Design Patterns (2)

CS580 Advanced Software Engineering
http://cs580.yusun.io
October 27, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

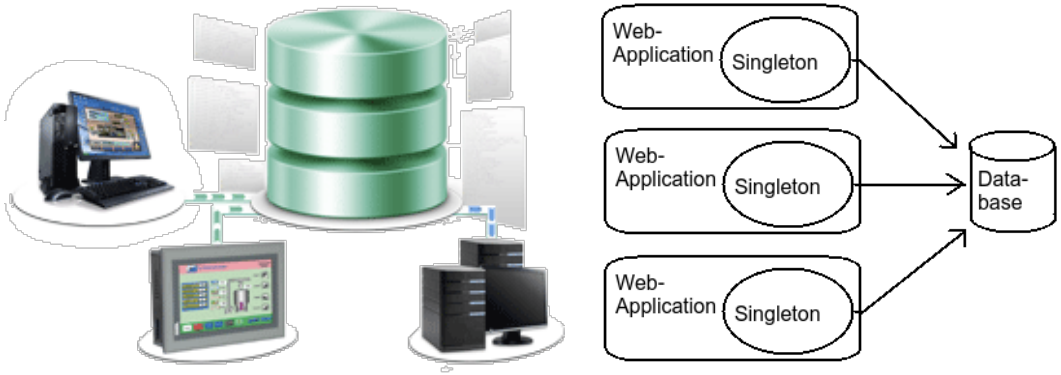
CAL POLY POMONA

Singleton Pattern

Singleton

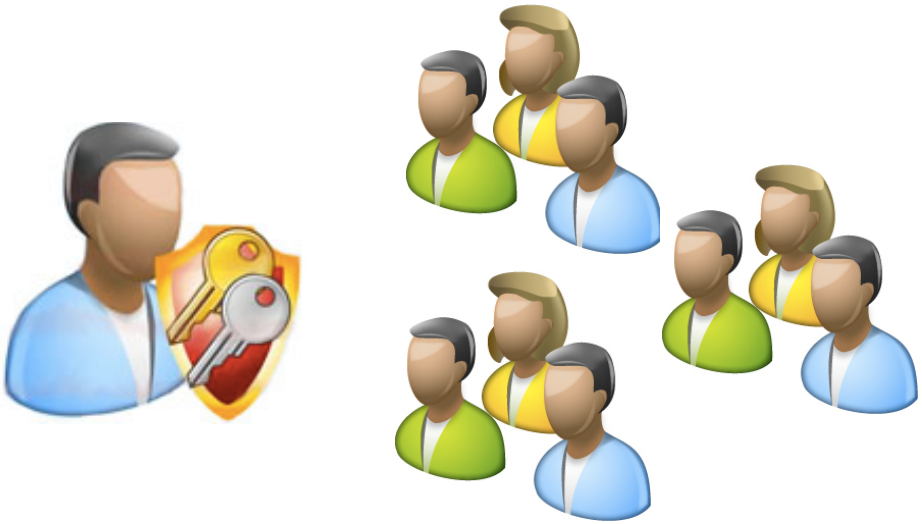
- Intent
  - Ensure a class has only one instance and provide a global point of access to it; class itself is responsible for sole instance
- Applicability
  - Want exactly one instance of a class
  - Accessible to clients from one point
  - Can also allow a countable number of instances
  - Global namespace provides a single object, but does not prevent other objects of the class from being instantiated

When do we need a Singleton?



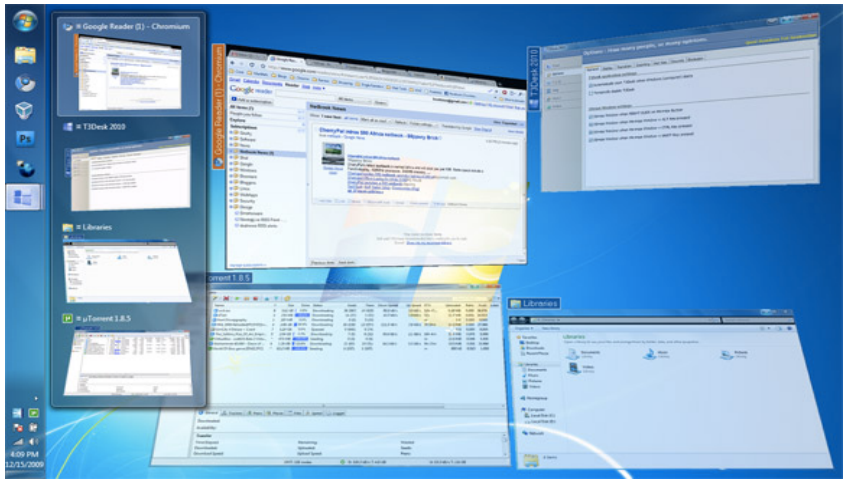
Database Connection

Camera API Object

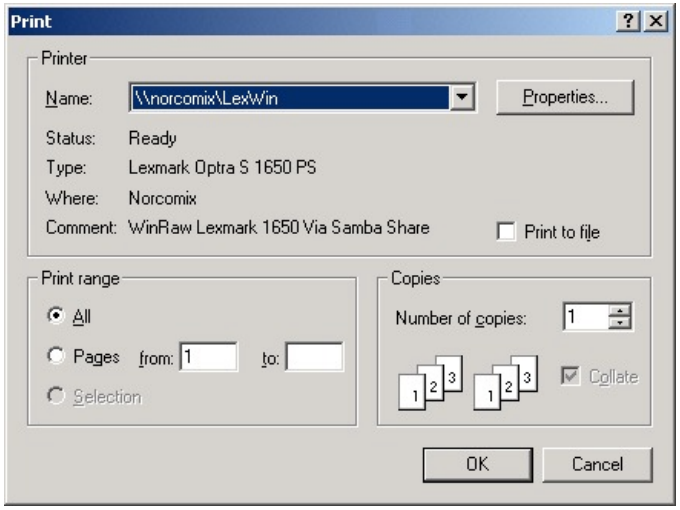


User Account Management

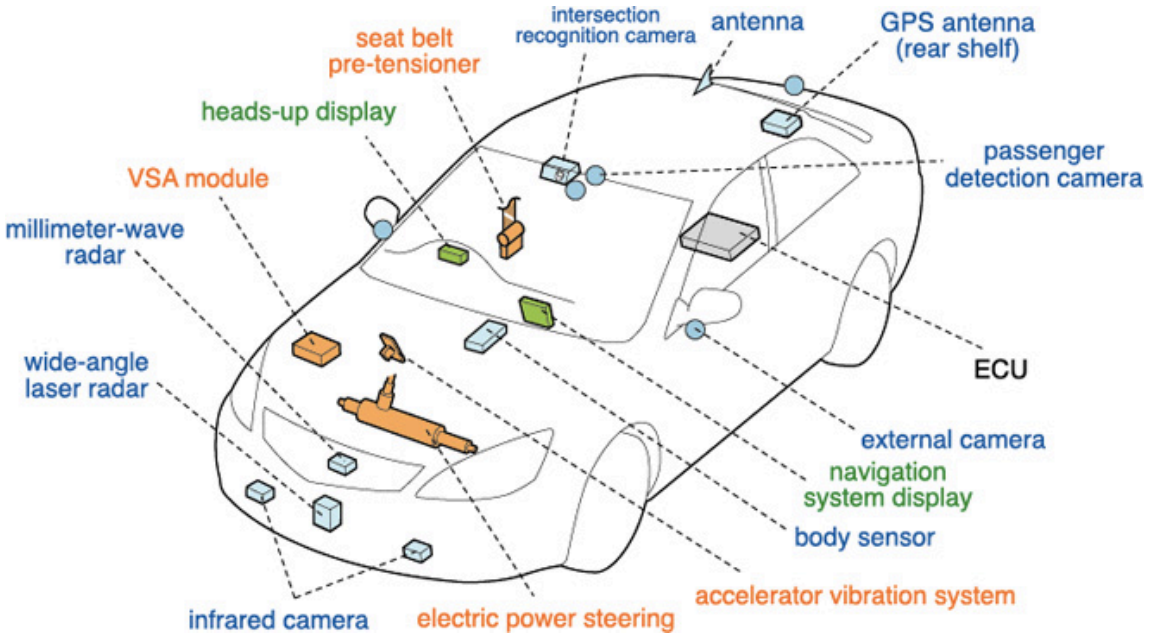
When do we need a Singleton?



Window Manager Object

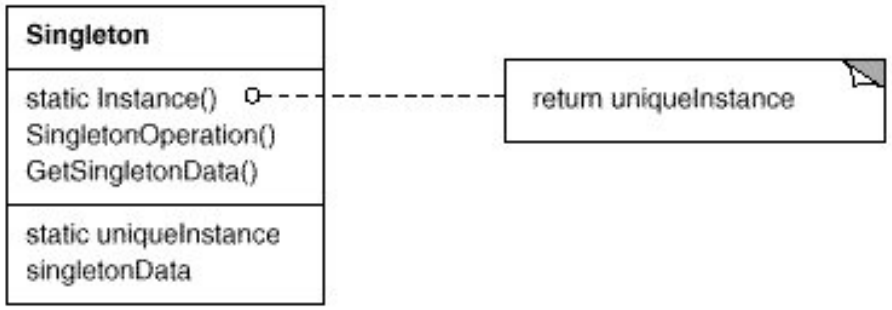


Printing Manager Object



Participants and Collaborations

- Singleton
  - Defines an `getInstance` method that becomes the single "gate" by which clients can access its unique instance.
    - `getInstance` is a class method (static method)
  - May be responsible for creating its own unique instance
  - Constructor placed in private/protected section
- Clients access Singleton instances solely through the `getInstance` method



Implementation: Ensuring a Unique Instance

```
public class Singleton {
  private static final Singleton instance = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
    return instance;
  }
}
```

Implementation: Lazy Instantiation

```
public class Singleton {
  private static Singleton instance = null;

  private Singleton() {}

  public static Singleton getInstance() {
    if(instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

What if there are subclasses?

```
public abstract class MazeFactory {

  private static MazeFactory instance = null;

  private MazeFactory() {}

  public static MazeFactory getInstance() {
    if (instance == null)
      return getInstance("enchanted"); // default instance
    else
      return instance;
  }

  public static MazeFactory getInstance(String name) {
    if(instance == null)
      if (name.equals(“bombed"))
        instance = new BombedMazeFactory();
      else if (name.equals(“enchanted"))
        instance = new EnchantedMazeFactory();

    return instance;
  }
}
```

Singleton with Subclasses

- Client code to create factory the first time

```
MazeFactory factory = MazeFactory.getInstance(“bombed");
```

- Client code to access the factory

```
MazeFactory factory = MazeFactory.getInstance();
```

- To add another subclass requires changing the instance() method!
- Constructors of BombedMazeFactory and EnchantedMazeFactory can not be private

Singleton with Subclasses (ver. 2)

```
public class EnchantedMazeFactory extends MazeFactory {

  private EnchantedMazeFactory() {}

  public static MazeFactory getInstance() {
    if(instance == null)
      instance = new EnchantedMazeFactory();

    return instance;
  }
}
```

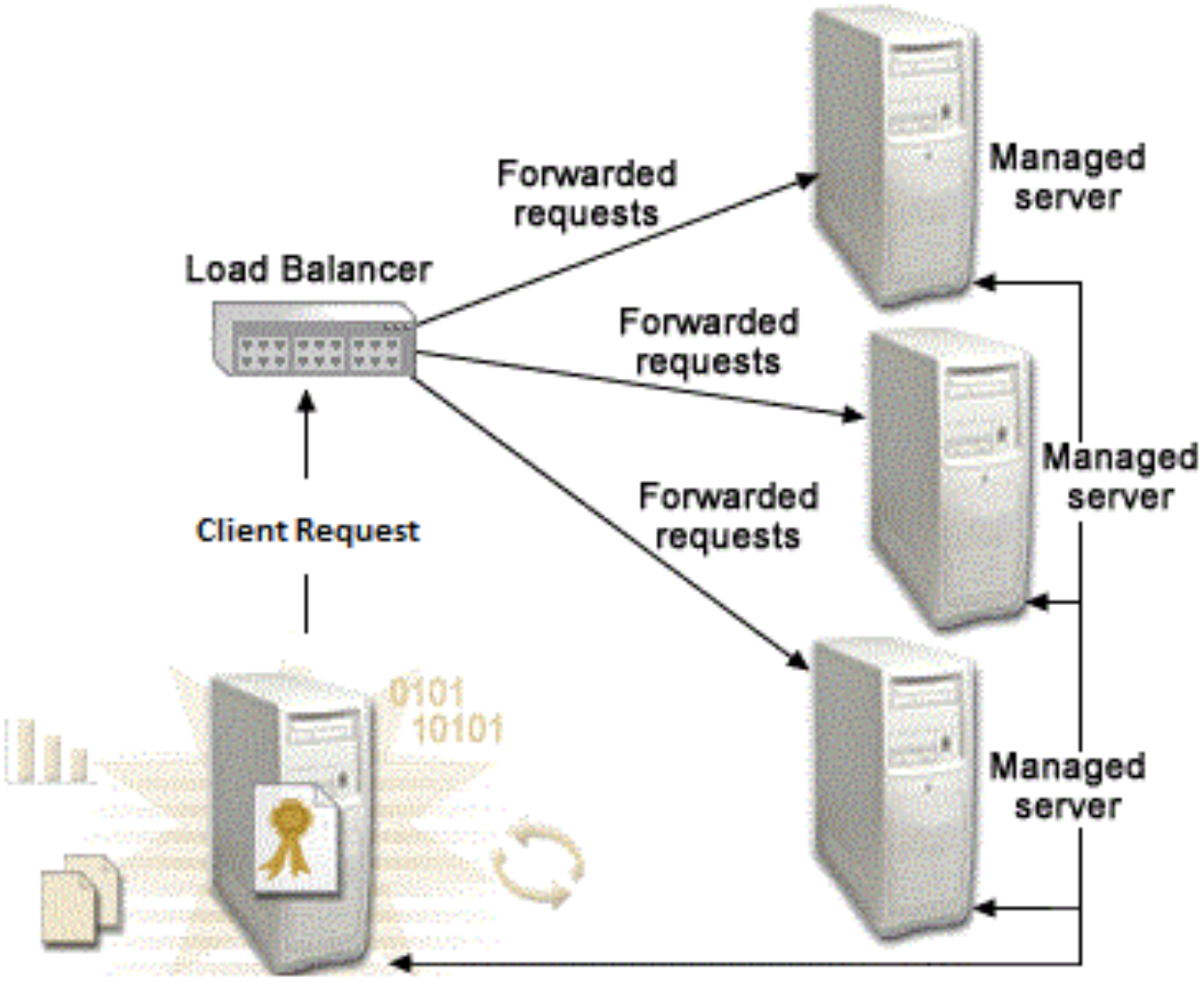
- Client code to create factory the first time

