

Design Patterns

Decorator Pattern

ebru@hacettepe.edu.tr

ebruakcapinarsezer@gmail.com





<http://yunus.hacettepe.edu.tr/~ebru/>

@ebru176

Ekim 2017



Let's try to design

- Each football player has the abilities of running, shooting, passing
- At initial stage these abilities are assigned to players randomly
- After training stage, these abilities are improved with different degrees and some players gain special abilities: leftfoot, long pass to the point
- While playing, in your PS4 dualshock unit,  means pass,  means shoot,  means run and  means stop
- Each player has to fulfill the order coming from dualshock according to his abilities with the consideration of best choice selection

Another attempt

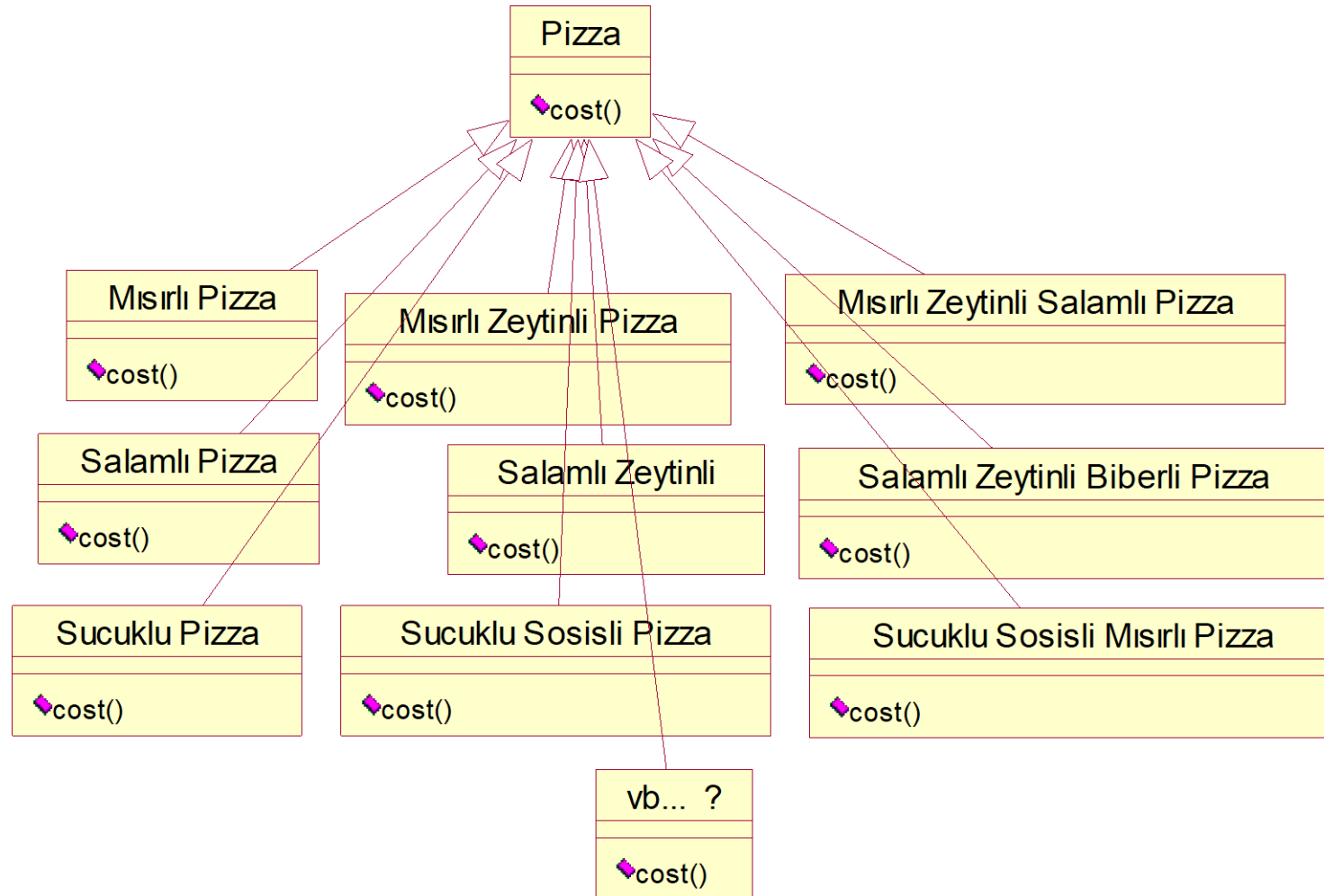
- What type of pizza would be ordered?
- You have various ingredient
sausage, corn, salami, pepper, vb.
- Each of customer can request different porsions from different ingredients
«corn, salami and pepper but pepper should used double porsions!»

