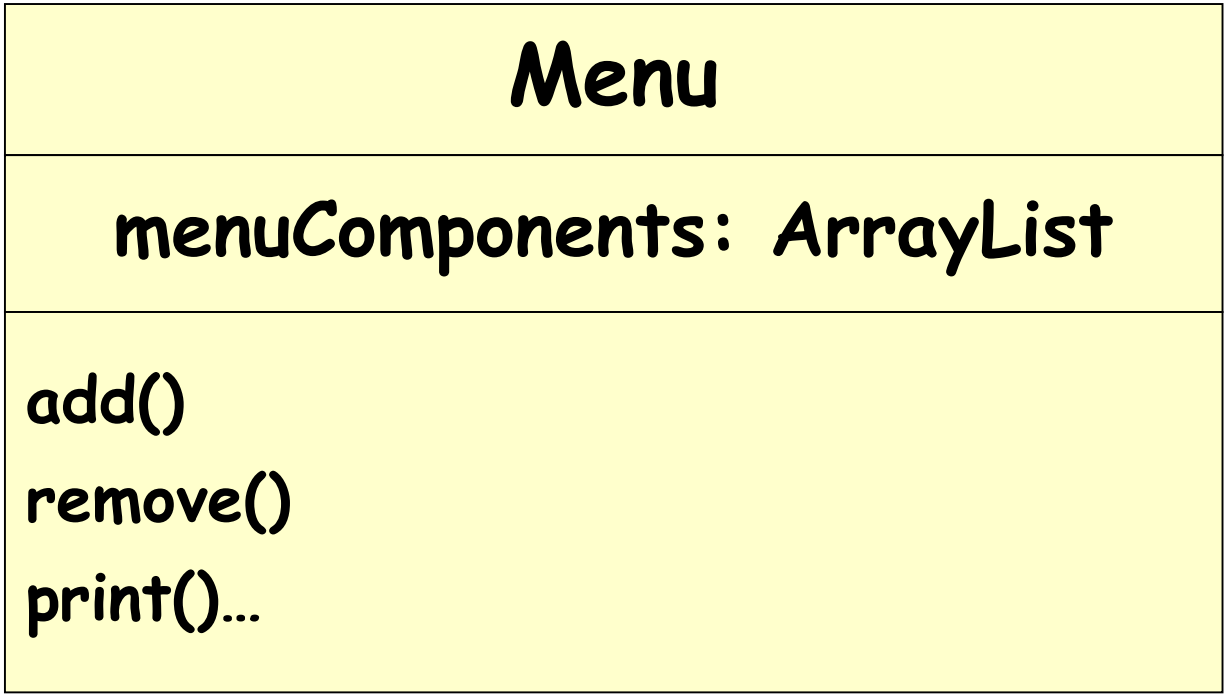
- However, this gives primitive and composite objects different interfaces:
  - Which is what the design pattern is to avoid

Composite Pattern: Example 1



How to print a menu?

Example 1



1. Menu to MenuComponents association implemented with an ArrayList data type.

2. Let us examine what is the implementation of print() in Menu and in MenuItem…

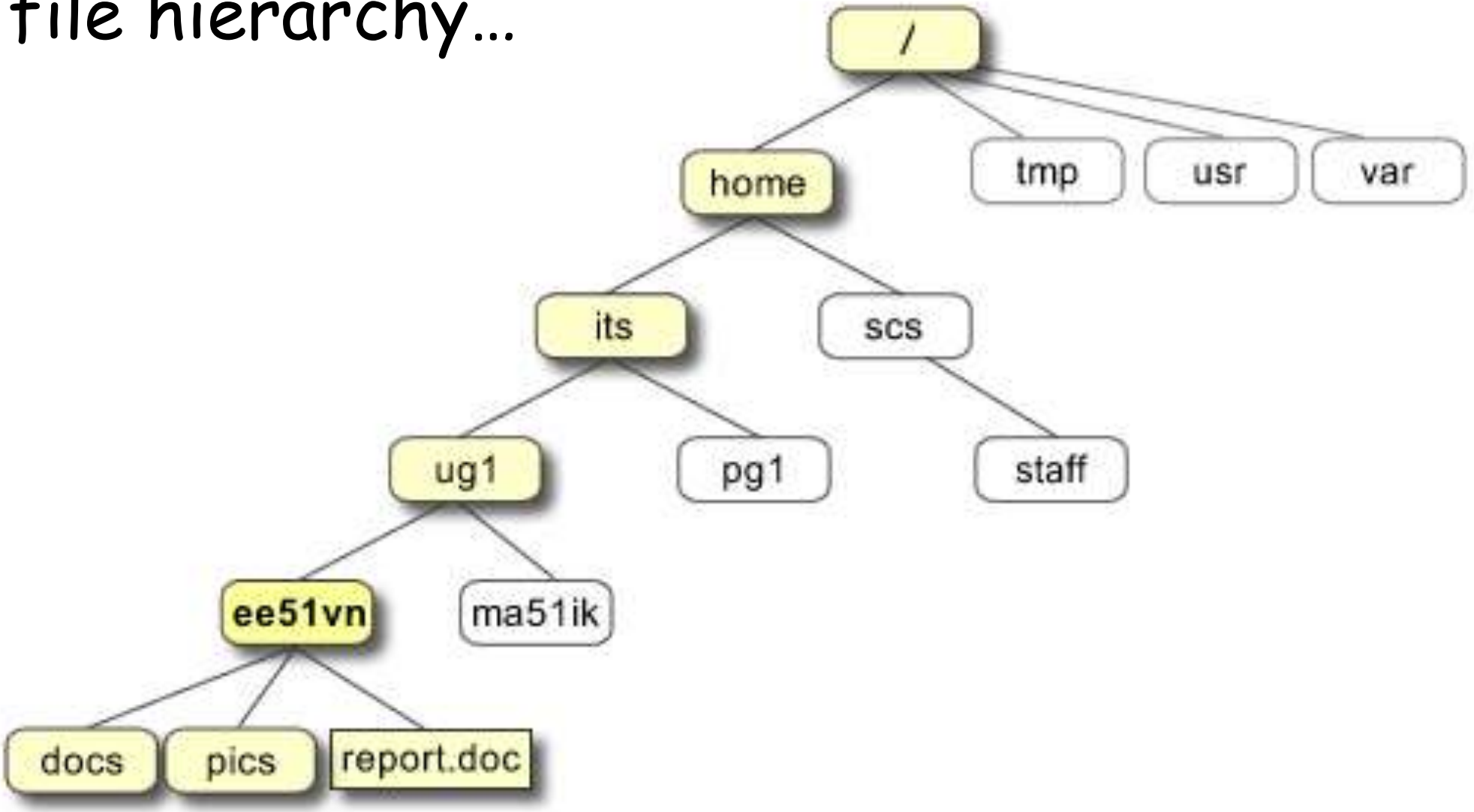
Example 1

```
Class Menu implements MenuComponent{… … …
public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("----------------");
        Iterator iterator= menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponents =
                (MenuComponent)iterator.next();
            menuComponents.print();
        }
    }
```

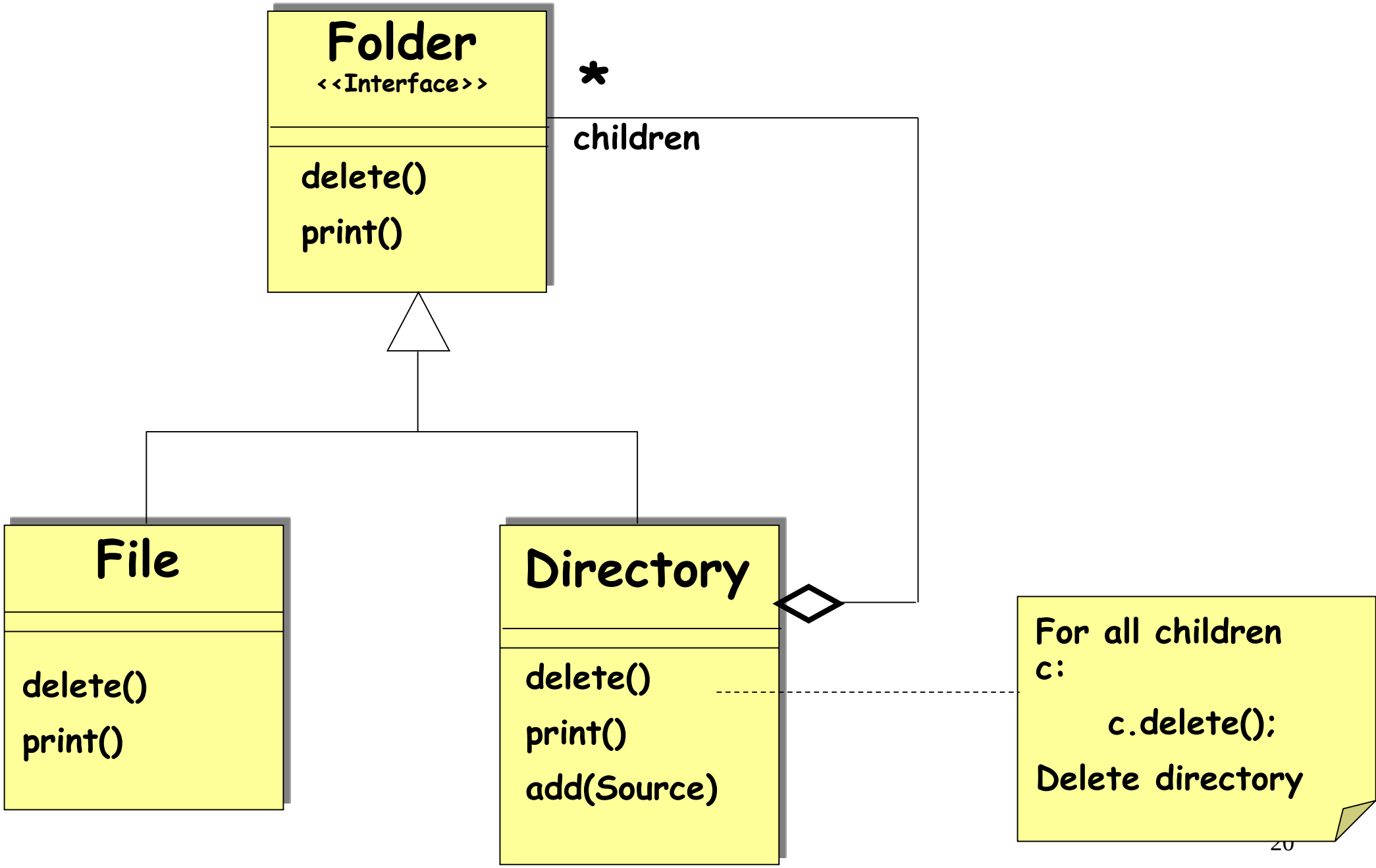
```
class MenuItem implements MenuComponent{ … …
public void print() {
    System.out.print("  " + getName());
    if (isVegetarian())  System.out.print("(v)");
    System.out.println(", " + getPrice());
    System.out.println("--" + getDescription());
  }
```

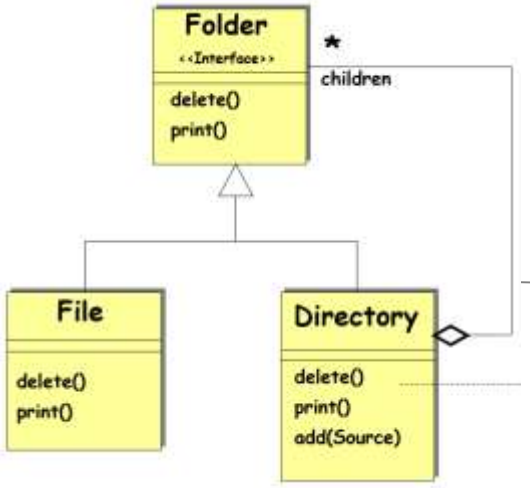
Exercise 1

- Design a class structure to model a Unix file hierarchy…



Exercise 1: Solution





Component A

setName();

getName();

getAllFiles(List theList);

getContents(List theList);A

*

Now Suppose file types are text, image, Video…

How to accommodate?

Directory

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

TextFile

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

ImageFile

setName();

getName();

getAllFiles(List theList);

getContents(List theList);

VideoFile

setName();

getName();

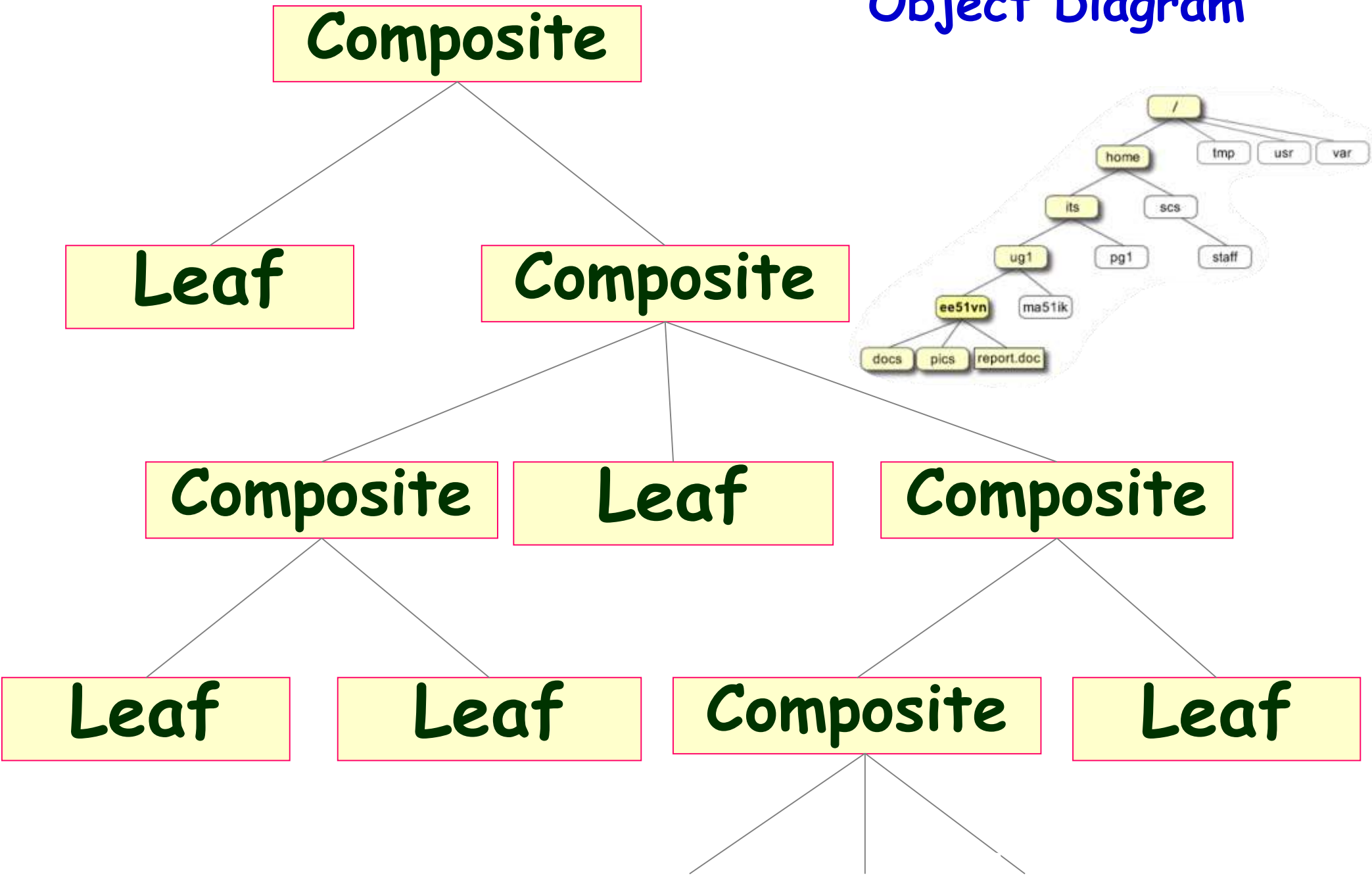
getAllFiles(List theList);

getContents(List theList);

This is the Composite

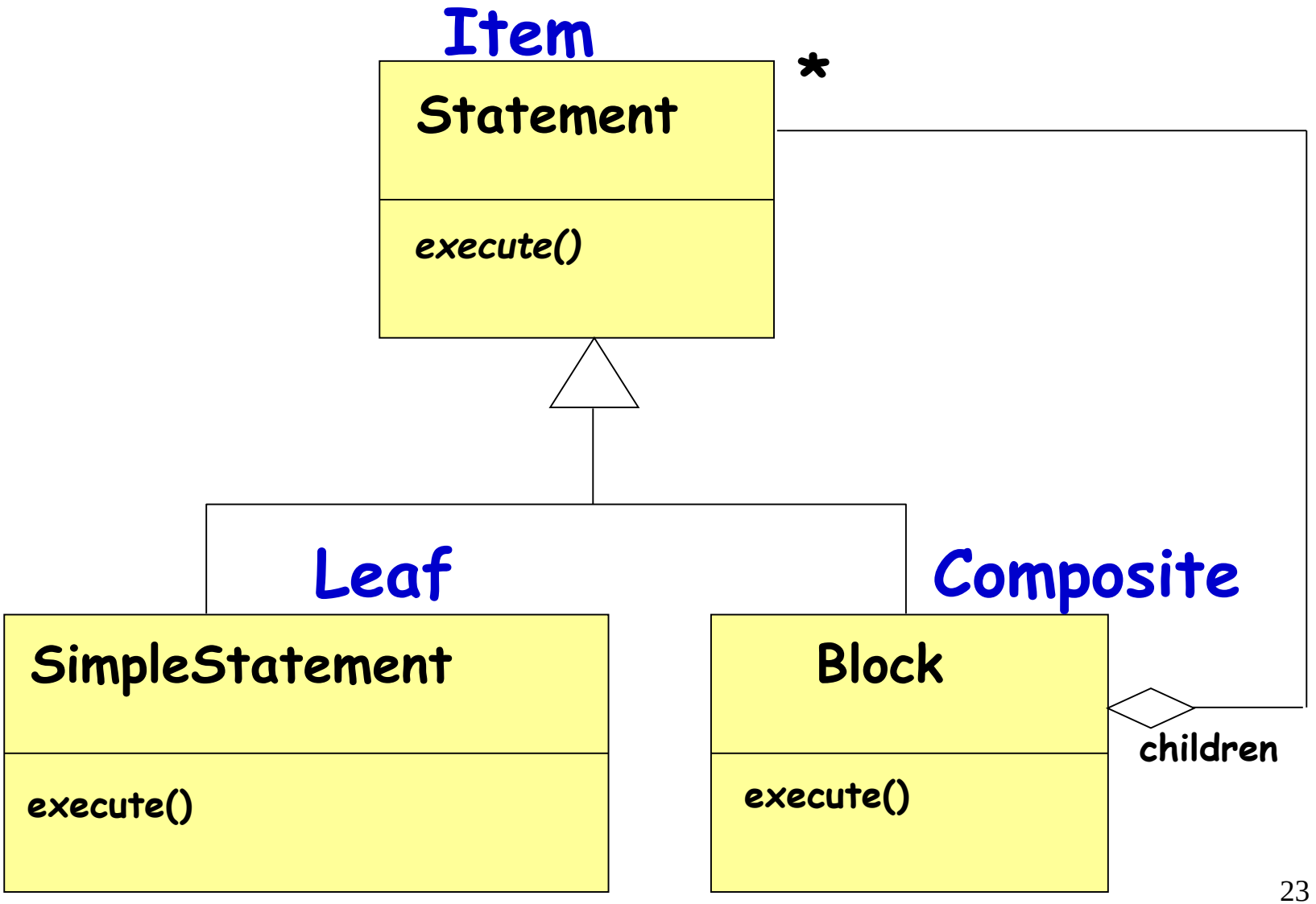
These objects are the leaves

Object Diagram



Exercise 3: Programming

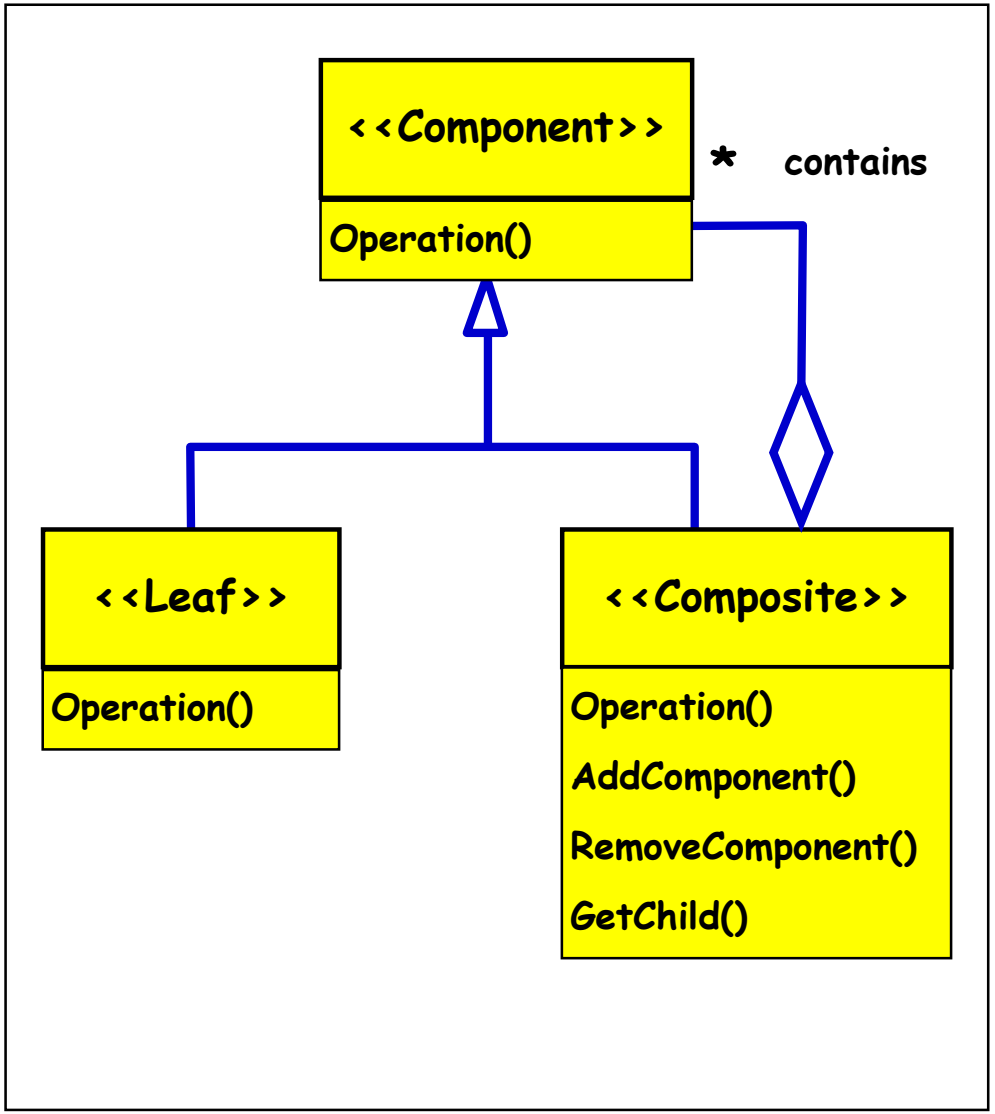
A program block can contain simple statements or other program blocks…



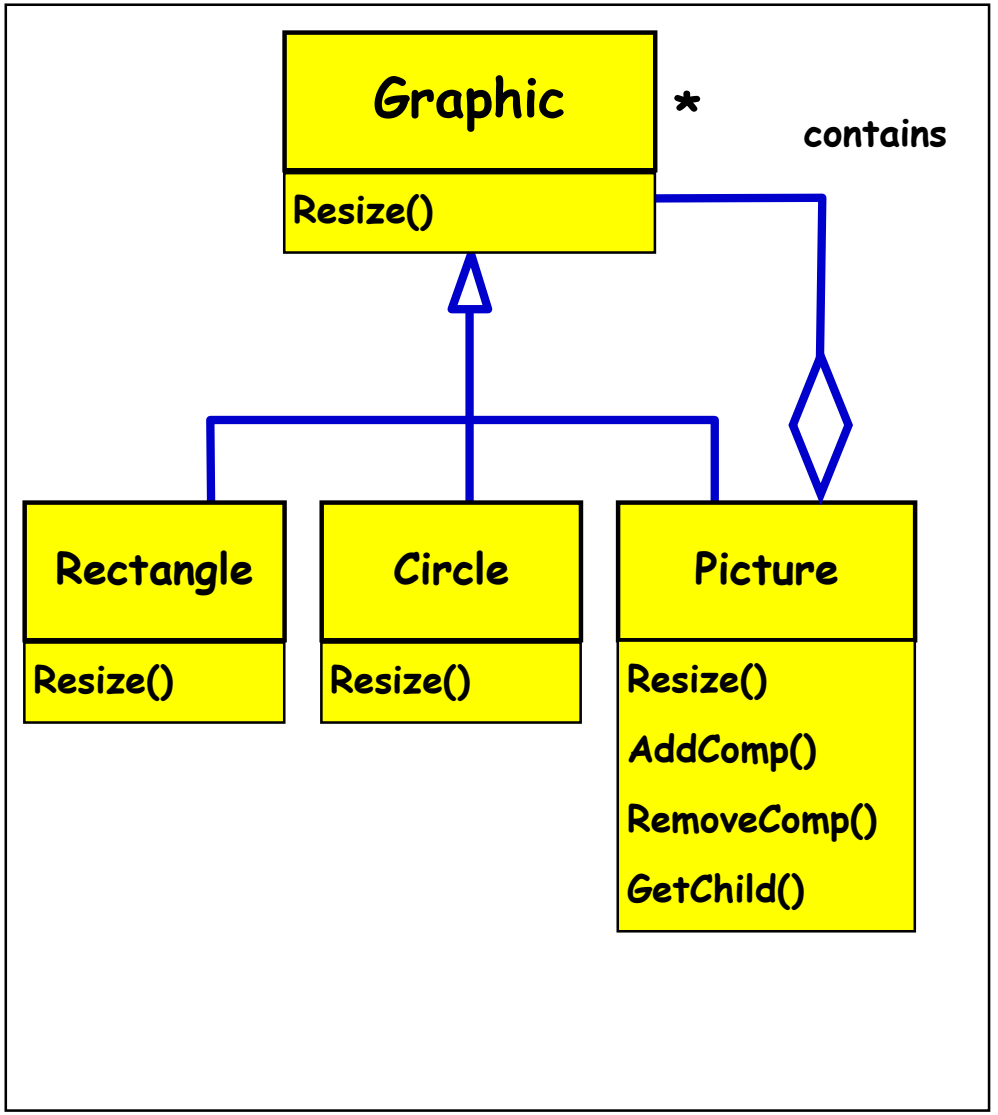
Exercise 4

- Develop class design of components to be handled by a Graphics editor:
  - Should let users group simple components into larger components.
    - Which in turn can be grouped to form still larger components.
    - Larger components should behave similarly w.r.t. select, copy, paste, move,delete, resize, ...