```
MazeFactory factory = EnchantedMazeFactory.getInstance();
```

- Client code to access the factory

```
MazeFactory factory = MazeFactory.getInstance();
```

Singleton Example – Load Balancer



Adapter Pattern

Adapter

- Problem
  - Have an object with an interface that's close to, but not exactly, what we need
- Context
  - Want to re-use an existing class
  - Can't change its interface
  - Impractical to extend class hierarchy more generally
    - May not have source code
- Solution
  - Wrap a particular class or object with the interface needed

Electrical Adapter…

Multi-Globe International Electrical Adapter - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Media

Address http://shop.store.yahoo.com/connectglobally/multiglobe.html Go Links

Google electrical adapter Search Web Search Site Page Info Up Highlight electrical adapter Norton AntiVirus

ELECTRICAL USB AC ADAPTERS Browse A Category

(888) 878 - 9327 24HRS

Multi-Globe International Electrical Adapter

Why buy multiple country specific adapters when you can have 1 small unit that covers them all for one low price? The Connect Globally Multi-Globe is a revolutionary electrical travel adapter that can be used in well over 140 countries worldwide.

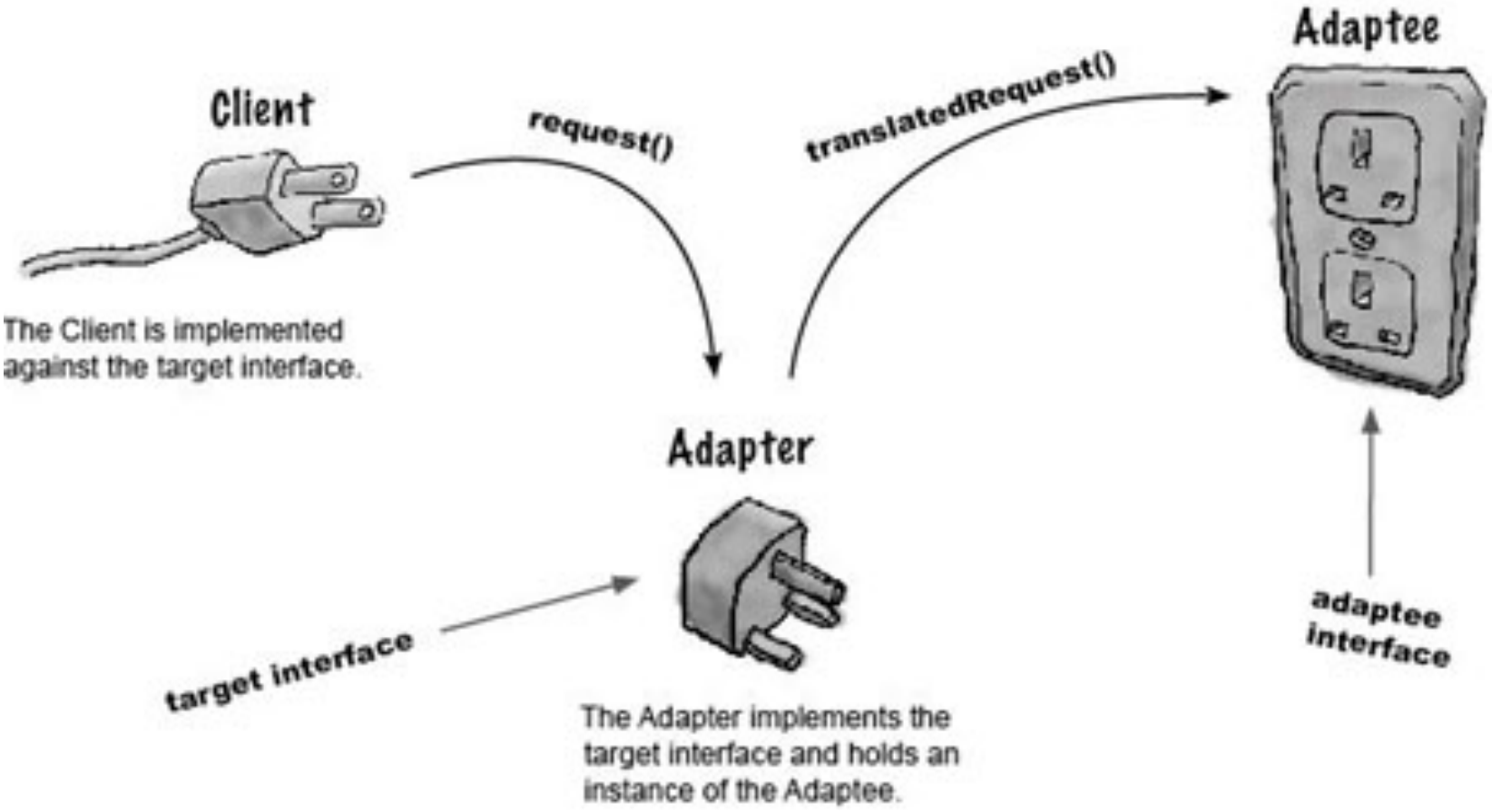
The Multi-Globe features a clever combination of fixed and swiveling pins ensuring the right connection every time. The unique Safe Connect sliding pin selector lets you plug into the desired country pin configuration and leaves the remaining pins. The lightweight and compact construction makes the Multi-Globe the one adapter you will not want to travel without. Designed in blue to catch your eye so you will never leave your Multi-Globe behind.

1. Safe Connect sliding pin selector only allows the pins that are plugged into the outlet to be live, leaving the remaining pins inactive.
2. Unique swiveling pins rotate 360º to achieve the correct Country specific outlet configuration.

For polarity specific cables and transformers a small adapter is required which fits neatly into the Safe Connect sliding pin selector. Item# CGEAGLOBE

http://shop.store.yahoo.com/connectglobally/acadaptors.html Internet

Electrical Adapter…



Some Additional Definitions

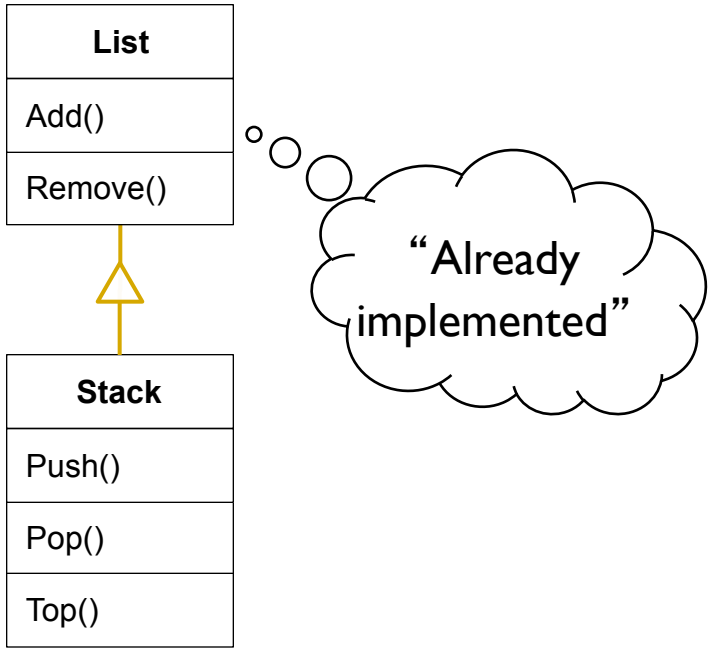
- Before we go into the adapter pattern let's review some motivation and introduce some terms

Reuse

- Main goal
  - Reuse knowledge from previous experience to current problem
  - Reuse functionality already available
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing components
- Inheritance (also called White Box Reuse)
  - New functionality is obtained by inheritance

Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation
- Problem with implementation inheritance
  - Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?



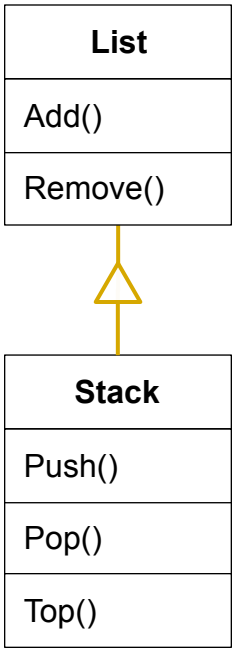
Delegation

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
  - Instead of "inheriting from" a class, we "delegate to" another object
- In Delegation, two objects are involved in handling a request
  - A receiving object delegates operations to its delegate
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object

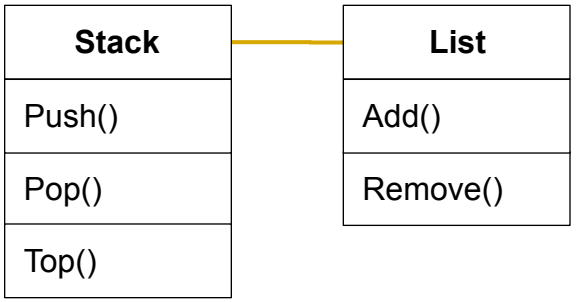
Delegation instead of Inheritance

- Delegation: Catching an operation and sending it to another object

Stack implemented by Inheritance



Stack implemented by Delegation




```
public class Stack {
  protected List delegatee;