Request it the calculation of the pizza price?



Solution-1

- Estimate all of the possibilities



Java Code – Comment Design

```
public abstract class Pizza {  
    public abstract double cost();  
}  
  
public class MısırlıPz extends Pizza{  
    public double cost(){  
        return 0.8;  
    }  
}  
  
public class MısırlıZeytinliPz extends Pizza{  
    public double cost(){  
        return 1.4;  
    }  
}  
  
public class MısırlıZeytinliSalamlıPz extends Pizza{  
    public double cost(){  
        return 2.1;  
    }  
}
```

```
public class Mc {  
    public static void main(String[] args) {  
  
        MısırlıPz msrPiz = new MisirliPz();  
  
        MısırlıZeytinliPz msrZeyPiz = new MisirliZeytinliPz();  
  
        MısırlıZeytinliSalamlıPz msrZeySalPiz = new MisirliZeytinliSalamlıPz();  
    }  
}
```

Java Code – Comment Design

```
public abstract class Pizza {  
    public abstract double cost();  
}
```

```
public class MısırlıPz extends Pizza {  
    public double cost(){  
        return 0.8;  
    }  
}
```

```
public class MısırlıZeytinliPz extends Pizza {  
    public double cost(){  
        return 1.4;  
    }  
}
```

```
public class MısırlıZeytinliSalamlıPz extends Pizza {  
    public double cost(){  
        return 2.1;  
    }  
}
```

```
public class Mc {  
    public static void main(String[] args) {
```

Too much subclasses

Hard to maintain

If cost of corn is changed?

If new ingredient is included?

```
        new MisirliZeytinliPz();
```

```
        pz = new MisirliZeytinliSalamliPz();
```

Solution-2: One class is enough, really?

Pizza	
	misir : Boolean
	salam : Boolean
	zeytin : Boolean
	hasMisir()
	hasSalam()
	hasZeytin()
	setMisiri()
	setSalam()
	setZeytin()
	cost()

Pls think about OO Thinking principles:

1-?

2-?

From GoF

3-?

4-?

5-?

The Decorator Pattern from GoF

- Intent
 - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing to extend flexibility
- Also Known As Wrapper

Motivation

- Sometimes we want to add responsibilities to individual objects, not to an entire class
- Inheritance is inflexible because the responsibilities statically
- A more flexible approach is to enclose the component in another object that adds the responsibilities
- The enclosing object is called Decorator.