Composite Pattern: Solution

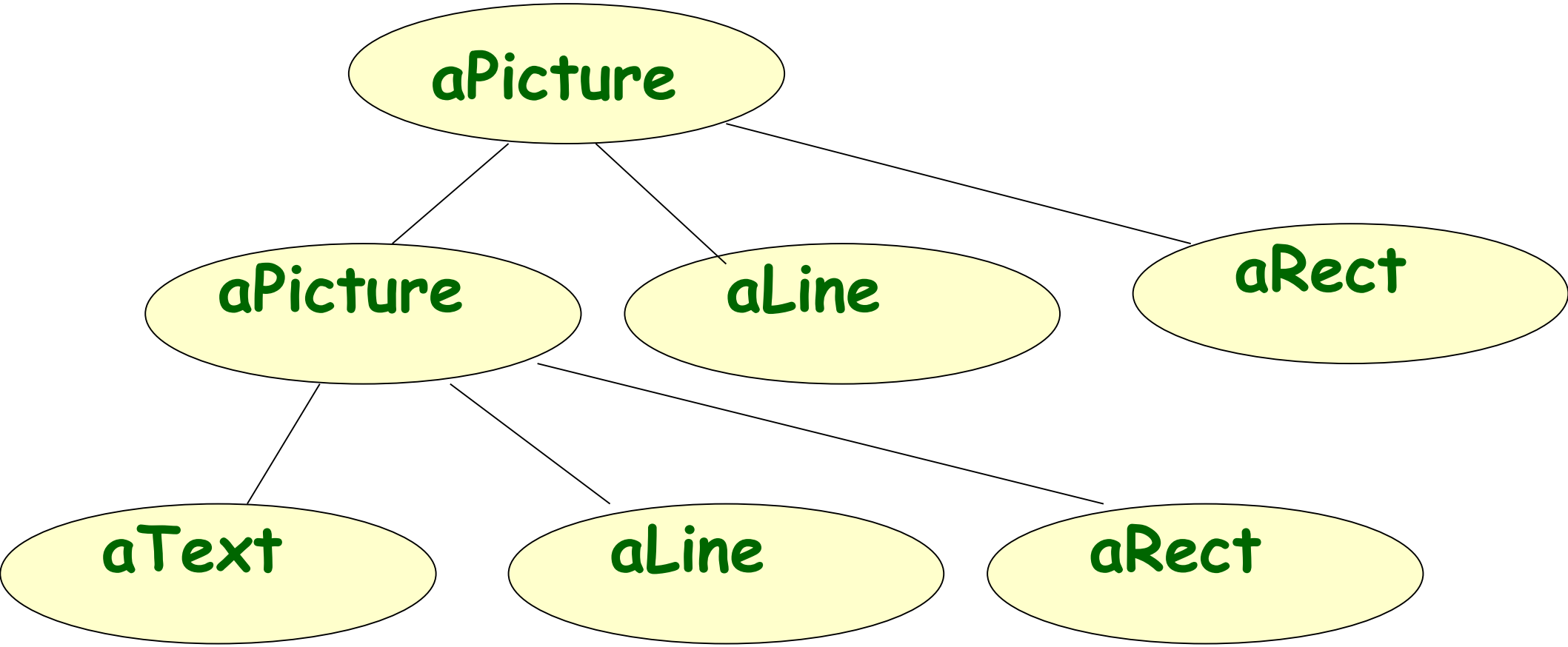


General Idea



Applied to the drawing example

Object Structure?



Solution Note

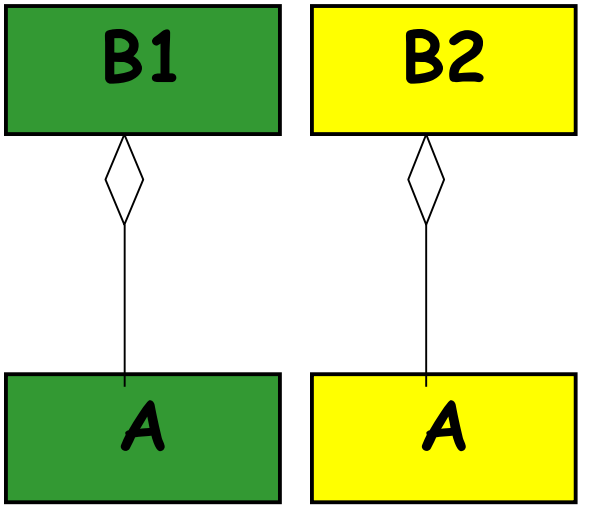
- The key to the Composite pattern:
  - An abstract class that represents both primitives and their containers.
  - The abstract class Graphic declares operations like Copy, Move, Delete, resize, etc. that are specific to graphical objects.
  - It also declares operations that only the composite objects need, such as
    - Operations for accessing and managing its children, like Add, UnGroup.

Alternatives…

- Component does not know what it is part of:
  - Component can be in many composites
  - Component can be accessed only through composite
- Component knows what it is a part of
  - Component can be in only one composite
  - Component can be accessed directly

Composite: Issues with Part-of

- Rules when component knows its single composite
  - A is a part of B if and only if B is the composite of A
  - However, duplicating information is dangerous!



- **Problem: How to ensure that references of components to composite and composite to components are consistent?**

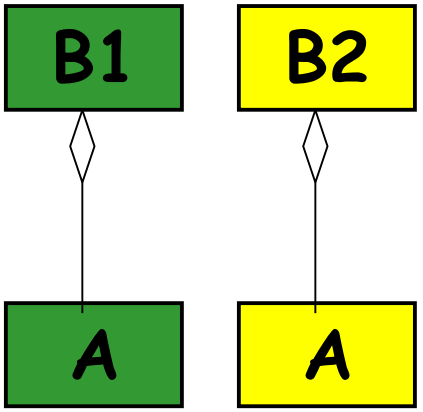
Composite Implementation: Alternatives

- Component does not know what it is a part of:
  - Component can be in many composites
  - Component can be accessed only through composite
- Component knows what it is a part of
  - Component can be in only one composite
  - Component can be accessed directly

Composite: Issues with Part-of

- Rules when component knows its single composite:
  - *A* is a part of *B* if and only if *B* is the composite of *A*
  - However, duplicating information is dangerous!

- **Problem: How to ensure that references of components to composite and composite to components are consistent?**



Ensuring consistency

- The public operations on components and composites are:
  - Composite can enumerate components
  - Component can name its container
  - Add/remove a component to/from the composite
    - **The operation to add a component to a composite updates the container of the component**

addChild() in Composite

```
public void addChild(Component child) {
    ensureCapacity();
    childArray.add(child);
    child.setParent(this);
}
```

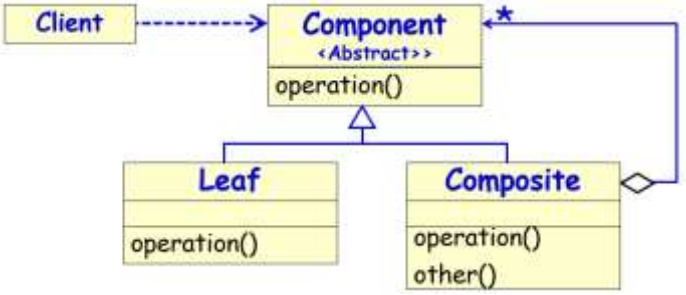
Exercise 4: Composing GUI

Q: How can we add any widget to another, for example panels to an applet?

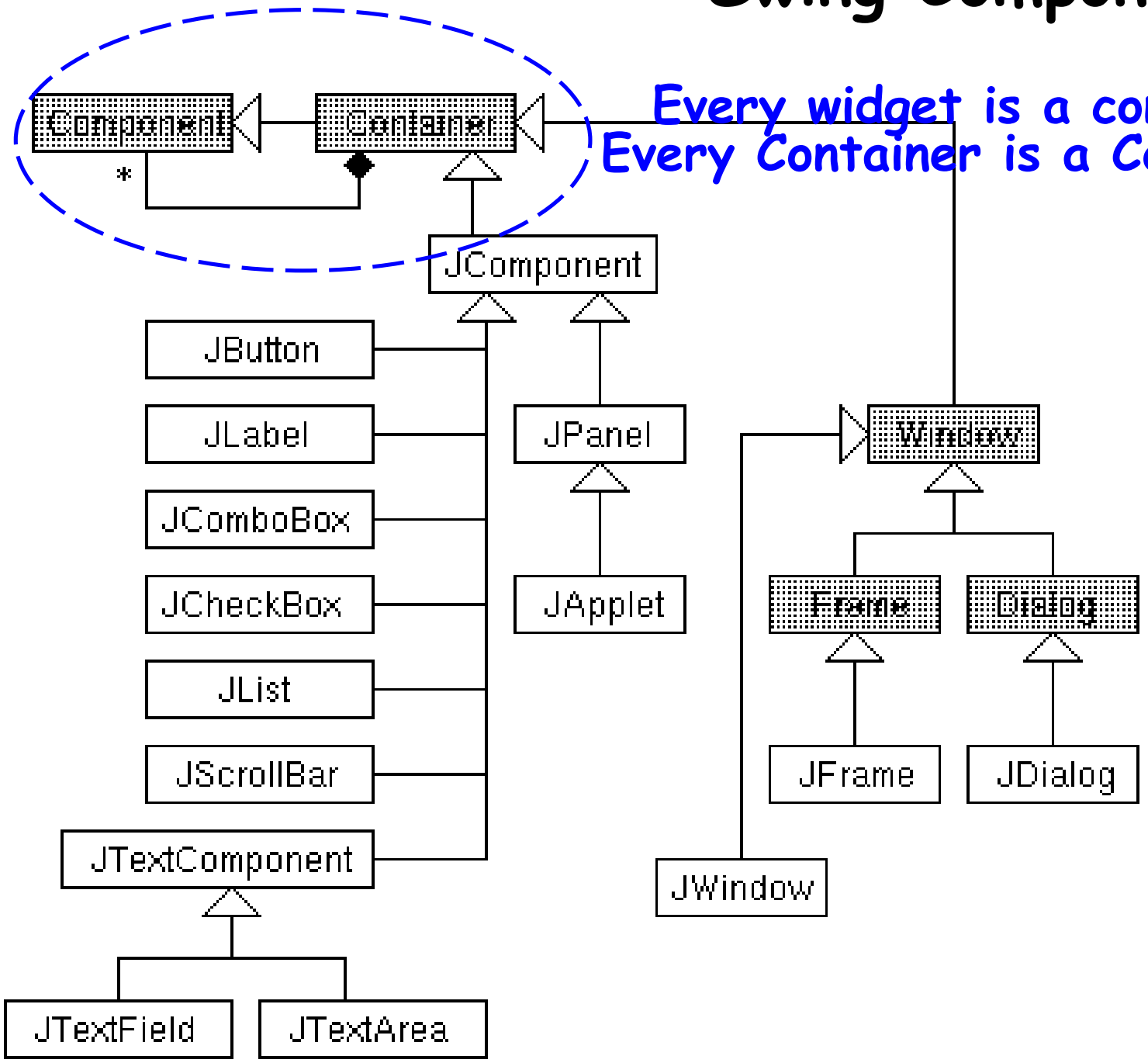
Solution: Composing GUI

```
public class MyApplet extends java.applet.Applet {
    public MyApplet() {
        add(new Label("My label"));
        add(new Button("My button"));
        Panel myPanel = new Panel();
        myPanel.add(new Label("Sublabel"));
        myPanel.add(new Button("Subbutton"));
        add(myPanel);
    }
}
```

AWT Components



Swing Components



Every widget is a container!
Every Container is a Component!

Composite: Consequences

- **Benefits:**
  - Makes it easy to add new kinds of components
  - Makes clients simpler, since they do not have to know if they are dealing with a leaf or a composite component
- **Liabilities:**
  - Makes it harder to restrict the type of components of a composite

Adapter Pattern

Adapter --a Wrapper Pattern

Intent

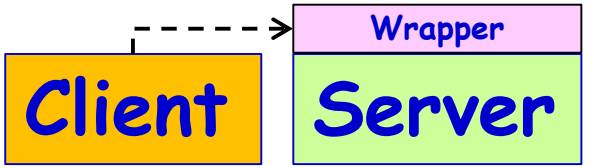
- **Convert the interface of a class to the interface expected by the users of the class.**
- Allows classes to work together even when they have incompatible interfaces.

Example (non-software)

- You went to U.S.-- found
- Had an Indian electrical appliance…
- How can you use it in U.S.?
- **Use Adapters!**

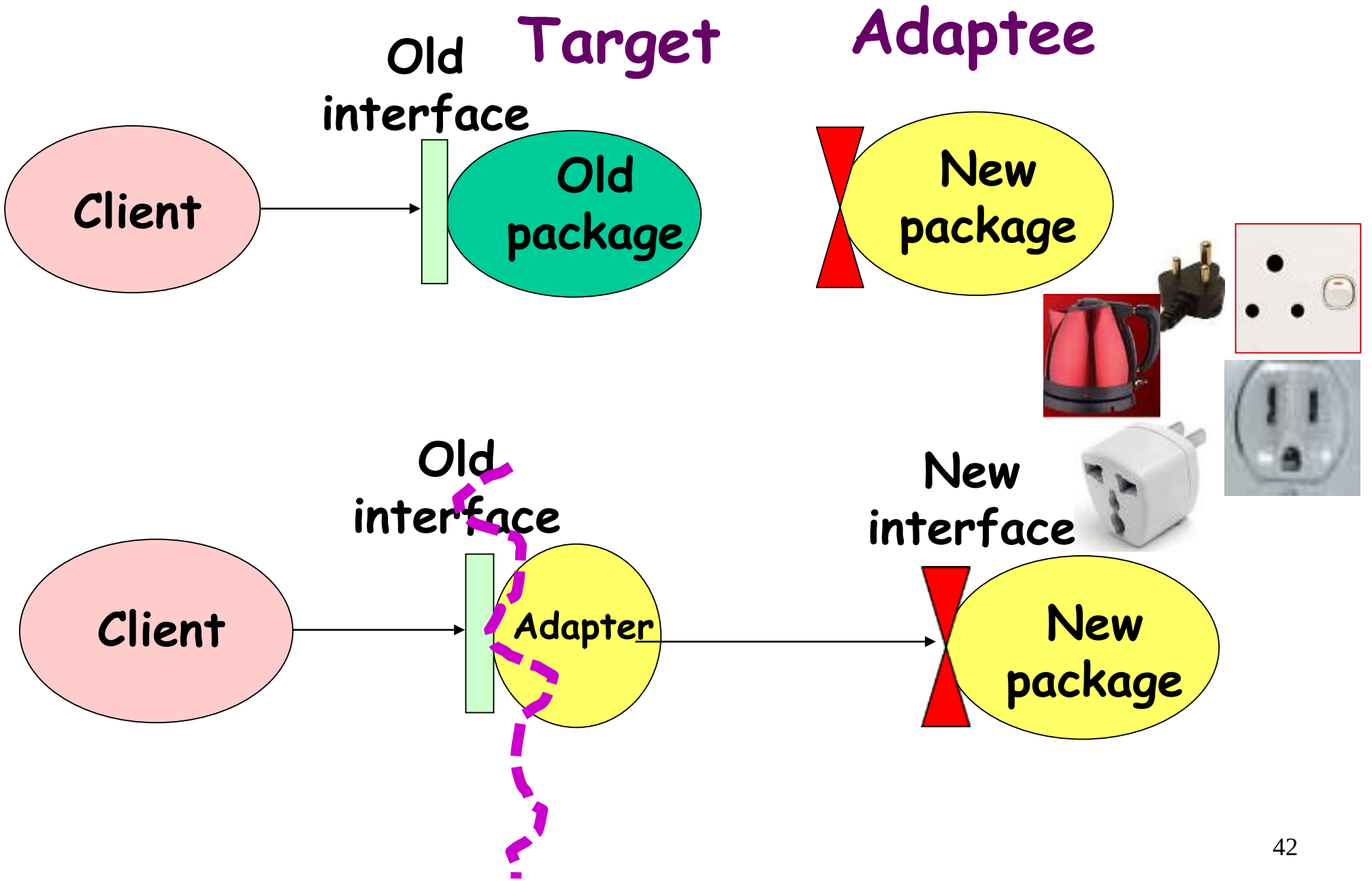
Also universal adapters?

Adapter Pattern



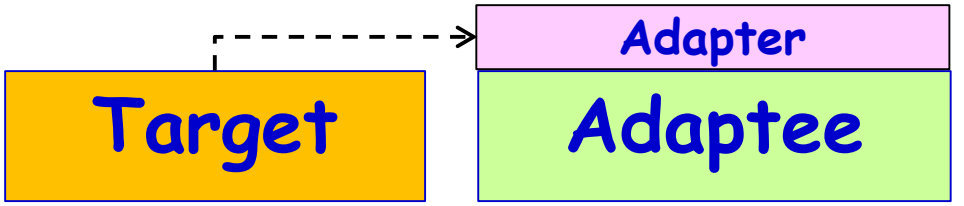
- It is a **wrapper pattern**
- **"Convert the interface of a class into one that a client expects."**
  - Lets two classes work together --- that couldn't otherwise --- because of incompatible interfaces
  - Used to provide a new interface to existing legacy components.
- Two main adapter variants:
  - **Class adapter:**
    - Uses interface implementation and inheritance
  - **Object adapter:**
    - Uses delegation to adapt one interface to another
- **Object adapters are much more common.**

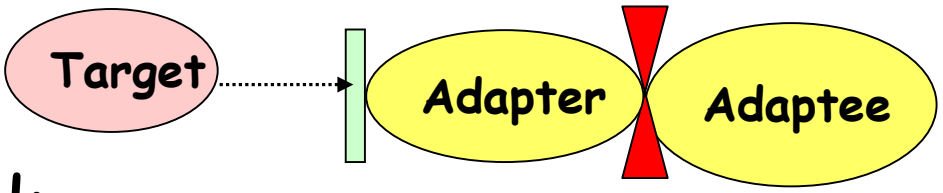
Essential Idea Behind Adapter Pattern



Adapter Pattern: Basics

- Helps two incompatible types to communicate.
  - A class expects an interface ---but that is not supported by a server class,
  - The adapter acts as a translator between the two types.





- 3 essential classes involved:
  - **Target** – Interface that client uses.
  - **Adapter** - class that wraps the operations of the Adaptee in interface familiar to client.
  - **Adaptee** - class with operations that the client class desires to use.

Lets Get Familiar With The Terminology

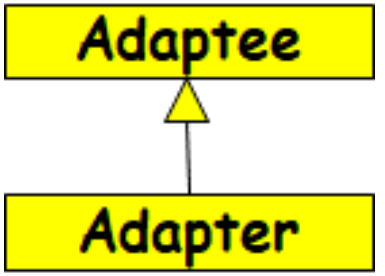
Target

Adapter

Adaptee

Class and Object Adapters

An adaptee may be given a new interface by an adapter in two ways:

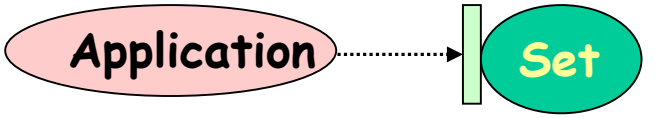


- **Inheritance**
  - The adapter sub-classes the adaptee;
  - This is the **Class Adapter pattern**

- **Delegation**
  - The adapter holds a reference to the adaptee and delegates work to it;
  - This is the **Object Adapter pattern**

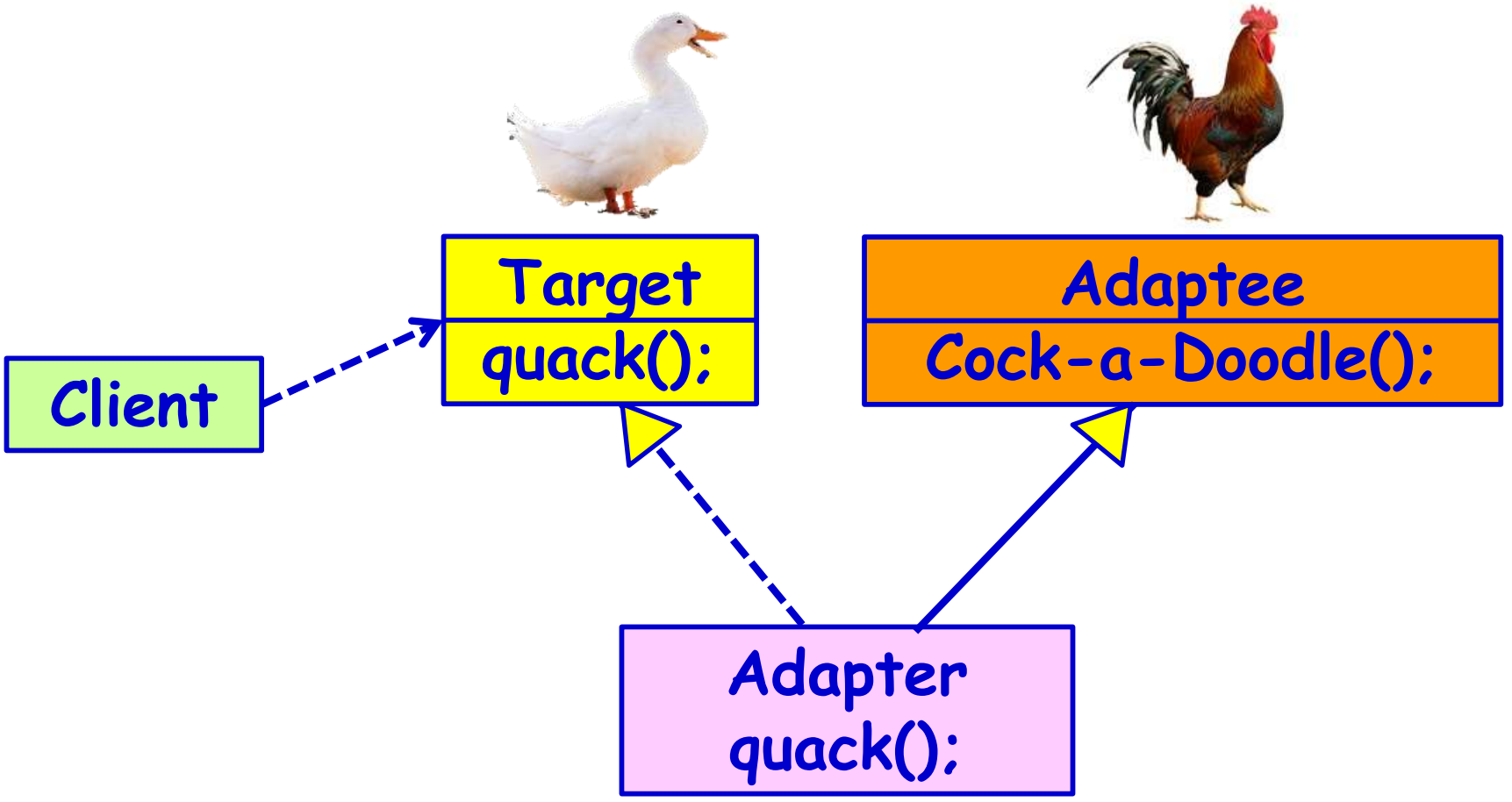
Example 1 – Sets

- There are many ways to implement a set
- Assume:
  - Your existing set implementation has poor performance.

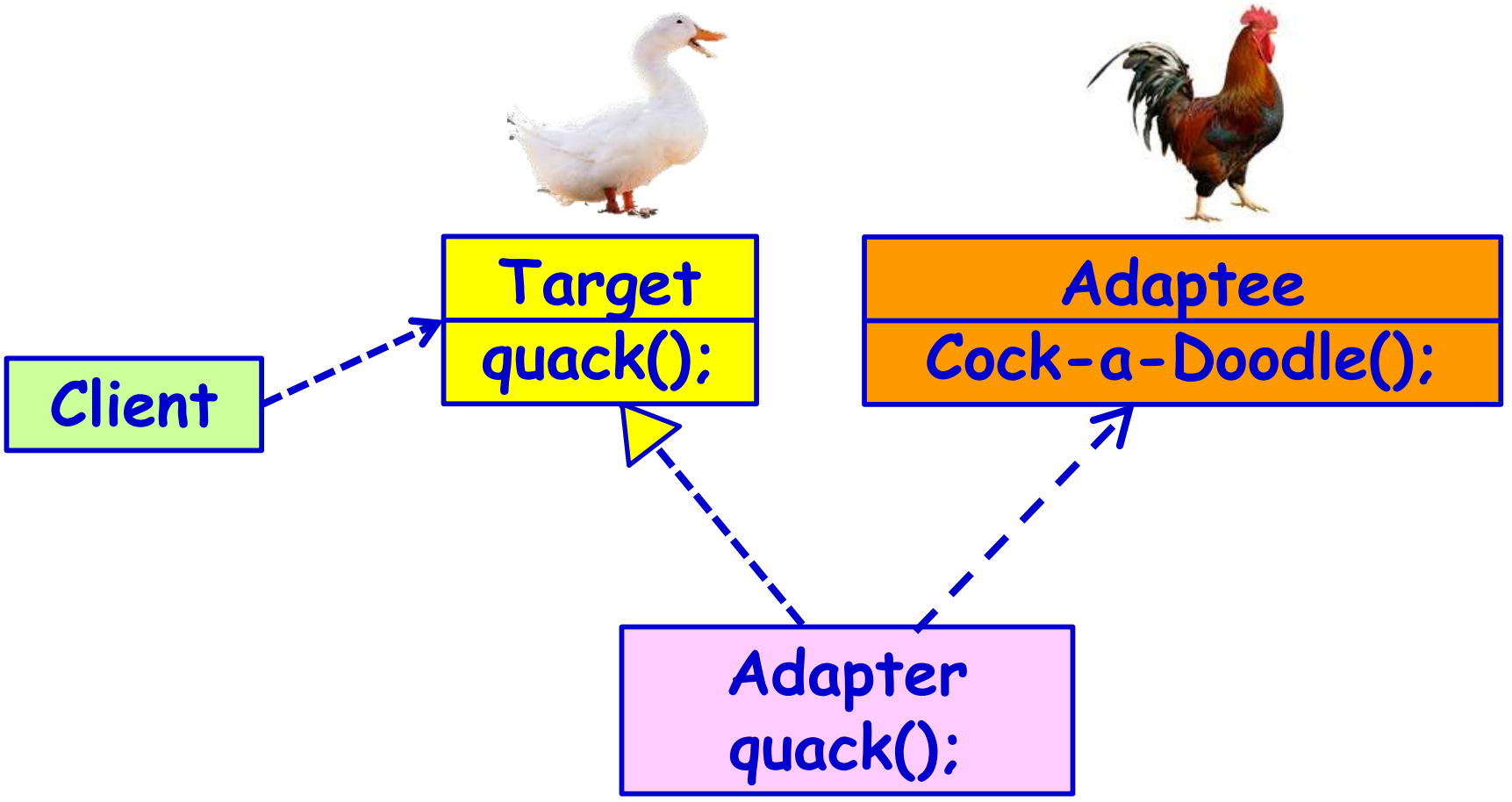


- You got hold of a more efficient set class,
  - BUT: The new set has a different interface.
  - Do not want to change voluminous client code
- **Solution: Design a class setAdapter:**
  - Same interface as the existing set..
  - Simply translates to the new set's interface.