  public Stack() {
    delegatee = new List();
  }

  public Object push(Object item) {
    delegatee.Add(item);
  }
  ...
```

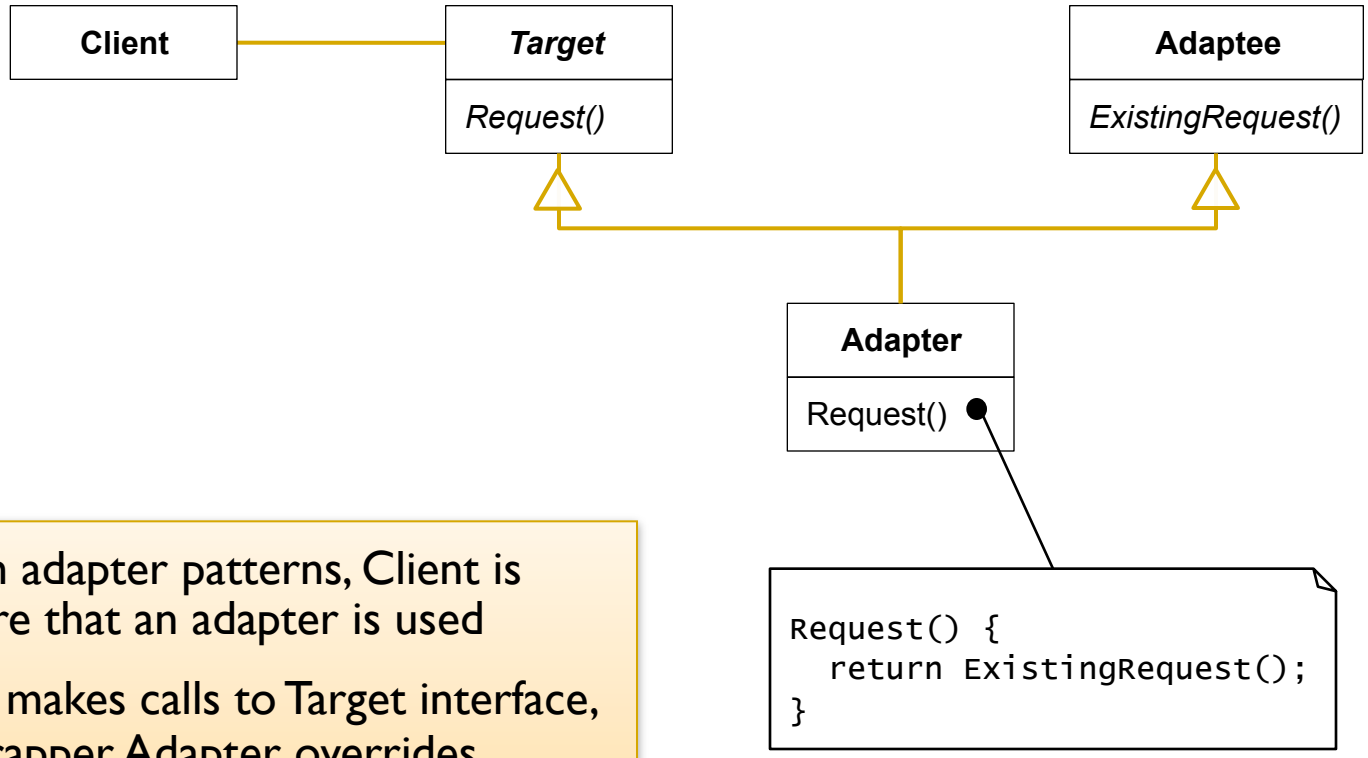
Key to Understanding

Many design patterns use a combination of inheritance and delegation

Adapter Pattern

- "Convert the interface of a class into another interface clients expect."
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering)
- Also known as a "wrapper"
- Two adapter patterns
  - Class adapter
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter
    - Uses single inheritance and delegation
- We will mostly use object adapters and call them simply adapters

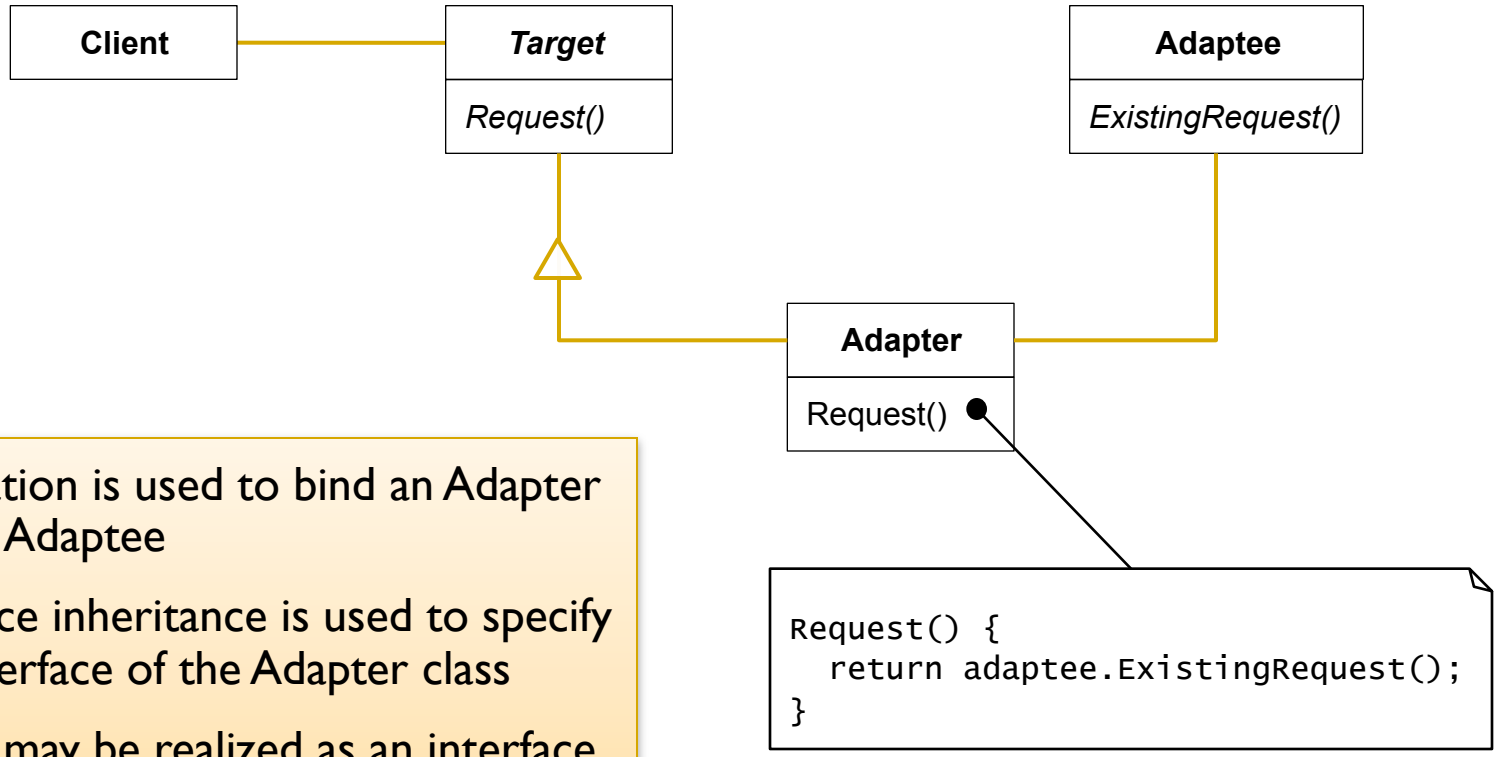
Class Adapter Pattern (Based on Multiple Inheritance)



In both adapter patterns, Client is unaware that an adapter is used

Simply makes calls to Target interface, and wrapper Adapter overrides Request with calls to legacy code

Adapter pattern (Object Adapter)



Delegation is used to bind an Adapter and an Adaptee

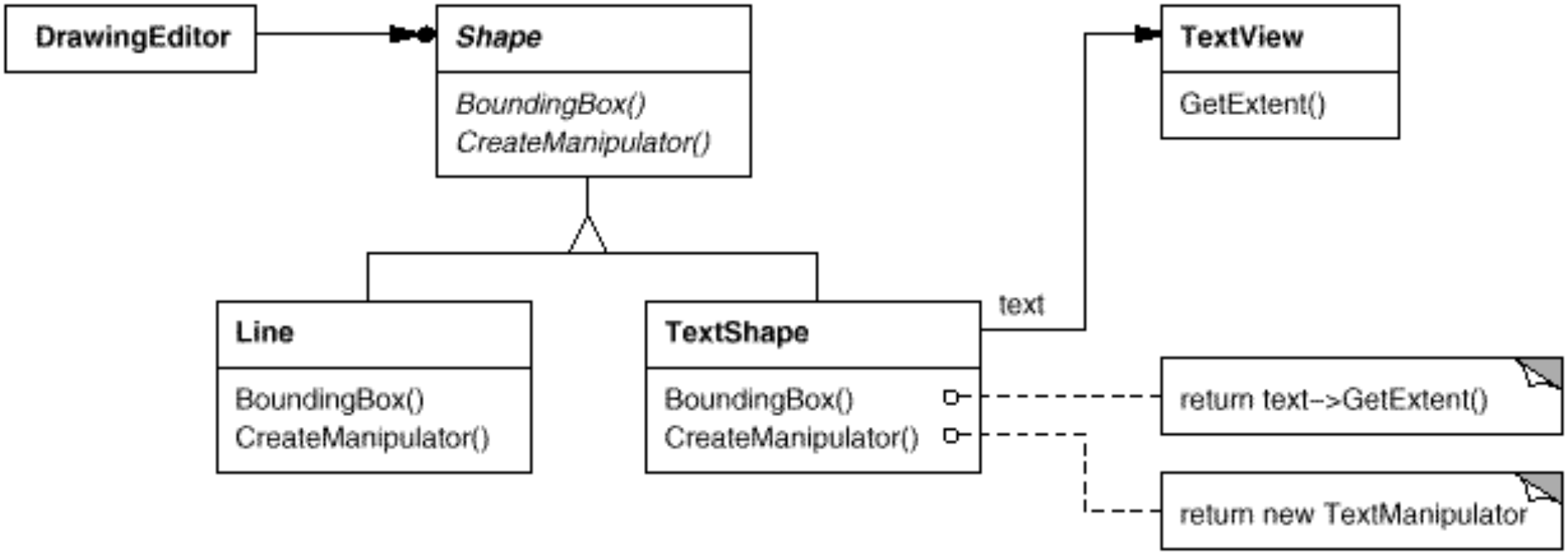
Interface inheritance is used to specify the interface of the Adapter class

Target may be realized as an interface in Java

Example of the Adapter Pattern

```
Interface SubmissionCounter {
   public boolean add(Object obj);
   public boolean addAll(Collection objs);
   public int getUniqueNumOfSubmissions();
   public int getNumOfAttemptedSubmissions();
}
```

Example of the Object Adapter Pattern



Class Shape and TextView

```
class Shape {
public:
  Shape();
  virtual void BoundingBox (Point& bottomLeft, Point& topRight);
  virtual Manipulator* CreateManipulator() const;
};
```

```
class TextView {
public:
  TextView();
  void GetOrigin(Coord& x, Coord& y);
  void GetExtent(Coord& width, Coord& height);
  virtual bool IsEmpty() const;
};
```

Class TextShape and Method BoundingBox

```
class TextShape : public Shape {
public:
  TextShape(TextView*);
  virtual void BoundingBox(Point& bottomLeft, Point& topRight);
  virtual bool IsEmpty();
  virtual Manipulator* CreateManipulator();
private:
  TextView* text;
};
```