Motivation

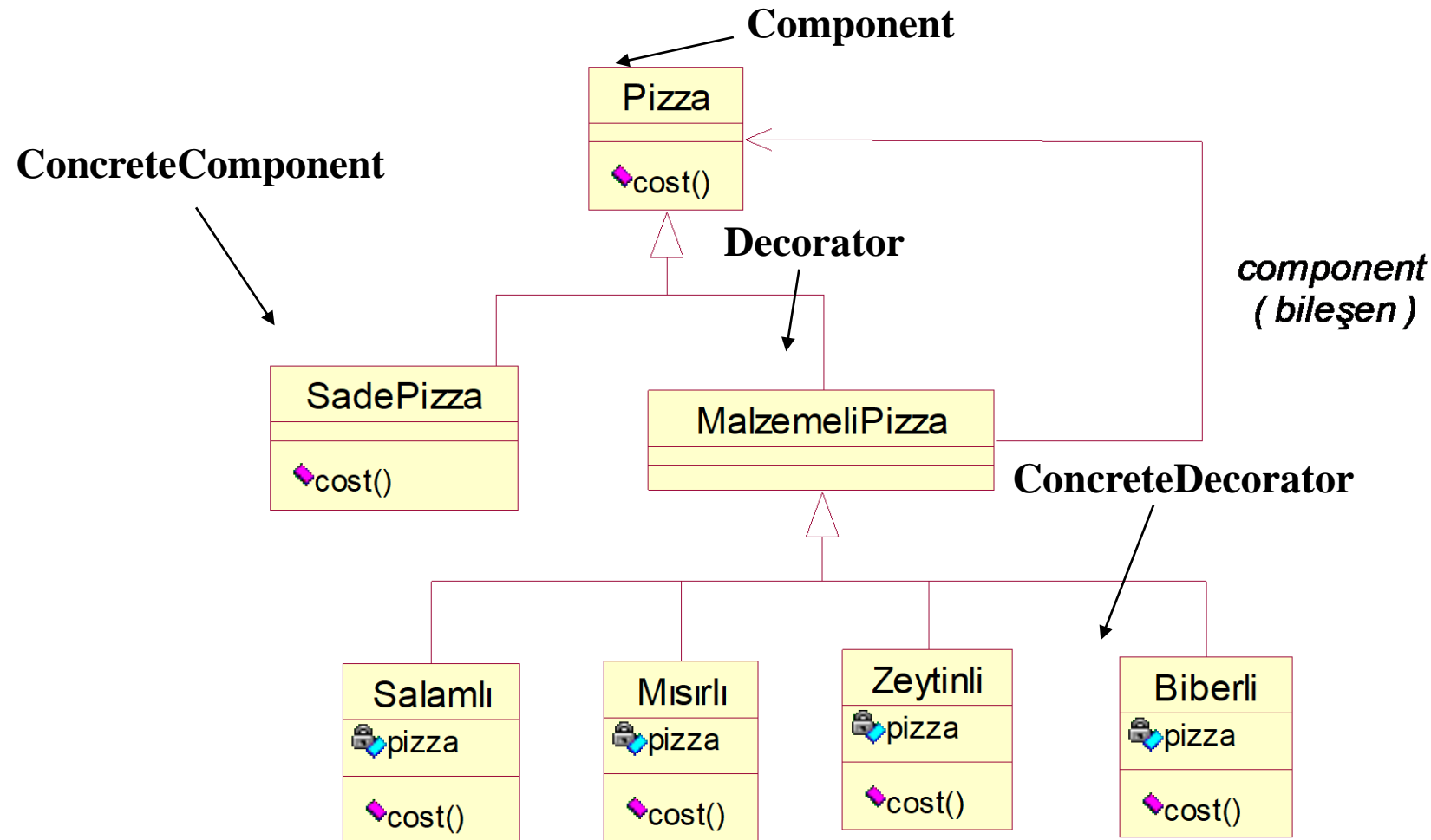
- The decorator conforms to the interface of the component it decorates so that **its presence is transparent to the clients**
- The decorator forwards **requests to the component** and may perform additional actions before and/or after forwarding
- Transparency lets you nest decorators recursively
- Decorator subclasses are free to add operations for specific functionality
- This pattern lets decorators appear anywhere a VisualComponent can

Applicability: as a nutshell

- Use Decorator
 - to add responsibilities to individual objects dynamically and transparently
 - for responsibilities that can be withdrawn
 - when extension by subclassing is impractical

Come back: Pizza Solution

- Each of ingredient should be assessed as new functionality



Java Code

```
public abstract class Pizza {
    public abstract double cost();
}

public abstract class MalzemeliPizza extends Pizza{
    /*...*/}

public class SadePizza extends Pizza{
    public double cost(){
        return 1.5;}}

public class Salamli extends MalzemeliPizza{
    protected Pizza pizza;
    public Salamli(Pizza pizza){
        this.pizza = pizza;
    }
    public double cost(){
        return 2.6 + pizza.cost();}}

public class Zeytinli extends MalzemeliPizza{
    protected Pizza pizza;
    public Zeytinli(Pizza pizza){
        this.pizza = pizza;}
    public double cost(){
        return 1.3 + pizza.cost();}}
```

```
public class Misirli extends MalzemeliPizza{
    protected Pizza pizza;
    public Misirli (Pizza pizza){
        this.pizza = pizza;    }
    public double cost(){
        return 1.8 + pizza.cost();    }
}

public classDominosPizza{
    Pizza pizza = new SadePizza();
    Sop( pizza.cost() );

    Pizza pizza2 = new SadePizza ( );
    pizza2 = new Salamli( pizza2 );
    pizza2 = new Zeytinli( pizza2 );
    pizza2 = new Misirli ( pizza2 );
    Sop( pizza2.cost() );

    Pizza pizza3 = new SadePizza();
    pizza3 = Salamli( pizza3 );
    pizza3 = Salamli( pizza3 );
    pizza3 = Misirli( pizza3 );
    Sop( pizza3.cost() );}
```