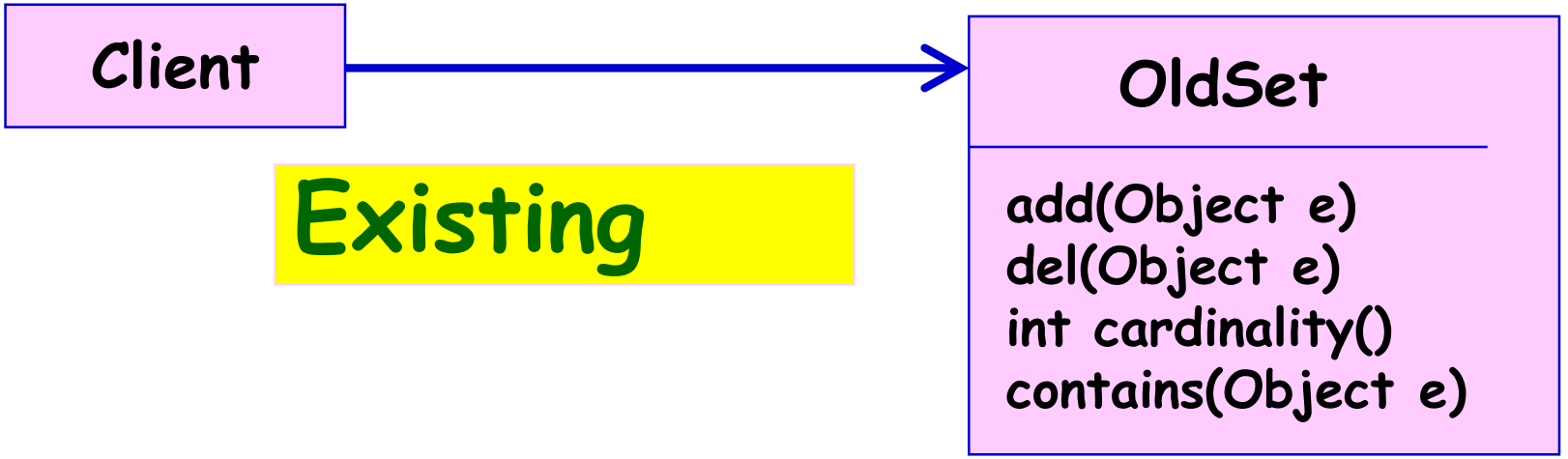
Class Adapter: Main Idea



Object Adapter: Main Idea



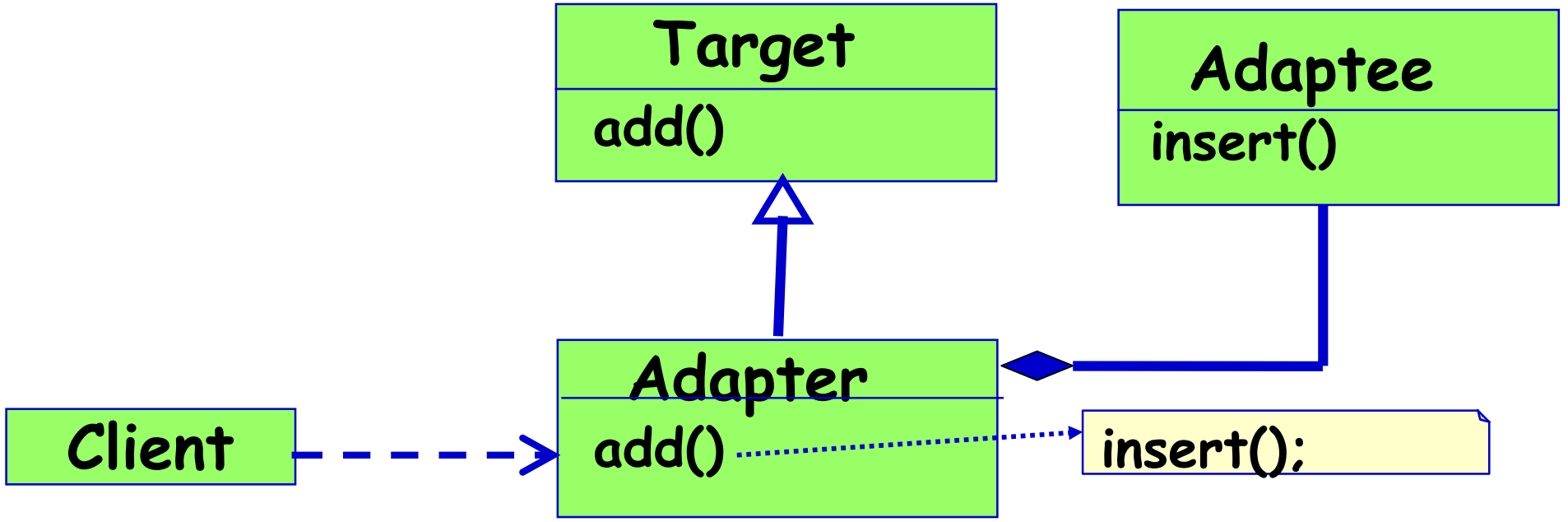
Example: Solution



But, do not want to change Client code…

Object Adapter: main idea -- delegation

- The Adapter internally holds an instance of the Adaptee or a reference to it…
- Uses it to call Adaptee operations from within the operations required by the Target.



Object Adapter - Code

Client Code:

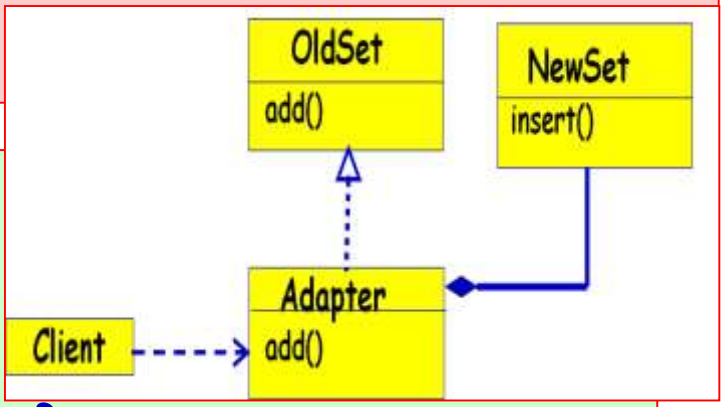
```
NewSet a =new NewSet(); OldSet t = new Adapter(a);
  public void test() {  t.add();  }
```

Target Code:

```
interface OldSet {
 public void  add(){}
}
```

Adaptee Code:

```
class NewSet {
  public void insert(){}
}
```

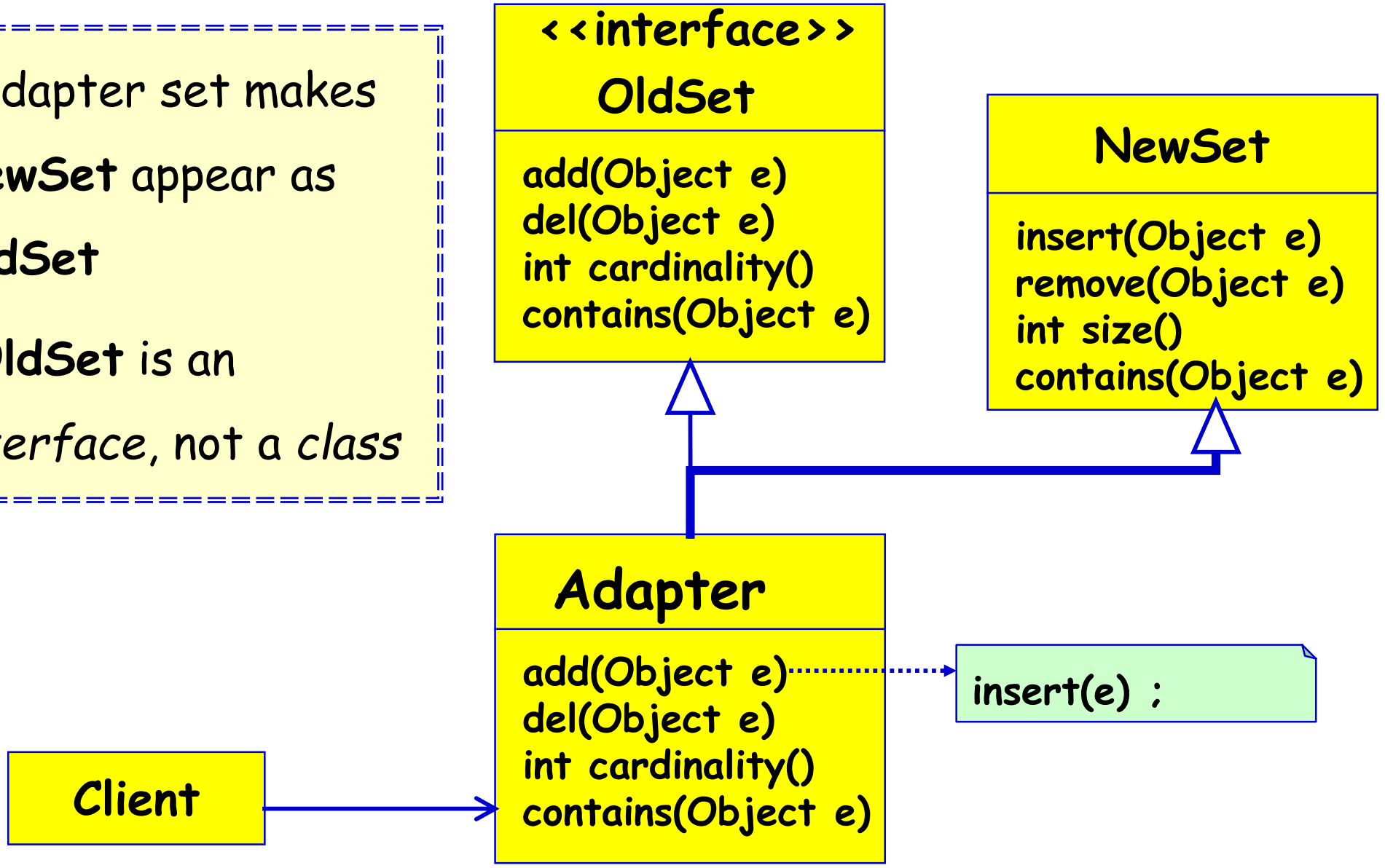


Adapter Code:

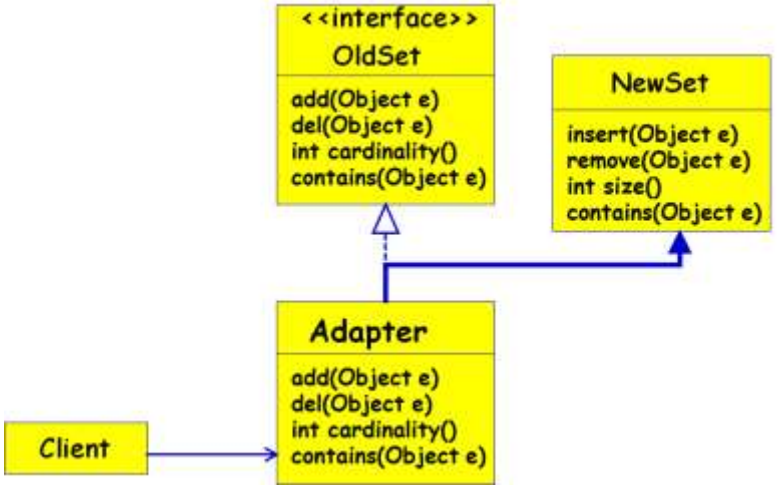
```
class Adapter implements OldSet {
  private NewSet nset;
  public Adapter(NewSet a) { nset = a;}
  public void add() { nset.insert();}
}
```

Class Adaptation

- Adapter set makes **NewSet** appear as **OldSet**
- **OldSet** is an *interface*, not a *class*



Class Adapter - Code




```
Client Code:
    OldSet t = new Adapter();
     public void test() {  t.add();  }
```

```
Target  Code:
interface OldSet {
 public void  add(){}
}
```

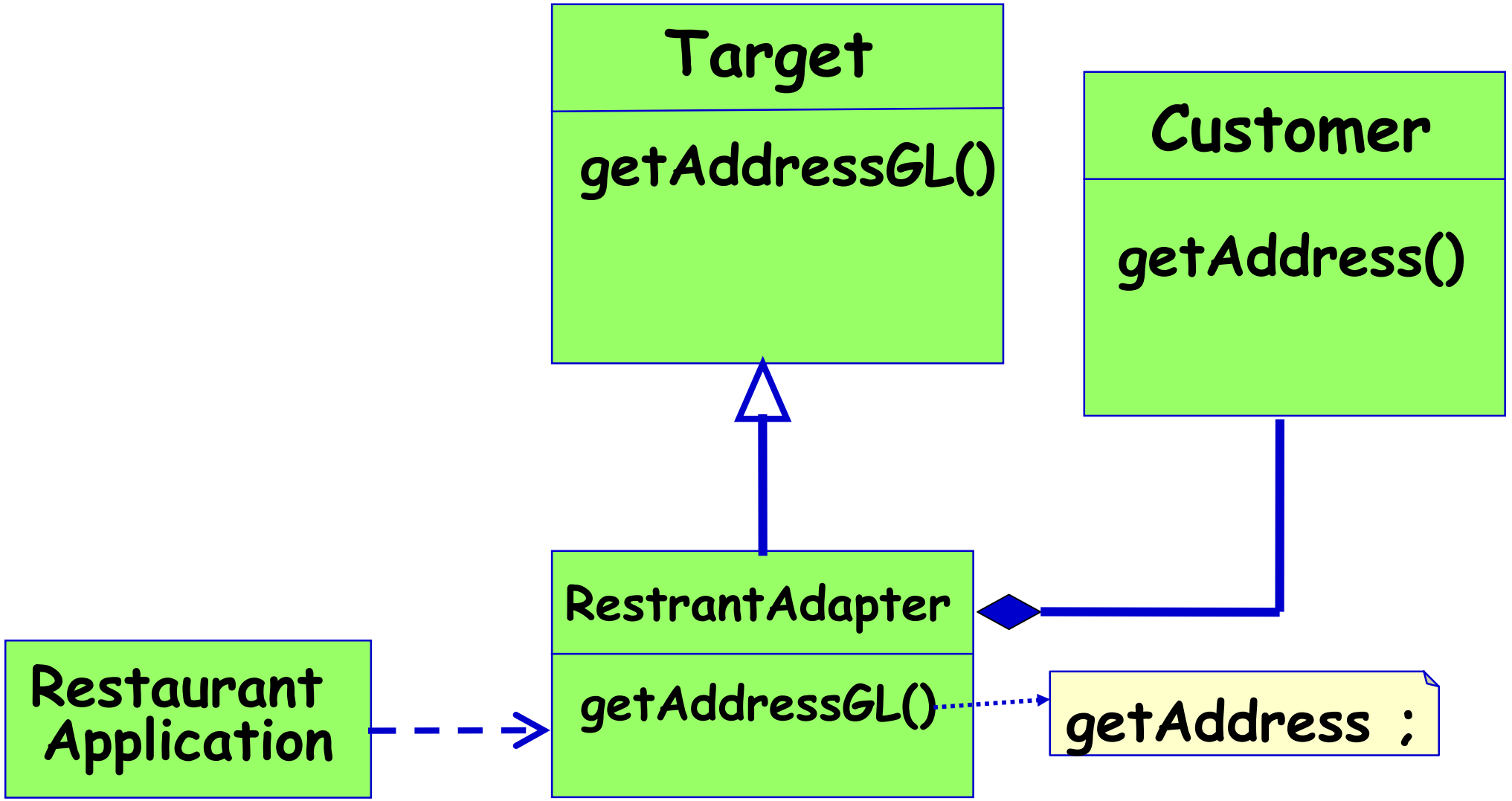
```
Adaptee  Code:
class NewSet {
   public void insert(){}
}
```

```
Adapter  Code:
class Adapter extends NewSet implements OldSet {
    public void add() { insert();}
}
```

Example: Restaurant Application

- A restaurant application uses geo-coded address for customer delivery (Longitude, Latitude).
- However, the customer application uses traditional address (House #, Street # etc).
- Both are working software:
  - You do not want to change either.

Solution: Object Adapter

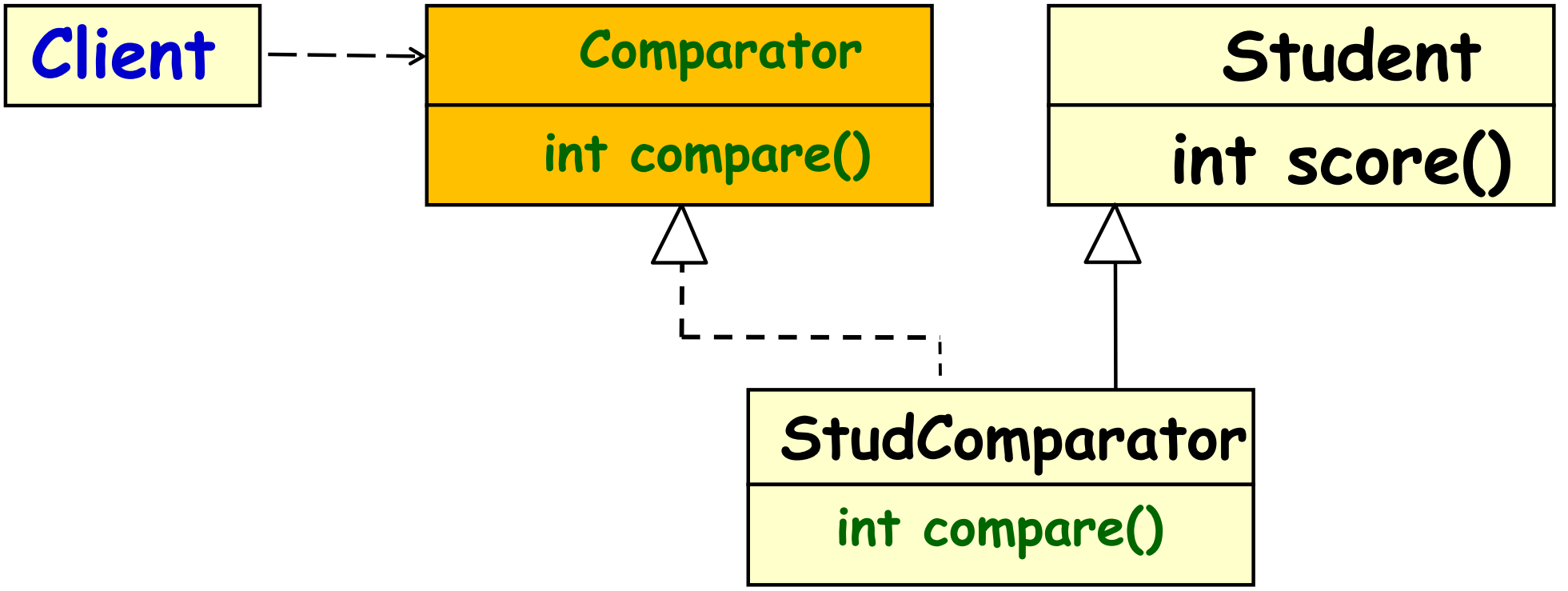


Example Code: Object Adapter

```
class Customer{

 Address getAddress(String
address, String city, String
state, String zip){

    // do some calculation

}

} -Adaptee


interface Target{

void getAdressGL(String lat,
String lng);

}  - Restaurant application uses
geocoded addresses.
```

```
class CustomerAdapter
implements Target{

private Customer customer
= new Customer();

public void
getAddressGL(String lat,
String lng){
     // calculate latitude and longitude
     // return address, city, state, zip
 Adress=customer.getAddress();

 GLAddress=conv(Address);

}

}
```

Adapter design pattern for comparing Objects



This idea has been used in implementing "Comparable" in Java

Java Comparator: Example

```
class Student{
    int rollno;
    String name, address;
}
ArrayList <Student> al=new ArrayList<Student>();

Collections.sort(al,new SortByName());
Collections.sort(al,new SortByRoll());
```

Example: StudentComparator

```
public class SortByName
    implements Comparator<Student> {

      public int compare(Student s1,
          Student s2){
      return s1.name.CompareTo(s2.name);}
  }
```

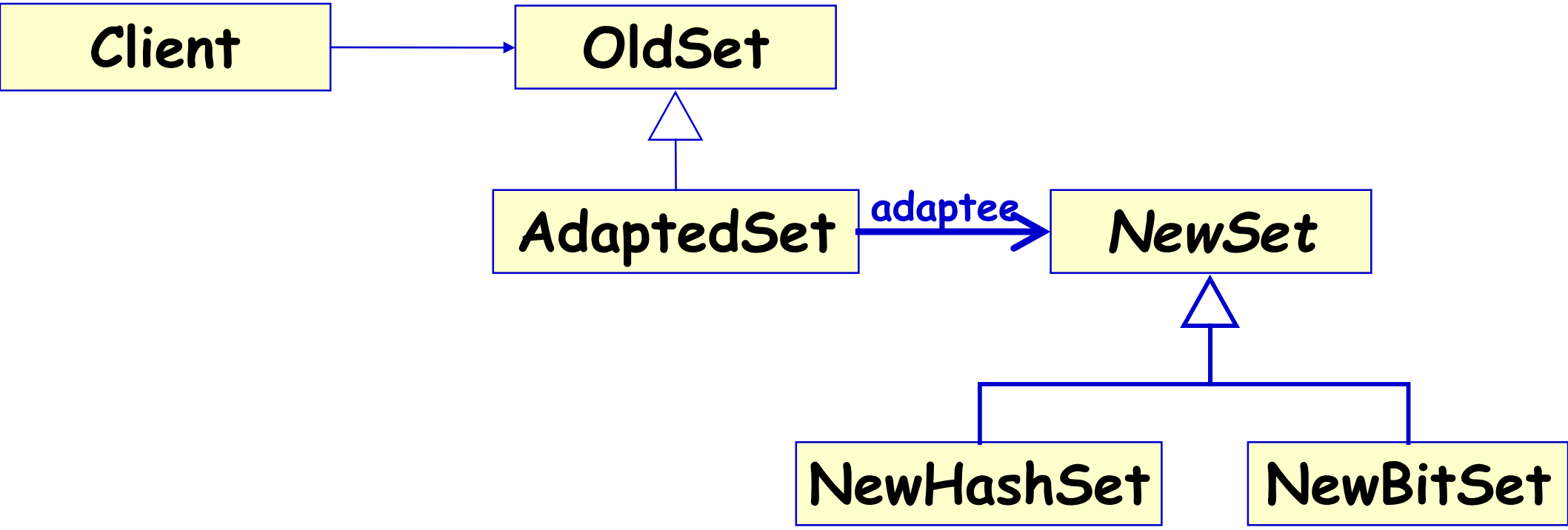
Example: StudentComparator

```
public class SortByRoll
    implements Comparator<Student> {

      public int compare(Student s1,
        Student s2){
          return s1.score() - s2.score();}
    }
```

Variant: Universal Adapter--Adapt Multiple Versions of NewSet

(Object only) Several subclasses to adapt:

- Too expensive to adapt each subclass.
- Create single adapter to superclass interface.
- Configure the **AdaptedSet** with the specific **NewSet** at run-time.



```
public class IPhoneCharger1 {
public void iPhoneCharge(){
 System.out.println("The iPhone is charging
..."); } }
```

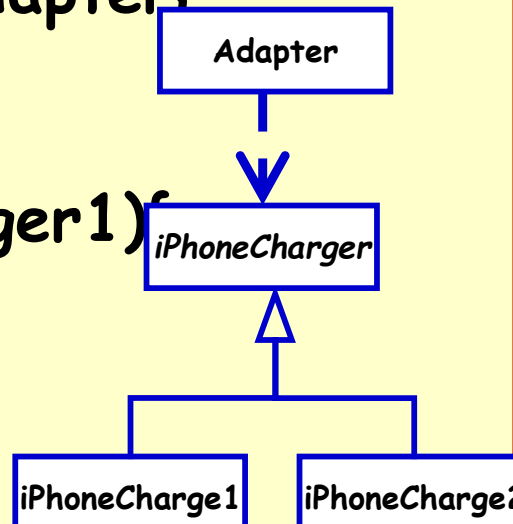
```
public interface ChargAdapter{
public void phoneCharge(); }
```

```
public class UniversalCharger extends IPhoneCharger1 implements ChargAdapter{
public void phoneCharge() {
super.iPhoneCharge(); } }
```

Class adapter:

Adapter object can have at most two faces

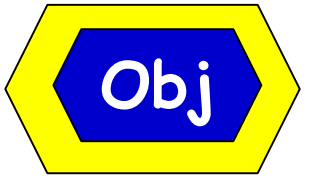
```
public class UniversalCharger implements ChargeAdapter{
IPhoneCharger iphoneCharger;
public UniversalCharger(IPhoneCharger iphoneCharger1){
this.iphoneCharger = iphoneCharger;  }
public void phoneCharge() {
iphoneCharger.iPhoneCharge(); }
```



Object adapter

Consequences - Class Adapters

- Creates concrete adapter for a specific Adaptee (e.g., **NewSet**):
  - Not really a wrapper pattern…



− Cannot adapt a class and all its subclasses…

+ Can override Adaptee (e.g., **NewSet**) behavior:

  - After all, Adapter is a subclass of Adaptee

Consequences - Object Adapters

- Single Adapter can handle many Adaptees **(Universal adaptor):**
  - Can adapt the Adaptee class and all its subclasses.
- The biggest benefit of Object Adapter compared to Class Adapter (and thus Inheritance):
  - **Loose coupling of client and adaptee.**
- **Hard to override Adaptee behavior**
  - Because the Adapter *uses* but does not *inherit from* Adaptee interface.

Other Issues

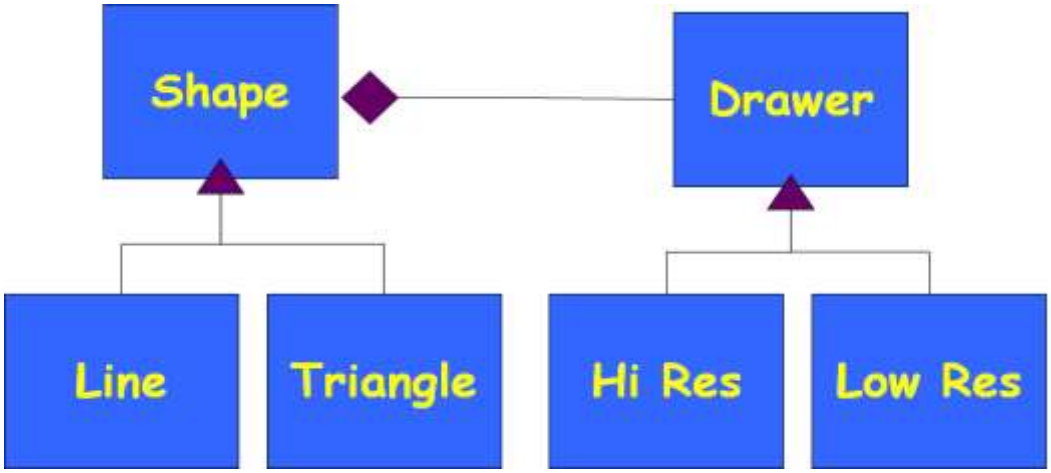
- How much adapting does adapter do?
  - Simple forwarding of requests (renaming)?
  - Different set of operations and semantics?
  - **At some point do the Adaptee and Adapter interfaces and functionality diverge so much that "adaption" is no longer the correct term…**

Advantages of Adapter Pattern

- Can help change behavior of existing software:
  - Without changing its source code.
- Can help use legacy software:
  - Without making any modifications to old source code

Bridge Pattern

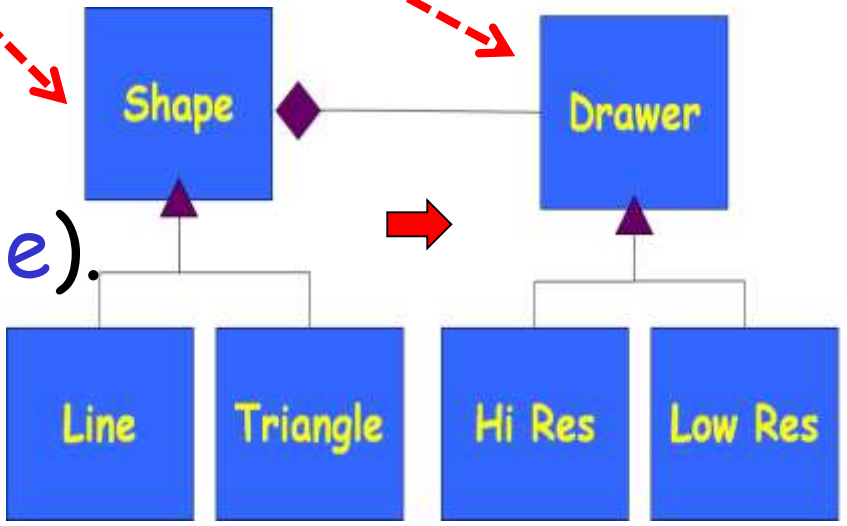
Bridge Pattern: Introduction



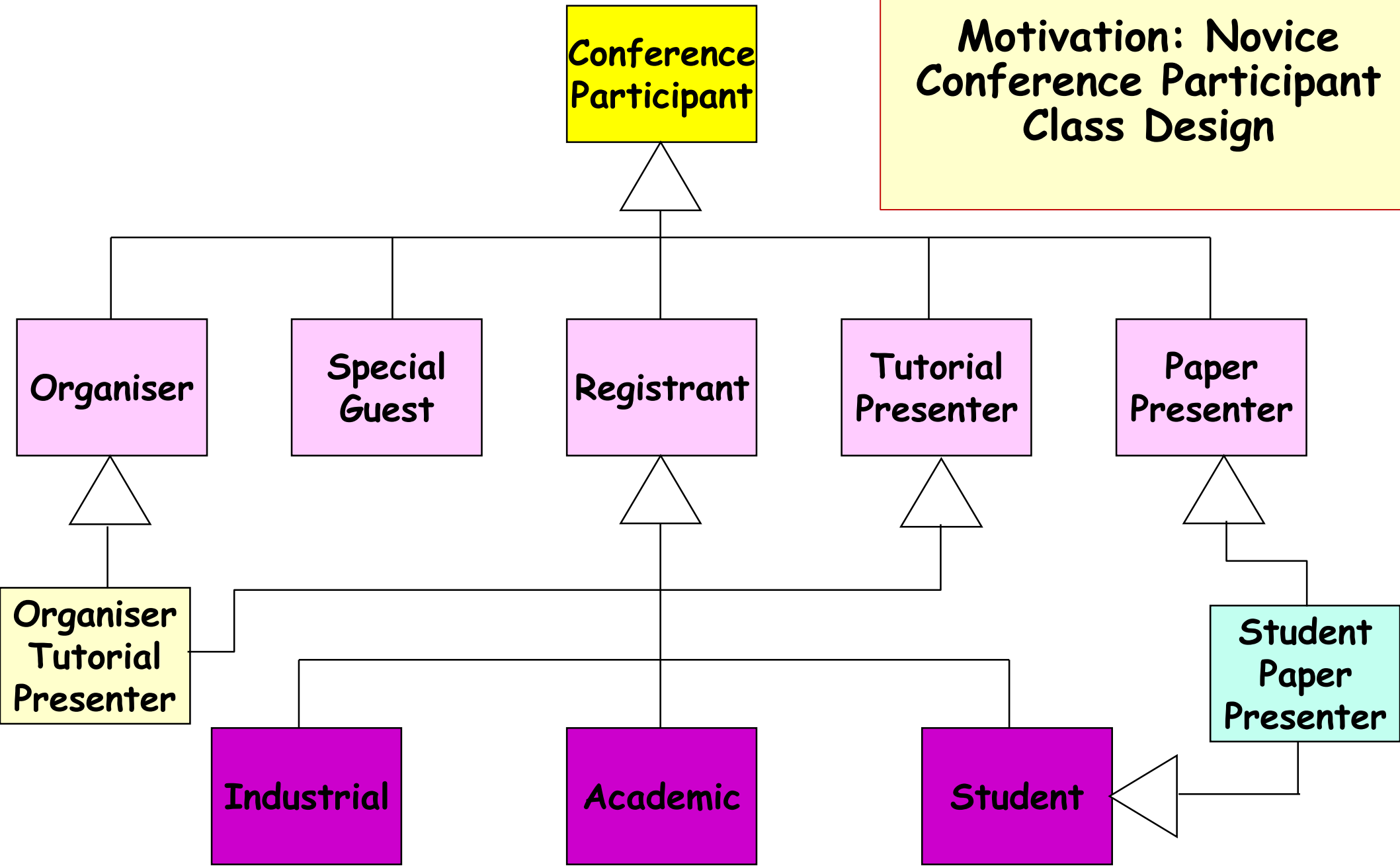
- Helps decouple an abstraction hierarchy from its implementation:
  - Lets implementations and abstractions to vary independently.
  - Allows using one of several implementations of an interface to be decided upon dynamically.