```
void TextShape::BoundingBox(Point& bottomLeft, Point& topRight) {
  Coord bottom, left, width, height;
  text->GetOrigin(bottom, left);
  text->GetExtent(width, height);
  bottomLeft = Point(bottom, left);
  topRight = Point(bottom-height, left+width);
}
```

Adapter Summary

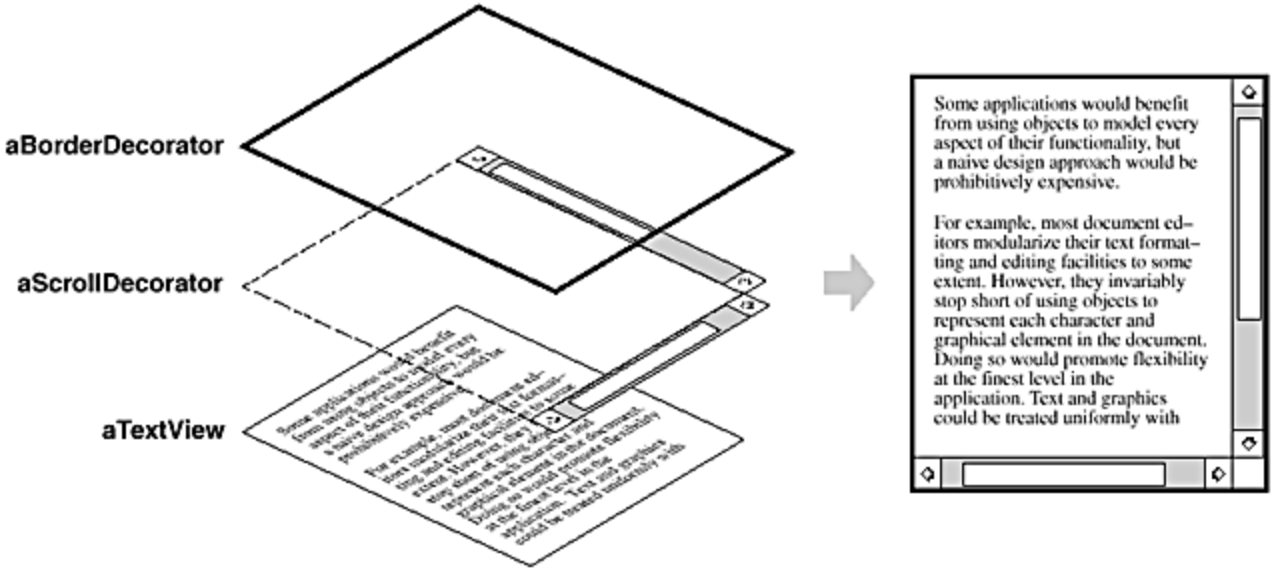
- Adapters are all about interface mapping between two artifacts
- Often, the goal is to find a "narrow" interface for Adaptee; that is, the smallest subset of operations that lets us do the adaptation
- Pay attention to Class Adapter (Inheritance)!

Decorator Pattern

Decorator

- *Intent*
  - Dynamically attach additional responsibilities to an object
  - Provide a flexible alternative to subclassing (static)
  - Decorating object is transparent to the core component
- *Also Known As* – Wrapper

Motivation



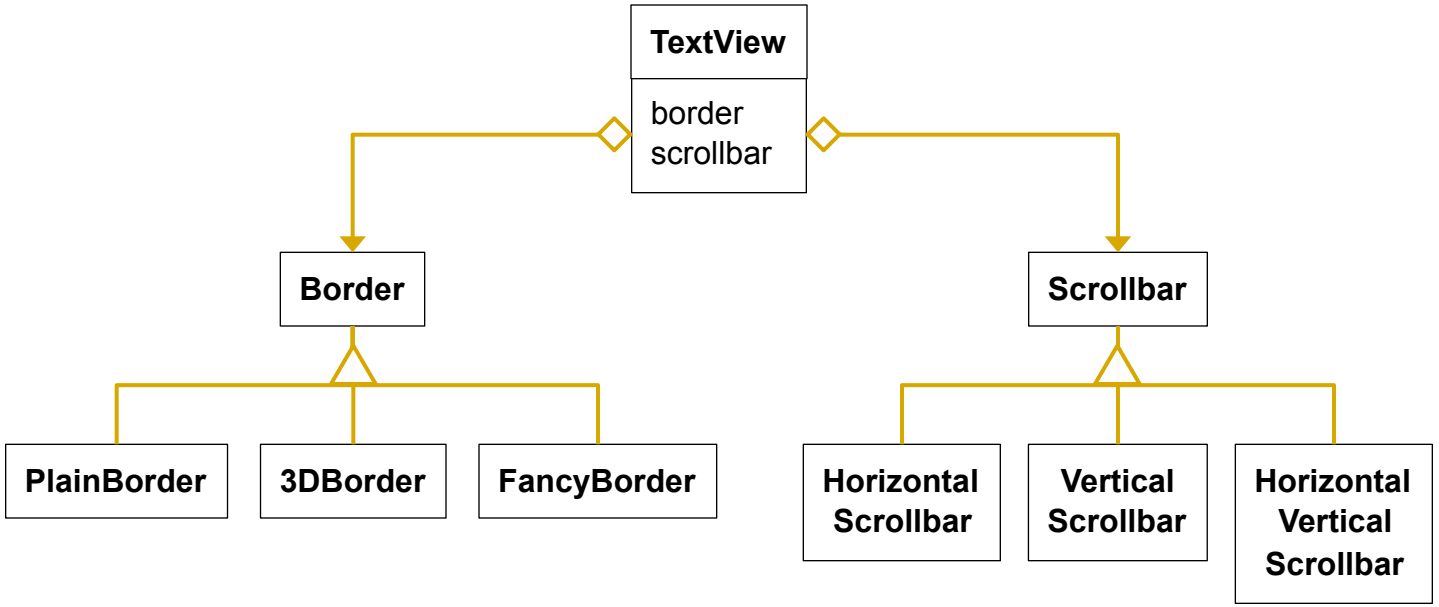
- We want to add different kinds of borders and/or scrollbars to a TextView GUI component
- *Borders* – Plain, 3D, or Fancy
- *Scrollbars* – Horizontal and/or Vertical

Motivation

- An inheritance solution requires 15 subclasses to represent each type of view

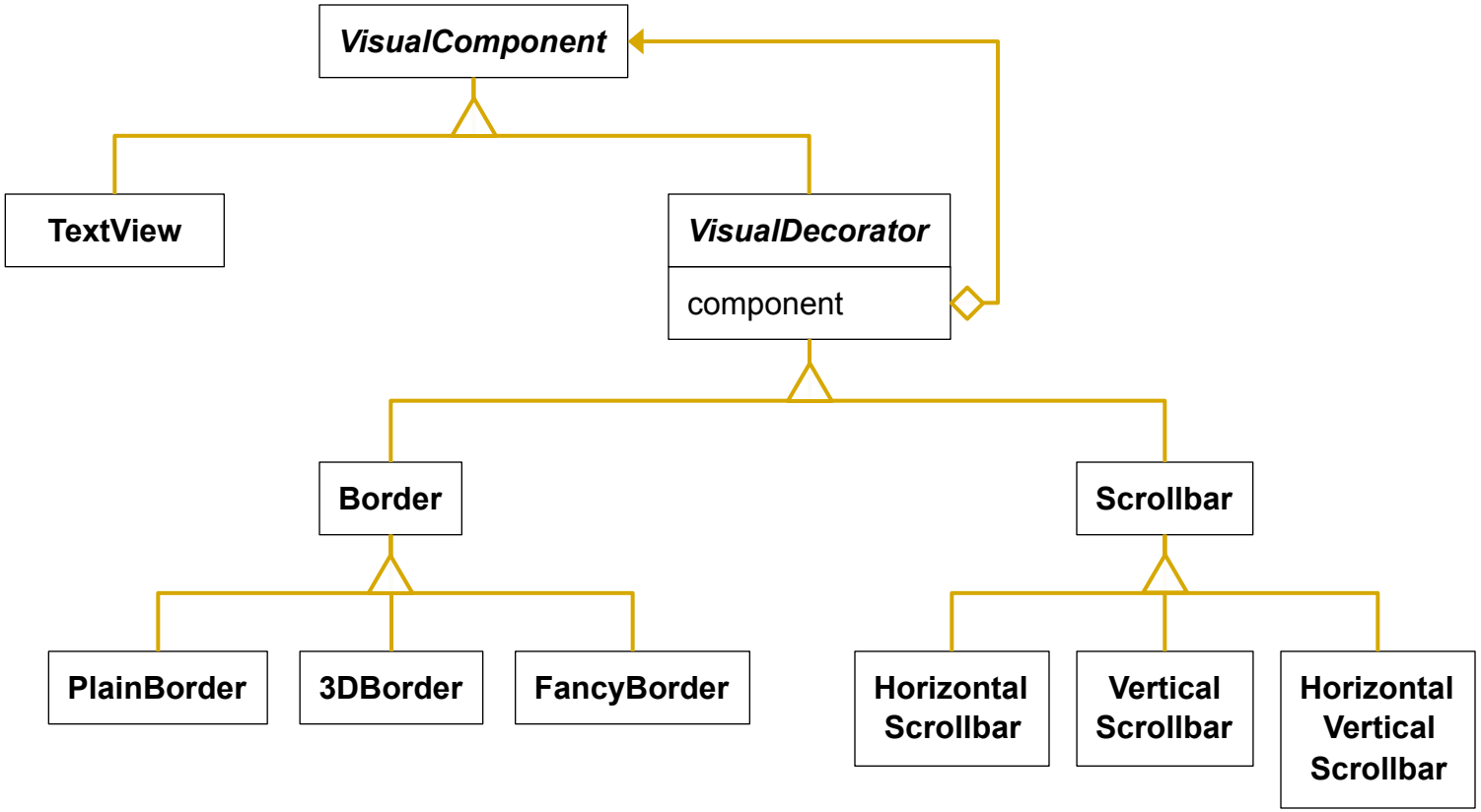
1. TextView-Plain
2. TextView-3D