* *Sop* => *System.out.println*

Java Code

```
public abstract class Pizza {  
    public abstract double cost();  
}
```

```
public abstract class MalzemeliPizza extends Pizza{  
    /*...*/}
```

```
public class SadePizza extends MalzemeliPizza{  
    public double cost(){  
        return 1.5;}}
```

```
public class Salamli extends MalzemeliPizza{  
    protected Pizza pizza;  
    public Salamli(Pizza p){  
        this.pizza = p;  
    }
```

```
    public double cost(){  
        return 2.6 + pizza.cost();}}
```

```
public class Zeytinli extends MalzemeliPizza{  
    protected Pizza pizza;  
    public Zeytinli(Pizza pizza){  
        this.pizza = pizza;}  
    public double cost(){  
        return 1.3 + pizza.cost();}}
```

```
public class Misirli extends MalzemeliPizza{  
    protected Pizza pizza;  
    public Misirli (Pizza pizza){  
        this.pizza = pizza;    }  
    public double cost(){  
        return 1.8 + pizza.cost();    }  
}
```

Lesser subclassess

Easy to change ingredients

public class YeniMalzeme extends MalzemeliPizza{...

Possible to aggregate all of ingredients in any order

```
Sop( pizza2.cost() );
```

```
Pizza pizza3 = new SadePizza();
```

```
pizza3 = Salamli( pizza3 );
```

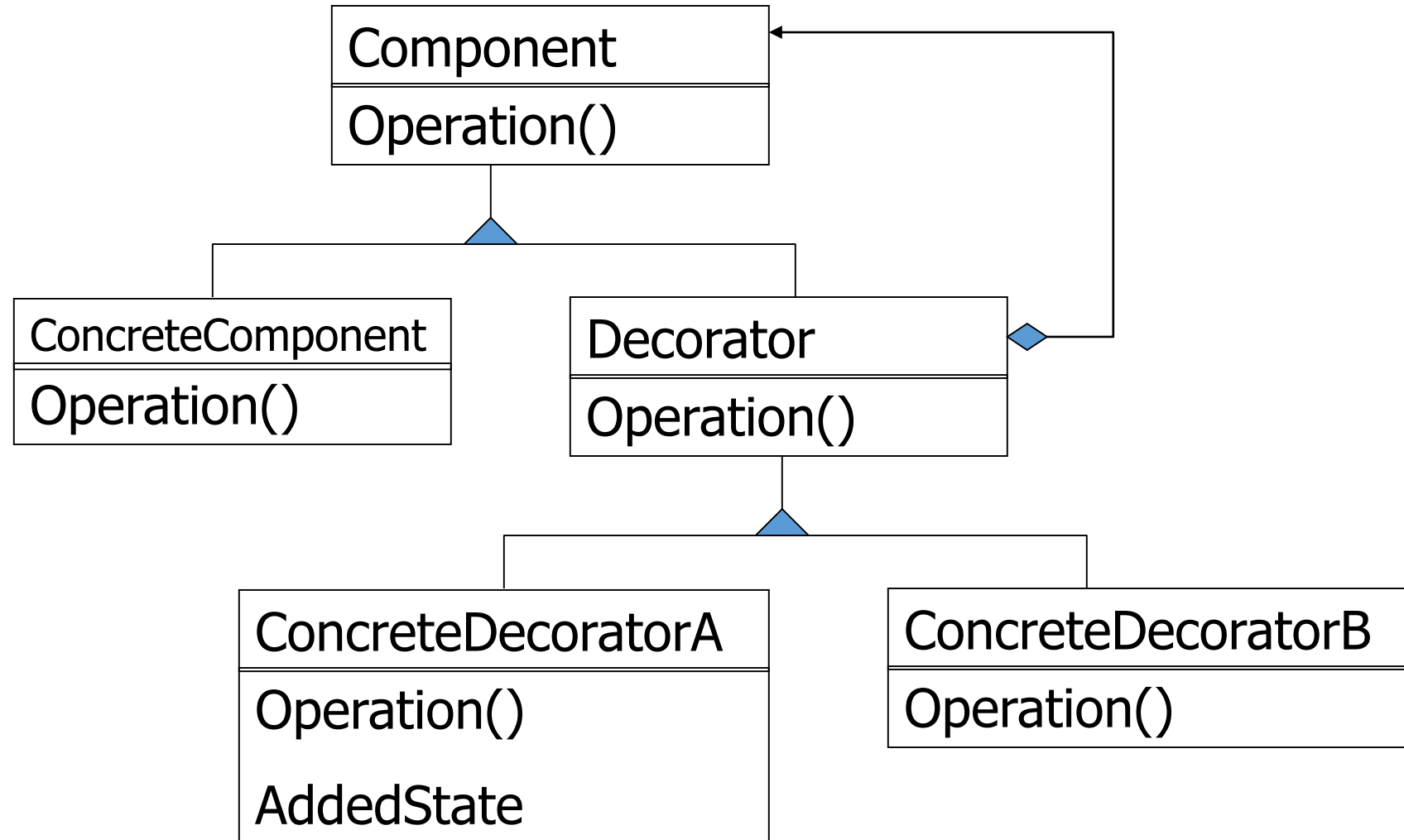
```
pizza3 = Salamli( pizza3 );
```

```
pizza3 = Misirli( pizza3 );
```

```
Sop( pizza3.cost() );}
```