Bridge Pattern

- Also known as a **Handle/Body pattern**.
  - Split a class design into two class hierarchies.
  - One represents the concepts (called the handle).
  - The other embodies the implementation, and is called the body.
  - *Handle* forwards any invocations to the body.

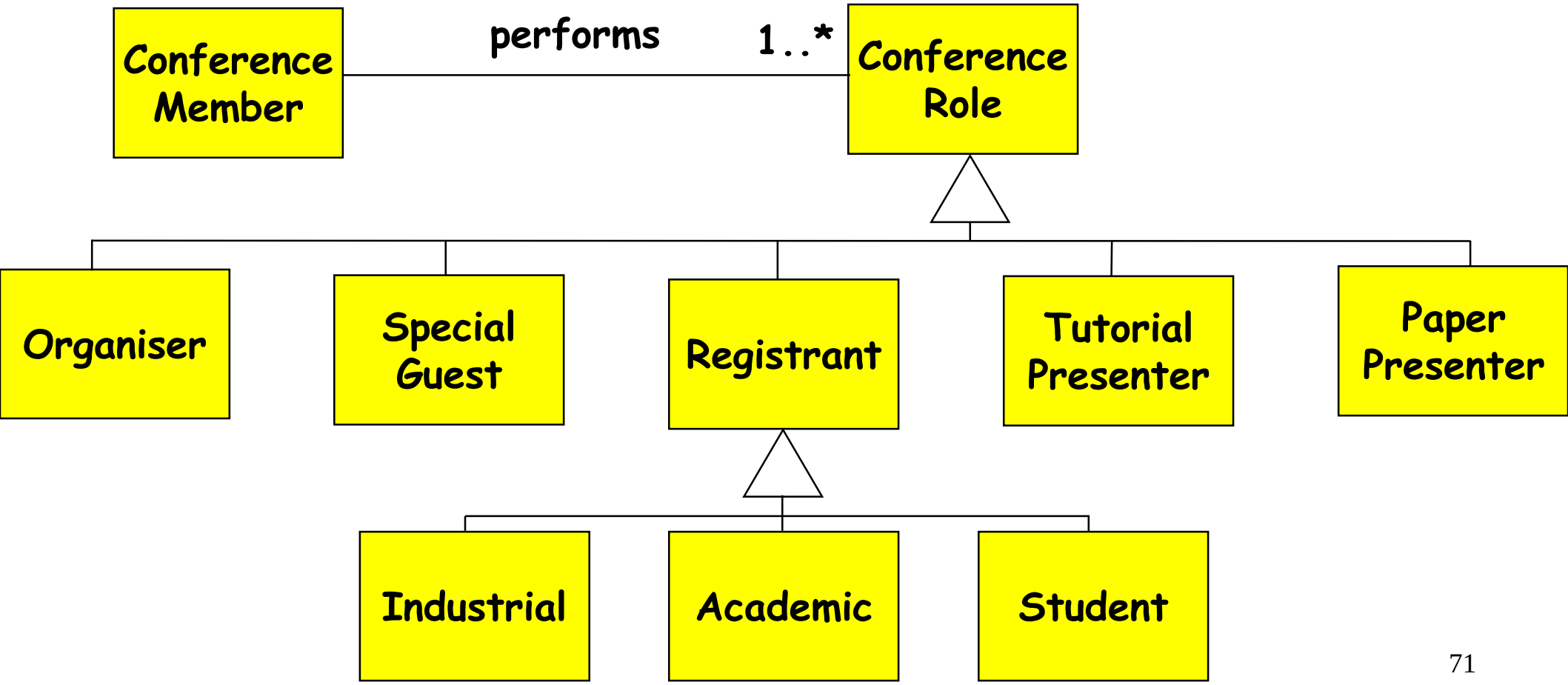


Motivation: Novice Conference Participant Class Design



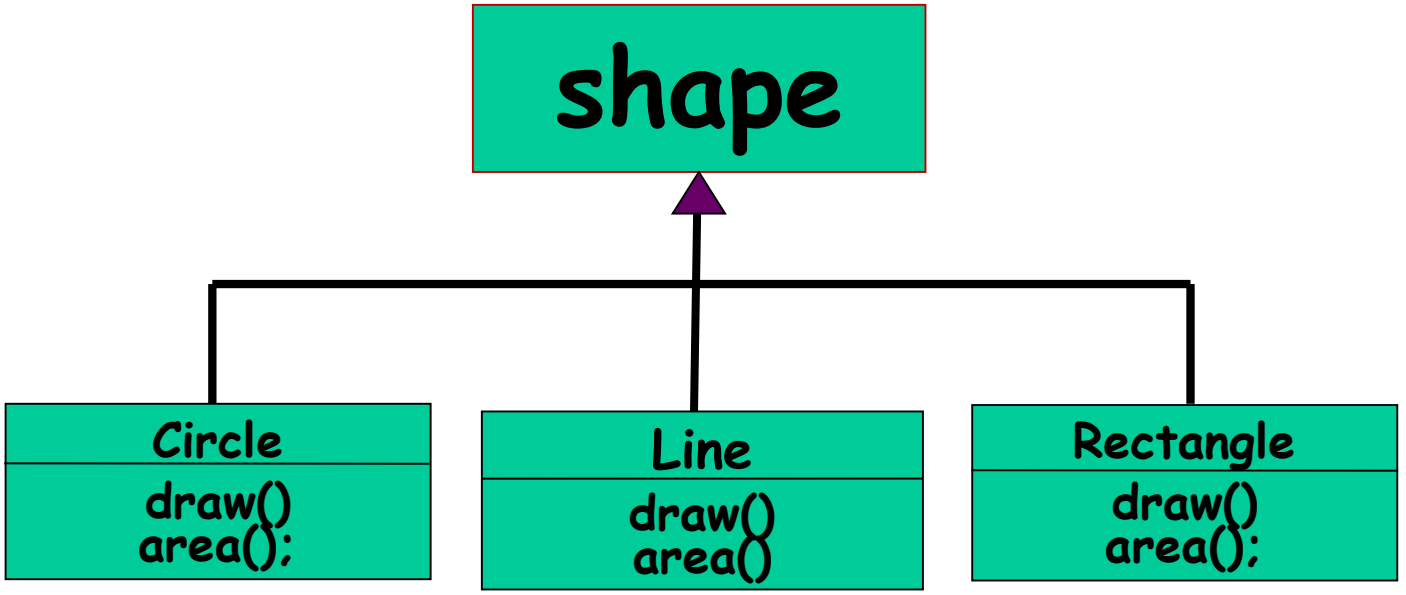
Solution: Delegate to Roles

- For the discussed example, multiple inheritance is not a good solution.
- ***Delegation to required roles* is a much better solution --- makes the solution more flexible.**



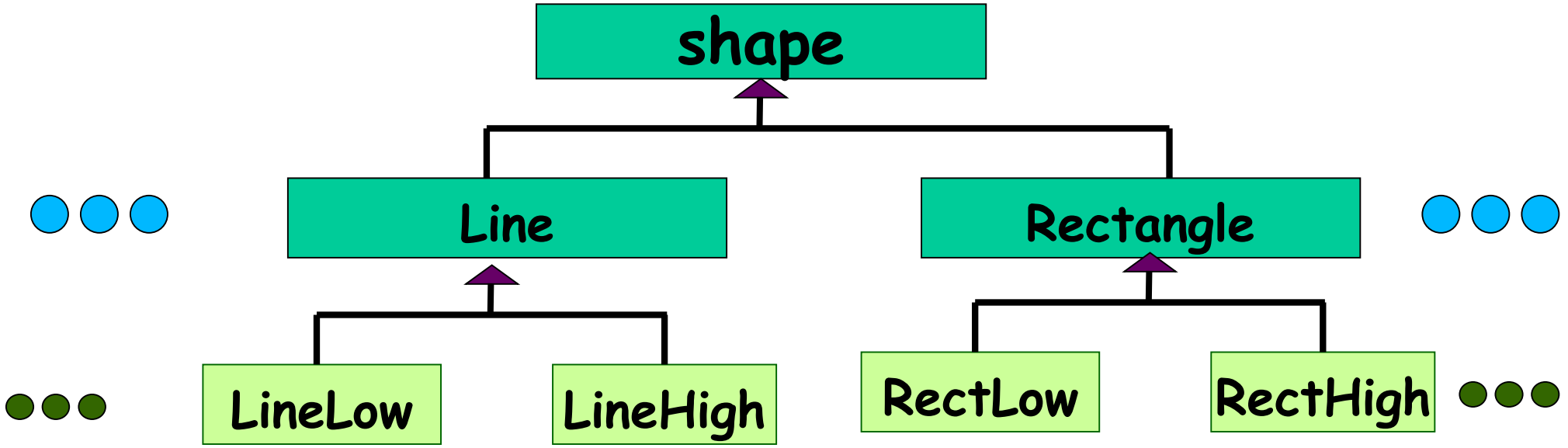
Motivating Example 2

- You designed a graphics package…



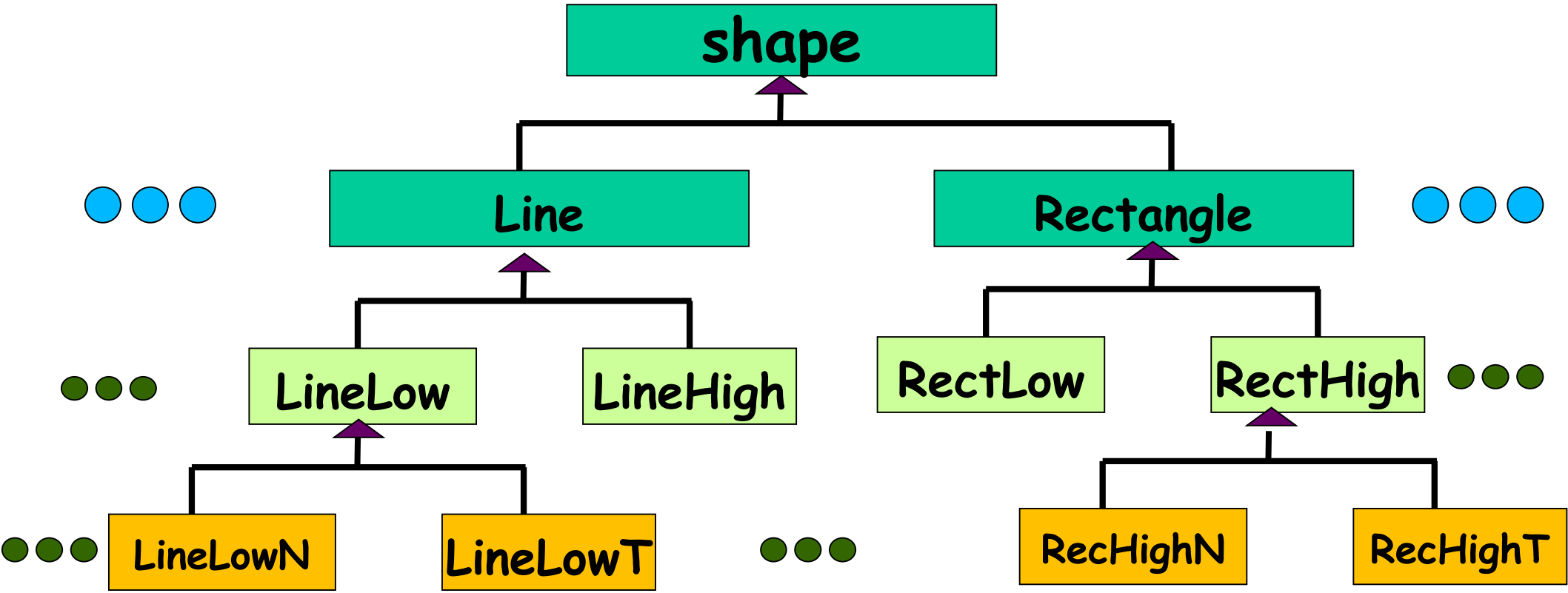
Motivating Example cont…

- Things worked fine:
  - Until you had to support mobile phones that can draw only low precision shapes.
  - **You extended your design….**



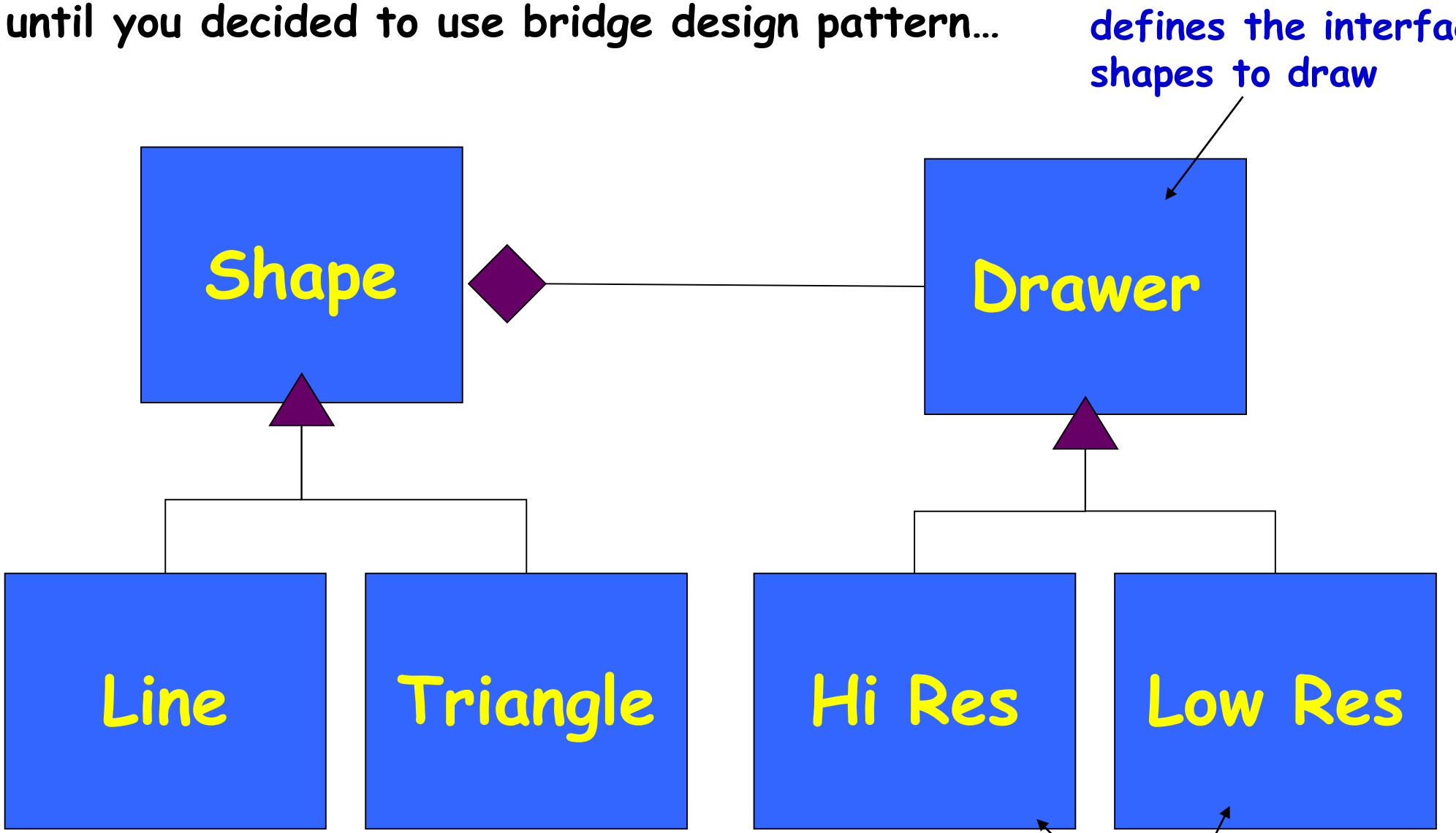
Motivating Example cont…

- You soon had to support a different way of drawing for efficient transient views for animation…
  - **You extended your design again ….**



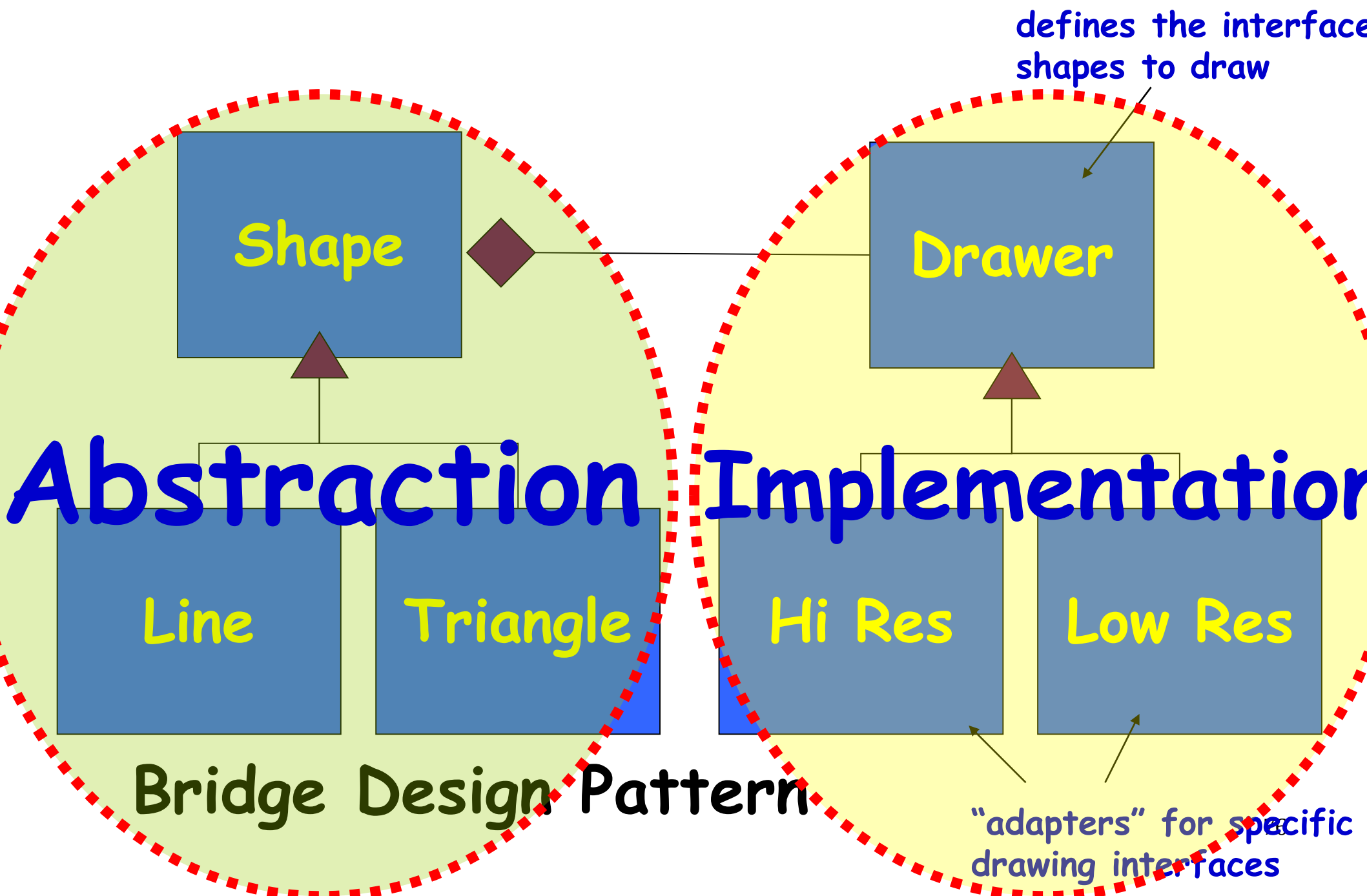
- You soon needed a different way of drawing on Smartphones…
- Things were becoming pretty complicated …

  until you decided to use bridge design pattern…



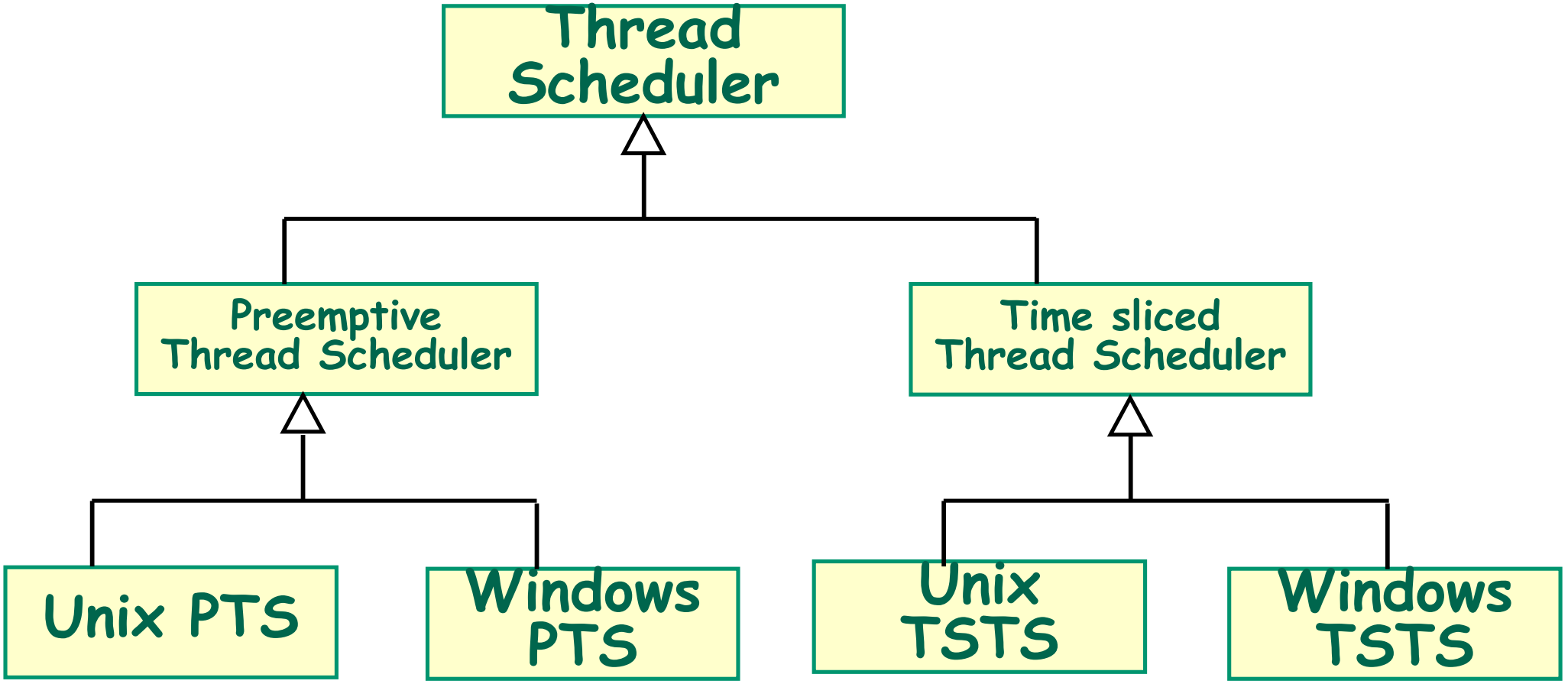
Bridge Design Pattern

Separate the abstraction from implementation



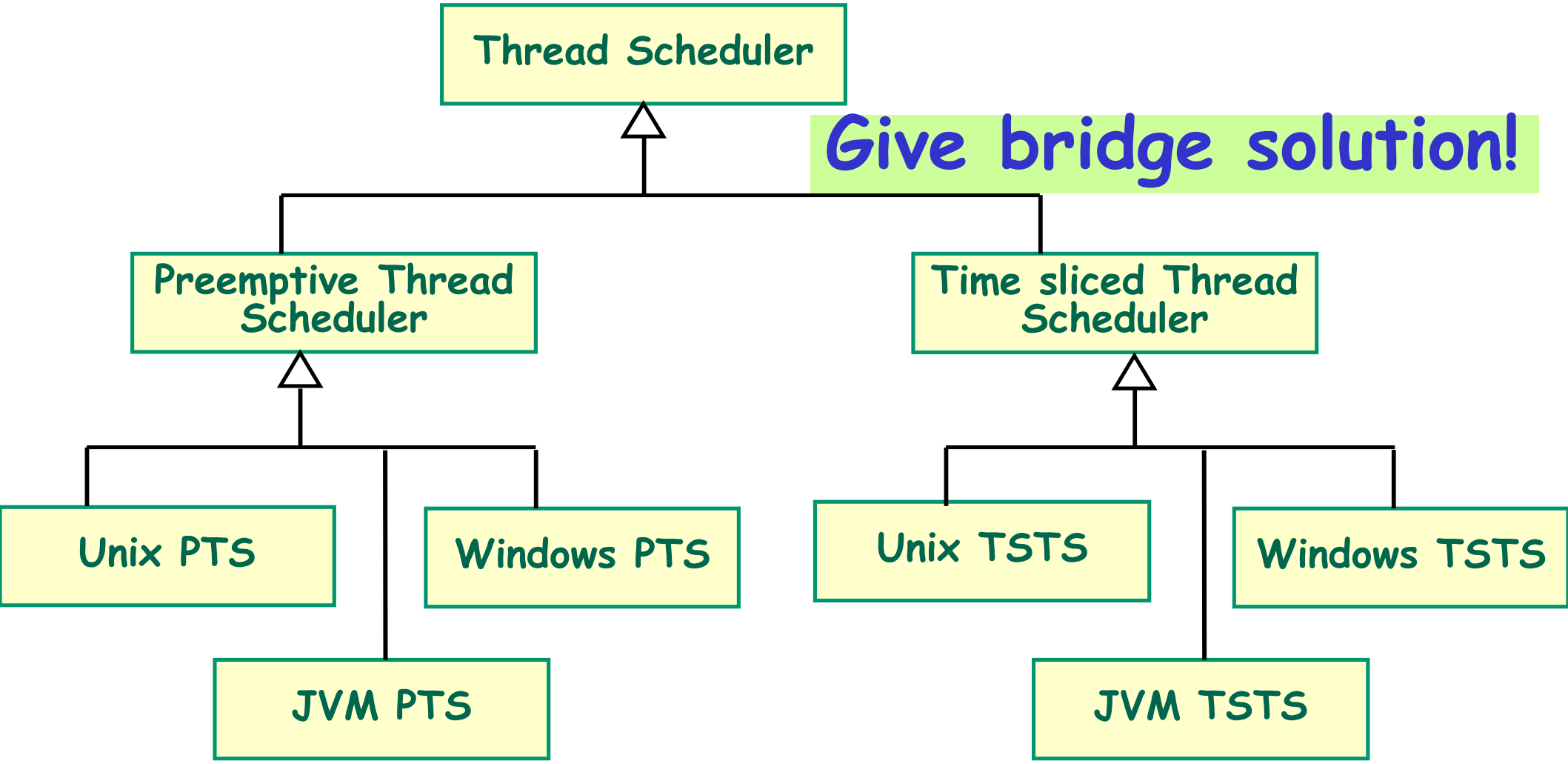
Exercise 1

Consider thread scheduling :