3. TextView-Fancy
4. TextView-Horizontal
5. TextView-Vertical
6. TextView-Horizontal-Vertical
7. TextView-Plain-Horizontal
8. TextView-Plain-Vertical
9. TextView-Plain-Horizontal-Vertical
10. TextView-3D-Horizontal
11. TextView-3D-Vertical
12. TextView-3D-Horizontal-Vertical
13. TextView-Fancy-Horizontal
14. TextView-Fancy-Vertical
15. TextView-Fancy-Horizontal-Vertical

Solution 1: Use Object Composition



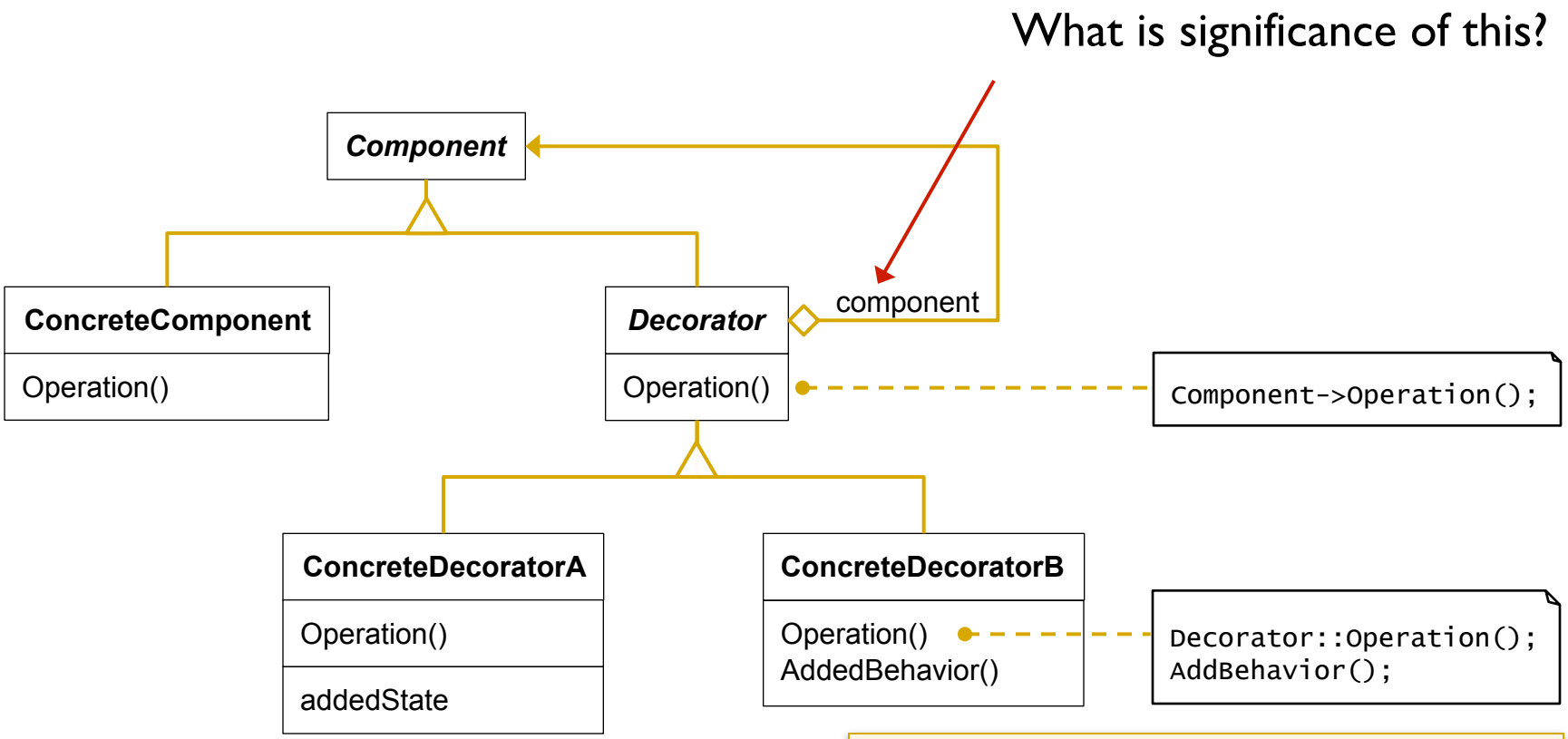
- Is it Open-Closed?
- Can you add new features without affecting TextView?
  - e.g., what about adding sound to a TextView?

Decorator Pattern Solution



- Change the Skin, not the Guts!
- TextView has no borders or scrollbars!
- Add borders and scrollbars on top of a TextView

Structure



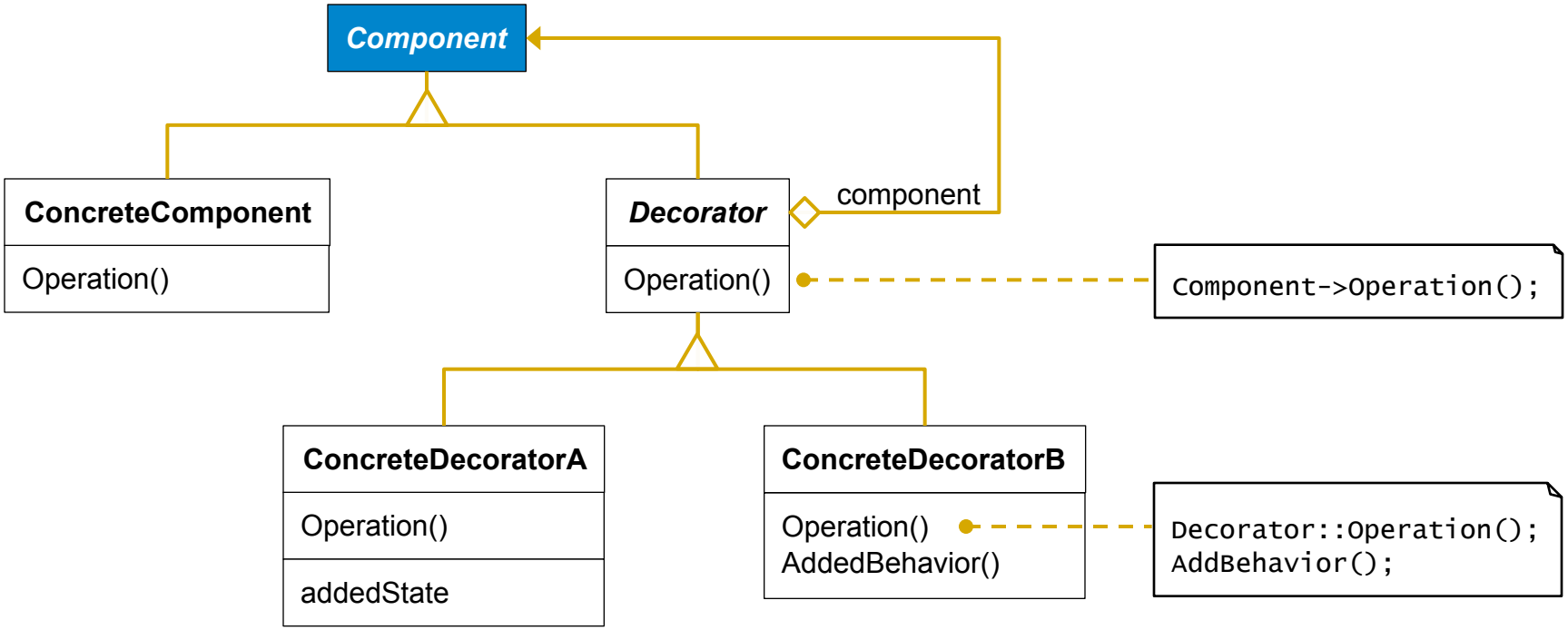
The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after any forwarding

Decorator

- *Applicability*
  - Dynamically and transparently attach responsibilities to objects
  - Responsibilities that can be withdrawn
  - Extension by subclassing is impractical
    - May lead to too many subclasses

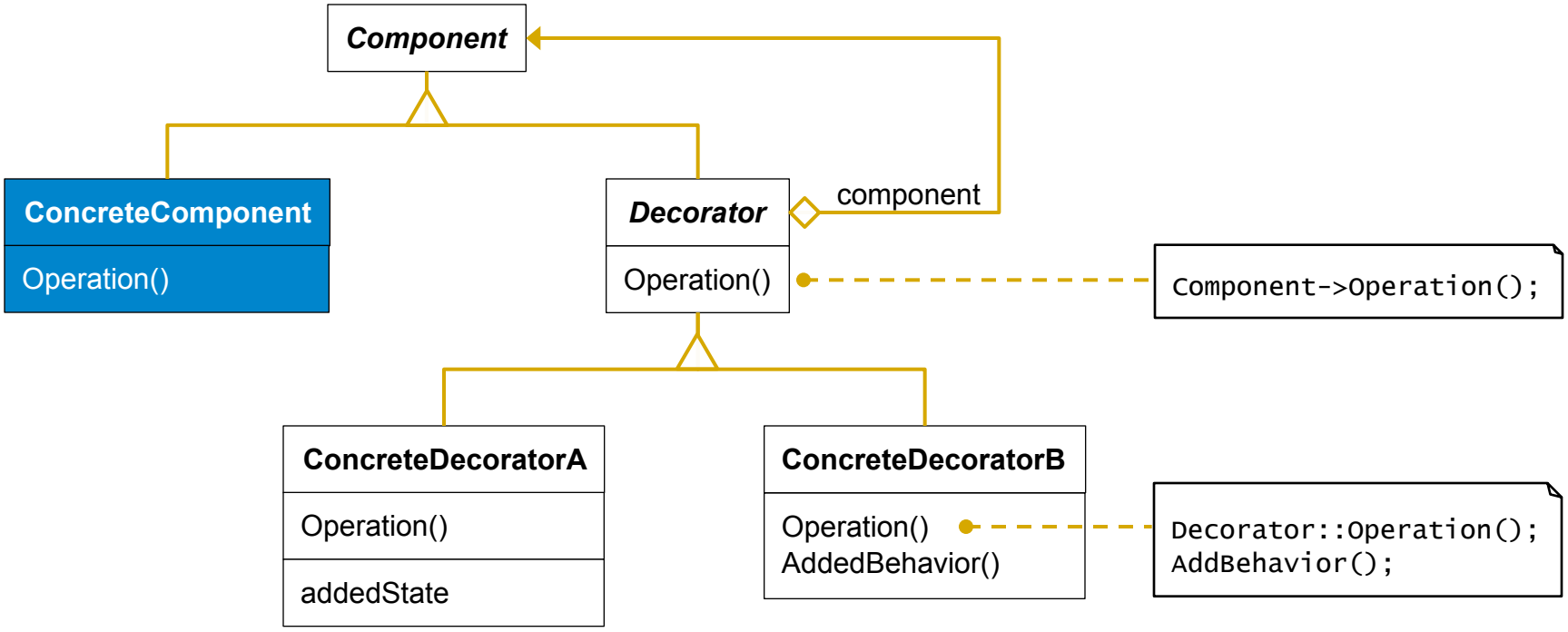
Component

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