* Sop => System.out.println

Structure



Participants

- Component:
defines the interface for objects that can have responsibilities added to them dynamically.
- Concrete Component
defines an object to which additional responsibilities can be attached.
- Decorator
maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- Concrete Decorator
adds responsibilities to the component.

Collaborations

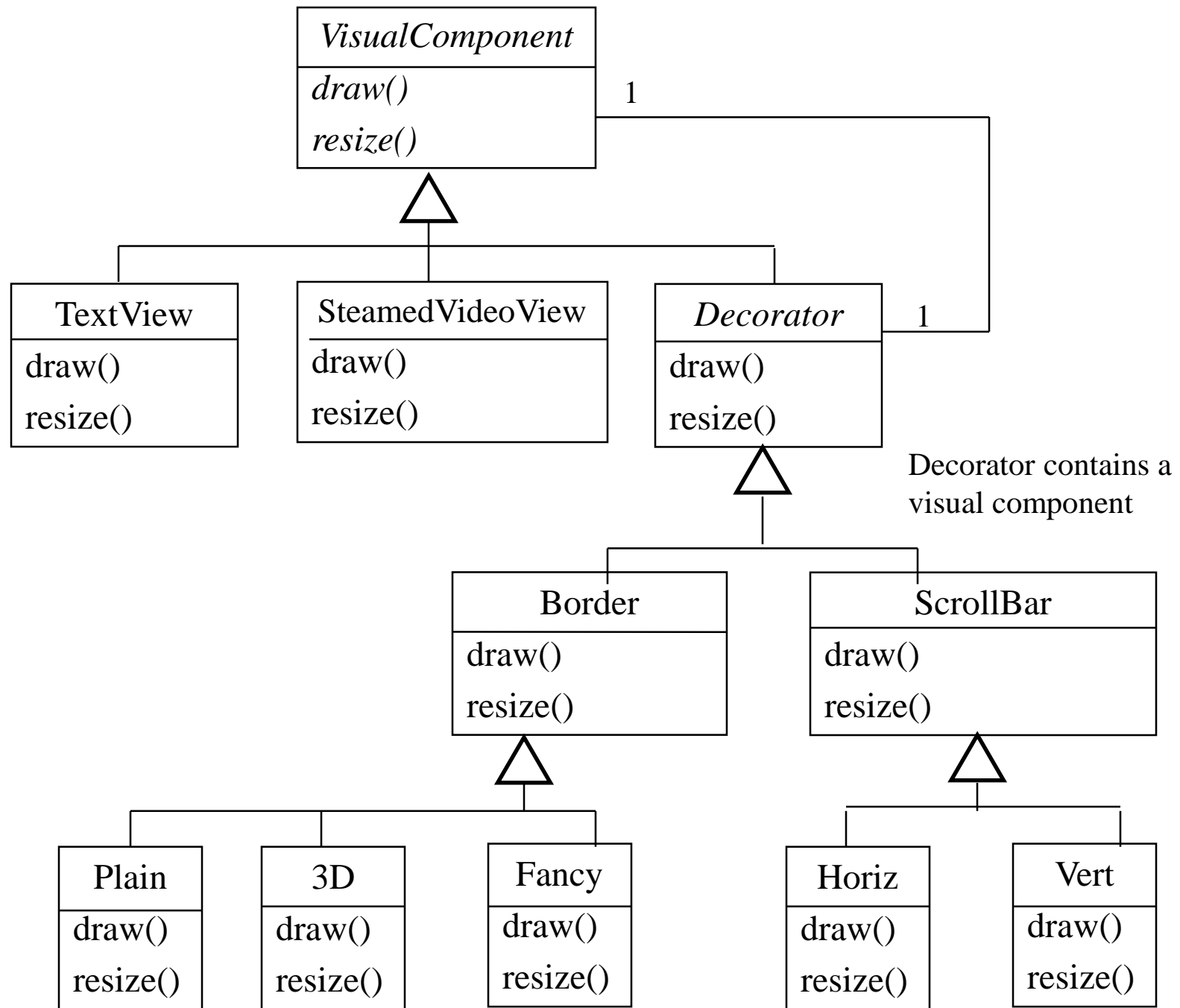
- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.
- In which DP category Decorator should be placed?