Exercise 1

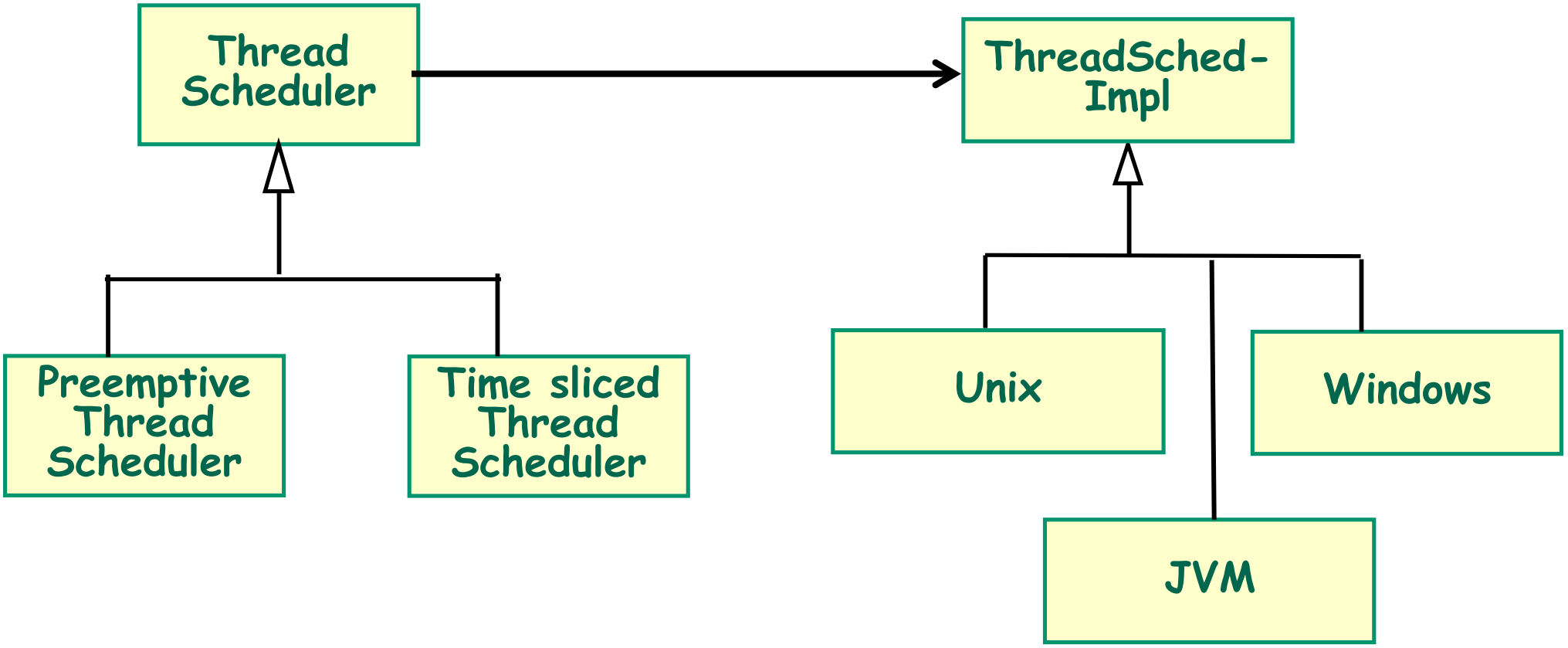
We need to now support Java platform also…



- **Explosive Class Hierarchy!**

Solution with Bridge Pattern

- Refactor into two orthogonal hierarchies:
  - Platform-independent abstractions and platform dependent implementations



Few Observations…

- Suppose an abstraction has several implementations:
  - **Inheritance is commonly used to accommodate these!!!**

1. But inheritance binds an implementation to the abstraction permanently:
   - It becomes difficult to modify and reuse abstraction and implementations independently.
2. Inheritance without a **Bridge**:.
   - **Leads to violation of single responsibilty principle**

Overusing of inheritance…

"As a beginning object-oriented analyst, I had a tendency to solve every kind of problem by using special cases, taking advantage of inheritance. I loved the idea of inheritance because it seemed new and powerful. I used it whenever I could. This usually seems to be normal to many beginning analysts, but it is naive: "given a new hammer, everything seems like a nail."

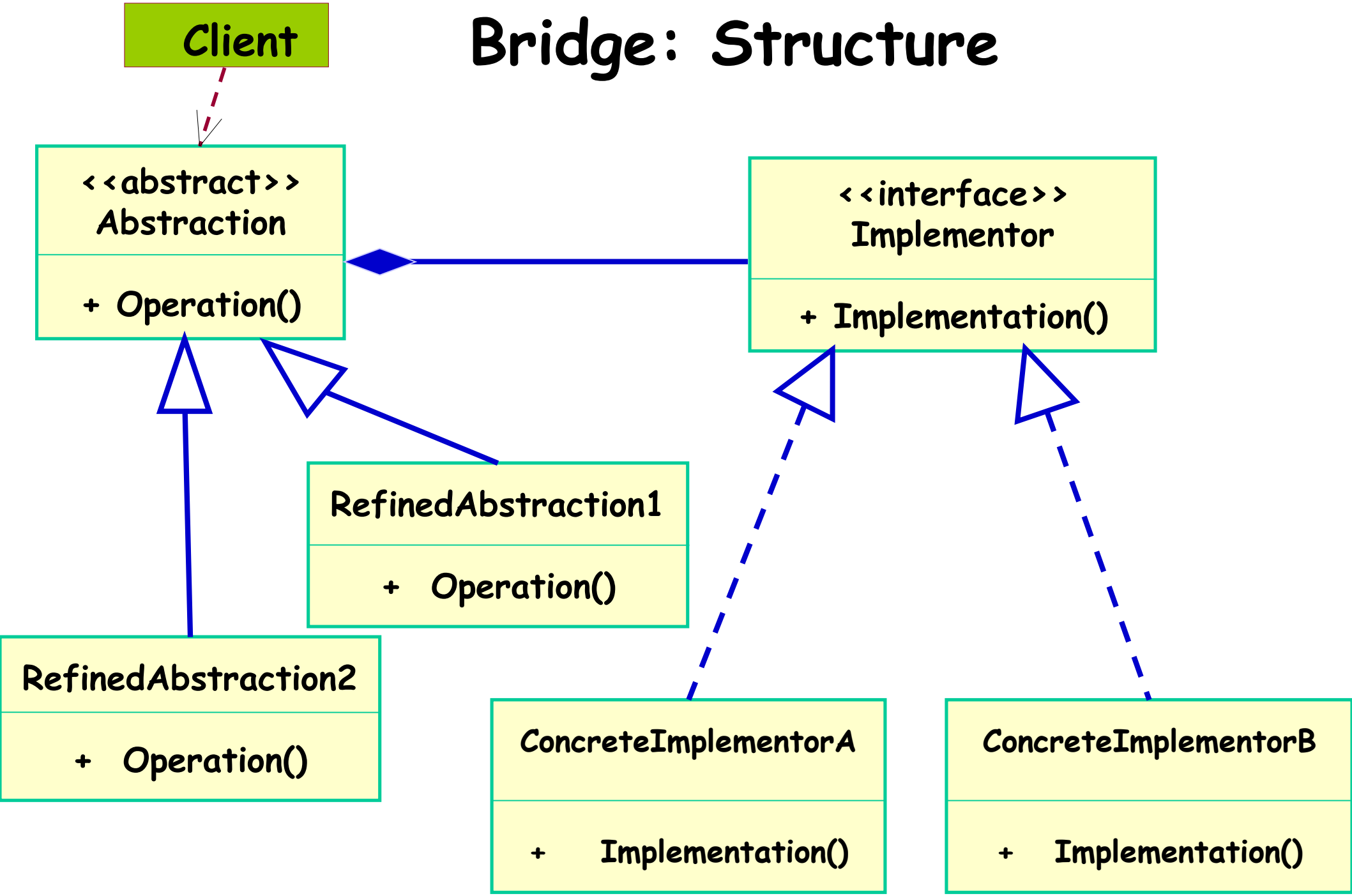
Bridge: Applicability

Use bridge Pattern when:

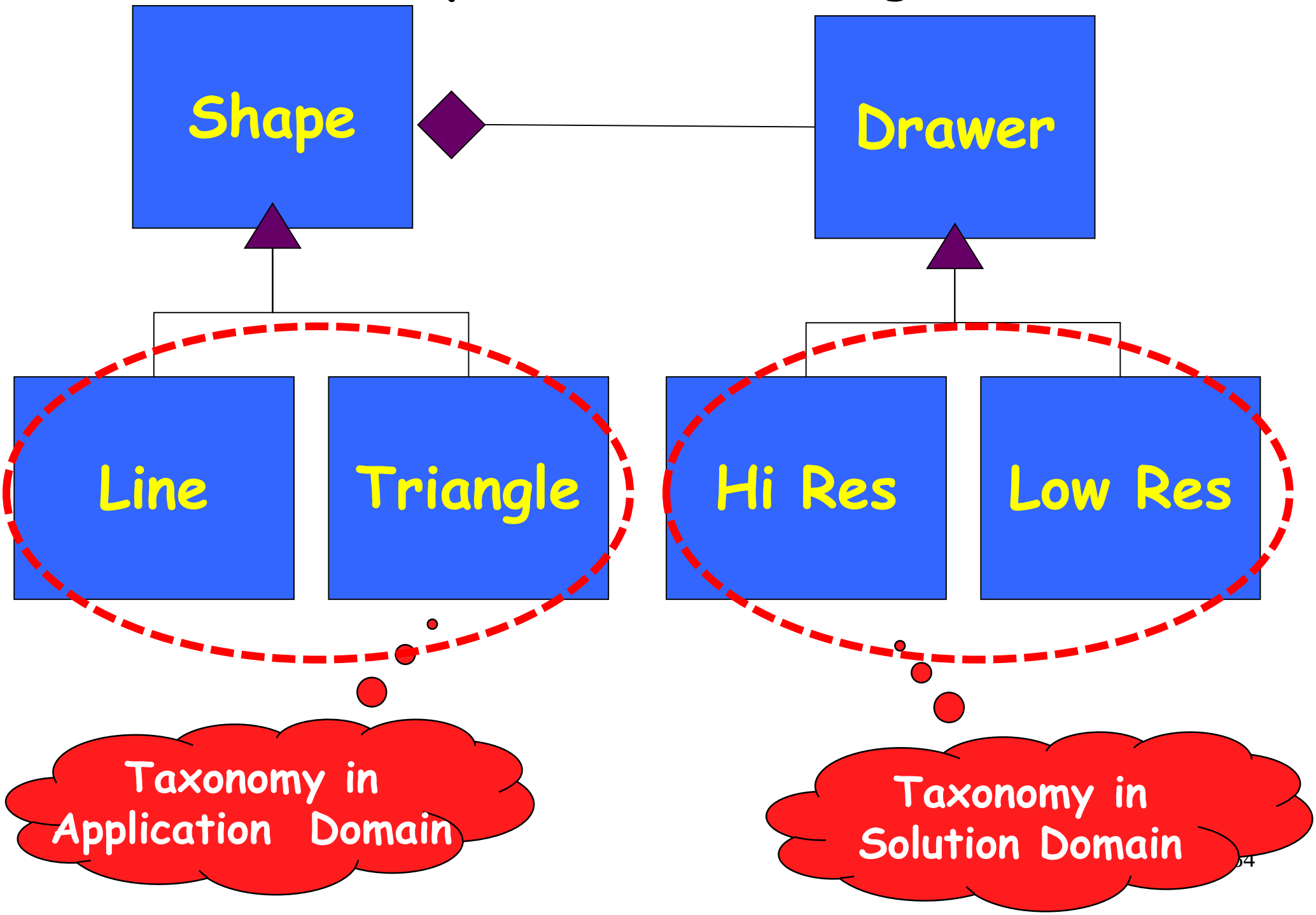
- You want to avoid a permanent binding between an abstraction and its implementation.
  - **Implementation may be selected or switched at run time.**

- Both the abstraction and their implementation should be extensible by subclassing without impacting the clients:
  - **Even client code would not need recompiling.**

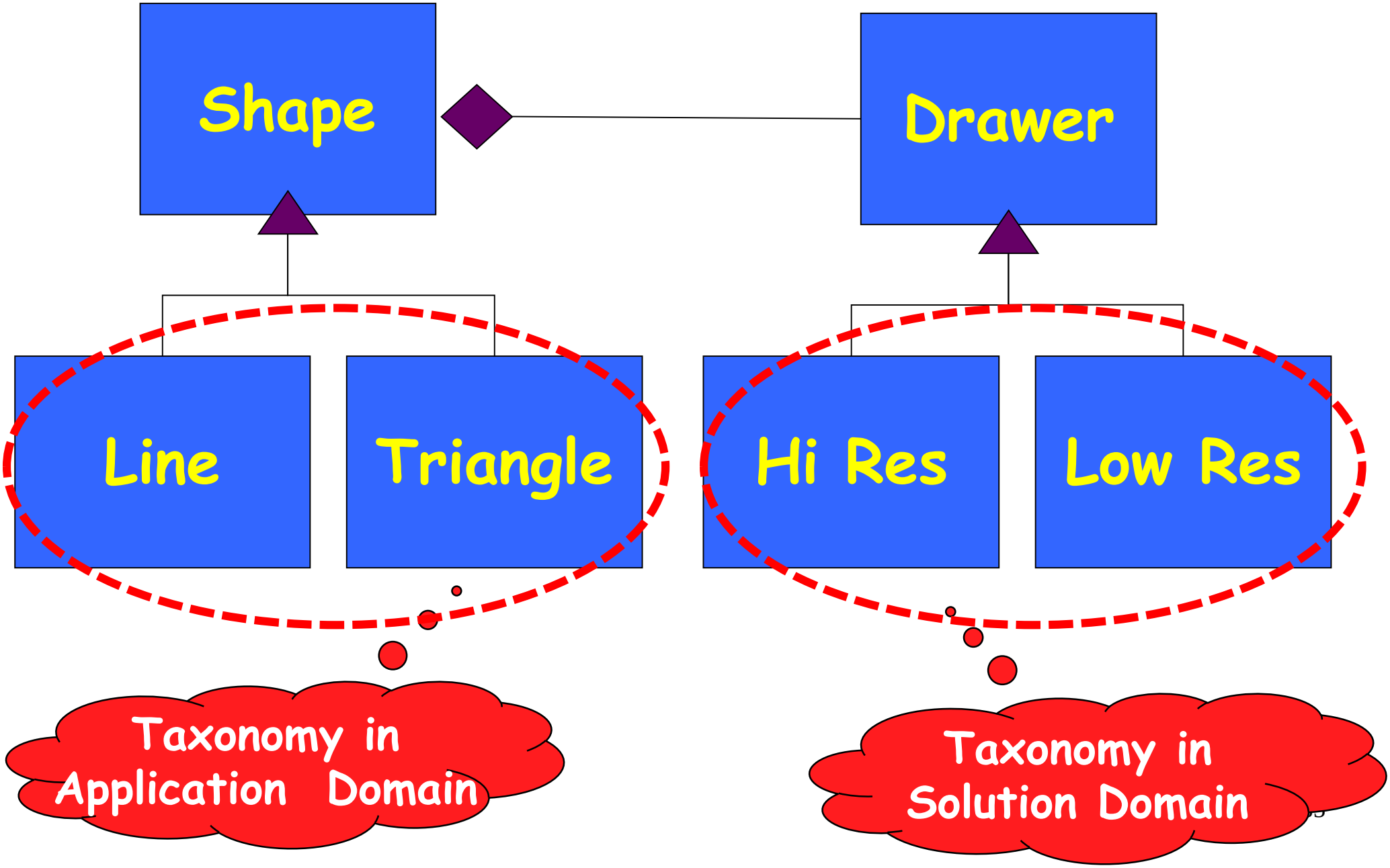
Bridge: Structure



Why the Name Bridge?

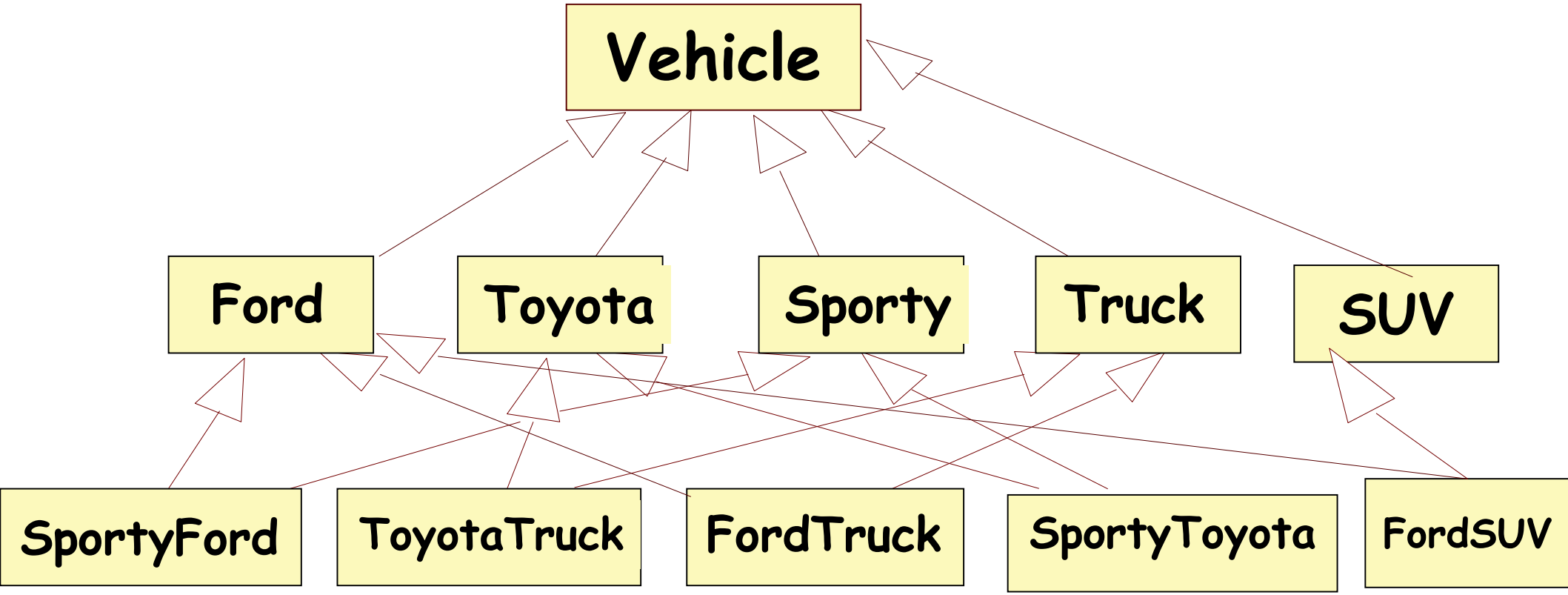


Provides A Bridge Between the Application and Solution domains…



Bridge Pattern: Exercise 2

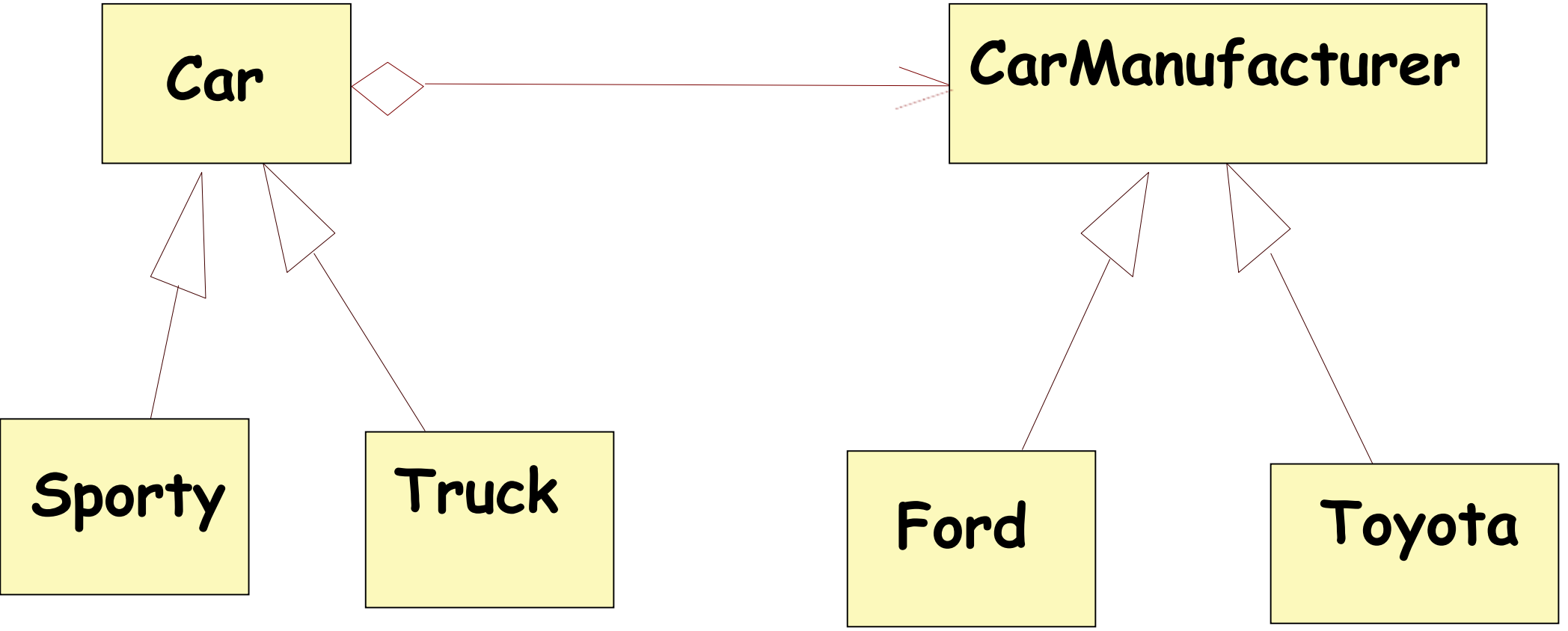
- How to improve this design?



- Existing design…

Exercise 1: Solution…

- Use Bridge when you might otherwise be tempted to use multiple inheritance…



When should we apply Bridge Pattern?

- **We want run-time binding with any required implementation.**
- **We need to overcome a proliferation of classes:**
  - Resulted from a coupled interface and numerous implementations
  - We need to map these into orthogonal class hierarchies…

Benefits

- Decoupling abstraction from implementation
- Reduction in number of sub-classes
- Reduction of program complexity and executable code size.
- Interface and Implementation can be varied independently.
- Improved extensibility:
  - Abstraction and Implementation can be extended independently…

Drawbacks?

- Runtime inefficiency
- Increased Complexity due to double Indirection :
  - Abstraction→ Implementor
    →ConcreteImplementor

Final Analysis

- Application of the time tested principle:
  - "Find what varies and encapsulate it" and
  - "Favor object composition over class inheritance"

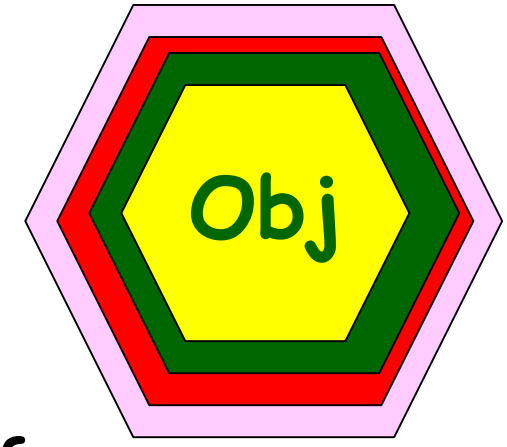
Decorator Pattern

Decorator Pattern: Another wrapper pattern!

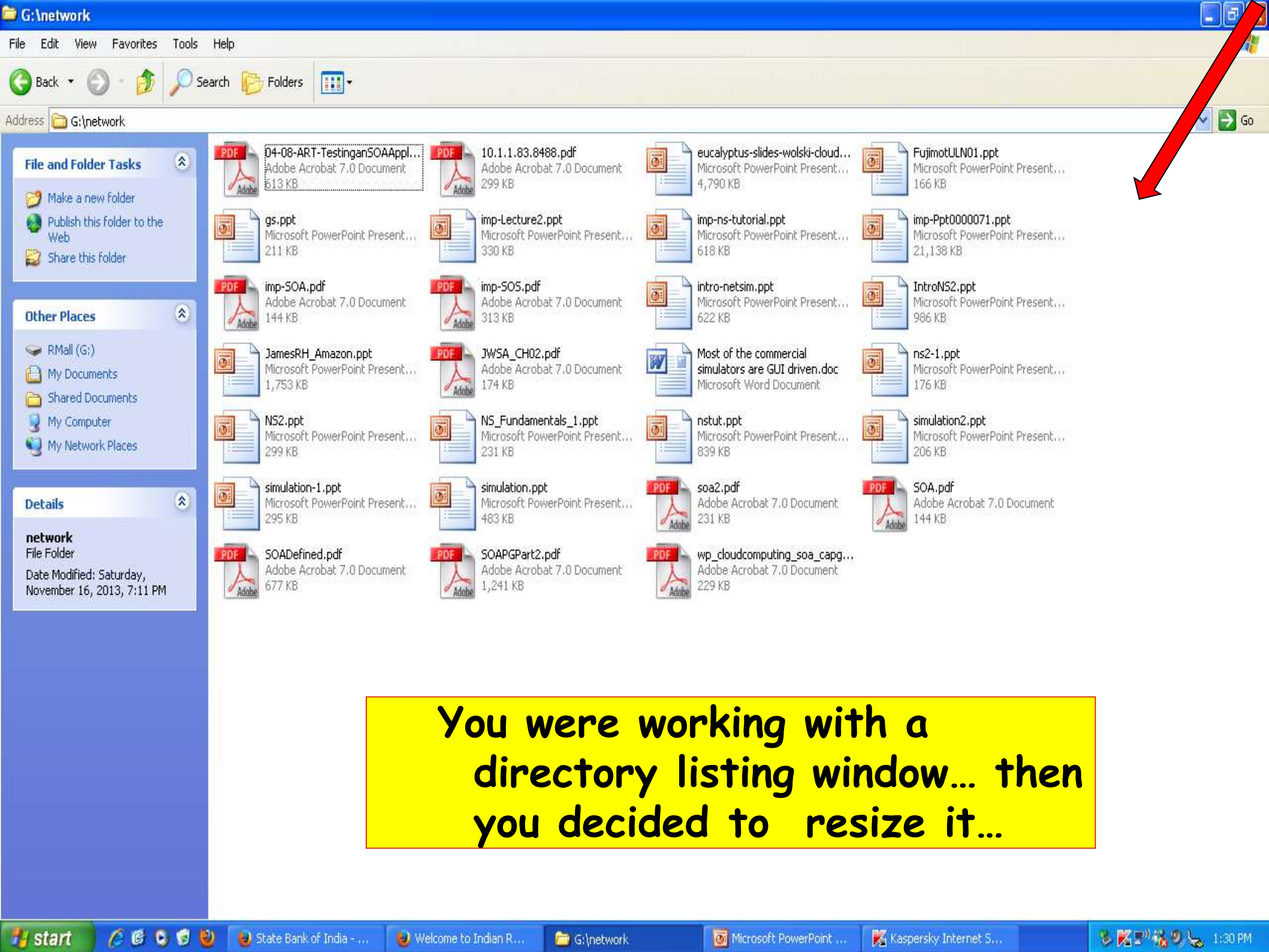
- **Intent:**
  - Attach additional responsibilities to an object dynamically.
  - Provides a flexible alternative to subclassing.