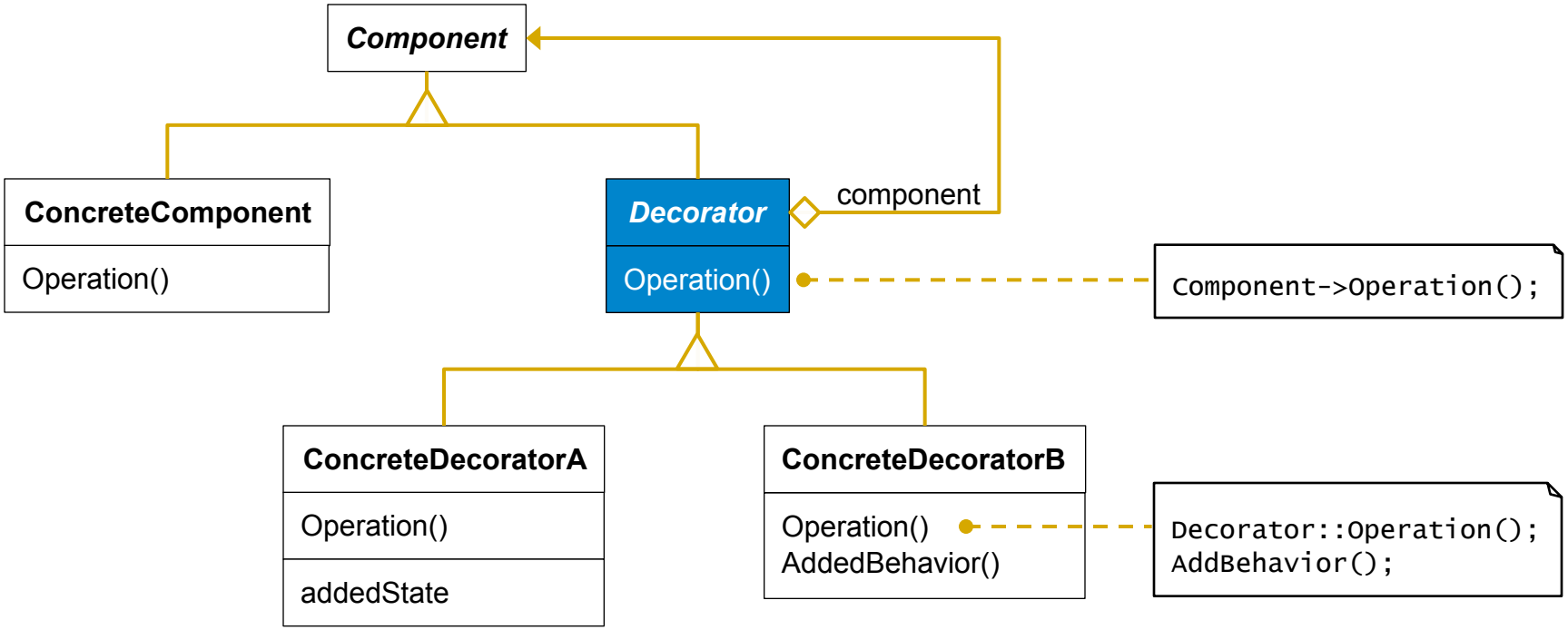
ConcreteComponent

- The "base" object to which additional responsibilities can be added



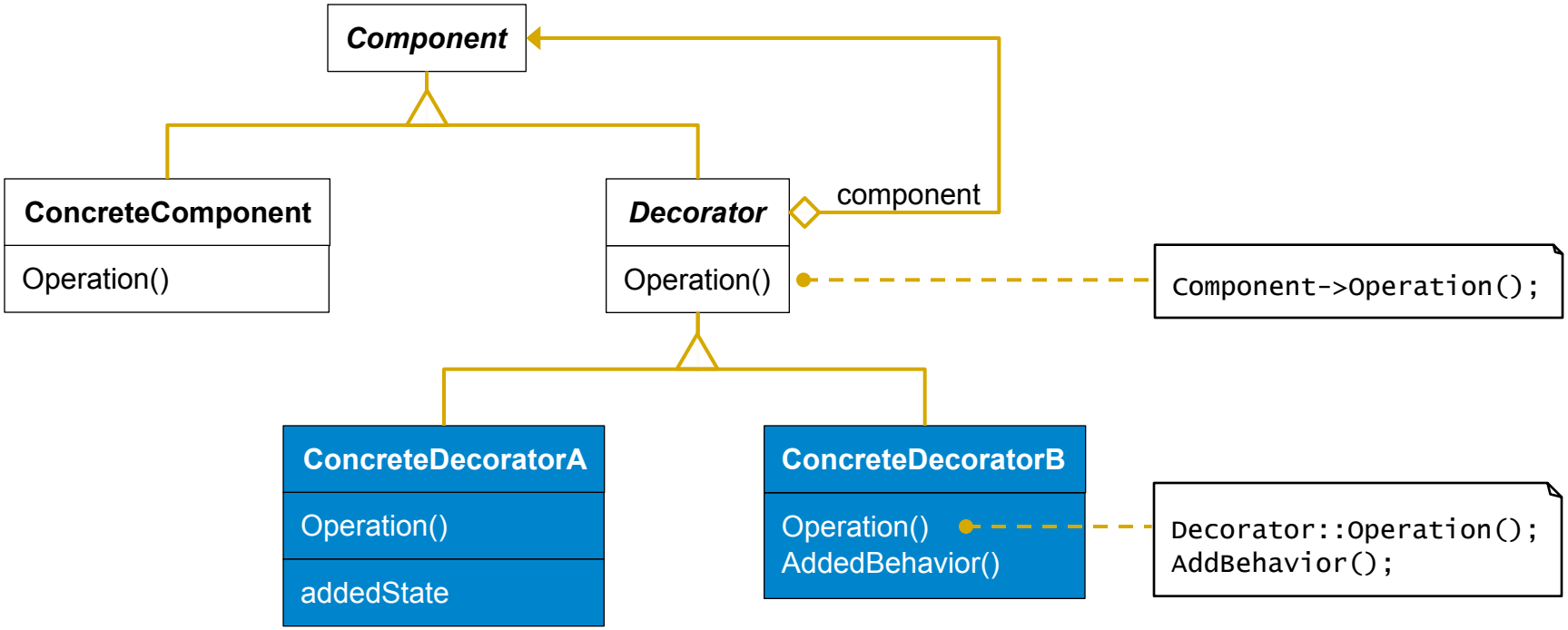
Decorator

- Maintains a reference to a Component object
- Defines an interface conformant to Component's interface



ConcreteDecorator

- Adds responsibilities to the component

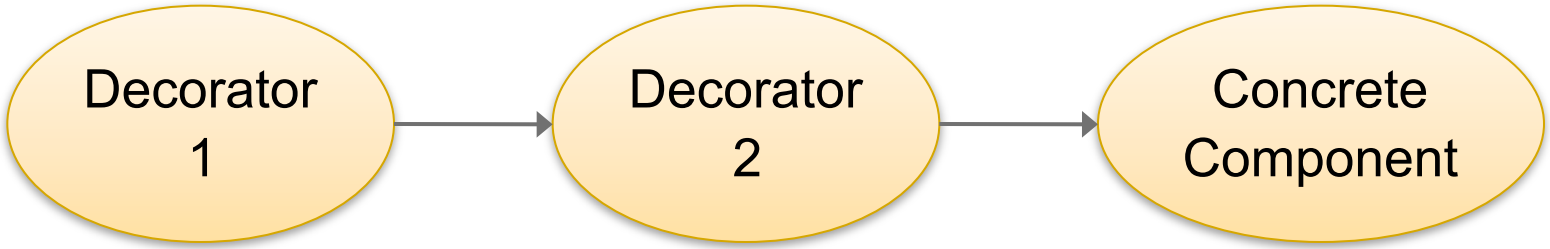


Example: Decorate Sales Ticket Printing

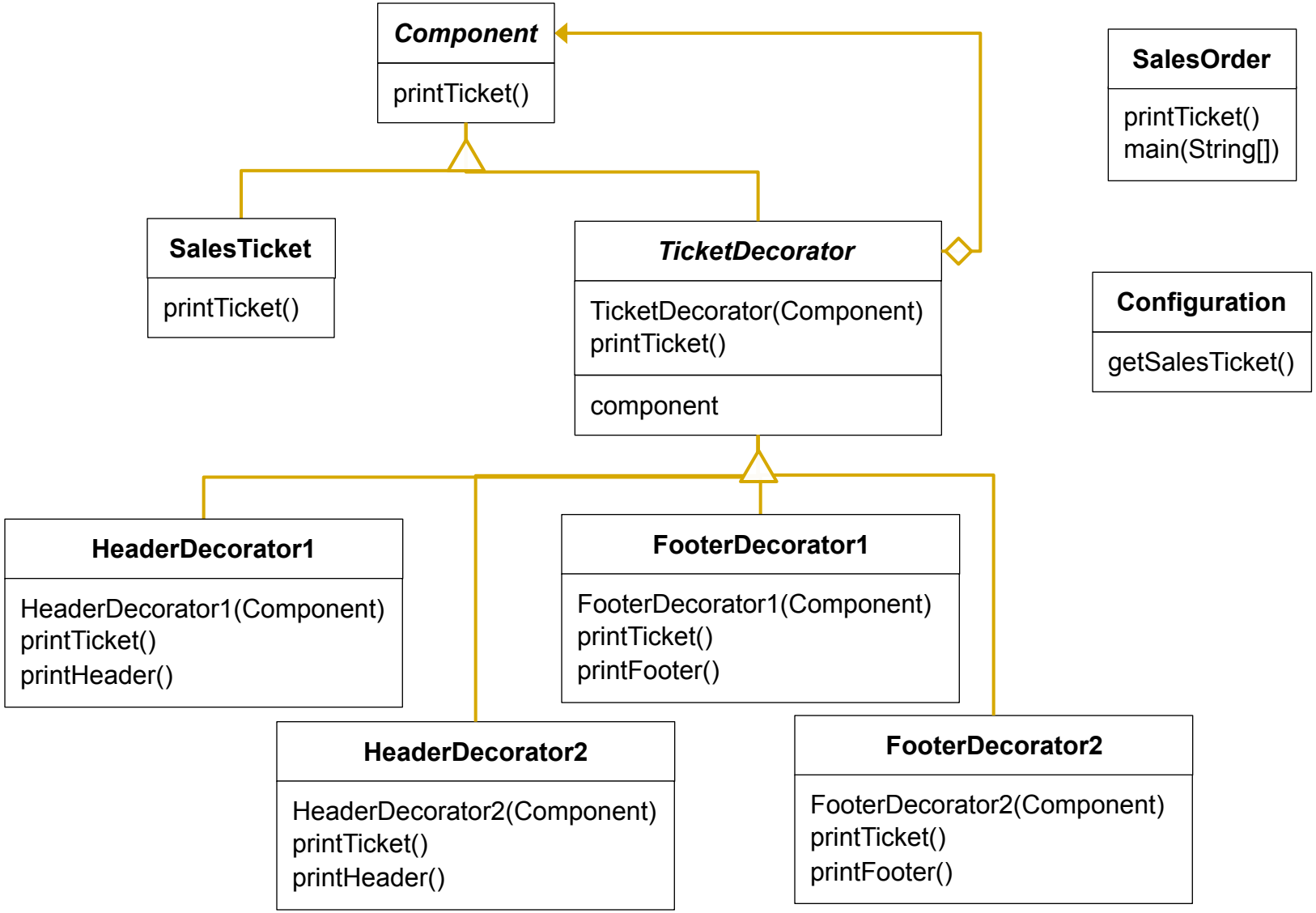
- Assume the SalesTicket currently creates an html sales receipt for an Airline Ticket
- New Requirements
  - Add header with company name
  - Add footer that is an advertisement
  - During the holidays add holiday relevant header(s) and footer(s)
  - We're not sure how many such things
- One solution
  - Place control in SalesTicket
    - Then you need flags to control what header(s) get printed

Decorator Approach

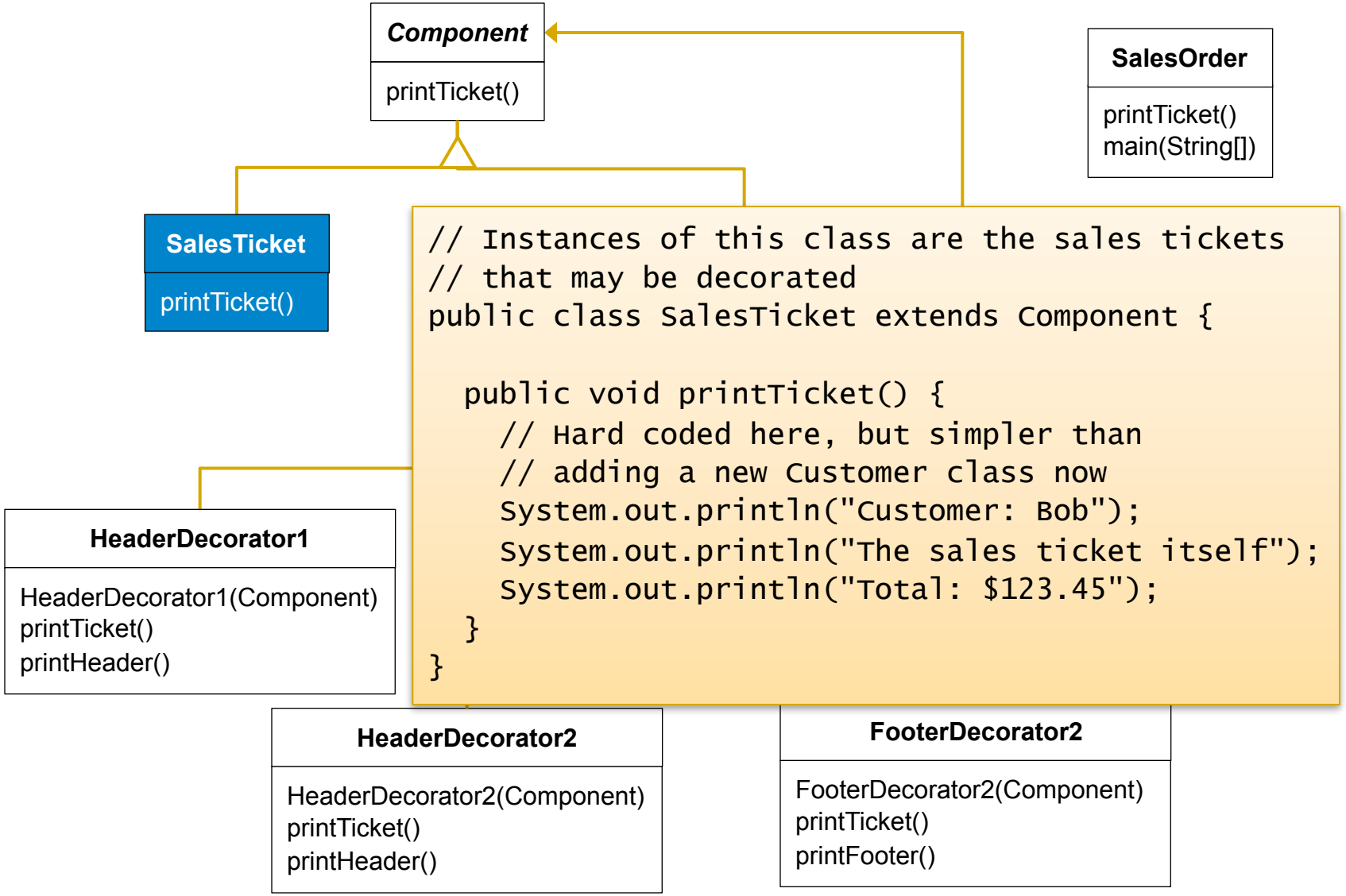
- A layered approach
  - Start chain with decorators
  - End with original object



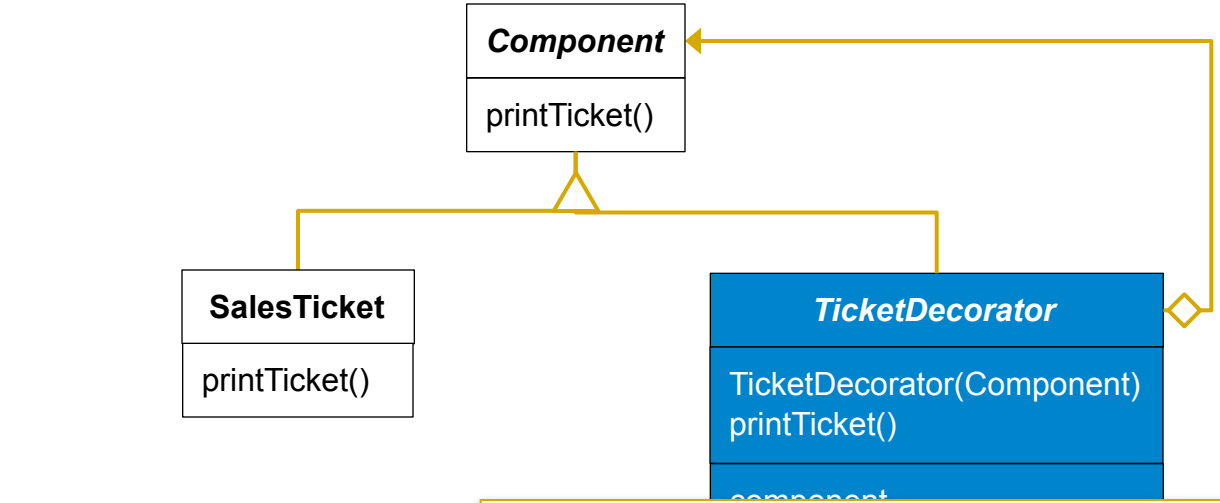
Example – Sales Ticket Printing



A SalesTicket Implementation

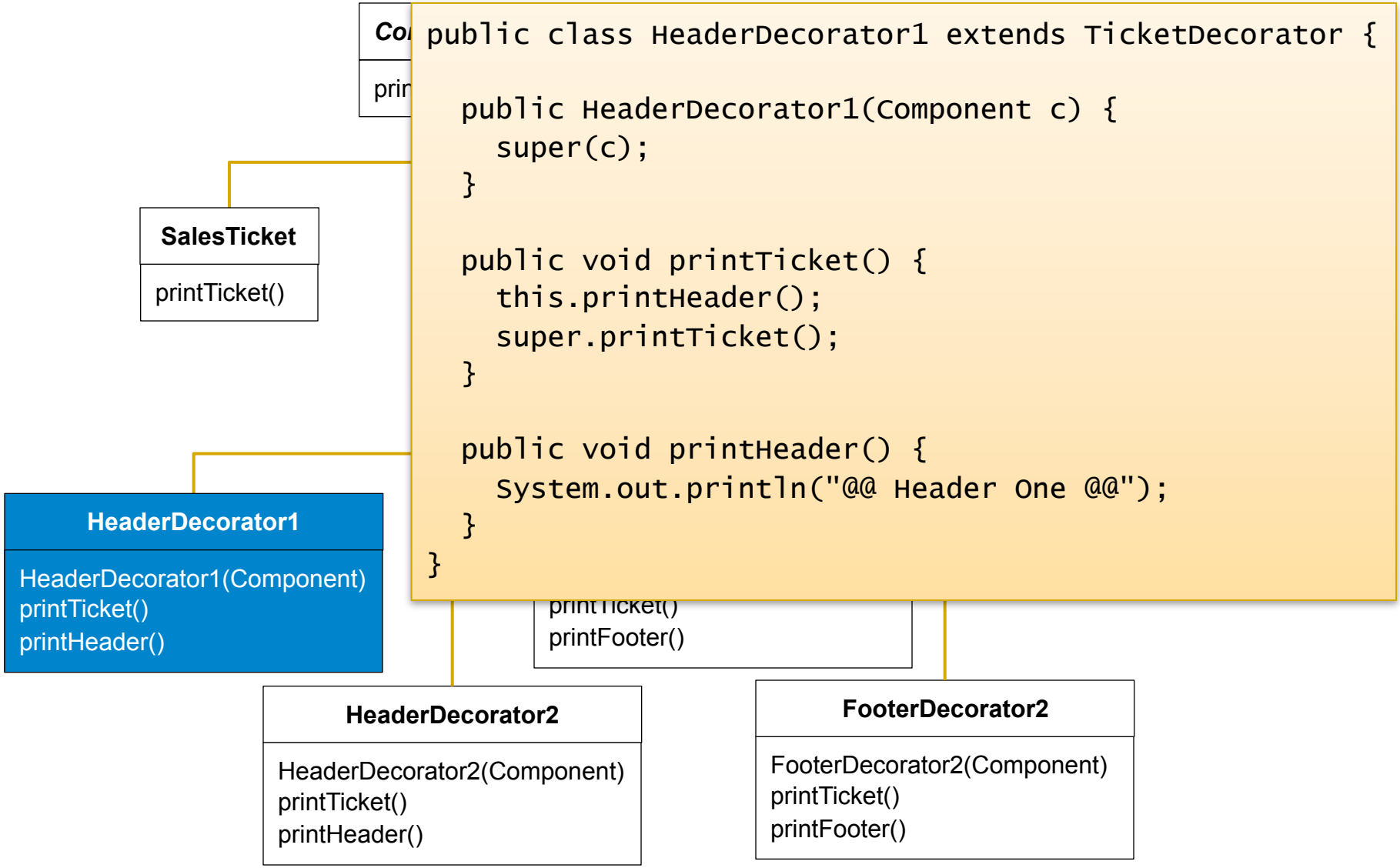


TicketDecorator



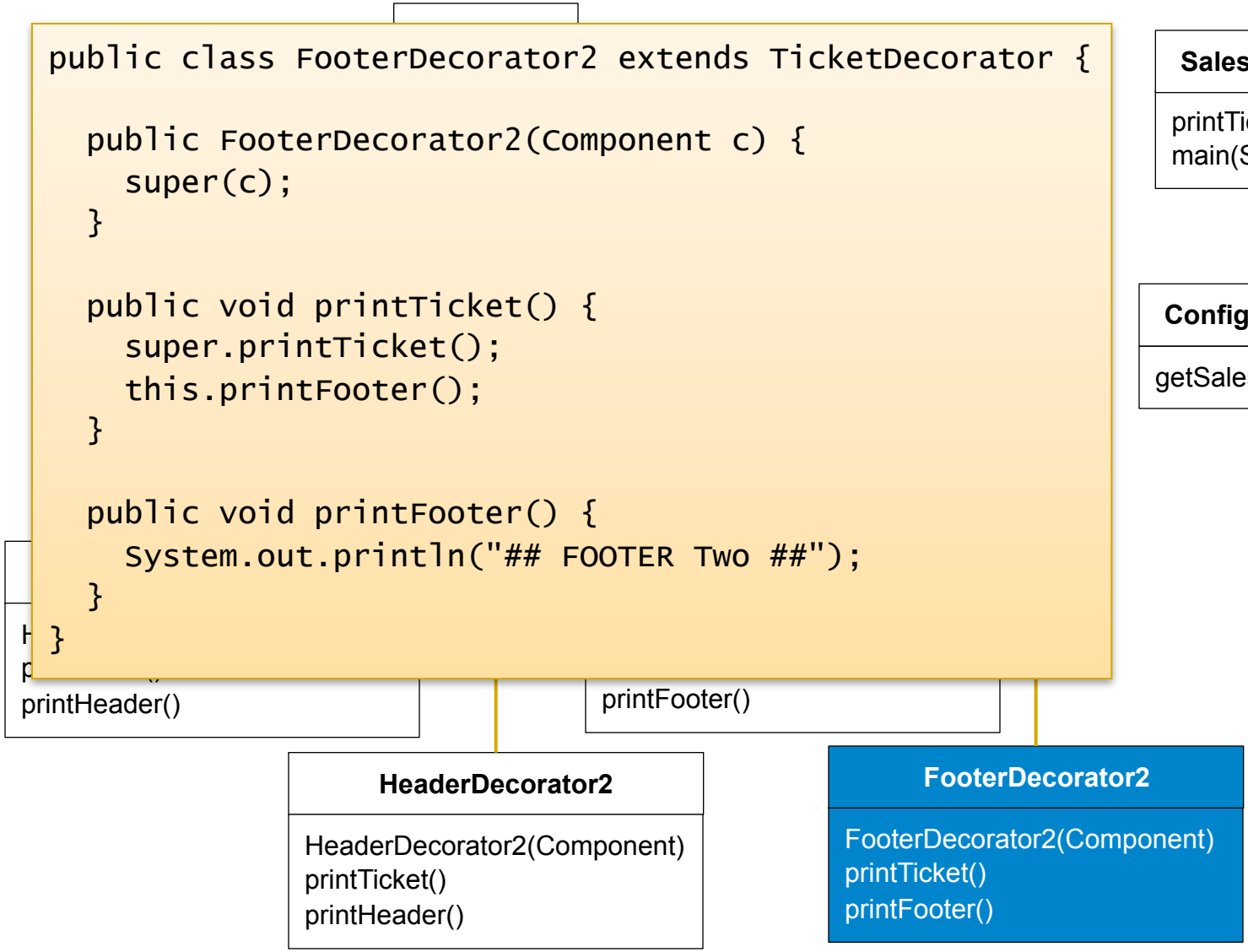