An Application

- Suppose there is a TextView GUI component and you want to add different kinds of borders and/or scrollbars to it
- You can add 3 types of borders
 - Plain, 3D, Fancy
- and , 1, or 2 two scrollbars
 - Horizontal and Vertical
- An inheritance solution requires 15 classes for one view

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



Sample Code(1)

- Sample code for the base Component class

```
class VisualComponent {  
public:  
    VisualComponent();  
    virtual void Draw();  
    virtual void Resize();  
}
```

- Sample code for base Decorator

```
class Decorator : public  
    VisualComponent  
{public:  
    Decorator(VisualComponent *);  
    virtual void Draw();  
    virtual void Resize();  
private:  
    VisualComponent * _component;  
}
```

Sample Code(2)

- The default implementation that passes the request on to _component:

```
void Decorator::Draw(){  
    _component->Draw();  
}  
void Decorator::Resize(){  
    _component->Resize();  
}
```

Sample Code(4)

- Sample code for concrete decorator

```
class BorderDecorator: public Decorator{
public:
    BorderDecorator(VisualComponent*,
        int borderWidth);
    virtual void Draw();
private:
    void DrawBorder(int);
    int _width;
}
void BorderDecorator::Draw(){
    Decorator::Draw();
    DrawBorder(_width);
}
```

- Use decorators

- void Window::SetContents(VisualComponent * contents){ ... }
- Window *window = new Window;
window->SetContents(
 new BorderDecorator(
 new ScrollDecortor(textView), 1));

Consequences(1)

- More flexibility than static inheritance.
 - responsibilities can be added and removed at run-time simply by attaching and detaching them.
 - Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities.

Consequences(2)

- Avoids feature-laden classes high up in the hierarchy.
 - Offers a pay-as-you-go approach to adding responsibilities.
 - You can define a simple class and add functionality incrementally with Decorator objects.
 - Also easy to define new kinds of Decorators independently.

Consequences(3)

- A decorator and its component aren't identical.
 - From an object identity point of view, a decorated component is not identical to the component itself.
 - Hence you shouldn't rely on object identity when you use decorators.

Consequences(4)

- Lots of little objects
 - Often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables.
 - Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Implementation(1)

- Interface conformance
 - A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must inherit from a common class.
- Omitting the abstract Decorator class
 - There's no need to define an abstract Decorator class when you only need to add one responsibility.
 - That's often the case when you're dealing with an existing class hierarchy rather than designing a new one.