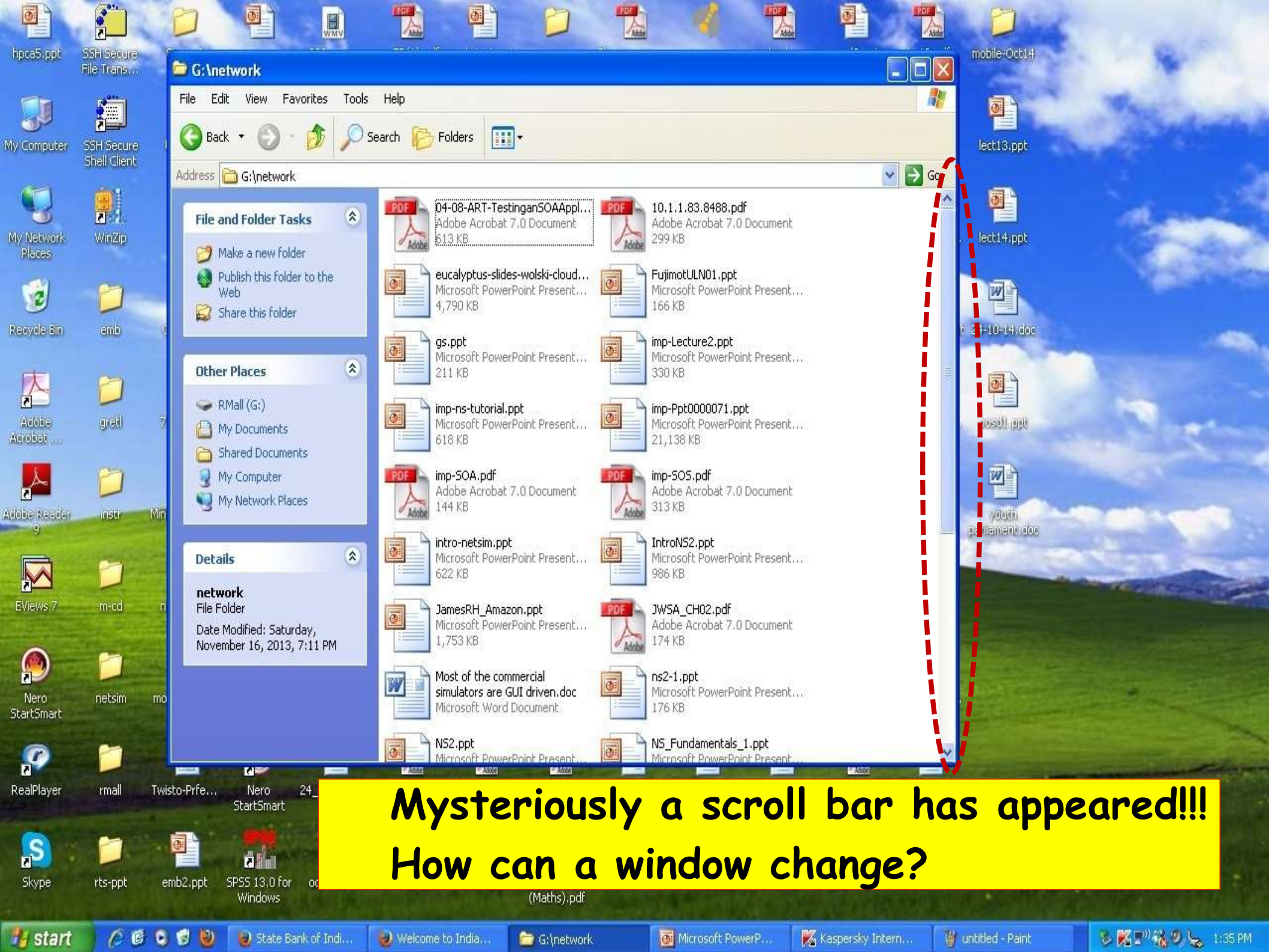
- **Motivation:**
  - **Add responsibilities to individual objects as and when required and not to an entire class**
  - Should conform to the interface of the object being decorated.

Decorator: In Simple Words



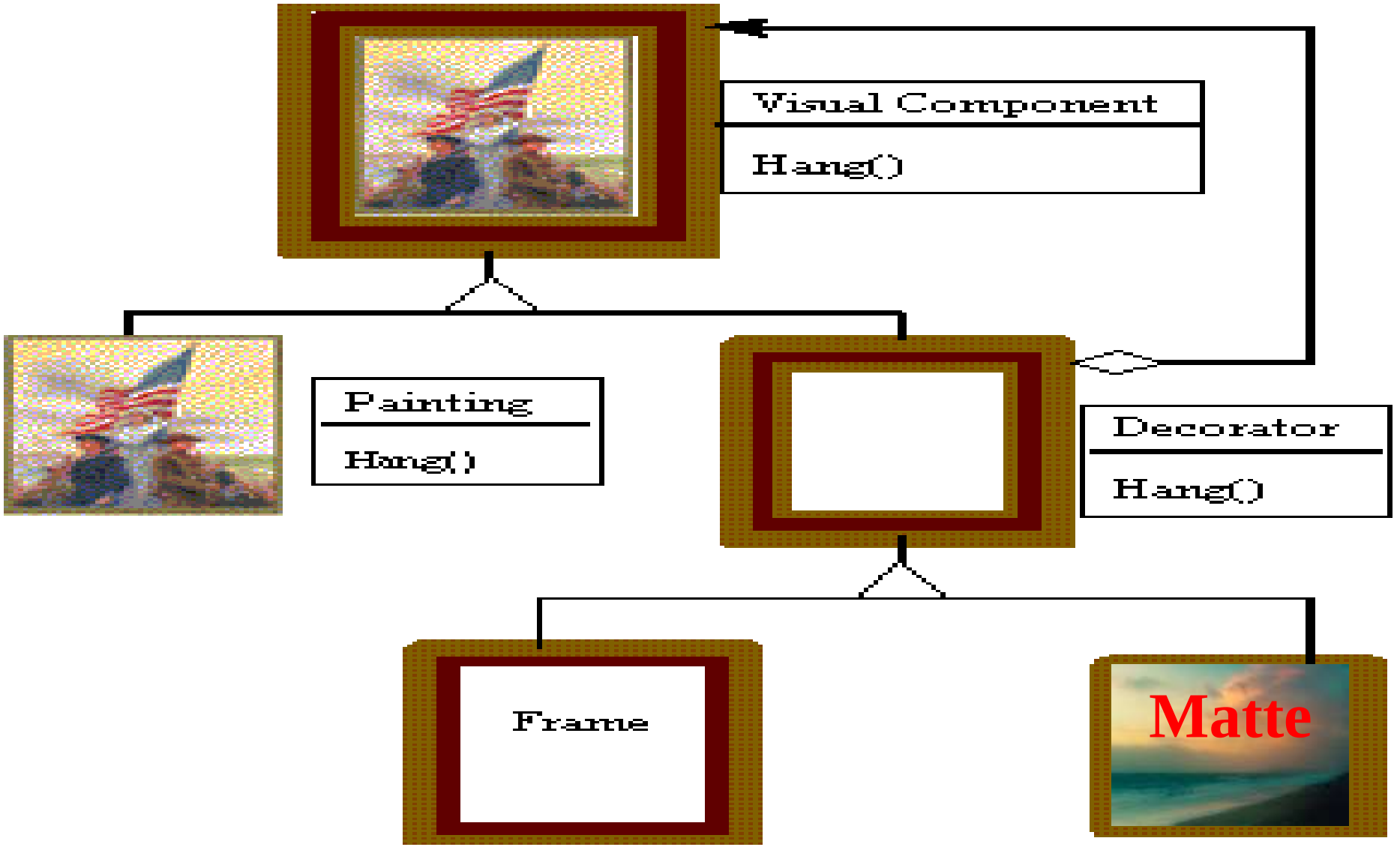
- **You have an object:**
  - You wrap it with another object.
  - They both support the same interface.
  - Later possibly wrap with more objects..
  - **The ones on outside are "decorators"**
- **Clients use the one on the outermost.**
  - Each decorator either masks, changes, or passes on method calls to one inside it…


G:\network
File Edit View Favorites Tools Help
Back Search Folders
Address G:\network
Go
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder
Other Places
RMall (G:)
My Documents
Shared Documents
My Computer
My Network Places
Details
network
File Folder
Date Modified: Saturday, November 16, 2013, 7:11 PM
04-08-ART-TestinganSOAAppl... Adobe Acrobat 7.0 Document 613 KB
10.1.1.83.8488.pdf Adobe Acrobat 7.0 Document 299 KB
eucalyptus-slides-wolski-cloud... Microsoft PowerPoint Present... 4,790 KB
FujimotULN01.ppt Microsoft PowerPoint Present... 166 KB
gs.ppt Microsoft PowerPoint Present... 211 KB
imp-Lecture2.ppt Microsoft PowerPoint Present... 330 KB
imp-ns-tutorial.ppt Microsoft PowerPoint Present... 618 KB
imp-Ppt0000071.ppt Microsoft PowerPoint Present... 21,138 KB
imp-SOA.pdf Adobe Acrobat 7.0 Document 144 KB
imp-SOS.pdf Adobe Acrobat 7.0 Document 313 KB
intro-netsim.ppt Microsoft PowerPoint Present... 622 KB
IntroNS2.ppt Microsoft PowerPoint Present... 986 KB
JamesRH_Amazon.ppt Microsoft PowerPoint Present... 1,753 KB
JWSA_CH02.pdf Adobe Acrobat 7.0 Document 174 KB
Most of the commercial simulators are GUI driven.doc Microsoft Word Document
ns2-1.ppt Microsoft PowerPoint Present... 176 KB
NS2.ppt Microsoft PowerPoint Present... 299 KB
NS_Fundamentals_1.ppt Microsoft PowerPoint Present... 231 KB
nstut.ppt Microsoft PowerPoint Present... 839 KB
simulation2.ppt Microsoft PowerPoint Present... 206 KB
simulation-1.ppt Microsoft PowerPoint Present... 295 KB
simulation.ppt Microsoft PowerPoint Present... 483 KB
soa2.pdf Adobe Acrobat 7.0 Document 231 KB
SOA.pdf Adobe Acrobat 7.0 Document 144 KB
SOADefined.pdf Adobe Acrobat 7.0 Document 677 KB
SOAPGPart2.pdf Adobe Acrobat 7.0 Document 1,241 KB
wp_cloudcomputing_soa_capg... Adobe Acrobat 7.0 Document 229 KB
You were working with a directory listing window… then you decided to resize it…
start
State Bank of India - ...
Welcome to Indian R...
G:\network
Microsoft PowerPoint ...
Kaspersky Internet S...
1:30 PM


G:\network
File Edit View Favorites Tools Help
Back Search Folders
Address G:\network
File and Folder Tasks
Make a new folder
Publish this folder to the Web
Share this folder
Other Places
RMall (G:)
My Documents
Shared Documents
My Computer
My Network Places
Details
network
File Folder
Date Modified: Saturday, November 16, 2013, 7:11 PM
04-08-ART-TestinganSOAAppl...
Adobe Acrobat 7.0 Document
613 KB
10.1.1.83.8488.pdf
Adobe Acrobat 7.0 Document
299 KB
eucalyptus-slides-wolski-cloud...
Microsoft PowerPoint Present...
4,790 KB
FujimotULN01.ppt
Microsoft PowerPoint Present...
166 KB
gs.ppt
Microsoft PowerPoint Present...
211 KB
imp-Lecture2.ppt
Microsoft PowerPoint Present...
330 KB
imp-ns-tutorial.ppt
Microsoft PowerPoint Present...
618 KB
imp-Ppt0000071.ppt
Microsoft PowerPoint Present...
21,138 KB
imp-SOA.pdf
Adobe Acrobat 7.0 Document
144 KB
imp-SOS.pdf
Adobe Acrobat 7.0 Document
313 KB
intro-netsim.ppt
Microsoft PowerPoint Present...
622 KB
IntroNS2.ppt
Microsoft PowerPoint Present...
986 KB
JamesRH_Amazon.ppt
Microsoft PowerPoint Present...
1,753 KB
JWSA_CH02.pdf
Adobe Acrobat 7.0 Document
174 KB
Most of the commercial simulators are GUI driven.doc
Microsoft Word Document
ns2-1.ppt
Microsoft PowerPoint Present...
176 KB
NS2.ppt
NS_Fundamentals_1.ppt
Mysteriously a scroll bar has appeared!!!
How can a window change?

A Non-Software Example

- Frames are often added to pictures.
- Prior to hanging, the paintings may be matted



Matte Painting? --Digression

- Helps create the illusion of an environment that is nonexistent or is too expensive or impossible to build or use.

The government warehouse in **Raiders of the Lost Ark** (1981) was painted on glass by Michael Pangrazio at Industrial Light & Magic, and combined with live-action footage of a government worker, pushing his cargo up the center aisle.

A Matte Painting

Decorator: Non-software example 2

Suppose you would like to give a gift to someone:.

- First you select the gift…

- Next you wrap the gift…

The gift can be wrapped in several ways…

Gift-Wrap Options

Various options for wrapping gift:

- box wrapped with gift-pa
- box wrapped with gift-paper-with-crepe-paper
- box wrapped with gift-paper-with-bow-without-crepe-paper
- box wrapped with gift-paper with bow and crepe-paper and card
- box wrapped with gift-paper without bow with crepe-paper and card
- box wrapped with gift-paper without bow with crepe-paper without card
- And so on…

Gift Paper

Kraft Paper

Crepe Paper

Bow

Card

Bow

Card

Card

Card

Solution

- To overcome the problem, manufacturers sell the following materials separately:
  - **Boxes**
  - **Gift Papers**
  - **Cards**
  - **Bows**
  - **Crepe-paper**

- **Clients select any number of materials and ask them to be put in any order required!**

Decorator Pattern: Some Examples…

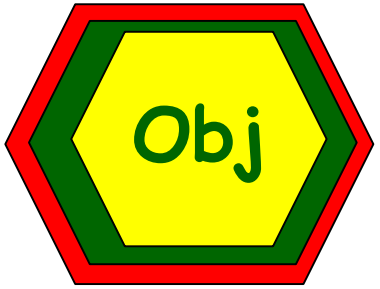
- Add borders or scrollbars to a GUI component
- Add headers and footers to an advertisement
- Add functionality to a stream :
  - Reading input line by line or word by word; or compress a file before sending it over.

Decorator: Some General Concepts

- A Decorator adds responsibilities to individual objects **(not to all objects of a class!)** dynamically:
  - **In situations where a large number of independent extensions required…**
  - **An explosion of subclasses would occur if every combination to be supported.**
  - **Difficult to understand, remember and apply…**

Decorator: Recounting the Ideas

- A Decorator is an object that has an interface identical to an object that it contains.
  - Used for adding additional functionality to a particular object at run time as opposed to adding to a class.

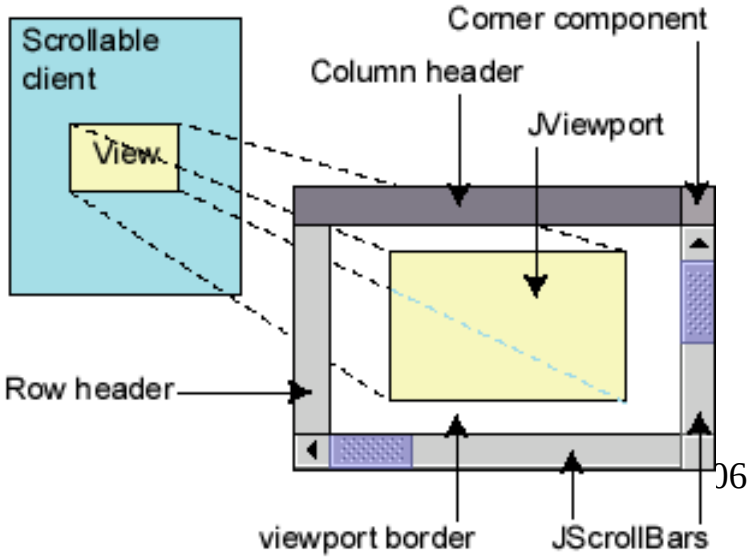


  - **Any call that the decorator gets, it relays to the object that it contains, and adds its own functionality along the way, either before or after the call.**

Decorator example: GUI

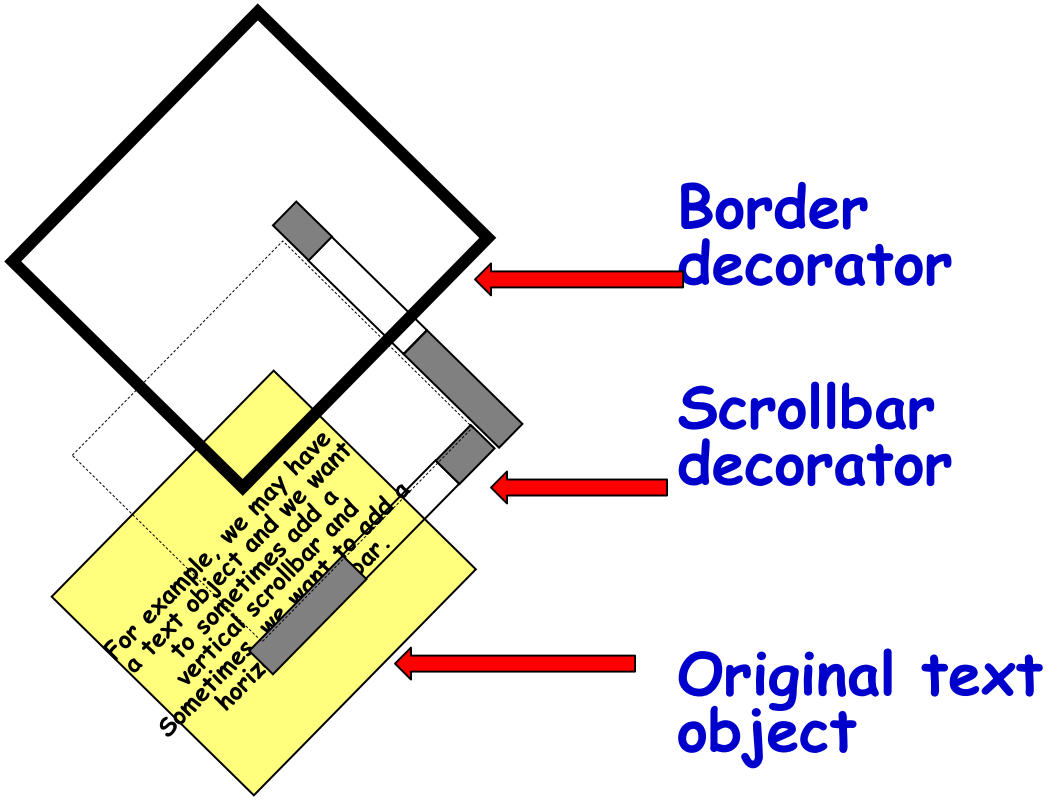
- Normal GUI components don't have scroll bars
- JScrollPane is a container with scroll bars to which you can add any component to make it scrollable

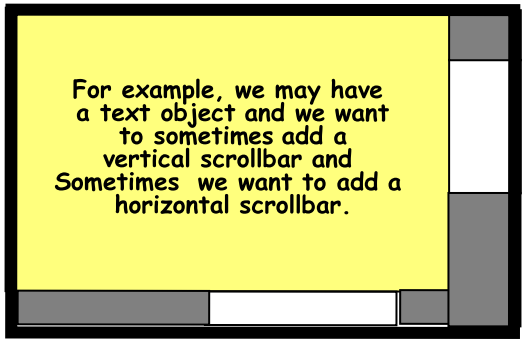
```
// JScrollPane decorates GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane =
   new JScrollPane(area);
contentPane.add(scrollPane);
```



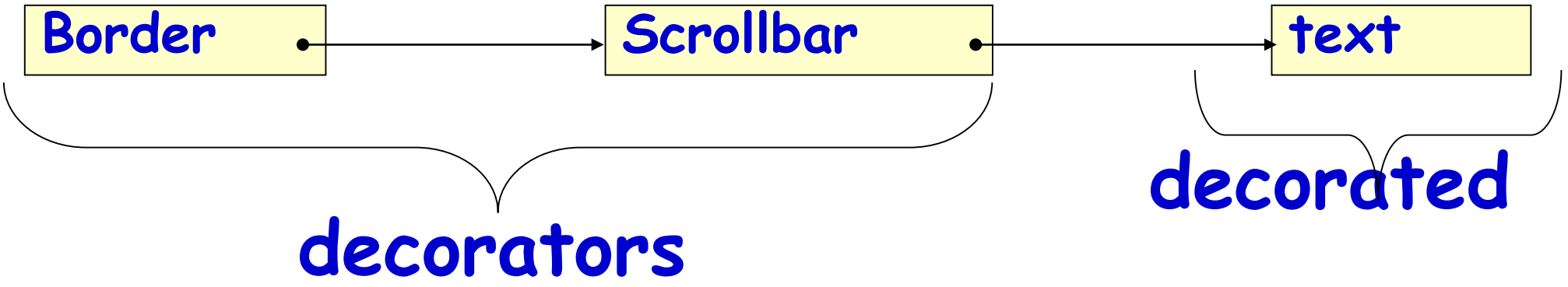
Decorator: Programming Example

- We have a text object and …
  - We want to add a border
  - Sometimes we want to also add a scrollbar.





Decorator: Object Diagram



- The objects refer each other like a linked list or chain of objects.
- The last in the list is the decorated object.

Widget and Stream Examples

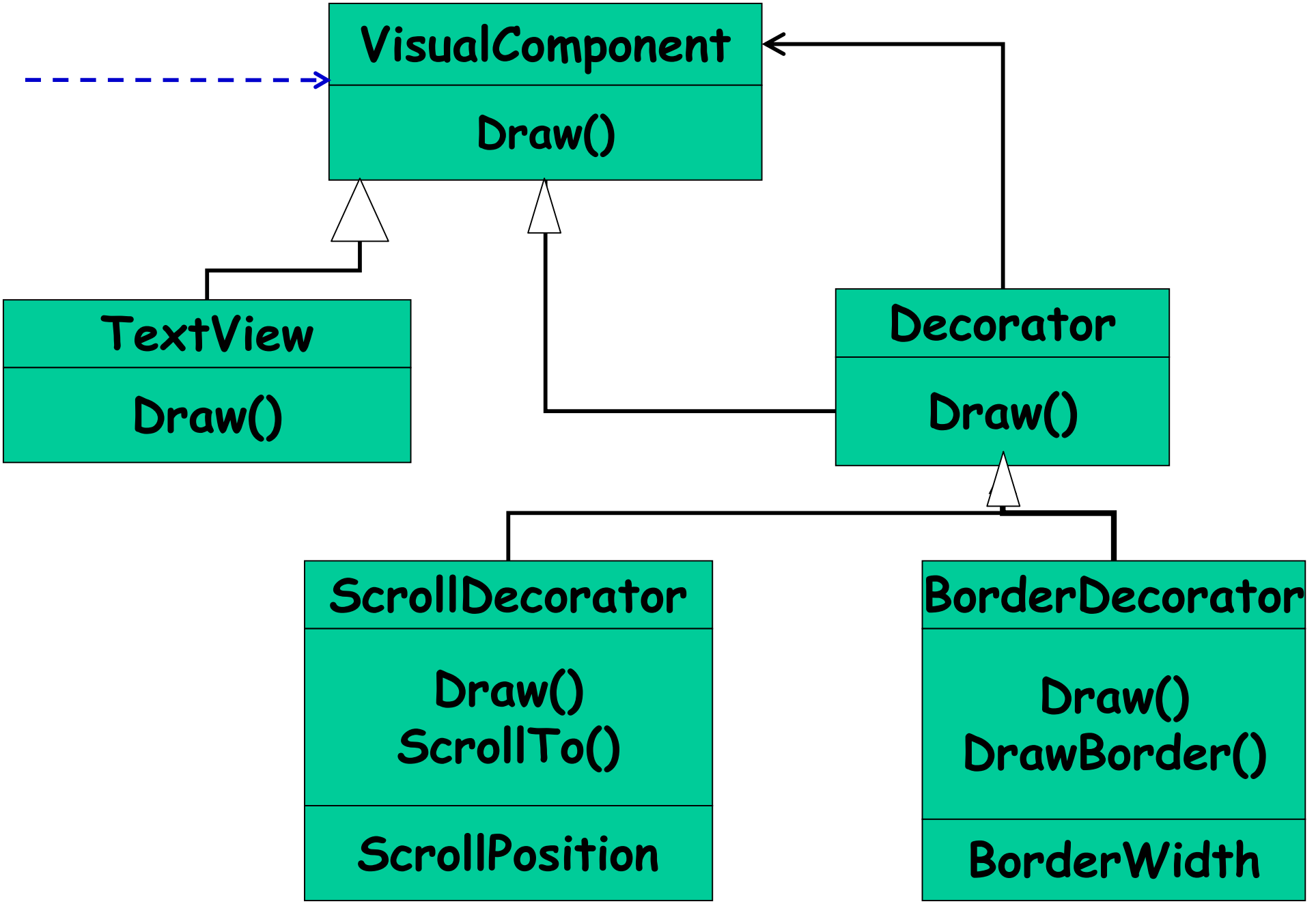
- Suppose you have a user interface toolkit and you wish to provide a choice of border or scrolling to clients.
- The client "attaches" the border or scrolling to only those objects requiring these capabilities.

```
Widget  aWidget = new BorderDecorator(
    new ScrollDecorator(new TextView(), 1);
aWidget.draw();
```

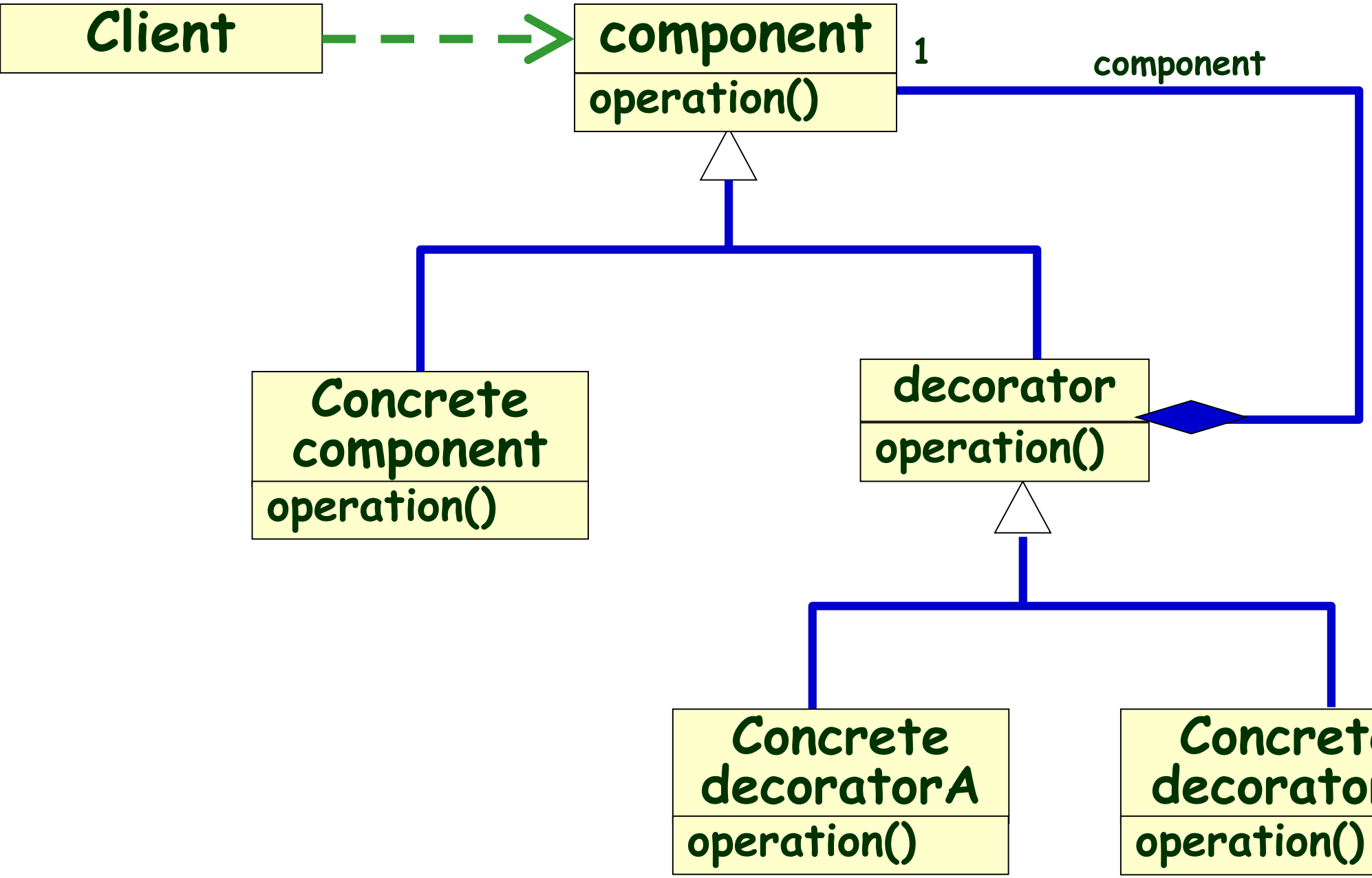
- **Stream Example**
  - Cascading responsibilities to an output stream

```
Stream aStream = new CompressingStream(
    new ASCII7Stream(new
  FileStream( "fileName.dat" )));
aStream.putString( "Hello world" );
```