```
public abstract class TicketDecorator extends Component {

  private Component component;

  public TicketDecorator(Component c) {
    component = c;
  }

  public void printTicket() {
    if(component != null)
      component.printTicket();
  }
}
```

A Header Decorator

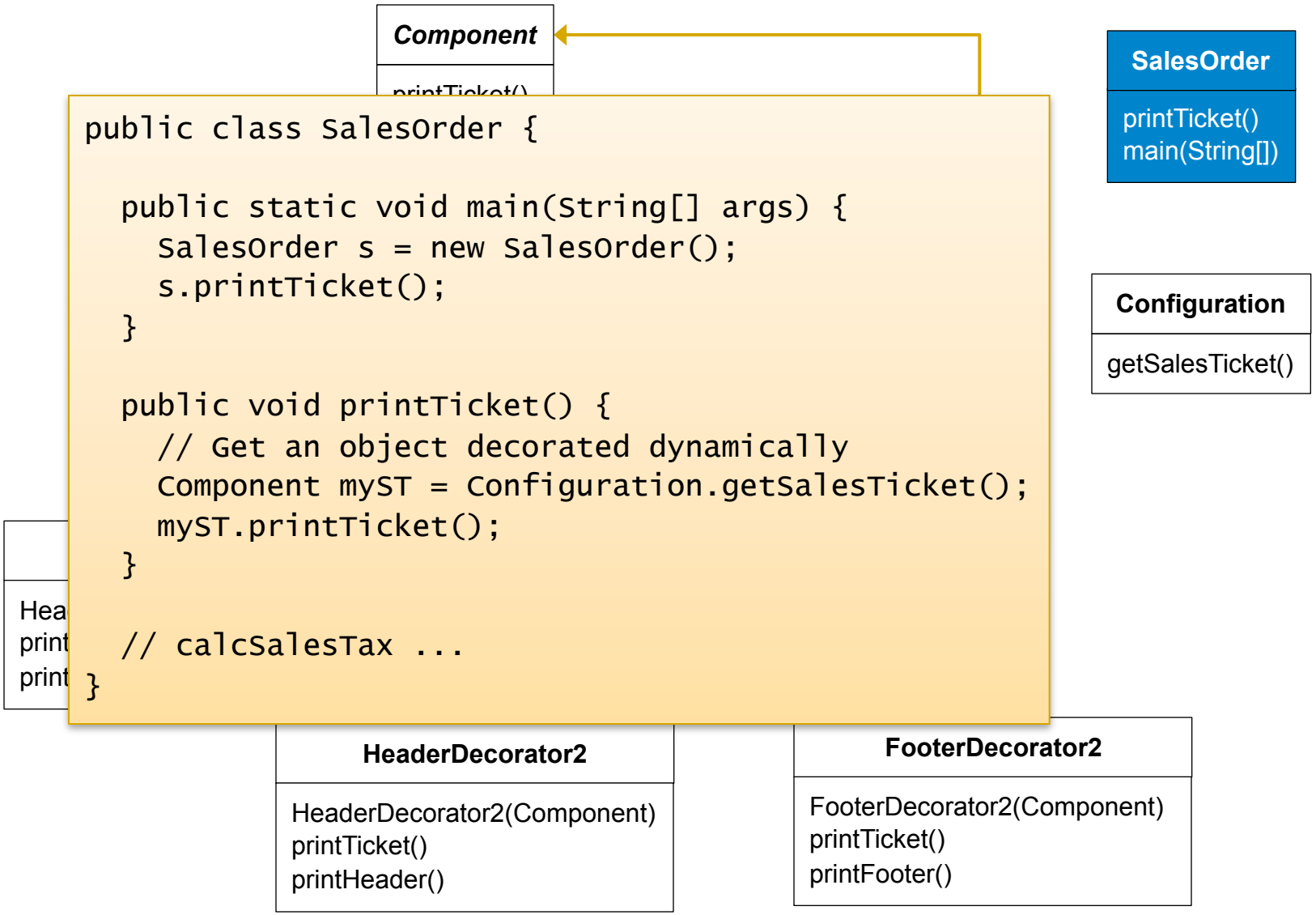


Example – Sales Ticket Printing

```
public class FooterDecorator2 extends TicketDecorator {

  public FooterDecorator2(Component c) {
    super(c);
  }

  public void printTicket() {
    super.printTicket();
    this.printFooter();
  }

  public void printFooter() {
    System.out.println("## FOOTER Two ##");
  }
}
```

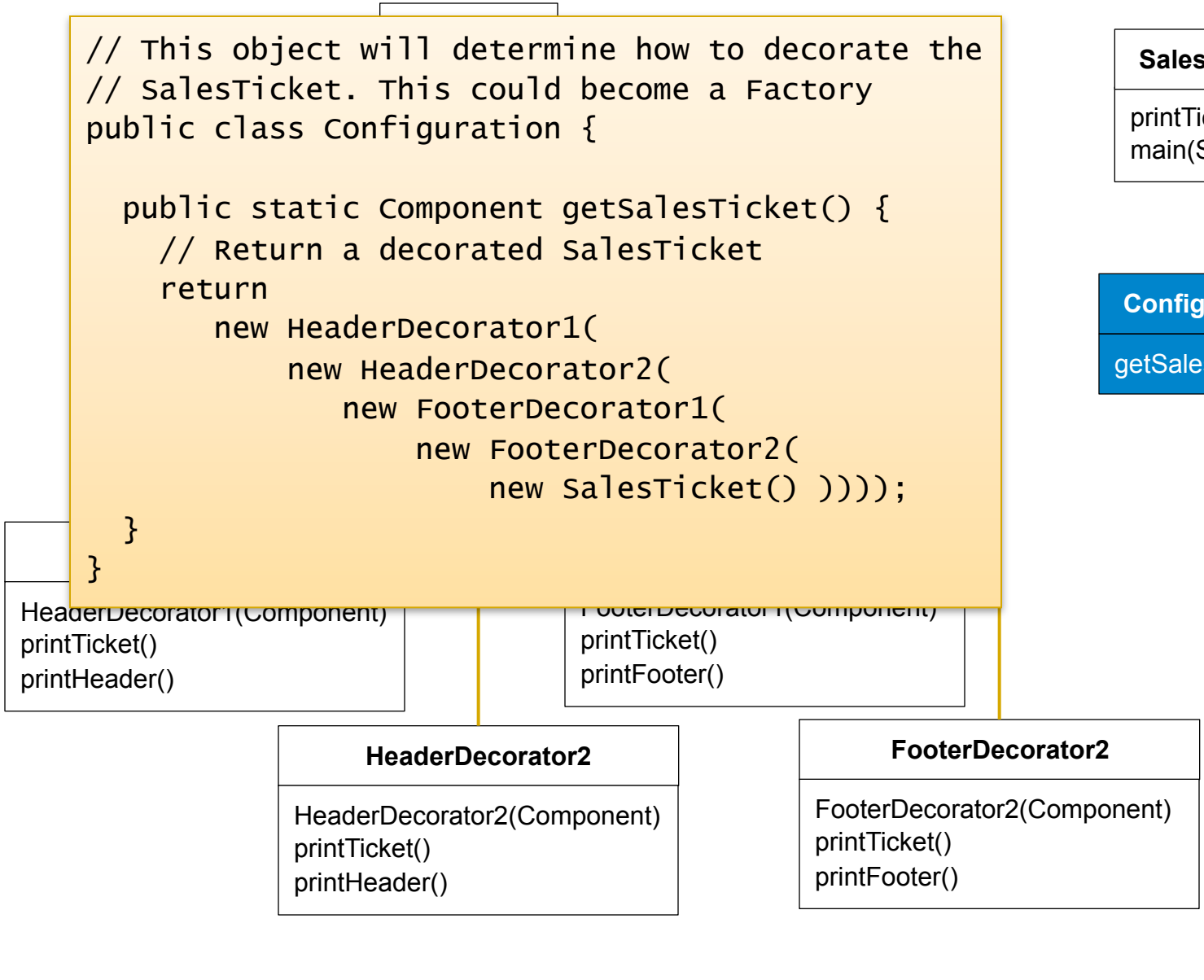


SalesOrder (Client)



Example Configuration