Implementation(2)

- Keeping Component classes lightweight.
 - Components and decorators must descend from a common Component class. It's important to keep this common class lightweight: focus on defining an interface, not on storing data.
 - Putting a lot of functionality into component also increases the probability that concrete subclasses will pay for features they don't need.

Implementation(3)

- Changing the skin of an object versus changing its guts
 - think of a decorator as a skin over an object that changes its behavior.
 - An alternative is to change the object's guts like Strategy pattern does.
 - Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly.

Implementation(4)

- Changing the skin of an object versus changing its guts
 - Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorator. With strategies, the component itself knows about possible extensions.
 - A strategy can have its own specialized interface, whereas a decorator's interface must conform to the component's.

Know Uses in Java

- Many object-oriented user interface toolkits use decorators to add graphical embellishment to widgets.
- Streaming classes.

Java Borders

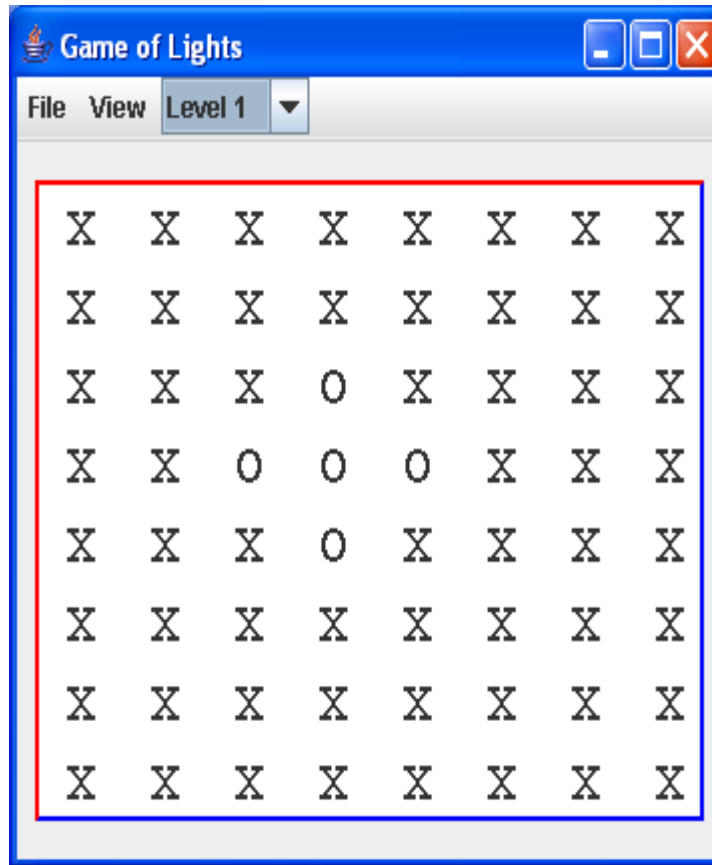
- Any JComponent can have 1 or more borders
- Borders are useful objects that, while not themselves components, know how to draw the edges of Swing components
- Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components

For more on Borders, The Java Tutorial

<http://java.sun.com/docs/books/tutorial/uiswing/components/border.html>

Java Code: Add a Beveled Border

```
toStringView.setBorder(new BevelBorder(  
    BevelBorder.LOWERED, Color.BLUE, Color.RED));
```



Decorator Pattern in Java

- `InputStreamReader(InputStream in)` *System.in is an InputStream object*
 - ... bridge from byte streams to character streams: It reads bytes and translates them into characters using the specified character encoding.
Java™API
- `BufferedReader`
 - Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
Java™API
- What we has to do for console input up to Java 1.4 *before Scanner*

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

Let's try to desing

- Each football player has the abilities of running, shooting, passing
- At initial stage this abilities are assigned to players randomly
- After training stage, these abilities are improved with different degrees and some players gain special abilities: leftfoot, long pass to the point
- While playing, in your PS4 dualshock unit, ▲ means pass, ■ means shoot, ● means run and × means stop
- Each player has to fulfill the order coming from dulashock according to his abilities with the consideration of best choice selection