Example: Decorator Class Structure



Decorator Structure



An Example Application

- Consider a **TextView** GUI component:
  - You want to add different kinds of borders and/or scrollbars to it.
- You can add 3 types of borders:
  - Plain, 3D, or Fancy
- Also scrollbars:
  - Horizontal or Vertical

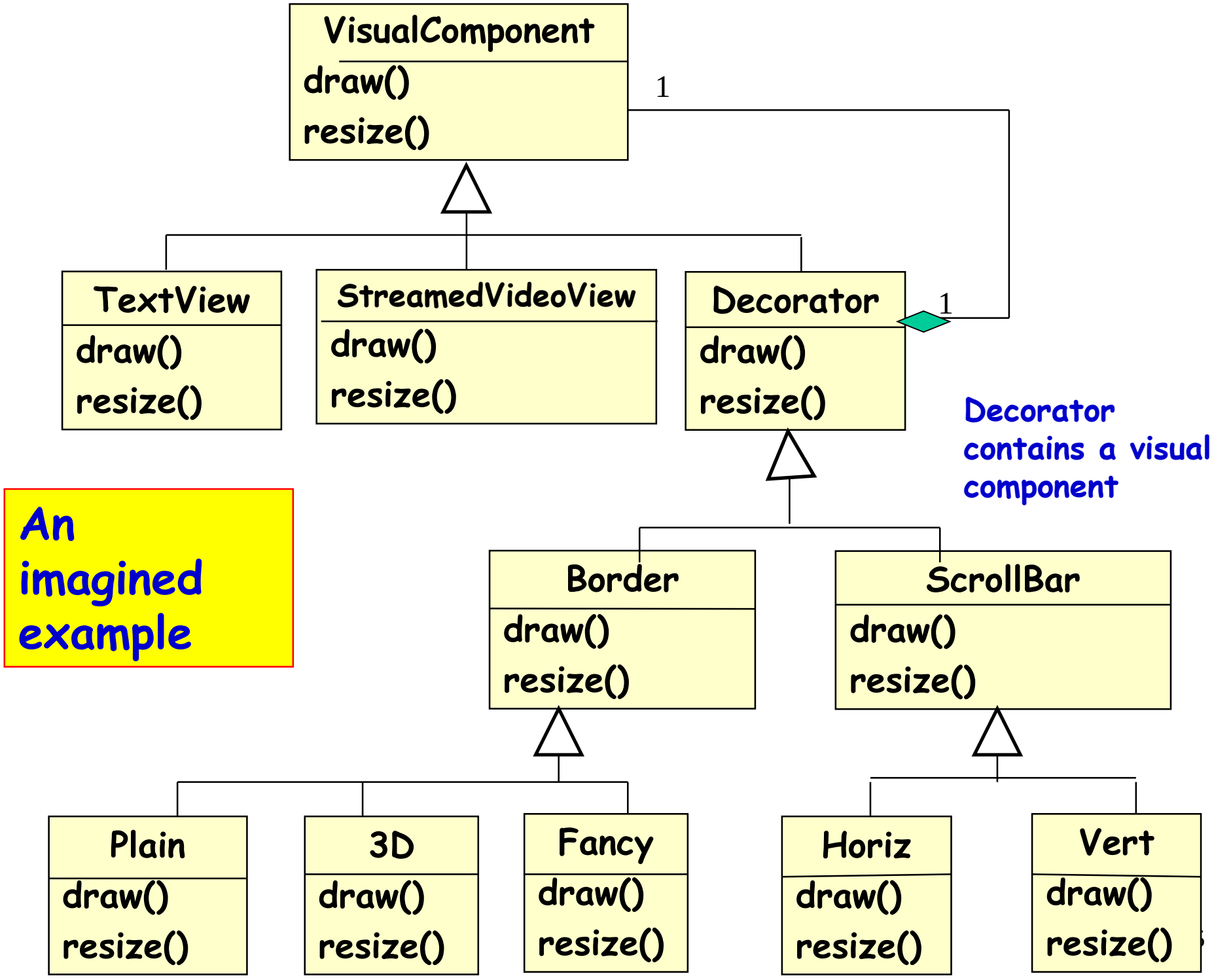
- **An inheritance solution would require 15 classes!**

That's a lot of classes!

1. TextView_Plain
2. TextView_Fancy
3. TextView_3D
4. TextView_Horizontal
5. TextView_Vertical
6. TextView_Horizontal_Vertical
7. TextView_Plain_Horizontal
8. TextView_Plain_Vertical
9. TextView_Plain_Horizontal_Vertical
10. TextView_3D_Horizontal
11. TextView_3D_Vertical
12. TextView_3D_Horizontal_Vertical
13. TextView_Fancy_Horizontal
14. TextView_Fancy_Vertical
15. TextView_Fancy_Horizontal_Vertical

A Simpler Solution

- **The component is contained in another object (decorator) that adds the border.**
- The decorator conforms to the interface of the component:
  - **So its presence is transparent to clients**
- The decorator forwards requests to the component:
  - May perform additional actions before or after forwarding.


VisualComponent
draw()
resize()
1
TextView
draw()
resize()
StreamedVideoView
draw()
resize()
Decorator
draw()
resize()
1
Decorator contains a visual component
An imagined example
Border
draw()
resize()
ScrollBar
draw()
resize()
Plain
draw()
resize()
3D
draw()
resize()
Fancy
draw()
resize()
Horiz
draw()
resize()
Vert
draw()
resize()

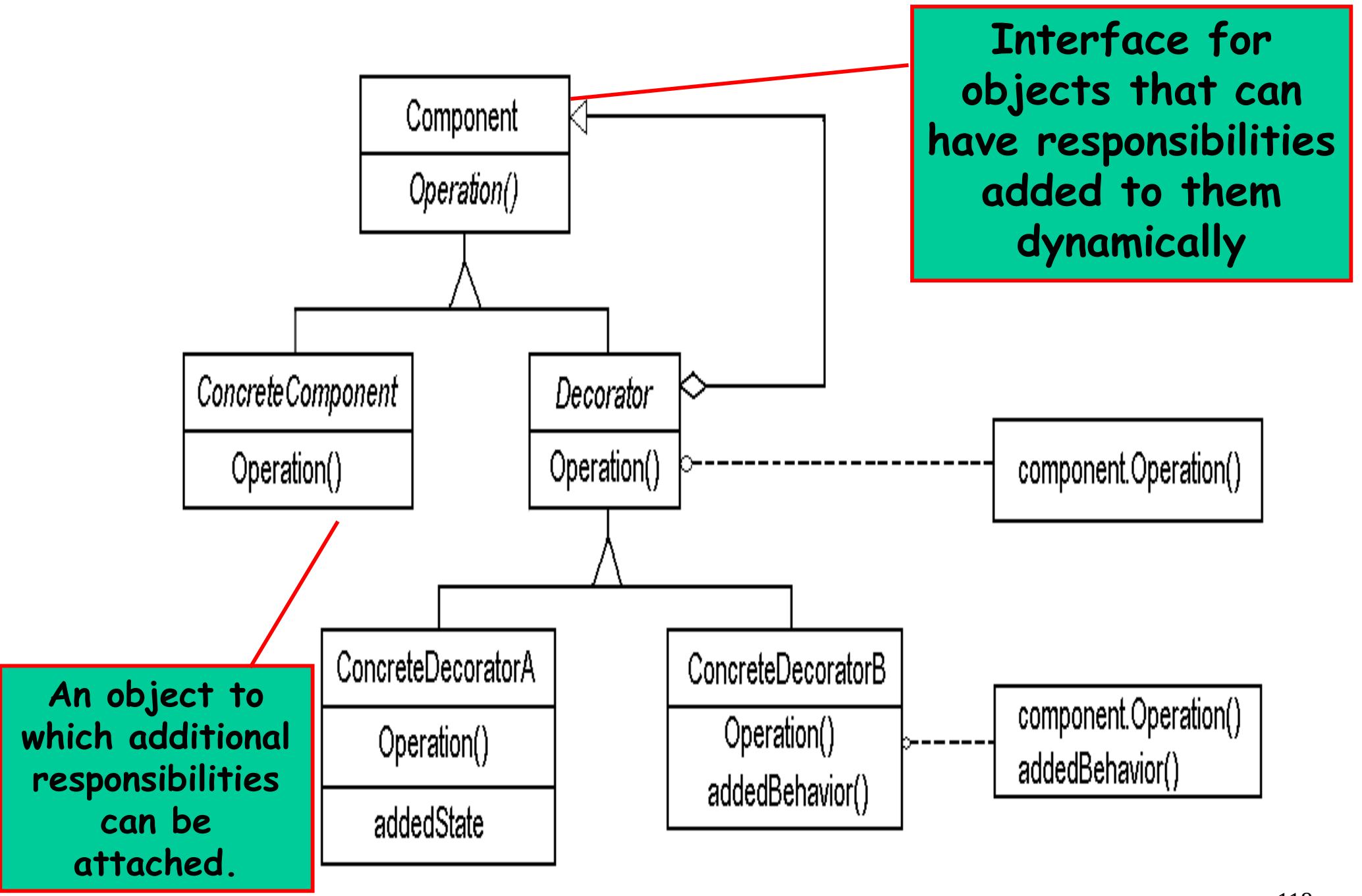
Disadvantages of Inheritance

- Use of inheritance leads to an explosion of classes …
- With another type of border added:
  - Many more classes would be needed with this design.
- If another view were added such as StreamedVideoView:
  - Double the number of Borders/Scrollbar classes
- **Use the Decorator Pattern instead!**

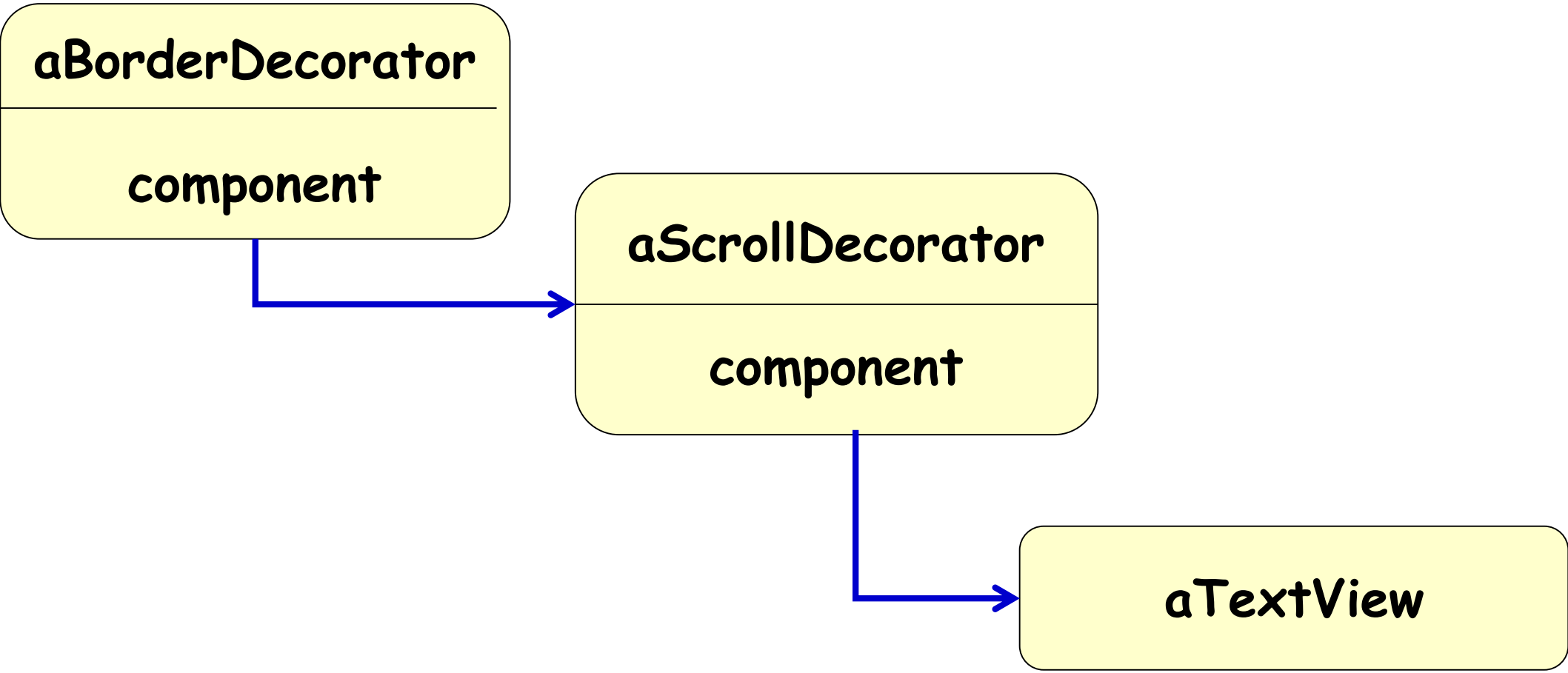
Decorator Disadvantages

- When tempted to add many decorators:
  - A package may become hard to understand…
  - **Like Java I/O streams!!!** 
- Solutions become complex:
  - A factory class may help

Decorator: Review



Decorator Example: Object Diagram



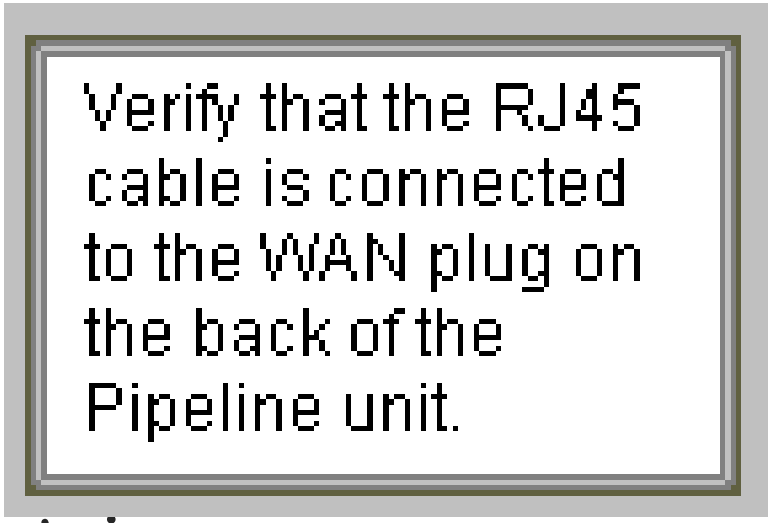
Java Borders

- Any JComponent can have 1 or more borders
- Borders are useful objects that, while not themselves components:
  - Know how to draw the edges of Swing components…
- **Borders are useful not only for drawing lines and fancy edges:**
  - **But also for providing titles and empty space around components**

JTextField, JTextArea

An input control for typing text values
(field = single line; area = multi-line)

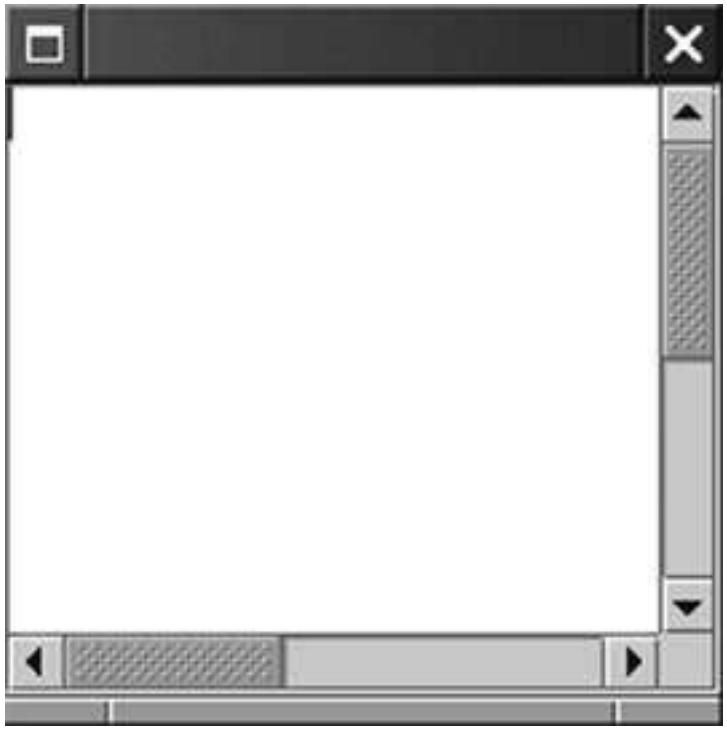
- **public JTextField(int columns)**
  public JTextArea(int lines, int columns)
  Creates a new field, the given number of letters wide.

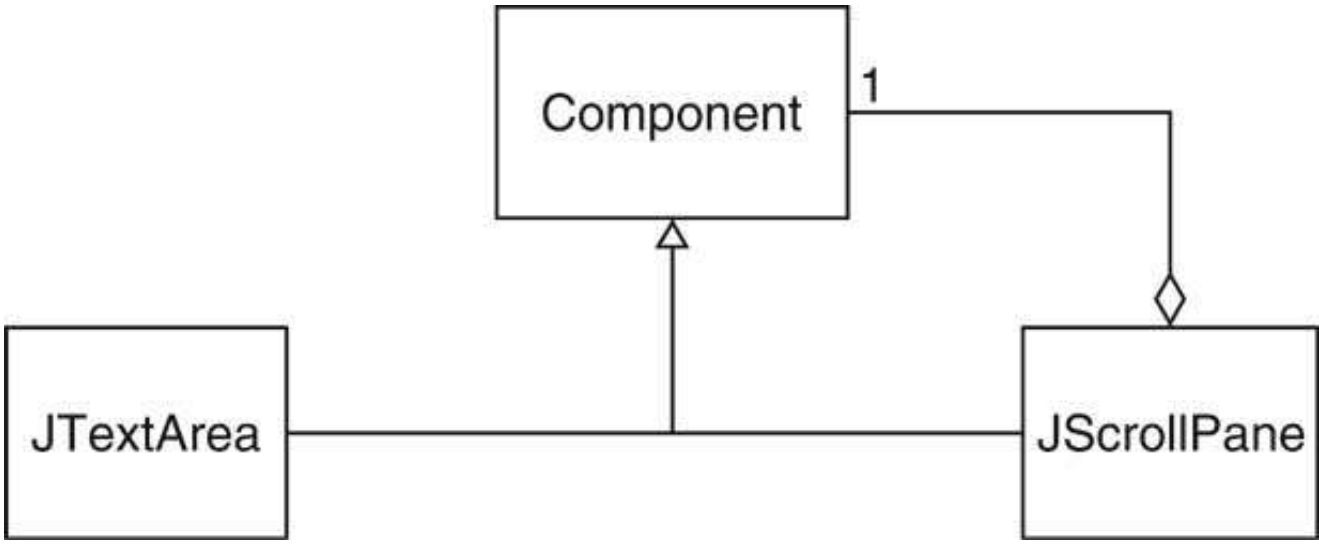


- **public String getText()**
  Returns the text currently in field.

- **public void setText(String text)**
  Sets field's text to be the given string.
  - *What if the text area is too big to fit in the window?*

Example: Scroll Bars

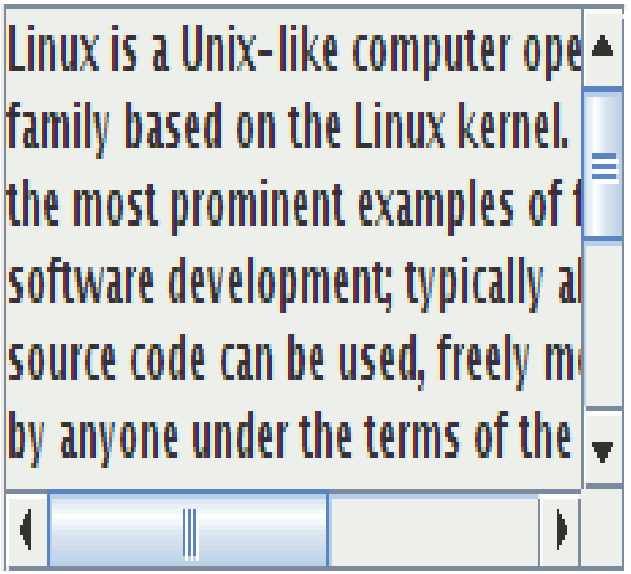




- Scroll bars can be attached to components
- Approach #1: Component class can turn on scroll bars
- Approach #2: Scroll bars can surround component
  JScrollPane pane = new JScrollPane(component);
- Swing uses approach #2
- **JScrollPane** is again a component

JScrollPane

A container that adds scrollbars around any other component



- **public JScrollPane(Component comp)**
  Wraps the given component with scrollbars.
  - After constructing the scroll pane, you must add the scroll pane, to the onscreen container:

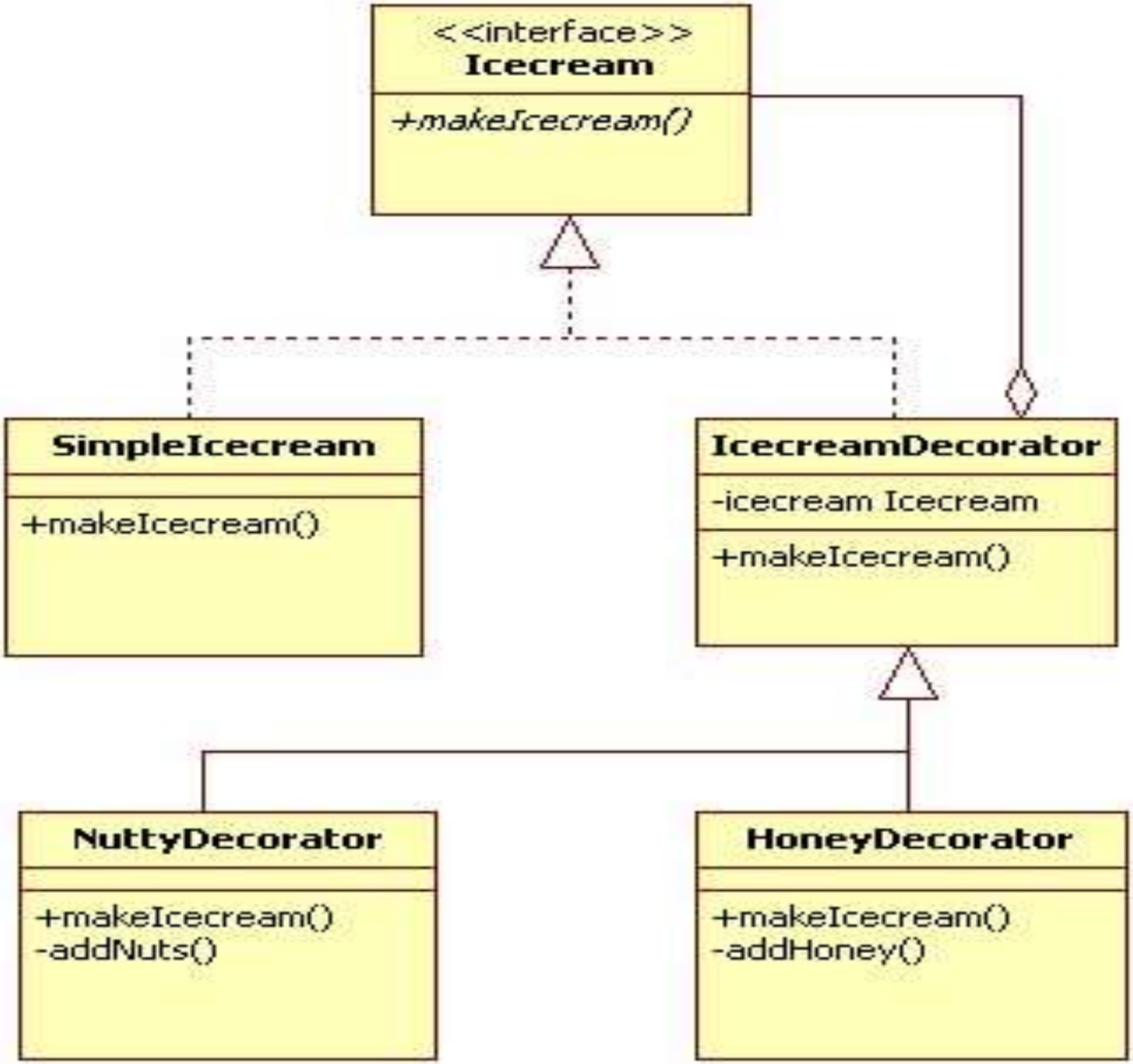
```
myContainer.add(new JScrollPane(textarea),
                BorderLayout.CENTER);
```

Quiz

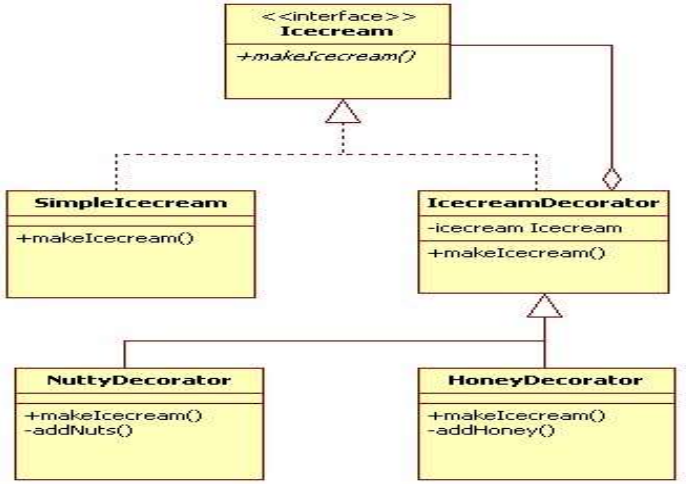
- An ice cream can be made with any of the following types of toppings in any combination and order:
  - **Nutty**
  - **Honey**
  - **Fruity**
  - **Chocolate**
  - **Vanilla**

1. **Draw class diagram**
2. **Write Java Code**

Quiz: Solution



Quiz: Java Code




```
public interface Icecream {
  public String makeIcecream();
}
```

```
public class SimpleIcecream implements Icecream {
public String makeIcecream() {
    return "Base Icecream";
  }
```

```
abstract class IcecreamDecorator implements Icecream {
  protected Icecream specialIcecream;
  public IcecreamDecorator(Icecream specialIcecream) {
    this.specialIcecream = specialIcecream;}
  public String makeIcecream() {
    return specialIcecream.makeIcecream();}
}
```

```
public class NuttyDecorator extends IcecreamDecorator {
  public NuttyDecorator(Icecream specialIcecream) {
    super(specialIcecream);  }
  public String makeIcecream() {
    return specialIcecream.makeIcecream() + addNuts();  }
  private String addNuts() {
    return " + crunchy nuts";}
}
```

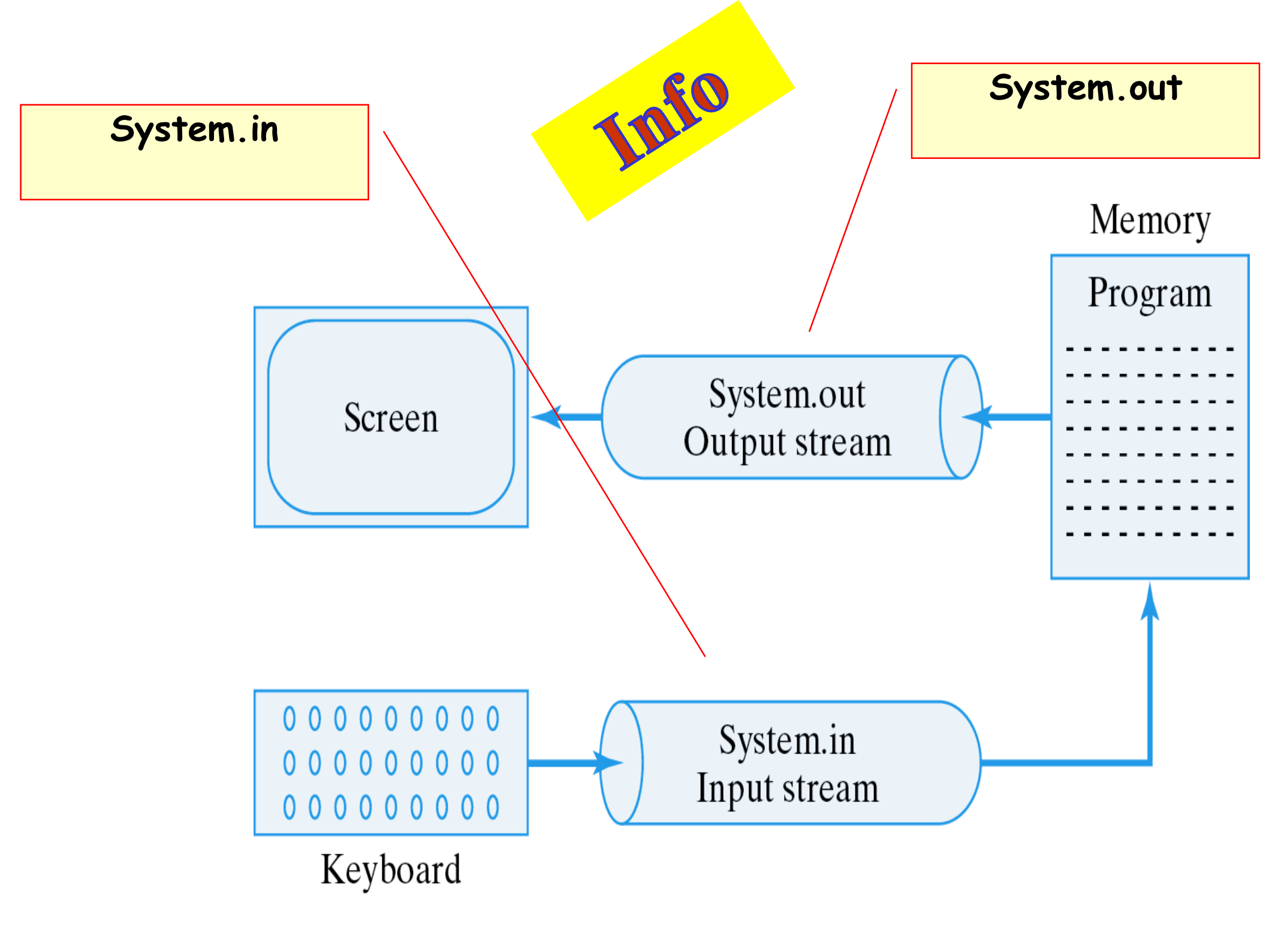
```
public class HoneyDecorator extends IcecreamDecorator {
    public HoneyDecorator(Icecream specialIcecream) {
        super(specialIcecream);  }
    public String makeIcecream() {
        return specialIcecream.makeIcecream() + addHoney();  }
    private String addHoney() {
        return " + sweet honey";}
  }
```

Making Sense of Stream Classes…

- What are **System.in.read()** , **System.out.print(), etc**… ?
- System class contains a variable called in -- an object created from a subclass of InputStream.
  - The period character after in states that read() belongs to **in**

```
public final class System extends Object;
```

- In other words:
  - read() is a method that belongs to an object called **in**, which in turn belongs to a class called System.