```
// This object will determine how to decorate the
// SalesTicket. This could become a Factory
public class Configuration {

  public static Component getSalesTicket() {
    // Return a decorated SalesTicket
    return
       new HeaderDecorator1(
           new HeaderDecorator2(
              new FooterDecorator1(
                  new FooterDecorator2(
                      new SalesTicket() ))));
  }
}
```



Output with Current Configuration

- Output:

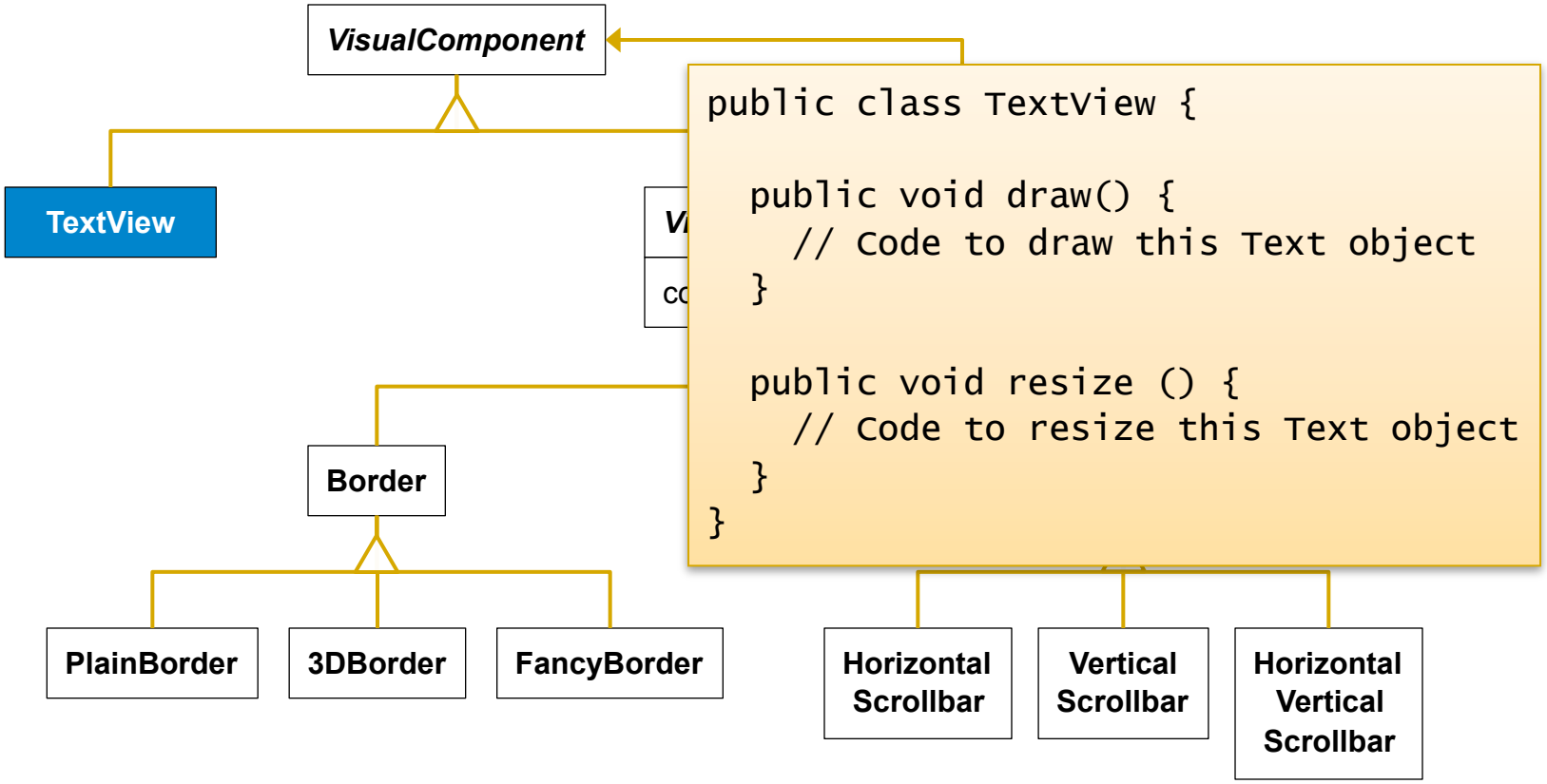
```
@@ Header One @@
>> Header Two <<
Customer: Bob
The sales ticket itself
Total: $123.45
%% FOOTER One %%
## FOOTER Two ##
```

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

Return to TextView Example

- The TextView class knows nothing about Borders and Scrollbars



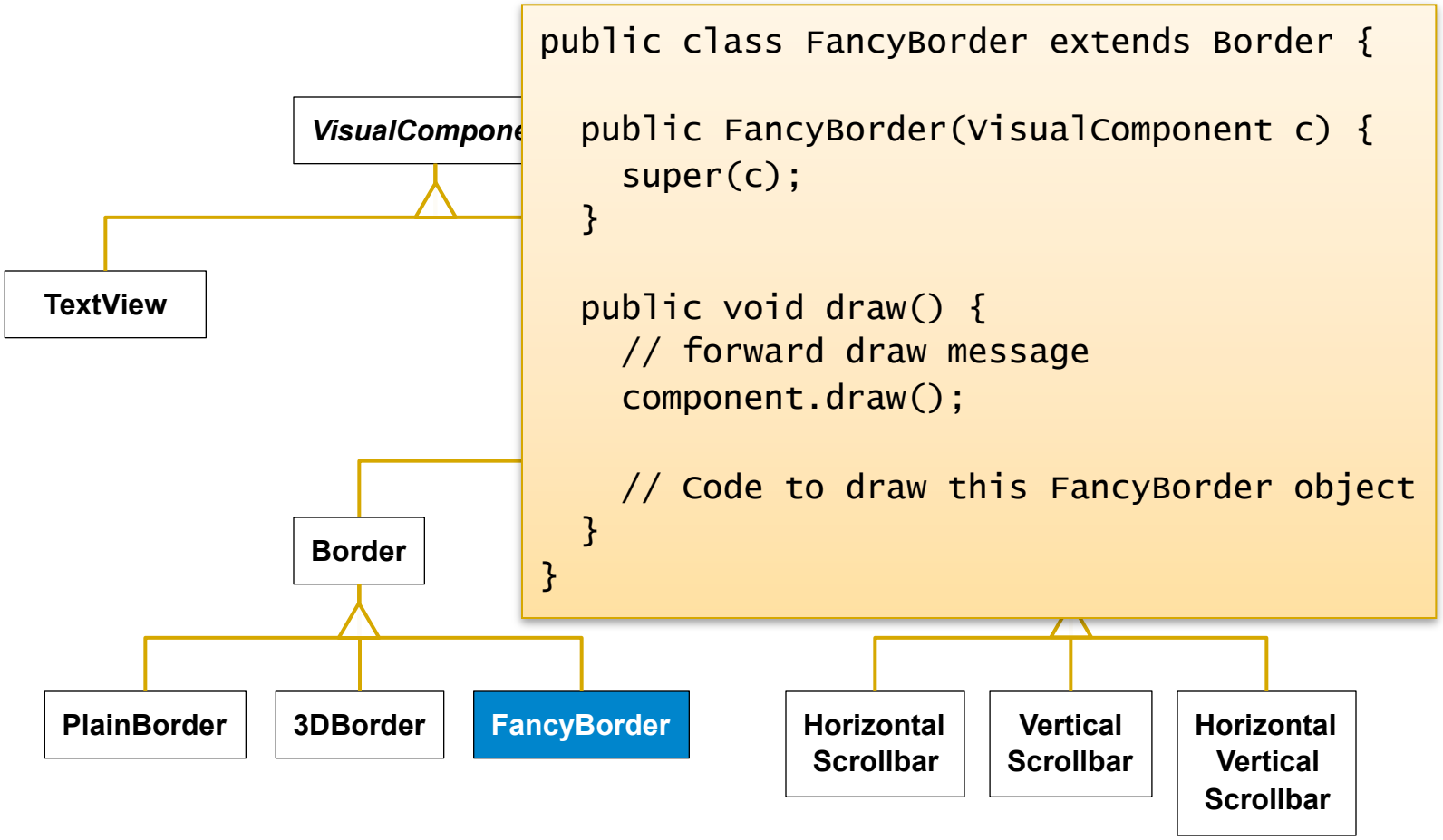
A New Class

- The new ImageView class knows nothing about Borders and Scrollbars

```
public class ImageView {

  public void draw() {
    // Code to draw this Image Object
  }

  public void resize () {
    // Code to resize this Image Object
  }
}
```

Decorators Contain Components

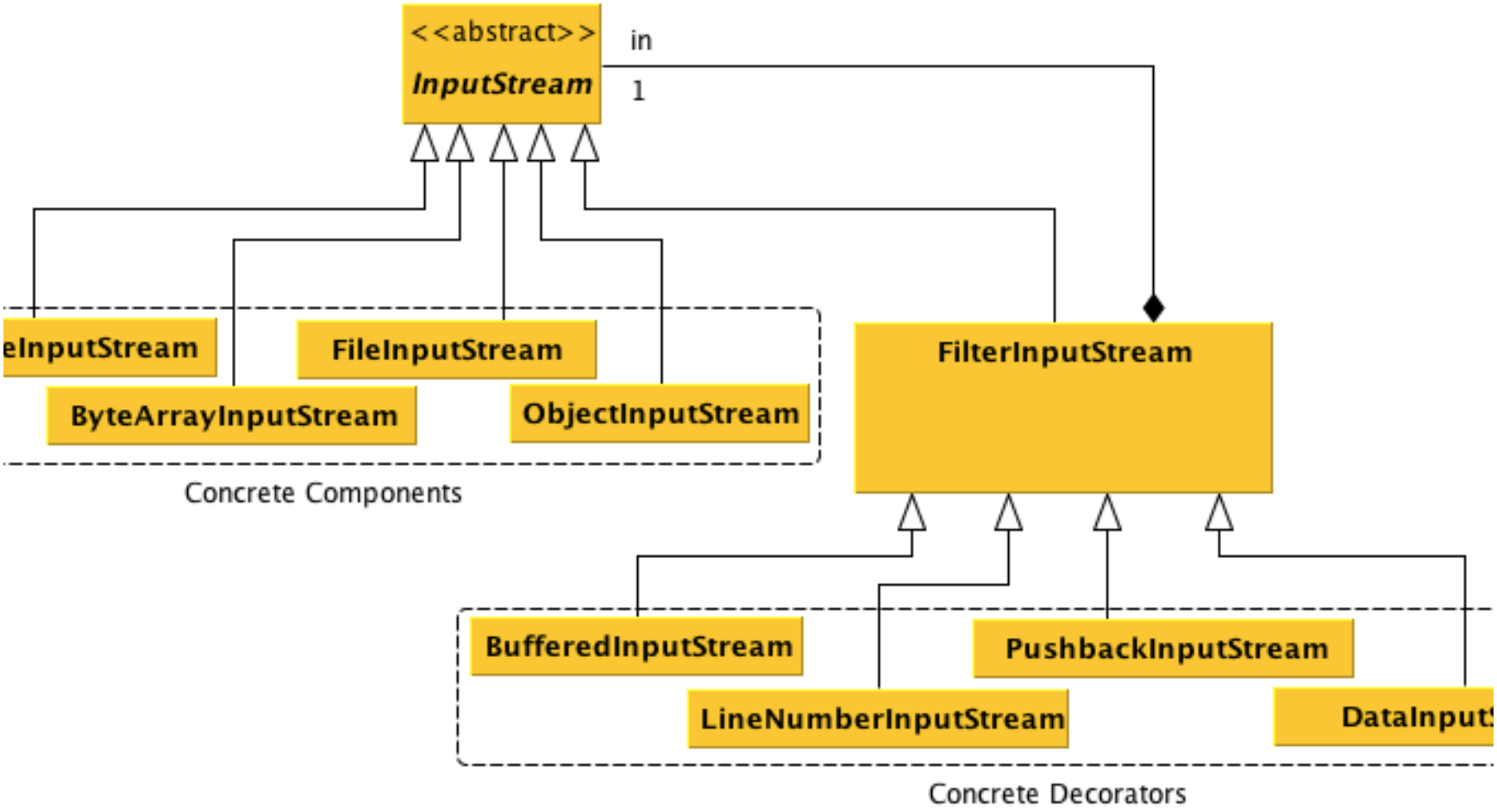
- The decorators don't need to know about components



How to Use Decorators

```
public class Client {

  public static void main(String[] args) {
   TextView data = new TextView();

   Component borderData = new FancyBorder(data);

   Component scrolledData = new VertScrollbar(data);

   Component borderAndScrolledData = new HorzScrollbar(borderData);
  }
}
```

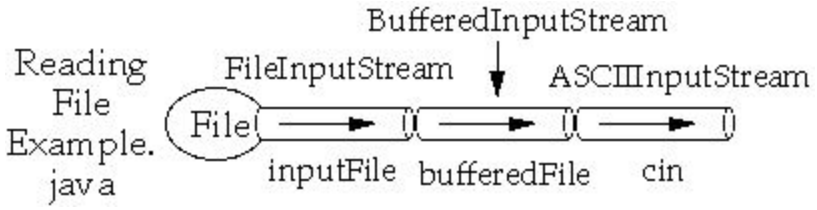
Decorator Pattern in Java



Decorator Pattern in Java

```
public class JavaIO {

  public static void main(String[] args) {
    // Open an InputStream.
    FileInputStream in = new FileInputStream("test.dat");
    // Create a buffered InputStream.
    BufferedInputStream bin = new BufferedInputStream(in);
    // Create a buffered, data InputStream.
    DataInputStream dbin = new DataInputStream(bin);
    // Create a buffered, pushback, data InputStream.
    PushbackInputStream pbdbin = new PushbackInputStream(dbin);
  }
}
```

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



Java Streams

- With > 60 streams in Java, you can create a wide variety of input and output streams
  - This provides flexibility (good)
  - It also adds complexity (bad)
  - Flexibility made possible with inheritance and classes that accept many different classes that extend the parameter
- You can have an InputStream instance or any instance of a class that extends InputStream

```
public InputStreamReader(InputStream in)
```

Consequences

- + Transparency – very good
- + 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

- × A decorator and its component aren't identical
- × Lots of little objects
  - Easy to customize, but hard to learn and debug