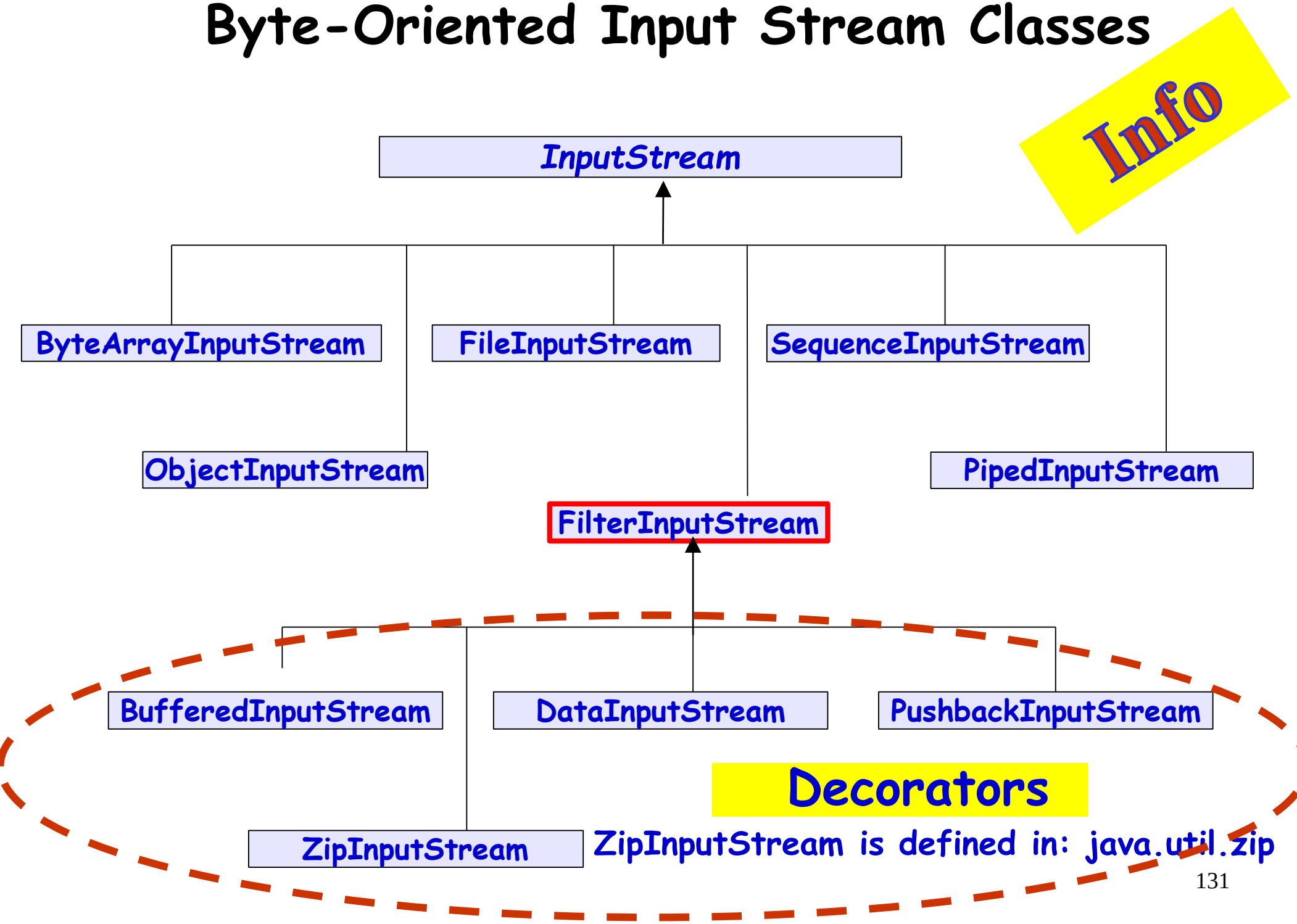

System.in
Info
System.out
Memory
Program
Screen
System.out
Output stream
System.in
Input stream
Keyboard

What are Streams?

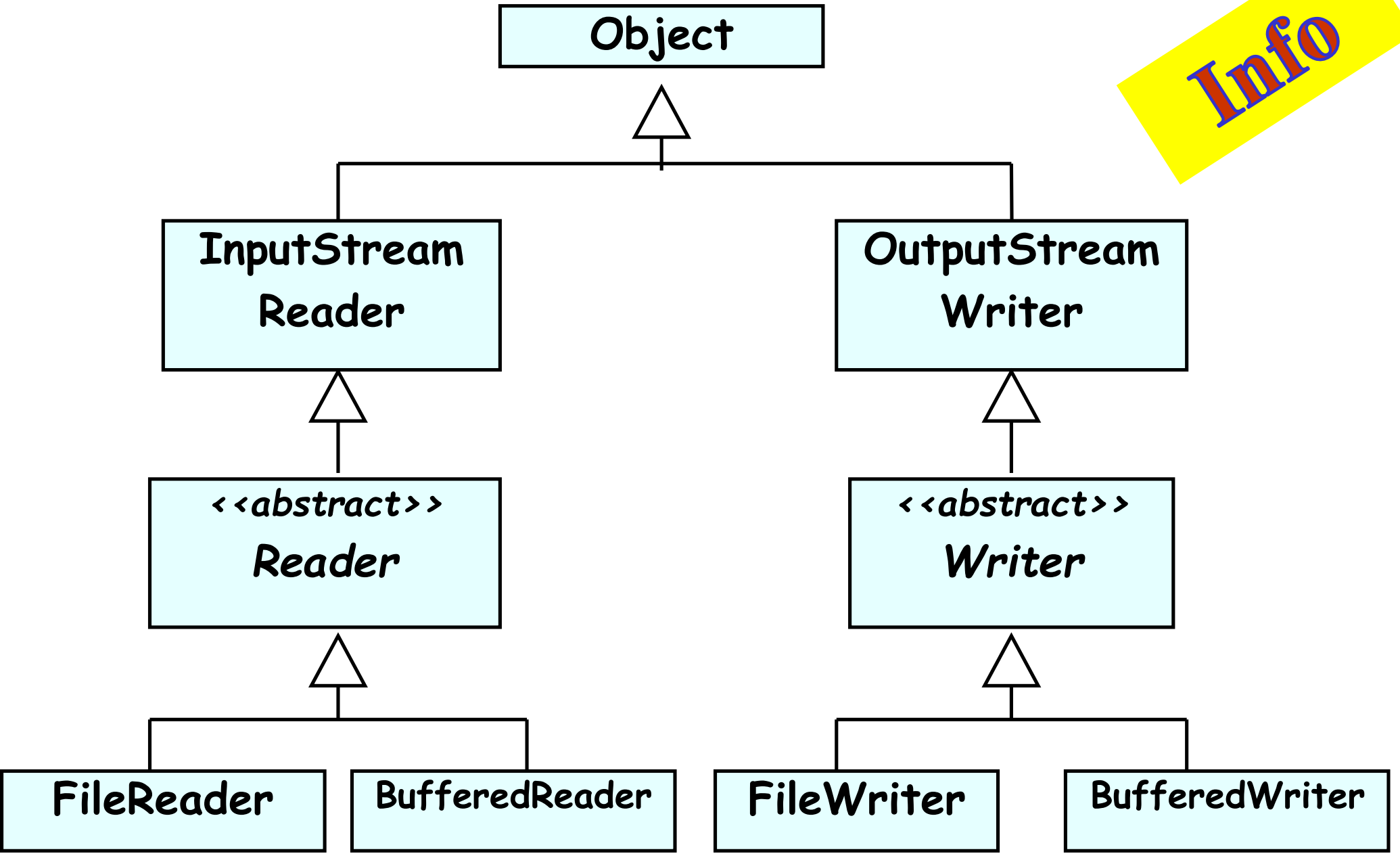
- The I/O System in Java is based on Streams
  - Input Streams are data sources
    - Programmers read data from input streams
  - Output Streams are data sinks
    - Programmers write data to output streams

- **Java has two main types of Streams**
  - **Byte Oriented**
    - Each datum is a byte
    - uses InputStream class hierarchy & OutputStream class hierarchy
  - **Character-based I/O streams**
    - each datum is a Unicode character
    - uses Reader & Writer class hierarchy

Byte-Oriented Input Stream Classes



Character-Oriented Stream Classes



Creating an InputStream

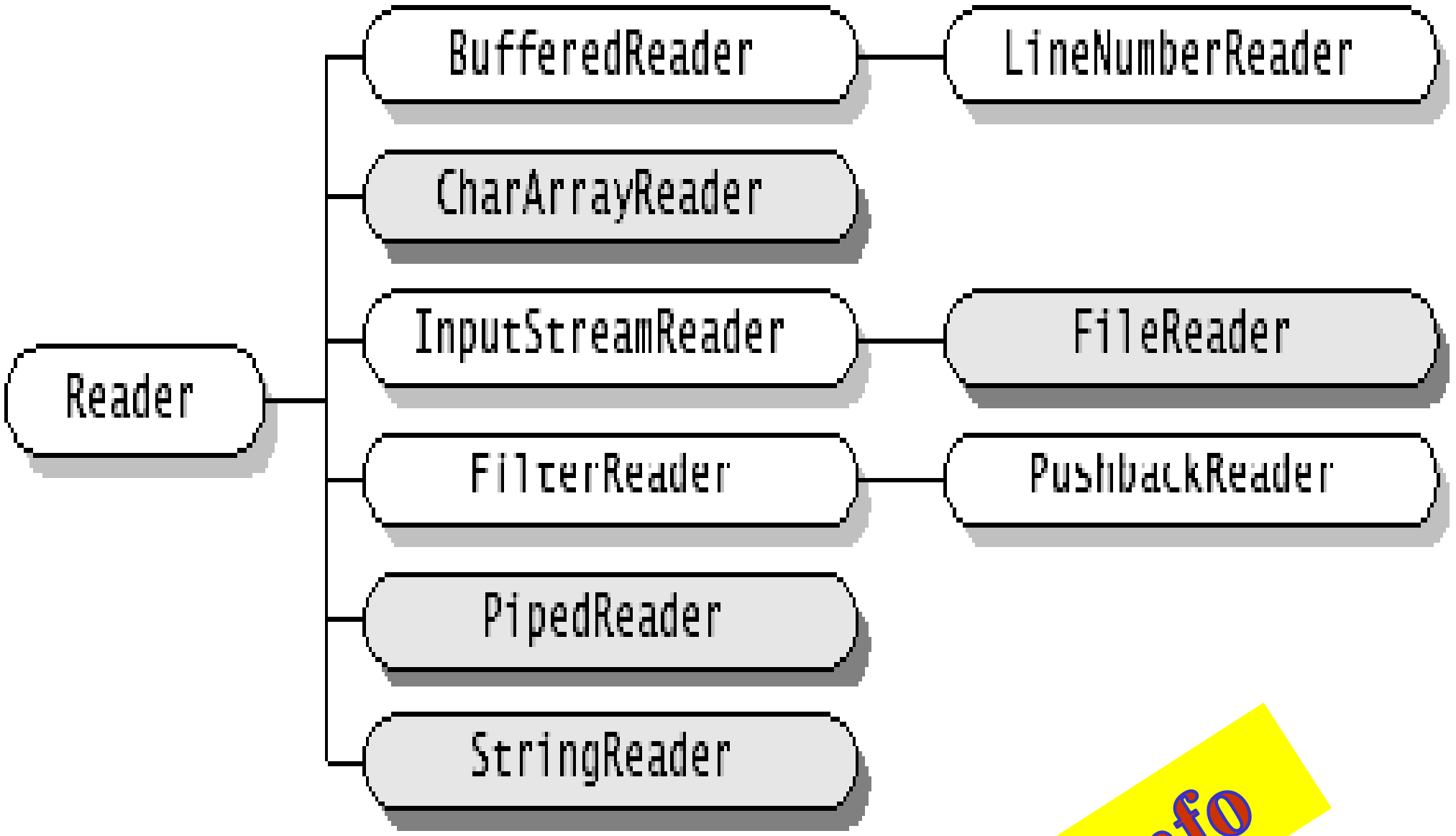
- **InputStream is an abstract class**
  - Programmers can only instantiate subclasses.
- **ByteArrayInputStream:**
  - Constructor is provided with a byte array.
  - This byte array contains all the bytes provided by this stream
- **FileInputStream:**
  - Constructor takes a filename or a FileDescriptor Object.
  - Opens a stream to a file.
- **FilterInputStream:**
  - Provides a basis for filtered input streams
  - **Our focus…**

Creating an InputStream

- **ObjectInputStream**
  - Created from another input stream (such as FileInputStream)
  - Reads bytes from the stream (which represent serialized Objects) and converts them back into Objects

- **PipedInputStream:**
  - Connects to an Instance of PipedOutputStream
  - A pipe represents a one-way stream through which 2 threads may communicate
    - Thread1 writes to a PipedOutputStream
    - Thread2 reads from the PipedInputStream

- **SequenceInputStream:**
  - Constructor takes multiple InputStreams
  - Allows reading. When one stream ends, it continues reading from next stream in the list


Reader
BufferedReader
LineNumberReader
CharArrayReader
InputStreamReader
FileReader
FilterReader
PushbackReader
PipedReader
StringReader

Info

Decorator Pattern in Java

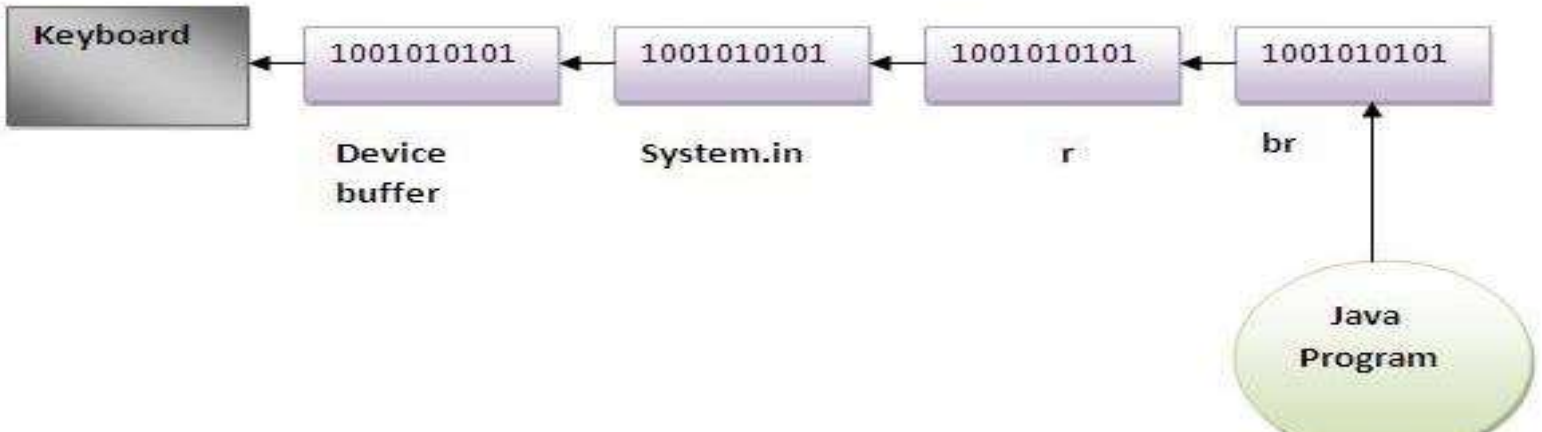
- **System.in is associated with keyboard input stream**
  - ... **InputStreamReader**: Reads bytes and translates them into characters using the specified char encoding.

- **BufferedReader**
  - Read text from a character-input stream, buffering characters so as to provide for efficient reading of characters, arrays, and lines.
- **Example:**

```
BufferedReader keyboard = new BufferedReader(new
InputStreamReader(System.in));
```

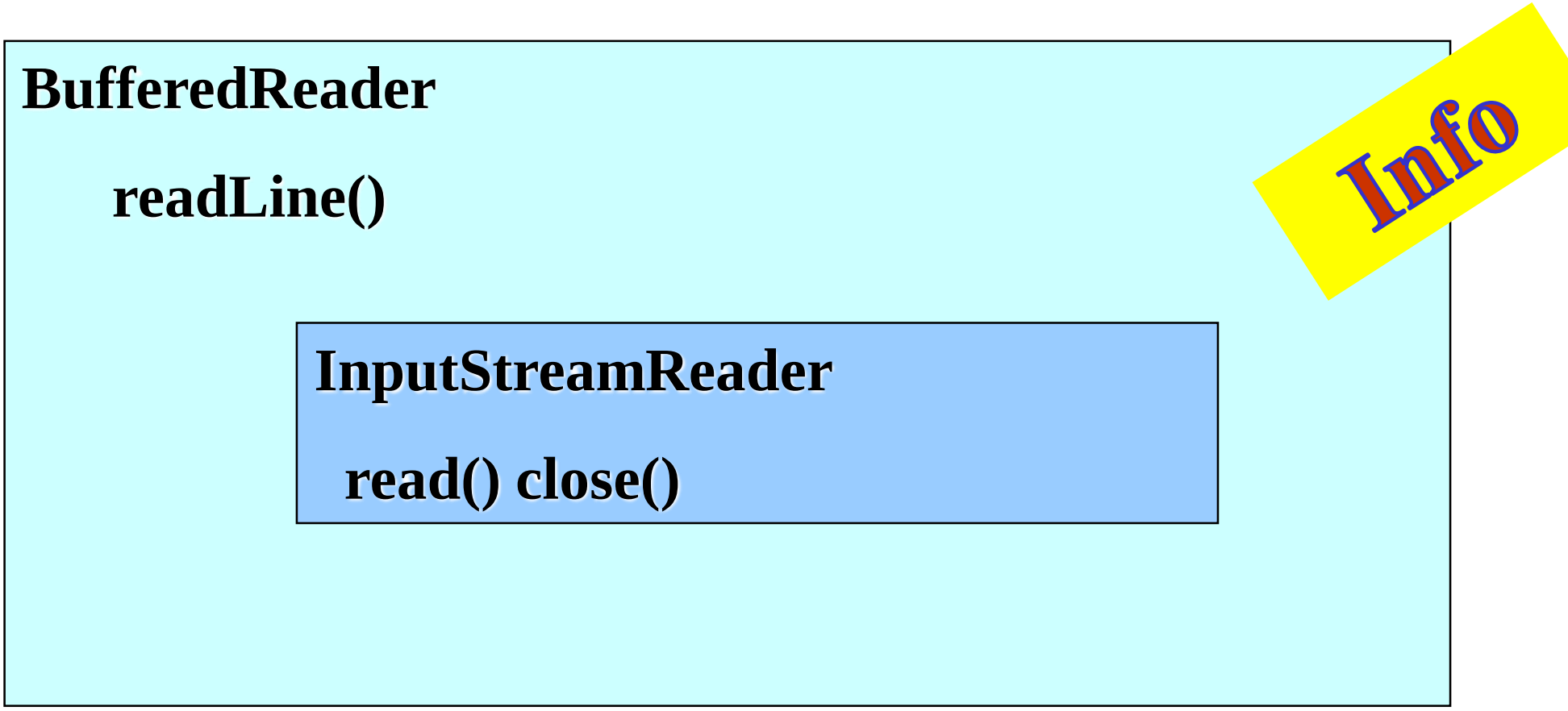
```
 import java.io.*;
class Test{
public static void main(String args[])throws Exception{
InputStreamReader r=new
                              InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
 }
}
```

Info



Example of decorator pattern use

BufferedReader *decorates* InputStreamReader



I/O Explanation

- Normal **InputStream** class has only **public int read()** method to read one letter at a time

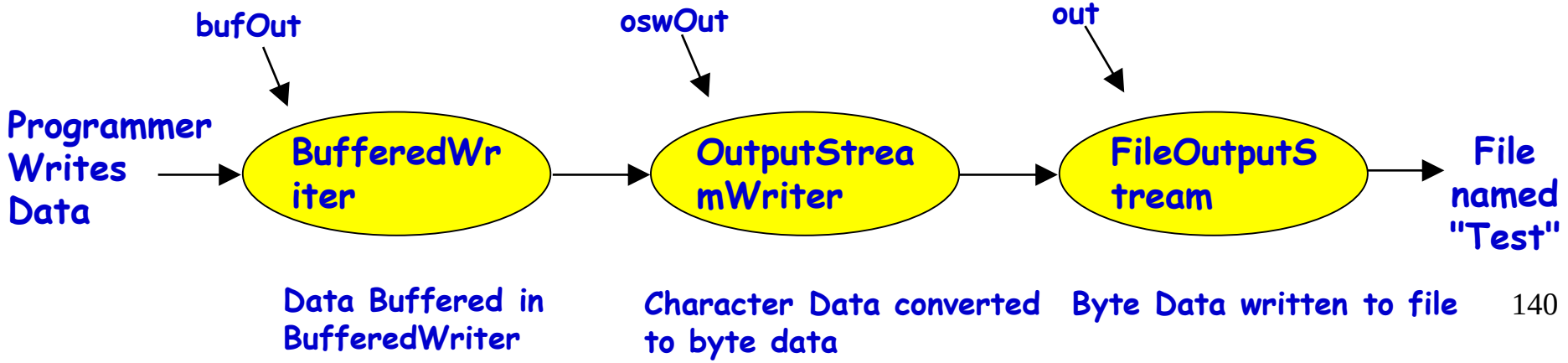
- Decorators such as **BufferedReader** add functionality to read the stream more easily

```
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);
// Thanks to decorator streams, can read an
// entire line from the file in one call else would read characters
// (InputStream only provides public int read() )
String wholeLine = br.readLine();
```

Filter Output Streams - Example

```
import java.io.*;

public class MyClass{
  public void test() {
        try {
              FileOutputStream out = new FileOutputStream("Test");
              OutputStreamWriter oswOut = new OutputStreamWriter(out);
              BufferedWriter bufOut = new BufferedWriter(oswOut);
              // programmer now uses bufOut…
        }
        catch (IOException x){
        }
  }
}
```

Info



Another Decorator Example

- We can decorate a **FileInputStream** with an **ObjectInputStream** :
  - So you can read objects that implement **Serializable Interface**

Java streams

- With > 60 stream decorators in Java, you can create a wide variety of input and output streams:
  - This provides flexibility -- **good**
  - It also adds complexity -- **bad**

Decorator: Pros

- **More flexible than static inheritance**:
  - Responsibilities can be added and removed at run-time
  - Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. **Inheriting from a Border class twice is error.**
- **Avoids inheriting from feature-laden classes.**
  - An application needn't pay for features it doesn't use.
  - It's also easy to define new kinds of Decorators independently from the classes of objects they extend.

Decorator: Cons

- **Lots of little objects:**
  - The objects differ only in the way they are interconnected, not in their class or in the value of their variables.
  - Although these are easy to customize by those who understand them, **they can be hard to learn and debug.**

Aggregation vs. inheritance

- **Both are ways to re-use functionality**
- **Inheritance:**
  - Re-use functionality of parent class
  - Statically decided
  - Weakens encapsulation
- **Aggregation:**
  - Re-use functionality of objects at run-time
  - Invoked through the base interface
  - Dynamic: multiple types with same interface
  - Black-box re-use

Aggregation vs. Inheritance cont…

- Inheritance is a quick and easy way to design new components:
  - These are variants of existing ones
- However, indiscriminate use of inheritance creates bloated hierarchies
  - Code is more difficult to maintain
  - **Unnecessary baggage for many classes**
- **Aggregation drawback:** it is harder to understand the behavior of a program by looking only at its source code…
  - **Semantics of interaction are decided at run-time**

- **Decorator: Consequences:**
  - + Responsibilities can be added/removed at run-time
  - + Avoids subclass explosion
  - + Recursive nesting allows attachment of multiple responsibilities
- **Implementation Issues:**
  - – Use lightweight classes as Decorators
  - – Heavyweight classes make Strategy pattern more attractive

Decorator: Final Comments

- The Decorator pattern gives you flexibility:
  - You can change a decorator at runtime, as opposed to having responsibility bound statically and determined at compile time by subclassing.
- Since a Decorator complies with the interface of the object that it contains:
  - The Decorator is indistinguishable from the object that it contains and from any other concrete instances, including other decorated objects.
- This is powerful and can be used to great advantage:
  - You can recursively nest decorators without any other objects being able to tell the difference, allowing significant customization.